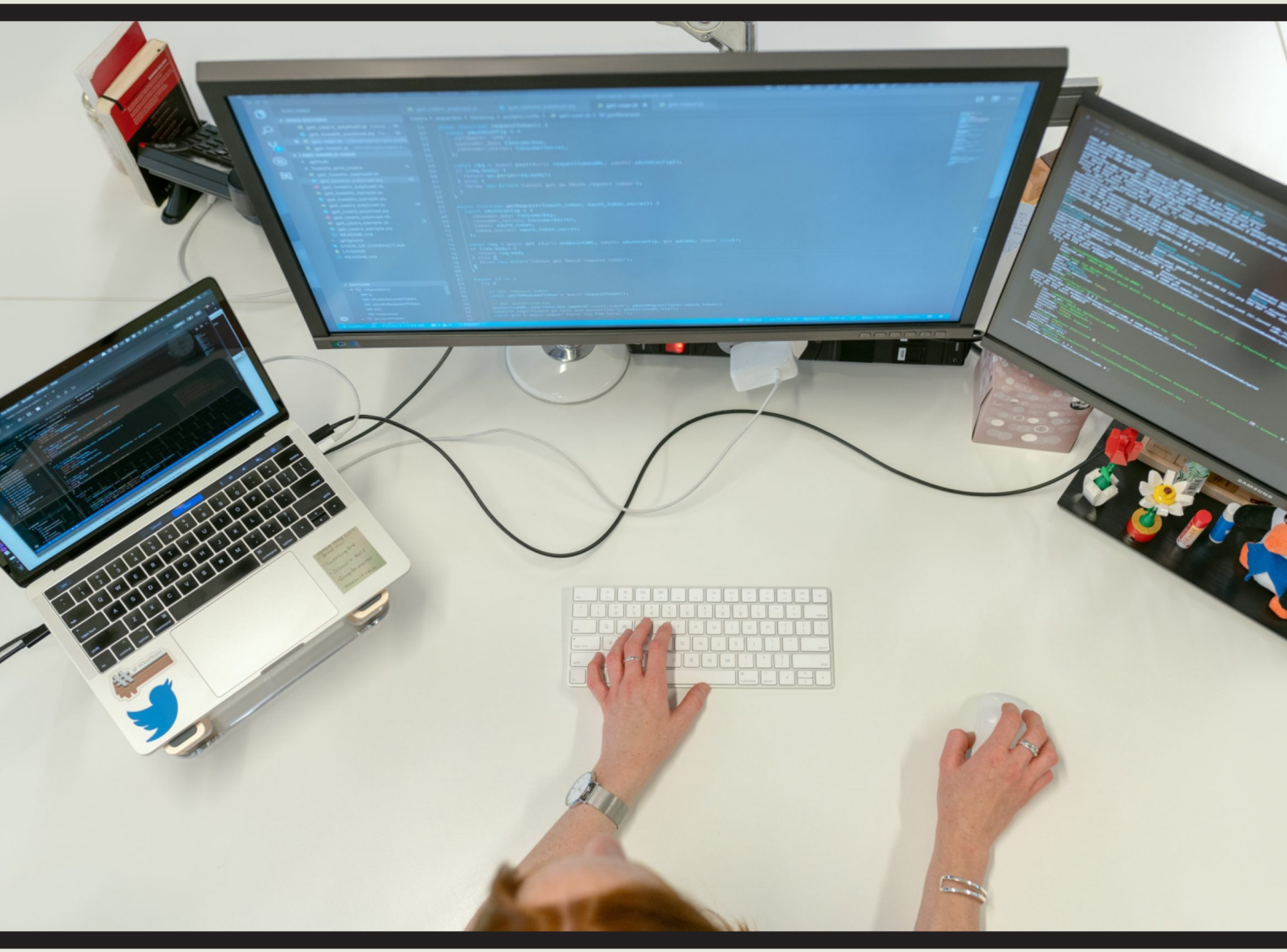
CPU Scheduling Algorithms

Bob Huber



CPU SCHEDULING ALGORITHMS

CPU SCHEDULING ALGORITHMS

Bob Huber

BIBLIOTEX
Digital Library

CPU Scheduling Algorithms
by Bob Huber

www.bibliotex.com

Ebook ISBN: 9781984664259

Published by:
Bibliotex
Canada
Website: www.bibliotex.com

Contents

1

Introduction

- What is CPU scheduling? Determining which processes run when there are multiple runnable processes. Why is it important? Because it can have a big effect on resource utilization and the overall performance of the system.
- By the way, the world went through a long period (late 80's, early 90's) in which the most popular operating systems (DOS, Mac) had NO sophisticated CPU scheduling algorithms. They were single threaded and ran one process at a time until the user directs them to run another process. Why was this true? More recent systems (Windows NT) are back to having sophisticated CPU scheduling algorithms. What drove the change, and what will happen in the future?
- Basic assumptions behind most scheduling algorithms:

  - There is a pool of runnable processes contending for the CPU.
  - The processes are independent and compete for resources.
  - The job of the scheduler is to distribute the scarce resource of the CPU to the different processes "fairly" (according to some definition of fairness) and in a way that optimizes some performance criteria.

In general, these assumptions are starting to break down. First of all, CPUs are not really that scarce - almost everybody has several, and pretty soon people will be able to afford lots. Second, many applications are starting to be structured as multiple cooperating processes. So, a view of the scheduler as mediating between competing entities may be partially obsolete.

- How do processes behave? First, CPU/IO burst cycle. A process will run for a while (the CPU burst), perform some IO (the IO burst), then run for a while more (the next CPU burst). How long between IO operations? Depends on the process.
  - IO Bound processes: processes that perform lots of IO operations. Each IO operation is followed by a short CPU burst to process the IO, then more IO happens.
  - CPU bound processes: processes that perform lots of computation and do little IO. Tend to have a few long CPU bursts.

One of the things a scheduler will typically do is switch the CPU to another process when one process does IO. Why?

The IO will take a long time, and don't want to leave the CPU idle while wait for the IO to finish.

- When look at CPU burst times across the whole system, have the exponential or hyperexponential distribution in Fig.
- What are possible process states?
  - Running - process is running on CPU.
  - Ready - ready to run, but not actually running on the CPU.
  - Waiting - waiting for some event like IO to happen.
- When do scheduling decisions take place? When does CPU choose which process to run? Are a variety of possibilities:
  - When process switches from running to waiting. Could be because of IO request, because wait for child to terminate, or wait for synchronization operation (like lock acquisition) to complete.
  - When process switches from running to ready - on completion of interrupt handler, for example. Common example of interrupt handler - timer interrupt in interactive systems. If scheduler switches processes in this case, it has preempted the running process. Another common case interrupt handler is the IO completion handler.
  - When process switches from waiting to ready state (on completion of IO or acquisition of a lock, for example).
  - When a process terminates.
- How to evaluate scheduling algorithm? There are many possible criteria:

  - CPU Utilization: Keep CPU utilization as high as possible. (What is utilization, by the way?).
  - Throughput: number of processes completed per unit time.
  - Turnaround Time: mean time from submission to completion of process.
  - Waiting Time: Amount of time spent ready to run but not running.
  - Response Time: Time between submission of requests and first response to the request.
  - Scheduler Efficiency: The scheduler doesn't perform any useful work, so any time it takes is pure overhead. So, need to make the scheduler very efficient.
- Big difference: Batch and Interactive systems. In batch systems, typically want good throughput or turnaround time. In interactive systems, both of these are still usually important (after all, want some computation to happen), but response time is usually a primary consideration. And, for some systems, throughput or turnaround time is not really relevant - some processes conceptually run forever.
- Difference between long and short term scheduling. Long term scheduler is given a set of processes and decides which ones should start to run. Once they start running, they may suspend because of IO or because of preemption. Short term scheduler decides which of the available jobs that long term scheduler has decided are runnable to actually run.

- Let's start looking at several vanilla scheduling algorithms.
- First-Come, First-Served. One ready queue, OS runs the process at head of queue, new processes come in at the end of the queue. A process does not give up CPU until it either terminates or performs IO.
- Consider performance of FCFS algorithm for three compute-bound processes. What if have 4 processes P1 (takes 24 seconds), P2 (takes 3 seconds) and P3 (takes 3 seconds). If arrive in order P1, P2, P3, what is
  - Waiting Time? (24 + 27)/3 = 17
  - Turnaround Time? (24 + 27 + 30) = 27.
  - Throughput? 30/3 = 10.

What about if processes come in order P2, P3, P1? What is
  - Waiting Time? (3 + 3)/2 = 6
  - Turnaround Time? (3 + 6 + 30) = 13.
  - Throughput? 30/3 = 10.
- Shortest-Job-First (SJF) can eliminate some of the variance in Waiting and Turnaround time. In fact, it is optimal with respect to average waiting time. Big problem: how does scheduler figure out how long will it take the process to run?
- For long term scheduler running on a batch system, user will give an estimate. Usually pretty good - if it is too short, system will cancel job before it finishes. If too long, system will hold off on running the process. So, users give pretty good estimates of overall running time.

- For short-term scheduler, must use the past to predict the future. Standard way: use a time-decayed exponentially weighted average of previous CPU bursts for each process. Let T_n be the measured burst time of the nth burst, s_n be the predicted size of next CPU burst. Then, choose a weighting factor w, where 0 <= w <= 1 and compute $s_{n+1} = w\ T_n + (1 - w)s_n$. s_0 is defined as some default constant or system average.
- w tells how to weight the past relative to future. If choose w =.5, last observation has as much weight as entire rest of the history. If choose w = 1, only last observation has any weight. Do a quick example.
- Preemptive vs. Non-preemptive SJF scheduler. Preemptive scheduler reruns scheduling decision when process becomes ready. If the new process has priority over running process, the CPU preempts the running process and executes the new process. Non-preemptive scheduler only does scheduling decision when running process voluntarily gives up CPU. In effect, it allows every running process to finish its CPU burst.
- Consider 4 processes P1 (burst time 8), P2 (burst time 4), P3 (burst time 9) P4 (burst time 5) that arrive one time unit apart in order P1, P2, P3, P4. Assume that after burst happens, process is not reenabled for a long time (at least 100, for example). What does a preemptive SJF scheduler do? What about a non-preemptive scheduler?
- Priority Scheduling. Each process is given a priority, then CPU executes process with highest priority. If

multiple processes with same priority are runnable, use some other criteria - typically FCFS. SJF is an example of a priority-based scheduling algorithm. With the exponential decay algorithm above, the priorities of a given process change over time.

- Assume we have 5 processes P1 (burst time 10, priority 3), P2 (burst time 1, priority 1), P3 (burst time 2, priority 3), P4 (burst time 1, priority 4), P5 (burst time 5, priority 2). Lower numbers represent higher priorities. What would a standard priority scheduler do?
- Big problem with priority scheduling algorithms: starvation or blocking of low-priority processes. Can use aging to prevent this - make the priority of a process go up the longer it stays runnable but isn't run.
- What about interactive systems? Cannot just let any process run on the CPU until it gives it up - must give response to users in a reasonable time. So, use an algorithm called round-robin scheduling. Similar to FCFS but with preemption. Have a time quantum or time slice. Let the first process in the queue run until it expires its quantum (*i.e.* runs for as long as the time quantum), then run the next process in the queue.
- Implementing round-robin requires timer interrupts. When schedule a process, set the timer to go off after the time quantum amount of time expires. If process does IO before timer goes off, no problem - just run next process. But if process expires its quantum, do

a context switch. Save the state of the running process and run the next process.

- How well does RR work? Well, it gives good response time, but can give bad waiting time. Consider the waiting times under round robin for 3 processes P1 (burst time 24), P2 (burst time 3), and P3 (burst time 4) with time quantum 4. What happens, and what is average waiting time? What gives best waiting time?
- What happens with really a really small quantum? It looks like you've got a CPU that is 1/n as powerful as the real CPU, where n is the number of processes. Problem with a small quantum - context switch overhead.
- What about having a really small quantum supported in hardware? Then, you have something called multithreading. Give the CPU a bunch of registers and heavily pipeline the execution. Feed the processes into the pipe one by one. Treat memory access like IO - suspend the thread until the data comes back from the memory. In the meantime, execute other threads. Use computation to hide the latency of accessing memory.
- What about a really big quantum? It turns into FCFS. Rule of thumb - want 80 percent of CPU bursts to be shorter than time quantum.
- Multilevel Queue Scheduling - like RR, except have multiple queues. Typically, classify processes into separate categories and give a queue to each category. So, might have system, interactive and batch processes, with the priorities in that order. Could also allocate a percentage of the CPU to each queue.

- Multilevel Feedback Queue Scheduling - Like multilevel scheduling, except processes can move between queues as their priority changes. Can be used to give IO bound and interactive processes CPU priority over CPU bound processes. Can also prevent starvation by increasing the priority of processes that have been idle for a long time.
- A simple example of a multilevel feedback queue scheduling algorithm. Have 3 queues, numbered 0, 1, 2 with corresponding priority. So, for example, execute a task in queue 2 only when queues 0 and 1 are empty.
- A process goes into queue 0 when it becomes ready. When run a process from queue 0, give it a quantum of 8 ms. If it expires its quantum, move to queue 1. When execute a process from queue 1, give it a quantum of 16. If it expires its quantum, move to queue 2. In queue 2, run a RR scheduler with a large quantum if in an interactive system or an FCFS scheduler if in a batch system. Of course, preempt queue 2 processes when a new process becomes ready.
- Another example of a multilevel feedback queue scheduling algorithm: the Unix scheduler. We will go over a simplified version that does not include kernel priorities. The point of the algorithm is to fairly allocate the CPU between processes, with processes that have not recently used a lot of CPU resources given priority over processes that have.
- Processes are given a base priority of 60, with lower numbers representing higher priorities. The system

clock generates an interrupt between 50 and 100 times a second, so we will assume a value of 60 clock interrupts per second. The clock interrupt handler increments a CPU usage field in the PCB of the interrupted process every time it runs.

- The system always runs the highest priority process. If there is a tie, it runs the process that has been ready longest. Every second, it recalculates the priority and CPU usage field for every process according to the following formulas.
  - CPU usage field = CPU usage field/2
  - Priority = CPU usage field/2 + base priority
- So, when a process does not use much CPU recently, its priority rises. The priorities of IO bound processes and interactive processes therefore tend to be high and the priorities of CPU bound processes tend to be low (which is what you want).
- Unix also allows users to provide a "nice" value for each process. Nice values modify the priority calculation as follows:
  - Priority = CPU usage field/2 + base priority + nice value

So, you can reduce the priority of your process to be "nice" to other processes (which may include your own).

- In general, multilevel feedback queue schedulers are complex pieces of software that must be tuned to meet requirements.
- Anomalies and system effects associated with schedulers.

- Priority interacts with synchronization to create a really nasty effect called priority inversion. A priority inversion happens when a low-priority thread acquires a lock, then a high-priority thread tries to acquire the lock and blocks. Any middle-priority threads will prevent the low-priority thread from running and unlocking the lock. In effect, the middle-priority threads block the high-priority thread.
- How to prevent priority inversions? Use priority inheritance. Any time a thread holds a lock that other threads are waiting on, give the thread the priority of the highest-priority thread waiting to get the lock. Problem is that priority inheritance makes the scheduling algorithm less efficient and increases the overhead.
- Preemption can interact with synchronization in a multiprocessor context to create another nasty effect - the convoy effect. One thread acquires the lock, then suspends. Other threads come along, and need to acquire the lock to perform their operations. Everybody suspends until the lock that has the thread wakes up. At this point the threads are synchronized, and will convoy their way through the lock, serializing the computation. So, drives down the processor utilization.
- If have non-blocking synchronization via operations like LL/SC, don't get convoy effects caused by suspending a thread competing for access to a resource. Why not? Because threads don't hold resources and prevent other threads from accessing them.

- Similar effect when scheduling CPU and IO bound processes. Consider a FCFS algorithm with several IO bound and one CPU bound process. All of the IO bound processes execute their bursts quickly and queue up for access to the IO device. The CPU bound process then executes for a long time. During this time all of the IO bound processes have their IO requests satisfied and move back into the run queue. But they don't run - the CPU bound process is running instead - so the IO device idles. Finally, the CPU bound process gets off the CPU, and all of the IO bound processes run for a short time then queue up again for the IO devices. Result is poor utilization of IO device - it is busy for a time while it processes the IO requests, then idle while the IO bound processes wait in the run queues for their short CPU bursts. In this case an easy solution is to give IO bound processes priority over CPU bound processes.
- In general, a convoy effect happens when a set of processes need to use a resource for a short time, and one process holds the resource for a long time, blocking all of the other processes. Causes poor utilization of the other resources in the system.

Scheduling Mechanisms

A multiprogramming operating system allows more than one process to be loaded into the executabel memory at a time and for the loaded process to share the CPU using time-multiplexing. Part of the reason for using multiprogramming is that the operating system itself is implemented as one or

more processes, so there must be a way for the operating system and application processes to share the CPU. Another main reason is the need for processes to perform I/O operations in the normal course of computation. Since I/O operations ordinarily require orders of magnitude more time to complete than do CPU instructions, multiprograming systems allocate the CPU to another process whenever a process invokes an I/O operation.

Goals for Scheduling

Make sure your scheduling strategy is good enough with the following criteria:

- Utilization/Efficiency: keep the CPU busy 100% of the time with useful work
- Throughput: maximize the number of jobs processed per hour.
- Turnaround time: from the time of submission to the time of completion, minimize the time batch users must wait for output
- Waiting time: Sum of times spent in ready queue - Minimize this
- Response Time: time from submission till the first response is produced, minimize response time for interactive users
- Fairness: make sure each process gets a fair share of the CPU.

Context Switching

Typically there are several tasks to perform in a computer system.

So if one task requires some I/O operation, you want to initiate the I/O operation and go on to the next task. You will come back to it later.

This act of switching from one process to another is called a "Context Switch"

When you return back to a process, you should resume where you left off. For all practical purposes, this process should never know there was a switch, and it should look like this was the only process in the system.

To implement this, on a context switch, you have to:

- Save the context of the current process
- Select the next process to run
- Restore the context of this new process.

Scheduling in Context of Process

- Programme Counter
- Stack Pointer
- Registers
- Code + Data + Stack (also called Address Space)
- Other state information maintained by the OS for the process (open files, scheduling info, I/O devices being used *etc.*)

All this information is usually stored in a structure called Process Control Block (PCB). All the above has to be saved and restored.

What Does a Context_Switch() Routine Look Like

```
context_switch()
{
Push registers onto stack
```

```
   Save ptrs to code and data.
   Save stack pointer
   Pick next process to execute
   Restore stack ptr of that process/* You have now switched the
stack */
   Restore ptrs to code and data.
   Pop registers

   Return
   }
```

Non-Preemptive Vs Preemptive Scheduling

- *Non-Preemptive*: Non-preemptive algorithms are designed so that once a process enters the running state(is allowed a process), it is not removed from the processor until it has completed its service time (or it explicitly yields the processor). context_switch() is called only when the process terminates or blocks.
- *Preemptive*: Preemptive algorithms are driven by the notion of prioritized computation. The process with the highest priority should always be the one currently using the processor. If a process is currently using the processor and a new process with a higher priority enters, the ready list, the process on the processor should be removed and returned to the ready list until it is once again the highest-priority process in the system. context_switch() is called even when the process is running usually done via a timer interrupt.

First In First Out (FIFO)

This is a *Non-Premptive* scheduling algorithm. FIFO strategy assigns priority to processes in the order in which they request the processor.

The process that requests the CPU first is allocated the CPU first.When a process comes in, add its PCB to the tail of ready queue. When running process terminates, dequeue the process (PCB) at head of ready queue and run it.

Consider the example with P1=24, P2=3, P3=3

Gantt Chart for FCFS: 0 - 24 P1, 25 - 27 P2, 28 - 30 P3

Turnaround time for P1 = 24

Turnaround time for P1 = 24 + 3

Turnaround time for P1 = 24 + 3 + 3

Average Turnaround time = (24*3 + 3*2 + 3*1)/3

In general we have (n*a + (n-1)*b +....)/n

If we want to minimize this, a should be the smallest, followed by b and

so on.

Comments: While the FIFO algorithm is easy to implement, it ignores the service time request and all other criteria that may influence the performance with respect to turnaround or waiting time.

Problem: One Process can monopolize CPU

Solution: Limit the amount of time a process can run without a context switch. This time is called a time slice.

Round Robin

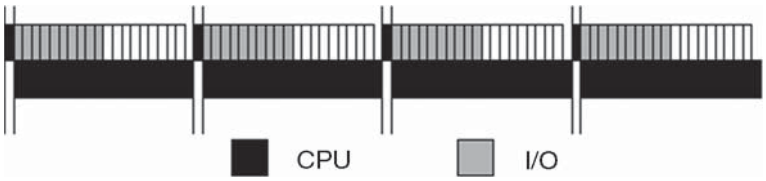
Round Robin calls for the distribution of the processing time equitably among all processes requesting the processor.

Run process for one time slice, then move to back of queue. Each process gets equal share of the CPU. Most systems use some variant of this.

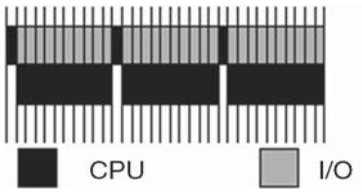
Choosing Time Slice

What happens if the time slice isnt chosen carefully?

- For example, consider two processes, one doing 1 ms computation followed by 10 ms I/O, the other doing all computation. Suppose we use 20 ms time slice and round-robin scheduling: I/O process runs at 11/21 speed, I/O devices are only utilized 10/21 of time.



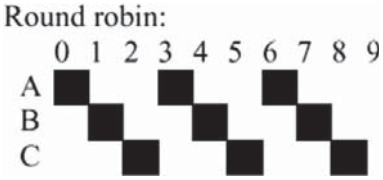
- Suppose we use 1 ms time slice: then compute-bound process gets interrupted 9 times unnecessarily before I/O-bound process is runnable



Problem: Round robin assumes that all processes are equally important; each receives an equal portion of the CPU. This sometimes produces bad results.

Consider three processes that start at the same time and each requires three time slices to finish. Using FIFO how long does it take the average job to complete (what is the average response time)? How about using round robin?

* Process A finishes after 3 slices, B 6, and C 9. The average is (3+6+9)/3 = 6 slices.



* Process A finishes after 7 slices, B 8, and C 9, so the average is (7+8+9)/3 = 8 slices.

Round Robin is fair, but uniformly enefficient.

Solution: Introduce priority based scheduling.

Priority Based Scheduling

Run highest-priority processes first, use round-robin among processes of equal priority. Re-insert process in run queue behind all processes of greater or equal priority.

- Allows CPU to be given preferentially to important processes.
- Scheduler adjusts dispatcher priorities to achieve the desired overall priorities for the processes, *e.g.* one process gets 90% of the CPU.

Comments: In priority scheduling, processes are allocated to the CPU on the basis of an externally assigned priority. The key to the performance of priority scheduling is in choosing priorities for the processes.

Problem: Priority scheduling may cause low-priority processes to starve

Solution: (AGING) This starvation can be compensated for if the priorities are internally computed. Suppose one parameter in the priority assignment function is the amount of time the process has been waiting. The longer a process waits, the higher its priority becomes. This strategy tends to eliminate the starvation problem.

Shortest Job First

Maintain the Ready queue in order of increasing job lengths. When a job comes in, insert it in the ready queue based on its length. When current process is done, pick the one at the head of the queue and run it.

This is provably the most optimal in terms of turnaround/ response time.

But, how do we find the length of a job?

Make an estimate based on the past behaviour.

Say the estimated time (burst) for a process is E0, suppose the actual

time is measured to be T0.

Update the estimate by taking a weighted sum of these two

ie. E1 = aT0 + (1-a)E0

in general, E(n+1) = aTn + (1-a)En (Exponential average)

if a=0, recent history no weightage

if a=1, past history no weightage.

typically a=1/2.

E(n+1) = aTn + (1-a)aTn-1 + (1-a)^jatn-j +...

Older information has less weightage

Comments: SJF is proven optimal only when all jobs are available simultaneously.

Problem: SJF minimizes the average wait time because it services small processes before it services large ones. While it minimizes average wait time, it may penalize processes with high service time requests. If the ready list is saturated, then processes with large service times tend to be left in the ready list while small processes receive service. In extreme case, where the system has little idle time, processes with large service times will never be served. This total starvation of large processes may be a serious liability of this algorithm.

Solution: Multi-Level Feedback Queques

Multi-Level Feedback Queue

Several queues arranged in some priority order.

Each queue could have a different scheduling discipline/ time quantum.

Lower quanta for higher priorities generally.

Defined by:

- # of queues
- Scheduling algo for each queue
- When to upgrade a priority
- When to demote

Attacks both efficiency and response time problems.

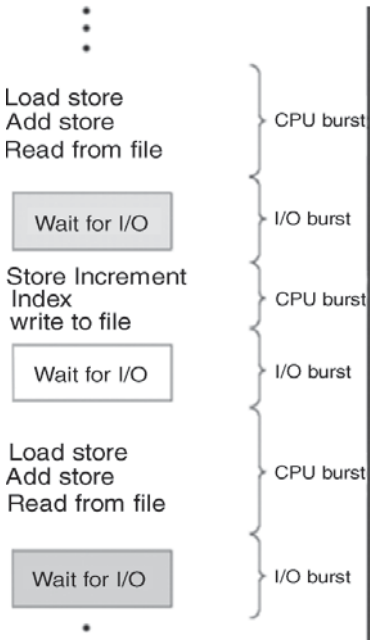
- Give newly runnable process a high priority and a very short time slice. If process uses up the time slice without blocking then decrease priority by 1 and double its next time slice.
- Often implemented by having a separate queue for each priority.
- How are priorities raised? By 1 if it doesn't use time slice? What happens to a process that does a lot of computation when it starts, then waits for user input?

Scheduling Concept

Within recent years the concept of the simultaneous multithreading (SMT) processor has been gaining in popularity. This hardware allows multiple processes to run on the processor at the same time providing more potential for instruction level parallelism. These new processors suggest that the rules an operating system (OS) scheduler follows need to be changed or at least modified. Our study shows the combination of jobs selected to run on these threads can significantly affect system performance. Our research shows that scheduling policies are greatly affected by the system workload and there most likely does not exist a single, best scheduling policy. However, it can be shown that a scheduler that tries to schedule processes doing a large number of loads and stores together with jobs doing few loads and stores consistently performs at levels close to or better than all other scheduling policies examined. It can also be seen that the more possibilities there are for scheduling, the more necessary it is to have an intelligent scheduler. In contrast, the few number of decisions to make (few threads and/or few processes) the less important the decision of a scheduler becomes.

Concepts

- Maximum CPU utilization obtained with multiprogramming - several processes are kept in memory, while one is waiting for I/O, the OS gives the CPU to another process
- OS does CPU scheduling
- CPU scheduling depends on the observation that processes cycle between CPU execution and I/O wait.



CPU Scheduler

Scheduling Decision

Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state (*e.g.* I/O request)
2. Switches from running to ready state (*e.g.* Interrupt)
3. Switches from waiting to ready (*e.g.* I/O completion)
4. Terminates
   - Scheduling under 1 and 4 is *non-preemptive (cooperative)*
   - All other scheduling is *preemptive* - have to deal with possibility that operations (system calls) may be incomplete

Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user programme to restart that programme
- Dispatch latency – time it takes for the dispatcher to stop one process and start another running
- Should be as low as possible.

Scheduling Criteria

- CPU utilization (max) – keep the CPU as busy as possible
- Throughput (max) – # of processes that complete their execution per time unit
- Turnaround time (min) – amount of time to execute a particular process
- Waiting time (min) – amount of time a process has been waiting in the ready queue
- Response time (min) – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- In typical OS, we optimize each to various degrees depending on what we are optimizing the OS.

Criteria for scheduling algorithms (performance)

- CPU utilization
- Throughput (*e.g.*, no. of processes completed/unit time)

- Waiting time (the amount of time a job waits in the queue)
- Turnaround time (the total time, including waiting time, to complete a job)
- Response time.

In general, it is not possible to optimize all these criteria for process scheduling using any algorithm (*i.e.,* some of the criteria may conflict, in some circumstances). Typically, the criteria are prioritized, with most attention paid to the most important criterion. *e.g.,* in an interactive system, response time may well be considered more important then CPU utilization. In the operating systems course, you will study some of the algorithms which are used for process scheduling.

CPU Scheduling is used to allocate the CPU to one of the processes in memory that are ready to execute. On of the main goals of scheduling is to maximize CPU utilization.

Typically, each process (programme) consists of CPU burst cycles (the process needs to use the CPU for some calculation or operation) and I/O cycles. When there are several concurrently running processes, instead of waiting the currently running process to perform its I/O operation, the CPU can be assigned to another process and hence achieve efficient use of system resources.

Performance Criteria

To compare the performance of different types of scheduling algorithms, the following criteria are generally used:

Scheduling Algorithms:

First Come First Served: the process that requests the CPU is allocated first.

Shortest Job First: the process with smallest CPU burst gets the CPU.

Shortest Remaining Time First: pre-emptive version of Shortest Job First

Round Robin: CPU is shared for intervals called *time quantum*. Each process takes turns and uses the CPU for that duration. algorithm and compute the values of different performance criteria given above for each algorithm. Repeat step

1. For prob2.cfg and prob3.cfg.
2. State which algorithm behaves better for which performance criteria and the reasons.

Lab Work

In this experiment, your will be assigned a scheduling problem. You are expected to encode the given problem for the simulator, draw the resultant Gantt Chart and compute the values of some performance criteria. You are also expected to comment on the relative changes on the performance criteria in the cases of minor modifications on the given problem.

Central Processing Unit and Signed Numbers

Addition

Adding Unsigned Numbers

Adding unsigned numbers in binary is quite easy. Recall that with 4 bit numbers we can represent numbers from 0 to 15. Addition is done exactly like adding decimal numbers, except that you have only two digits (0 and 1).

The only number facts to remember are that,

0+0 = 0, with no carry,

1+0 = 1, with no carry,

0+1 = 1, with no carry,

1+1 = 0, and you carry a 1.

so to add the numbers $06_{10}=0110_2$ and $07_{10}=0111_2$ (answer = $13_{10} = 1101_2$) we can write out the calculation (the results of any carry is shown along the top row, in italics).

Decimal	Unsigned binary
1 (carry)	110 (carry)
06	0110
+07	+0111
13	1101

The only difficulty adding unsigned numbers occurs when you add numbers that are too large. Consider 13+5.

Decimal	Unsigned binary
0 (carry)	1101 (carry)
13	1101
+05	+0101
18	10010

The result is a 5 bit number. So the carry bit from adding the two most significant bits represents a results that *overflows* (because the sum is too big to be represented with the same number of bits as the two addends).

Adding Signed Numbers

Adding signed numbers is not significantly different from adding unsigned numbers. Recall that signed 4 bit numbers (2's complement) can represent numbers between-8 and 7. To see how this addition works, consider three examples.

In this case the extra carry from the most significant bit has no meaning. With signed numbers there are two ways to get an overflow--if the result is greater than 7, or less than-8. Let's consider these occurrences now.

Decimal	Signed binary
	1110 (carry)
-2	1110
+3	+0011
1	0001

Decimal	Signed binary
	011 (carry)
-5	1011
+3	+0011
-2	1110

Decimal	Signed binary
	1100 (carry)
-4	1100
-3	+1101
-7	1001

In this case the extra carry from the most significant bit has no meaning. With signed numbers there are two ways to get an overflow—if the result is greater than 7, or less than-8. Let's consider these occurrences now.

Decimal	Signed binary
	110 (carry)
6	0110
+3	+0011
9	1001

Decimal	Signed binary
	1001 (carry)
-7	1001
-3	+1101
-10	0110

Obviously both of these results are incorrect, but in this case overflow is harder to detect. But you can see that if two numbers with the same sign (either positive or negative) are added and the result has the opposite sign, an overflow has occurred. Typically DSP's, including the 320C5x, can deal somewhat with this problem by using something called saturation arithmetic, in which results that result in overflow are replaced by either the most positive number (in this case

7) if the overflow is in the positive direction, or by the most negative number (-8) for overflows in the negative direction.

Adding Fractions

There is no further difficult in adding two signed fractions, only the interpretation of the results differs. For instance consider addition of two Q3 numbers shown (compare to the example with two 4 bit signed numbers, above).

Decimal	Fractional binary
	1110 (carry)
-0.25	1110
+0.375	+0011
0.125	0001

Decimal	Fractional binary
	011 (carry)
-0.625	1011
+0.375	+0011
-0.25	1110

Decimal	Fractional binary
	1100 (carry)
-0.5	1100
-0.375	+1101
-0.875	1001

If you look carefully at these examples, you'll see that the binary representation and calculations are the same as before, only the decimal representation has changed.

This is very useful because it means we can use the same circuitry for addition, regardless of the interpretation of the results.

Even the generation of overflows resulting in error conditions remains unchanged (again compare with above)

Decimal	Fractional binary
	110 (carry)
0.75	0110
+0.375	+0011
1.125	1001

Decimal	Fractional binary
	1001 (carry)
-0.875	1001
-0.375	+1101
-1.25	0110

2's Complement

We do not just place a 1 in the MSB of a binary number to make it negative. We must take the 2's Complement of the number. Taking the 2's Complement of the number will cause the MSB to become 1.

To obtain the 2's complement of a number is a two step process.

- Take the 1's complement of the number by changing every logic 1 bit in the number to a 0, and change every logic 0 bit to a 1.
- Add 1 to the 1's Complement of the binary number. You now have the 2's Complement of the original number. You will notice that the MSB has become a 1.

The complement of a binary number, also known as the 1's complement, requires us to change every logic 1 bit in a number to a logic 0, and every logic 0 bit to a logic 1. Let's find the 1's complement of 36H or 0011 0110 in binary. In the following table, the 1's Complement is shown in red.

Number format	D7	D6	D5	D4	D3	D2	D1	D0
Unsigned number	0	0	1	1	0	1	1	0
1's Complement	1	1	0	0	1	0	0	1

To obtain the 2's complement of a binary number, we must first obtain the 1's complement of the number and then add 1. In the following table, the 1's complement is RED and the 2's Complement is in BLUE. The 2's complement of 36H or 0011 0110 in binary is:

Number format	D7	D6	D5	D4	D3	D2	D1	D0
Unsigned number	0	0	1	1	0	1	1	0
1's Complement	1	1	0	0	1	0	0	1
2's Complement	1	1	0	0	1	0	1	0

If we are using signed binary numbers and the MSB is already logic 1, it means the value is the 2's complement of the number. The actual numeric value can be determined by taking the 2's complement of the number and keeping track of the minus sign mentally.

Find the decimal value of the signed binary number 0FFH.

Number format	D7	D6	D5	D4	D3	D2	D1	D0
Signed number	1	1	1	1	1	1	1	1
1's Complement	0	0	0	0	0	0	0	0
2's Complement	0	0	0	0	0	0	0	1

Therefore, 0FF is a signed number that has the value in decimal of-1.

Using 2's Complement Addition

If we add the 2's Complement of a signed number to another signed number, we are performing the mathematical operation of subtraction. This is, in fact. how many 8-bit microprocessors actually perform subtraction, they perform 2's complement addition.

Let us see if 0FFH is really-1. If we add it to 01H, we expect the result to be 0. One special condition about twos complement addition, overflows (a CARRY OUT which SETS the CARRY FLAG) from the register are ignored. So let's see what happens.

In Decimal	CO	D7	D6	D5	D4	D3	D2	D1	D0
1	X	0	0	0	0	0	0	0	1
–1	X	1	1	1	1	1	1	1	1
0	1	0	0	0	0	0	0	0	0

The last row of the table shows that the result is 0. The 1 in the CO (CARRY OUT) is ignored. If we ignore the CO, the result is 0000 0000. Okay, so it works for a byte. How about a Word? It is the same procedure. Only the MSB is used as the sign bit, so the MSB of the low order byte is a weighted position bit.

For 16-bit numbers:

- 8000H is negative because the high order bit is one.
- 0100H is positive because the high order bit is zero.
- 7FFFH is positive.
- 0FFFFH is negative.
- 0FFFH is positive.

Look Ahead Carry Adders

Purpose

- To Understand delays in digital circuits
- To Learn to use the Timing Simulator
- To Design a 16-bit carry-look ahead adder and measure its delay.

Background: Circuit Delays

Up to now we have been interested in the functional operation of logic circuits. In the previous labs you used the logic simulator to verify that the circuits function properly. However, there is one other important aspect to circuit design: the speed at which it operates. The speed of a digital circuit is very important, as it will determine the maximum frequency at which it can work. Let us consider a PC that has a clock frequency of 800 MHz. That means that each 1.25 ns (*i.e.* period T=1/frequency) the PC will perform a computation! As we will see, this will require clever circuit design.

There are several factors that contribute to the delay. One is the propagation delay due to the internal structure of the gates, another factor is the loading of the output buffers (due to fan out and net delays), and a third factor is the logic circuit itself.

Propagation Delay

When the input signal of a gate changes, the output signal will not change instantaneously as is shown in Figure below.

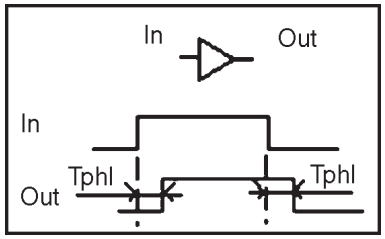


Fig. Propagation Delay of Gates.

The propagation delay (or gate delay) of a gate is the time difference between the change of the input and output signals. There are two types of gate delays, T_{PHL} and T_{PLH}, as indicated in Figure. The value of the propagation delay varies from gate to gate and from logic family to family. In general the more you are willing to pay for a device (or chip), the faster it will be. The FPGAs we are using in the lab have gate delays which vary between 1.5 and 4.5ns. The actual delay depends on the way the logic gates have been mapped into the LUTs (Look up table) of a CLB (Configurable Logic Block). The I/O buffers have delays in the range of 2-4ns.

Fanout and Net Delays

The propagation delay described above is caused by parasitic capacitors inside the gates and the physical limitations of the devices used to build these gates. Another

cause of delay is the capacitor associated with the loads seen by a gate. The more capacitors that need to be charged or discharged the longer it will take for the output to change. Also, the longer the interconnection, the more resistance the nets will have. The easiest way to visualize this is to use a hydraulic equivalent of a capacitor and a resistor: a bucket filled with water and a narrow pipe, respectively, as shown in Figure. The more buckets connected to the drain (*i.e.* the input inverter), the longer it will take to empty them. This delay is the result of the fan out of the inverter.

Delay as a Result of Circuit Topography

Circuits that perform the same function can vary significantly in their speeds. A good example is an adder circuit. The one you designed in the previous lab is called a *ripple-adder* and is considerably slower that a *carry-look-ahead adder* or *CLA [1,3,4].*

Measuring Circuit Delays

The overall speed of a digital system can be measured on an oscilloscope by comparing the input to the output signals. However, during the design phase, the circuit has not yet been fabricated and therefore, cannot be measured. In that case it is possible to determine the delay of circuits by doing a *Timing Simulation*. The advantage of a simulation is that one can also determine the delay of internal nodes of a circuit. This can be very helpful to understand which nodes or paths are the slowest and thus limit the overall speed of the circuits. These paths are called “Critical path”. It is important to understand which paths are critical in a circuit so that one can reduce their delay.

Ripple-carry vs. Carry-look-ahead Adders

One type of circuit where the effect of gate delays is particularly clear, is an ADDER. In this lab you will be measuring the delay of different types of adder circuits. The 4-bit adder you designed and implemented in the previous lab is called a ripple-carry adder because the result of an addition of two bits depends on the carry generated by the addition of the previous two bits. Thus, the Sum of the most significant bit is only available after the carry signal has rippled through the adder from the least significant stage to the most significant stage. This can be easily understood if one considers the addition of the two 4-bit words: $1\ 1\ 1\ 1_2$ + $0\ 0\ 0\ 1_2$, as shown in Figure.

$$
\begin{array}{l}
1\ \ 1\ \ 1\ \ 1 \longrightarrow \text{Carry bits} \\
\ \ \ 1\ \ 1\ \ 1\ \ 1 \\
+\ \ 0\ \ 0\ \ 0\ \ 1 \\
\hline
1\ \ 0\ \ 0\ \ 0\ \ 0 \\
C_0\ S_3\ S_2\ S_1\ S_0 \rightarrow \text{Sum bits} \\
/ \\
\text{Carry.out}
\end{array}
$$

Fig. Addition of two 4-bit Numbers Illustrating the Generation of the Carry-out Bit.

In this case, the addition of ($1+1 = 10_2$) in the least significant stage causes a carry bit to be generated. This carry bit will consequently generate another carry bit in the next stage, and so on, until the final carry-out bit appears at the output. As a result, the final Sum and Carry bits will be valid after a considerable delay. The carry-out bit of the first stage will be valid after 4 gate delays (2 associated with the XOR gate and 1 each associated with the AND and OR gates).

From the schematic of Figure, one finds that the next carry-out (C2) will be valid after an additional 2 gate delays (associated with the AND and OR gates) for a total of 6 gate delays. In general the carry-out of a N-bit adder will be valid after 2N+2 gate delays. The Sum bit will be valid an additional 2 gate delays after the carry-in signal. Thus the sum of the most significant bit S_{N-1} will be valid after 2(N-1) + 2 +2 = 2N +2 gate delays.

This delay may be in addition to any delays associated with interconnections. It should be mentioned that in case one implements the circuit in a FPGA, the delays may be different from the above expression depending on how the logic has been placed in the look up tables and how it has been divided among different CLBs.

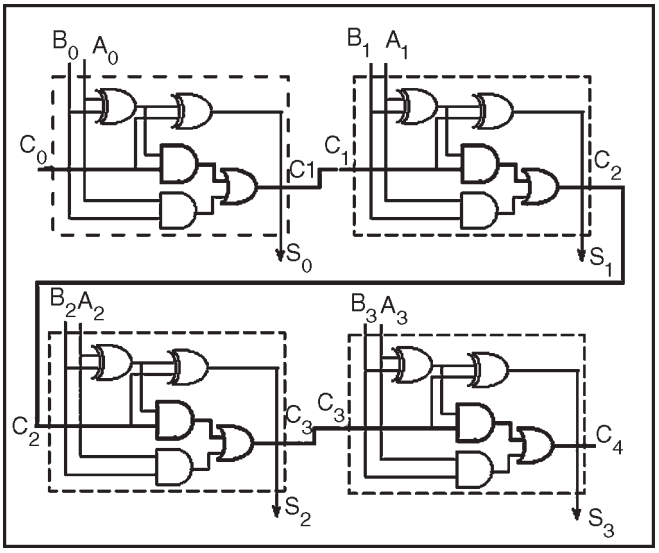


Fig. Ripple-carry Adder, Illustrating the Delay of the Carry Bit.

The disadvantage of the ripple-carry adder is that it can get very slow when one needs to add many bits. For instance, for a 32-bit adder, the delay would be about 66 ns if one assumes a gate delay of 1 ns. That would imply that the

maximum frequency one can operate this adder would be only 15 MHz! For fast applications, a better design is required.

The carry-look-ahead adder solves this problem by calculating the carry signals in advance, based on the input signals. It is based on the fact that a carry signal will be generated in two cases: (1) when both bits A_i and B_i are 1, or (2) when one of the two bits is 1 and the carry-in (carry of the previous stage) is 1.

Thus, one can write,

- $C_{OUT} = C_{i+1} = A_i.B_i + (A_i$ Å $B_i).C_i$. The "⊕" stands for exclusive OR or XOR. One can write this expression also, as
- $C_{i+1} = G_i + P_i.C_i$ *in which*:
- $G_i = A_i.B_i$
- $Pi = (A_i$ Å $B_i)$

  are called the Generate and Propagate term, respectively. Lets assume that the delay through an AND gate is one gate delay and through an XOR gate is two gate delays. Notice that the Propagate and Generate terms only depend on the input bits and thus will be valid after two and one gate delay, respectively. If one uses the above expression to calculate the carry signals, one does not need to wait for the carry to ripple through all the previous stages to find its proper value. Let's apply this to a 4-bit adder to make it clear.
- $C_1 = G_0 + P_0.C_0$
- $C_2 = G_1 + P_1.C_1 = G_1 + P1.G_0 + P_1.P_0.C_0$
- $C_3 = G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.C_0$
- $C_4 = G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3P_2.P_1.G_0 + P_3P_2.P_1.P_0.C_0$

Notice that the carry-out bit, C_{i+1}, of the last stage will be available after four delays (two gate delays to calculate the Propagate signal and two delays as a result of the AND and OR gate). The Sum signal can be calculated as follows,

- $S_i = A_i \oplus B_i \oplus C_i = P_i \oplus C_i$. The Sum bit will thus be available after two additional gate delays (due to the XOR gate) or a total of six gate delays after the input signals A_i and B_i have been applied. The advantage is that these delays will be the same independent of the number of bits one needs to add, in contrast to the ripple counter.

  The carry-look ahead adder can be broken up in two modules:

  – The Partial Full Adder, PFA, which generates Si, Pi and Gi as defined by equations 3, 4 and 9 above; and
  – The Carry Look-ahead Logic, which generates the carry-out bits according to equations 5 to 8.

  The 4-bit adder can then be built by using 4 PFAs and the Carry Look-ahead logic block as shown in Figure below. The disadvantage of the carry-lookahead adder is that the carry logic is getting quite complicated for more than 4 bits. For that reason, carry-look-ahead adders are usually implemented as 4-bit modules and are used in a hierarchical structure to realise adders that have multiples of 4 bits. Figure below shows the block diagram for a 16-bit CLA adder. The circuit makes use of the same CLA Logic block as the one used in the 4-bit adder.

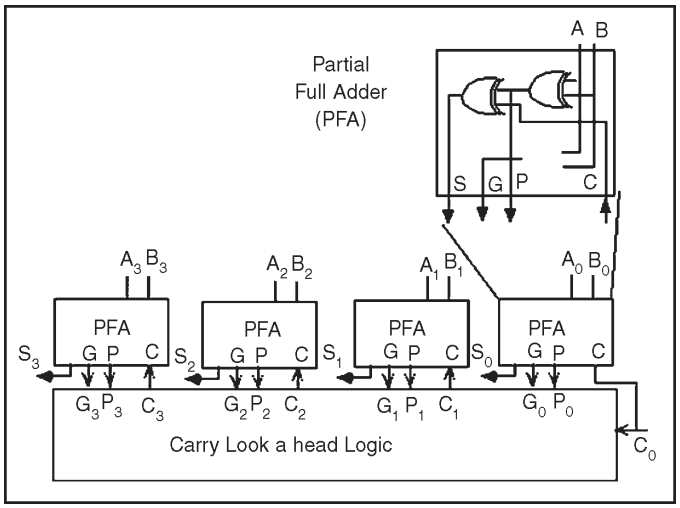


Fig. Block Diagram of a 4-bit CLA.

Notice that each 4-bit adder provides a group Propagate and Generate Signal, which is used by the CLA Logic block. The group Propagate P_G of a 4-bit adder will have the following expressions,

- $P_G = P_3.P_2.P_1.P_0$;
- $G_G = G_3 + P_3G_2 + P_3.P_2.G_1. + P_3.P_2.P_1.G_0$

The group Propagate P_G and Generate G_G will be available after 3 and 4 gate delays, respectively (one or two additional delays than the P_i and G_i signals, respectively).

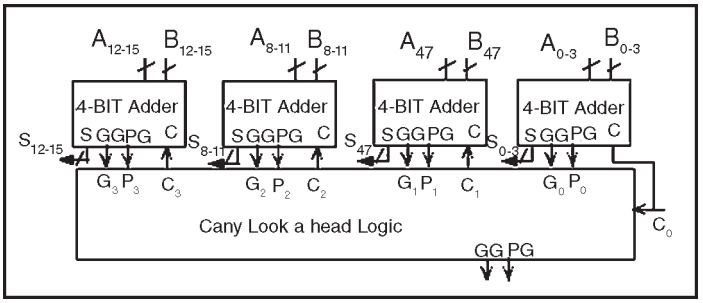


Fig. Block Diagram of a 16-bit CLA Adder.

Generate and Propagate

Why generate and propagate. If you look at the Boolean expressions for p_i and g_i, you will see that they both use only x_i and y_i. Neither depend on the carry. Sincex_i and y_i are available immediately, this gives us hope that we can avoid waiting for carries.

This is how we do it. First, let's write the c_1 which is the carry out for the adding bit 0 of x and y.

$$c_1 = g_0 + p_0c_0$$

Now, we write it for c_2.

$$c_2 = g_1 + p_1c_1$$

At this point, we've used c_1, which we'd rather avoid, because it means waiting for that carry to be computed. However, we *just* defined c_1 as $g_0 + c_0p_0$.

Let's plug that in:

$$\begin{aligned} c_2 &= g_1 + p_1(g_0 + p_0c_0) \\ &= g_1 + p_1g_0 + p_1p_0c_0 \end{aligned}$$

Notice that we no longer have c_1. That means we no longer have to wait for the carry!

Let's go one more step further:

$$c_3 = g_2 + p_2c_2$$

Again, we would prefer to avoid using c_2 since this requires us to wait for the result to propagate across two adders. However, we *already* have a Boolean expression for c_2 (we just computed it a moment ago) that doesn't use any carries except c_0 which we have right away.

$$\begin{aligned} c_3 &= g_2 + p_2c_2 \\ &= g_2 + p_2(g_1 + p_1g_0 + p_1p_0c_0) \\ &= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 \end{aligned}$$

Already, you should be able to detect a pattern. By following the same pattern, you'd expect:

$$c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$$

Multiplication

Multiplying unsigned numbers

Multiplying unsigned numbers in binary is quite easy. Recall that with 4 bit numbers we can represent numbers from 0 to 15.

Multiplication can be performed done exactly as with decimal numbers, except that you have only two digits (0 and 1). The only number facts to remember are that 0*1=0, and 1*1=1 (this is the same as a logical "and").

Multiplication is different than addition in that multiplication of an n bit number by an m bit number results in an n+m bit number. Let's take a look at an example where n=m=4 and the result is 8 bits

Decimal	Binary
10	1010
×6	×0110
60	0000
	1010
	1010
	+0000
	0111100

In this case the result was 7 bit, which can be extended to 8 bits by adding a 0 at the left. When multiplying larger numbers, the result will be 8 bits, with the leftmost set to 1, as shown.

Decimal	Binary
13	1101
×14	×1110
182	0000
	1101
	1101
	+1101
	10110110

As long as there are n+m bits for the result, there is no chance of overflow. For 2 four bit multiplicands, the largest possible product is 15*15=225, which can be represented in 8 bits.

Multiplying signed numbers

There are many methods to multiply 2's complement numbers. The easiest is to simply find the magnitude of the two multiplicands, multiply these together, and then use the original sign bits to determine the sign of the result. If the multiplicands had the same sign, the result must be positive, if the they had different signs, the result is negative. Multiplication by zero is a special case (the result is always zero, with no sign bit).

Multiplying fractions

As you might expect, the multiplication of fractions can be done in the same way as the multiplication of signed numbers. The magnitudes of the two multiplicands are multiplied, and the sign of the result is determined by the signs of the two multiplicands.

There are a couple of complications involved in using fractions. Although it is almost impossible to get an overflow (since the multiplicands and results usually have magnitude less than one), it is possible to get an overflow by multiplying-1x-1 since the result of this is +1, which cannot be represented by fixed point numbers.

The other difficulty is that multiplying two Q3 numbers, obviously results in a Q6 number, but we have 8 bits in our result (since we are multiplying two 4 bit numbers). This means that we end up with two bits to the left of the decimal

point. These are sign extended, so that for positive numbers they are both zero, and for negative numbers they are both one. Consider the case of multiplying-1/2 by-1/2:

Decimal	Fractional binary
–0.5	1100
×0.5	×0100
–0.25	0000
	0000
	+111100
	11110000

This obviously presents a difficulty if we wanted to store the number in a Q3 result, because if we took just the 4 leftmost bits, we would end up with two sign bits. So what we'd like to do is shift the number to the left by one and then take the 4 leftmost bit. This leaves us with 1110 which is equal to-1/4, as expected.

On a 16 bit DSP two Q15 numbers are multiplied to get a Q30 number with two sign bits. On the 320C50 there are two ways to accomplish this. The first is to use the p-scaler immediately after the multiplier, or the postscaler after the accumulator. to shift the result to the left by one

Example for Multiplication

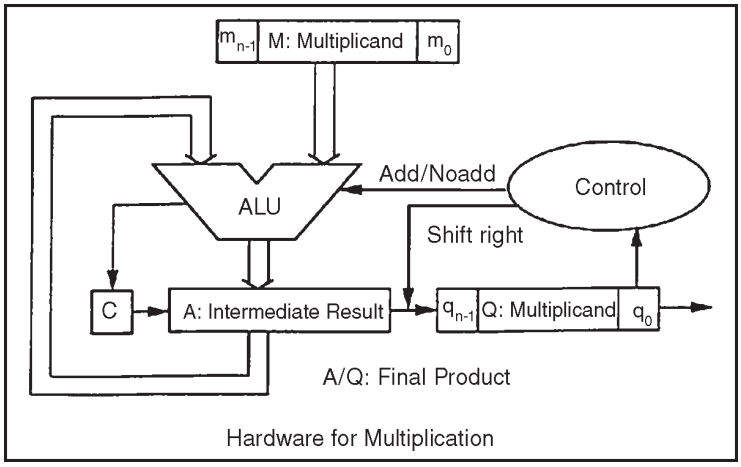
$$13 \times 11 = 143$$

The paper-and-pencel method:

$$\begin{array}{cccccccc} & & & & 1 & 1 & 0 & 1 \\ & & & \times & 1 & 0 & 1 & 1 \\ \hline & & & & 1 & 1 & 0 & 1 \\ & & & 1 & 1 & 0 & 1 & \\ & & 0 & 0 & 0 & 0 & & \\ + & 1 & 1 & 0 & 1 & & & \\ \hline 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}$$

Improve the method so that only 2 numbers are added each time:

```
                1 1 0 1
              × 1 0 1 1
  ---------------------
                1 1 0 1
            + 1 1 0 1
  ---------------------
          1 0 0 1 1 1 1
          0 0 0 0
  ---------------------
          1 0 0 1 1 1 1
      + 1 1 0 1
  ---------------------
  1 0 0 0 1 1 1 1
```



Hardware for Multiplication

Algorithm for Hardware Multiplication

```
Do n times:
{
  if (q0 = 1) then A ← A + M;
  right shift A and Q by 1 bit
}
```

Note: When A and Q are right shifted, the MSB of A is filled with 0 and the LSB of A becomes the MSB of Q, and the LSB of Q is lost.

Example

$$13 \times 11 = 143$$

Always use three registers M, A, and Q. Initially, the multiplicand $13 = (1101)_2$ is in M, the multiplier $11 = (10111)_2$ is in Q, and A is zero.

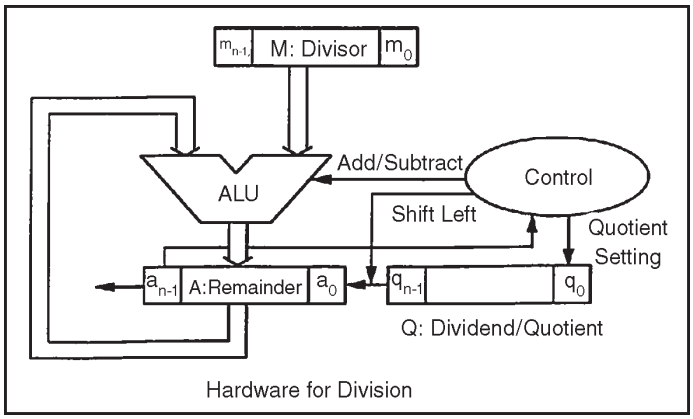
	[M]	1101		
	[A]	0000	[Q]	1011
q0 = 1, add A = [A] + [M]	+	1101		
	0	1101		1011
right shift A/Q	0	0110		1101
q0 = 1, add A = [A] + [M]	+	1101		
	1	0011		1101
right shift A/Q	0	1001		1110
q0 = 0,right shift, A/Q	0	0100		
q0 = 1, add A = [A] + [M]	+	1101		
	1	0001		1111
right shift A/Q	0	1000		1111

The upper half of the product $(1010001111)_2 = 143$ is in register A, while the lower half is in register Q.

Example for Division

$$274 \div 13 = 21\ \frac{1}{13}$$

```
                000010101          Quotient
Divisor  1101 ) 100010010          Dividend
                -1101
                 10000
                -1101
                  1110
                 -1101
                     1             Remainder
```



Hardware for Division

Algorithm for Hardware Division (Restoring)

Do n times:

```
  {
    left shift A and Q by 1 bit
    A ← A–M;
    if A < 0 (a_{n-1} = 1), then q0 ← 0,A ← A +
M  (restore)
    else q0 ← 1
  }
```

Note: When A and Q are left shifted, the MSB of Q becomes the LSB of A, and the MSB of A is lost. The LSB of Q is made available for the next quotient bit.

Example

$$8 \div 3 = 2\frac{2}{3}$$

Initially, the divisor $3 = (0011)_2$ is in register M, the dividend $8 = (1000)_2$ is in register Q, and register A is zero. Note that subtraction by $3 = (0011)_2$ is implemented by adding its 2's complement 1101.

	[M]	**1101**			
	[A]	**0000**	**[Q]**	**1011**	
left shift A/Q		0001		000.	
A = [A]–[M]	+	1101			
A < 0		1101		0000	
A = [A] + [M]	+	0011			
		0001		0000	
left shift A/Q		0010		000.	
A = [A]–[M]	+	1101			
A < 0		0011		0000	
A = [A] + [M]	+	0011			
		0010		0000	
left shift A/Q		1100		000.	
A = [A]–[M]	+	1101			
A > 0		0001		0001	left shift A/Q
0010		001.			
A = [A]–[M]	+	1101			

A < 0	1111	0010	A = [A] + [M]+ 0011
	0010	1111	

The quotient $(0010)_2 = 2$ is in register Q, and the reminder $(0010)_2 = 2$ is in register A.

Algorithm for Hardware Division (Non-restoring)

In the algorithm above, if the subtraction produces a non-positive result (A > 0), registers A and Q are left shifted and the next subtraction is carried out. But if the subtraction produces a negative result (A < 0), the dividend need be first restored by adding the divisor back before left shift A and Q and the next subtraction:

- If A ≤ 0, then 2A–M (left shift and subtract);
- If A ≥ 0, then 2(A + M)–M = 2A + M (restore, left shift and subtract).

Note that when A < 0, the restoration is avoided by combining the two steps. This leads to a faster non-restoring division algorithm:

Algorithm for Hardware Division (Non-restoring)

Do n times:

```
{ left shift A and Q by 1 bit
  if (previous A ≥ 0) then A → A–M
  else A → A + M;
  if (current A ≥ 0) then q0 → 1
  else q0 → 1
}
  if (A < 0) then  (remainder must be
positive)
```

	[M]	0011		
	[A]	0000	[Q]	1000
left shift A/Q		0001		000.
A = [A]–[M]	+	1101		
A < 0		1110		0000

left shift A/Q		1100	00<u>0.</u>	
A = [A] + [M]	+	0011		
A < 0		1111	0<u>000</u>	
left shift A/Q		1110	0<u>00.</u>	
A = [A] + [M]	+	0011		
A > 0		0001	0<u>001</u>	left shift A/Q
0010		<u>001.</u>		
A = [A]–[M]	+	1101		
A < 0		1111	<u>0010</u>	A = [A] + [M]+
0011				
		0010	1111	

The quotient $(0010)_2 = 2$ is in register Q, and the reminder $(0010)_2 = 2$ is in register A. The restoring division requires two operations (subtraction followed by an addition to restore) for each zero in the quotient. But non-restoring division only requires one operation (either addition or subtraction) for each bit in quotient.

Booths algorithm and Array Multiplier

Definition of an Algorithm

In the introduction, we gave an informal definition of an algorithm as "a set of instructions for solving a problem" and we illustrated this definition with a recipe, directions to a friend's house, and instructions for changing the oil in a car engine. You also created your own algorithm for putting letters and numbers in order. While these simple algorithms are fine for us, they are much too ambiguous for a computer. In order for an algorithm to be applicable to a computer, it must have certain characteristics. We will specify these characteristics in our formal definition of an algorithm.

An algorithm is a well-ordered collection of unambiguous and effectively computable operations that when

executed produces a result and halts in a finite amount of time. With this definition, we can identify five important characteristics of algorithms.

- Algorithms are well-ordered.
- Algorithms have unambiguous operations.
- Algorithms have effectively computable operations.
- Algorithms produce a result.
- Algorithms halt in a finite amount of time.

These characteristics need a little more explanation, so we will look at each one in detail.

Algorithms are Well-ordered

Since an algorithm is a collection of operations or instructions, we must know the correct order in which to execute the instructions. If the order is unclear, we may perform the wrong instruction or we may be uncertain which instruction should be performed next. This characteristic is especially important for computers. A computer can only execute an algorithm if it knows the exact order of steps to perform.

Algorithms have Unambiguous Operations

Each operation in an algorithm must be sufficiently clear so that it does not need to be simplified. Given a list of numbers, you can easily order them from largest to smallest with the simple instruction "Sort these numbers." A computer, however, needs more detail to sort numbers. It must be told to search for the smallest number, how to find the smallest number, how to compare numbers together, etc.

The operation "Sort these numbers" is ambiguous to a computer because the computer has no basic operations for

sorting. Basic operations used for writing algorithms are known as primitive operations or primitives. When an algorithm is written in computer primitives, then the algorithm is unambiguous and the computer can execute it.

Algorithms Produce a Result

In our simple definition of an algorithm, we stated that an algorithm is a set of instructions for solving a problem. Unless an algorithm produces some result, we can never be certain whether our solution is correct. Have you ever given a command to a computer and discovered that nothing changed? What was your response? You probably thought that the computer was malfunctioning because your command did not produce any type of result.

Without some visible change, you have no way of determining the effect of your command. The same is true with algorithms. Only algorithms which produce results can be verified as either right or wrong.

2

Scheduling Algorithms

First-Come-First-Served (FCFS) Scheduling

Other names of this algorithm are:

- First-In-First-Out (FIFO)
- Run-to-Completion
- Run-Until-Done

Perhaps, First-Come-First-Served algorithm is the simplest scheduling algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a nonpreemptive discipline, once a process has a CPU, it runs to completion. The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait.

FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling

interactive users because it cannot guarantee good response time. The code for FCFS scheduling is simple to write and understand. One of the major drawback of this scheme is that the average time is often quite long. The First-Come-First-Served algorithm is rarely used as a master scheme in modern operating systems but it is often embedded within other schemes.

Round Robin Scheduling

One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR). In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time called a time-slice or a quantum. If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list.

Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users. The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and appoximates FCFS. In any event, the average waiting time under round robin scheduling is often quite long.

Shortest-Job-First (SJF) Scheduling

Other name of this algorithm is Shortest-Process-Next (SPN). Shortest-Job-First (SJF) is a non-preemptive discipline

in which waiting job (or process) with the smallest estimated run-time-to-completion is run next. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst.

The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal. The SJF algorithm favors short jobs (or processors) at the expense of longer ones. The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available. The best SJF algorithm can do is to rely on user estimates of run times. In the production environment where the same jobs run regularly, it may be possible to provide reasonable estimate of run time, based on the past performance of the process. But in the development environment users rarely know how their programme will execute. Like FCFS, SJF is non preemptive therefore, it is not useful in timesharing environment in which reasonable response time must be guaranteed.

Shortest-Remaining-Time (SRT) Scheduling

- The SRT is the preemtive counterpart of SJF and useful in time-sharing environment.
- In SRT scheduling, the process with the smallest estimated run-time to completion is run next, including new arrivals.
- In SJF scheme, once a job begin executing, it run to completion.

- In SJF scheme, a running process may be preempted by a new arrival process with shortest estimated run-time.
- The algorithm SRT has higher overhead than its counterpart SJF.
- The SRT must keep track of the elapsed time of the running process and must handle occasional preemptions.
- In this scheme, arrival of small processes will run almost immediately. However, longer jobs have even longer mean waiting time.

Priority Scheduling

The basic idea is straightforward: each process is assigned a priority, and priority is allowed to run. Equal-Priority processes are scheduled in FCFS order. The shortest-Job-First (SJF) algorithm is a special case of general priority scheduling algorithm.

An SJF algorithm is simply a priority algorithm where the priority is the inverse of the (predicted) next CPU burst. That is, the longer the CPU burst, the lower the priority and vice versa.

Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities or qualities to compute priority of a process.

Examples of Internal priorities are:

- Time limits.
- Memory requirements.
- File requirements, for example, number of open files.
- CPU Vs I/O requirements.

Externally defined priorities are set by criteria that are external to operating system such as:

- The importance of process.
- Type or amount of funds being paid for computer use.
- The department sponsoring the work.
- Politics.

Priority scheduling can be either preemptive or non preemptive:

- A preemptive priority algorithm will preemptive the CPU if the priority of the newly arrival process is higher than the priority of the currently running process.
- A non-preemptive priority algorithm will simply put the new process at the head of the ready queue.

A major problem with priority scheduling is indefinite blocking or starvation. A solution to the problem of indefinite blockage of the low-priority process is *aging*. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long period of time.

Multilevel Queue Scheduling

A multilevel queue scheduling algorithm partitions the ready queue in several separate queues, for instance

In a multilevel queue scheduling processes are permanently assigned to one queues.

The processes are permanently assigned to one another, based on some property of the process, such as:

- Memory size
- Process priority
- Process type

Algorithm choose the process from the occupied queue that has the highest priority, and run that process either:

- Preemptive or
- Non-preemptively

Each queue has its own scheduling algorithm or policy.

Possibility I

If each queue has absolute priority over lower-priority queues then no process in the queue could run unless the queue for the highest-priority processes were all empty. For example, in the above figure no process in the batch queue could run unless the queues for system processes, interactive processes, and interactive editing processes will all empty.

Possibility II

If there is a time slice between the queues then each queue gets a certain amount of CPU times, which it can then schedule among the processes in its queue. For instance;

- 80% of the CPU time to foreground queue using RR.
- 20% of the CPU time to background queue using FCFS.

Since processes do not move between queue so, this policy has the advantage of low scheduling overhead, but it is inflexible.

Multilevel Feedback Queue Scheduling

Multilevel feedback queue-scheduling algorithm allows a process to move between queues. It uses many ready queues and associate a different priority with each queue.

The Algorithm chooses to process with highest priority from the occupied queue and run that process either preemptively

or unpreemptively. If the process uses too much CPU time it will moved to a lower-priority queue. Similarly, a process that wait too long in the lower-priority queue may be moved to a higher-priority queue may be moved to a highest-priority queue. Note that this form of aging prevents starvation.

- A process entering the ready queue is placed in queue 0.
- If it does not finish within 8 milliseconds time, it is moved to the tail of queue 1.
- If it does not complete, it is preempted and placed into queue 2.
- Processes in queue 2 run on a FCFS basis, only when queue 2 run on a FCFS basis, only when queue 0 and queue 1 are empty.

Interprocess Communication

Since processes frequently needs to communicate with other processes therefore, there is a need for a well-structured communication, without using interrupts, among processes.

Race Conditions

In operating systems, processes that are working together share some common storage (main memory, file *etc.*) that each process can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Concurrently executing threads that share data need to synchronize their operations and processing in order to avoid race condition on shared data. Only one 'customer' thread at a time should be allowed to examine and update the shared variable. Race conditions are also possible in

Operating Systems. If the ready queue is implemented as a linked list and if the ready queue is being manipulated during the handling of an interrupt, then interrupts must be disabled to prevent another interrupt before the first one completes. If interrupts are not disabled than the linked list could become corrupt.

Mutual Exclusion

A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion.

Note that mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.

Mutual Exclusion Conditions

If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

- No two processes may at the same moment inside their critical sections.
- No assumptions are made about relative speeds of processes or number of CPUs.
- No process should outside its critical section should block other processes.
- No process should wait arbitrary long to enter its critical section.

Proposals for Achieving Mutual Exclusion

The mutual exclusion problem is to devise a pre-protocol (or entry protocol) and a post-protocol (or exist protocol) to keep two or more threads from being in their critical sections at the same time.

Tanenbaum examine proposals for critical-section problem or mutual exclusion problem.

Problem

When one process is updating shared modifiable data in its critical section, no other process should allowed to enter in its critical section.

Proposal 1 -Disabling Interrupts (Hardware Solution)

Each process disables all interrupts just after entering in its critical section and re-enable all interrupts just before leaving critical section.

With interrupts turned off the CPU could not be switched to other process. Hence, no other process will enter its critical and mutual exclusion achieved.

Conclusion

Disabling interrupts is sometimes a useful interrupts is sometimes a useful technique within the kernel of an operating system, but it is not appropriate as a general mutual exclusion mechanism for users process. The reason is that it is unwise to give user process the power to turn off interrupts.

Proposal 2 - Lock Variable (Software Solution)

In this solution, we consider a single, shared, (lock) variable, initially 0. When a process wants to enter in its critical section, it first test the lock. If lock is 0, the process first sets it to 1 and then enters the critical section. If the lock is already 1, the process just waits until (lock) variable becomes 0. Thus, a 0 means that no process in its critical section, and 1 means hold your horses - some process is in its critical section.

Conclusion

The flaw in this proposal can be best explained by example. Suppose process A sees that the lock is 0. Before it can set the lock to 1 another process B is scheduled, runs, and sets the lock to 1. When the process A runs again, it will also set the lock to 1, and two processes will be in their critical section simultaneously.

Proposal 3 - Strict Alteration

In this proposed solution, the integer variable 'turn' keeps track of whose turn is to enter the critical section. Initially, process A inspect turn, finds it to be 0, and enters in its critical section. Process B also finds it to be 0 and sits in a

loop continually testing 'turn' to see when it becomes 1. Continuously testing a variable waiting for some value to appear is called the *Busy-Waiting*.

Conclusion

Taking turns is not a good idea when one of the processes is much slower than the other. Suppose process 0 finishes its critical section quickly, so both processes are now in their noncritical section. This situation violates above mentioned condition 3.

Using Systems Calls 'Sleep' and 'Wakeup'

Basically, what above mentioned solution do is this: when a processes wants to enter in its critical section, it checks to see if then entry is allowed. If it is not, the process goes into tight loop and waits (*i.e.,* start busy waiting) until it is allowed to enter. This approach waste CPU-time.

Now look at some interprocess communication primitives is the pair of steep-wakeup.

- Sleep
  - It is a system call that causes the caller to block, that is, be suspended until some other process wakes it up.
- Wakeup
  - It is a system call that wakes up the process.

Both 'sleep' and 'wakeup' system calls have one parameter that represents a memory address used to match up 'sleeps' and 'wakeups'.

The Bounded Buffer Producers and Consumers

The bounded buffer producers and consumers assumes that there is a fixed buffer size *i.e.,* a finite numbers of slots are available.

Statement

To suspend the producers when the buffer is full, to suspend the consumers when the buffer is empty, and to make sure that only one process at a time manipulates a buffer so there are no race conditions or lost updates. As an example how sleep-wakeup system calls are used, consider the producer-consumer problem also known as bounded buffer problem. Two processes share a common, fixed-size (bounded) buffer. The producer puts information into the buffer and the consumer takes information out.

Trouble arises when:

1. The producer wants to put a new data in the buffer, but buffer is already full. Solution: Producer goes to sleep and to be awakened when the consumer has removed data.
2. The consumer wants to remove data the buffer but buffer is already empty. Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

Conclusion

This approaches also leads to same race conditions we have seen in earlier approaches. Race condition can occur due to the fact that access to 'count' is unconstrained. The essence of the problem is that a wakeup call, sent to a process that is not sleeping, is lost.

Semaphores

Definition

A semaphore is a protected variable whose value can be accessed and altered only by the operations P and V and

initialization operation called 'Semaphoiinitislize'. Binary Semaphores can assume only the value 0 or the value 1 counting semaphores also called general semaphores can assume only nonnegative values. The P (or wait or sleep or down) operation on semaphores S, written as P(S) or wait (S), operates as follows:

```
P(S): IF S > 0
      THEN S:= S – 1
      ELSE (wait on S)
```

The V (or signal or wakeup or up) operation on semaphore S, written as V(S) or signal (S), operates as follows:

```
V(S): IF (one or more process are waiting on S)
      THEN (let one of these processes proceed)
      ELSE S:= S +1
```

Operations P and V are done as single, indivisible, atomic action. It is guaranteed that once a semaphore operations has stared, no other process can access the semaphore until operation has completed. Mutual exclusion on the semaphore, S, is enforced within P(S) and V(S). If several processes attempt a P(S) simultaneously, only process will be allowed to proceed. The other processes will be kept waiting, but the implementation of P and V guarantees that processes will not suffer indefinite postponement. Semaphores solve the lost-wakeup problem.

Producer-Consumer Problem Using Semaphores

The Solution to producer-consumer problem uses three semaphores, namely, full, empty and mutex. The semaphore 'full' is used for counting the number of slots in the buffer that are full. The 'empty' for counting the number of slots that are empty and semaphore 'mutex' to make sure that the producer and consumer do not access modifiable shared section of the buffer simultaneously.

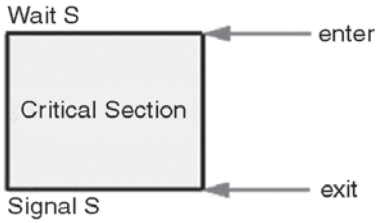
Initialization

- Set full buffer slots to 0. *i.e.*, semaphore Full = 0.
- Set empty buffer slots to N. *i.e.*, semaphore empty = N.
- For control access to critical section set mutex to 1. *i.e.*, semaphore mutex = 1.

```
Producer ()
WHILE (true)
     produce-Item ();
P (empty);
P (mutex);
enter-Item ()
V (mutex)
V (full);
Consumer ()
WHILE (true)
     P (full)
     P (mutex);
     remove-Item ();
     V (mutex);
     V (empty);
     consume-Item (Item)
```

System Critical Section

Critical Section



The key to preventing trouble involving shared storage is find some way to prohibit more than one process from reading and writing the shared data simultaneously. That part of the programme where the shared memory is accessed is called the *Critical Section*. To avoid race conditions and flawed results, one must identify codes in *Critical Sections* in each thread. The characteristic properties of the code that form a *Critical Section* are

- Codes that reference one or more variables in a "read-update-write" fashion while any of those variables is possibly being altered by another thread.
- Codes that alter one or more variables that are possibly being referenced in "read-updata-write" fashion by another thread.
- Codes use a data structure while any part of it is possibly being altered by another thread.
- Codes alter any part of a data structure while it is possibly in use by another thread.

Here, the important point is that when one process is executing shared modifiable data in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time.

Critical Section Objects

A *critical section object* provides synchronization similar to that provided by a mutex object, except that a critical section can be used only by the threads of a single process. Event, mutex, and semaphore objects can also be used in a single-process application, but critical section objects provide a

slightly faster, more efficient mechanism for mutual-exclusion synchronization (a processor-specific test and set instruction). Like a mutex object, a critical section object can be owned by only one thread at a time, which makes it useful for protecting a shared resource from simultaneous access. Unlike a mutex object, there is no way to tell whether a critical section has been abandoned.

Starting with Windows Server 2003 with Service Pack 1 (SP1), threads waiting on a critical section do not acquire the critical section on a first-come, first-serve basis. This change increases performance significantly for most code. However, some applications depend on FIFO ordering and may perform poorly or not at all on current versions of Windows (for example, applications that have been using critical sections as a rate-limiter). To ensure that your code continues to work correctly, you may need to add an additional level of synchronization. For example, suppose you have a producer thread and a consumer thread that are using a critical section object to synchronize their work. Create two event objects, one for each thread to use to signal that it is ready for the other thread to proceed. The consumer thread will wait for the producer to signal its event before entering the critical section, and the producer thread will wait for the consumer thread to signal its event before entering the critical section. After each thread leaves the critical section, it signals its event to release the other thread.

Windows Server 2003 and Windows XP/2000: Threads that are waiting on a critical section are added to a wait queue; they are woken and generally acquire the critical section in the order in which they were added to the queue. However, if

threads are added to this queue at a fast enough rate, performance can be degraded because of the time it takes to awaken each waiting thread.

The process is responsible for allocating the memory used by a critical section. Typically, this is done by simply declaring a variable of type CRITICAL_SECTION. Before the threads of the process can use it, initialize the critical section by using the Initialize Critical Section or Initialize Critical Section And Spin Count function.

A thread uses the EnterCriticalSection or TryEnterCriticalSection function to request ownership of a critical section. It uses the Leave Critical Section function to release ownership of a critical section. If the critical section object is currently owned by another thread, Enter Critical Section waits indefinitely for ownership. In contrast, when a mutex object is used for mutual exclusion, the wait functions accept a specified time-out interval. The Try Enter Critical Section function attempts to enter a critical section without blocking the calling thread.

When a thread owns a critical section, it can make additional calls to Enter Critical Section or Try Enter Critical Section without blocking its execution. This prevents a thread from deadlocking itself while waiting for a critical section that it already owns. To release its ownership, the thread must call Leave Critical Section one time for each time that it entered the critical section. There is no guarantee about the order in which waiting threads will acquire ownership of the critical section.

A thread uses the Initialize CriticalSection And Spin Count or SetCriticalSectionSpinCount function to specify a spin

count for the critical section object. Spinning means that when a thread tries to acquire a critical section that is locked, the thread enters a loop, checks to see if the lock is released, and if the lock is not released, the thread goes to sleep. On single-processor systems, the spin count is ignored and the critical section spin count is set to 0 (zero). On multiprocessor systems, if the critical section is unavailable, the calling thread spins *dwSpin Count* times before performing a wait operation on a semaphore that is associated with the critical section. If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.

Any thread of the process can use the DeleteCriticalSection function to release the system resources that are allocated when the critical section object is initialized. After this function is called, the critical section object cannot be used for synchronization.

When a critical section object is owned, the only other threads affected are the threads that are waiting for ownership in a call to EnterCriticalSection. Threads that are not waiting are free to continue running.

Semaphores

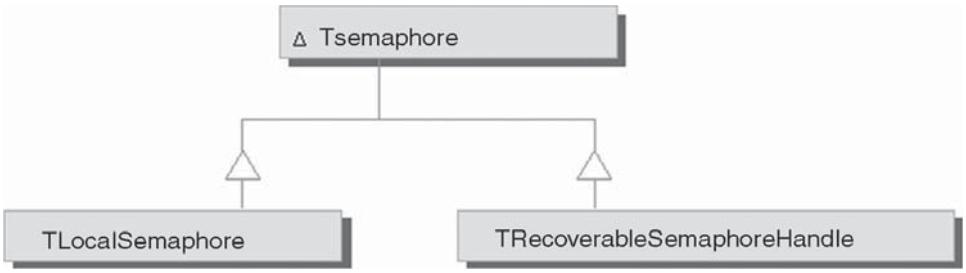
The CommonPoint application system provides two kinds of semaphores: *readers/writer locks* and *counting semaphores*, though the readers/writer lock style of semaphore is preferred.

A readers/writer lock semaphore contains no explicit reference associating it with the shared data it protects. You maintain the association simply by programming cooperating threads to acquire the semaphore before accessing the data.

This style of programming allows multiple threads (readers) to read the data associated with the semaphore simultaneously, but allows only one thread at a time (a writer) to modify the data.

When a thread acquires a readers/writer lock in *shared mode*, other threads can also hold the lock in shared mode at the same time. When a thread acquires the lock in *restricted mode*, no other thread can hold or acquire the lock until it's released. By convention you only use shared mode for reading the data associated with the semaphore; you use restricted mode for writing.

When using a semaphore to represent an operation rather than data, releasing restricted access to the semaphore could signal the end of the operation. Other threads that depend on the operation being complete could simply acquire the semaphore in restricted mode before proceeding. You can use a message stream to accomplish this coordination if you need to communicate detailed information between threads, but this method is slower than using a semaphore.



Local and Recoverable Semaphores

The CommonPoint application system provides two kinds of readers/writer lock semaphores: *local semaphores*, which can only be used within tasks, and *recoverable semaphores*, which can be shared among tasks. A recoverable semaphore

has the ability to recover when the thread holding the semaphore unexpectedly terminates without releasing the semaphore; in this case, the recoverable semaphore restores itself to an unlocked state and unblocks any threads waiting to acquire it.

This capability is essential for preventing deadlocks involving semaphores that are shared among tasks.

These rules govern the acquisition and release of readers/writer lock semaphores:

- A thread that acquires a semaphore in restricted mode becomes the *owner* of the semaphore. Once the semaphore is acquired in restricted mode, no other thread can acquire the semaphore in any mode, and only the owner is allowed to release the semaphore.
- When a thread attempts to acquire a semaphore in restricted mode while any other threads hold the semaphore in either mode, the thread blocks until all other threads release the semaphore.
- When a thread releases restricted-mode ownership of a semaphore, all threads waiting to acquire the semaphore in shared mode are unblocked (and allowed to acquire the semaphore) before threads waiting to acquire the semaphore in restricted mode.
- Threads waiting to acquire a semaphore in restricted mode are allowed to acquire the semaphore on a first-come-first-served basis.
- When a semaphore is destroyed, all threads waiting to acquire it receive a T Kernel Exception with an error code of Semaphore Deleted.

Software Metrics in Algorithmic

In computer science, efficiency is used to describe properties of an algorithm relating to how much of various types of resources it consumes. Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process, where the goal is to reduce resource consumption, including time to completion, to some acceptable, optimal level.

Software Metrics

The two most frequently encountered and measurable metrics of an algorithm are:-

- speed or running time - the time it takes for an algorithm to complete, and
- 'space' - the memory or 'non-volatile storage' used by the algorithm during its operation.

but also might apply to

- transmission size - such as required bandwidth during normal operation or
- size of external memory- such as temporary disk space used to accomplish its task

and perhaps even

- the size of required 'longterm' disk space required after its operation to record its output or maintain its required function during its required useful lifetime (examples: a data table, archive or a computer log) and also
- the performance per watt and the total energy, consumed by the chosen hardware implementation

> (with its System requirements, necessary auxiliary support systems including interfaces, cabling, switching, cooling and security), during its required useful lifetime. See Total cost of ownership for other potential costs that might be associated with any particular implementation.

(An extreme example of these metrics might be to consider their values in relation to a repeated simple algorithm for calculating and storing (ð+n) to 50 decimal places running for say, 24 hours, either on a "pocket calculator" sized processor such as an ipod or an early mainframe operating in its own purpose-built heated or air conditioned unit.) The process of making code more efficient is known as optimization and in the case of automatic optimization (i.e. compiler optimization - performed by compilers on request or by default), usually focus on space at the cost of speed, or vice versa.

There are also quite simple programming techniques and 'avoidance strategies' that can actually improve both at the same time, usually irrespective of hardware, software or language. Even the re-ordering of nested conditional statements - to put the least frequently occurring condition first (example: test patients for blood type ='AB-', before testing age > 18, since this type of blood occurs in only about 1 in 100 of the population - thereby eliminating the second test at runtime in 99% of instances), can reduce actual instruction path length, something an optimizing compiler would almost certainly not be aware of - but which a programmer can research relatively easily even without specialist medical knowledge.

History

The first machines that were capable of computation were severely limited by purely mechanical considerations. As later electronic machines were developed they were, in turn, limited by the speed of their electronic counterparts. As software replaced hard-wired circuits, the efficiency of algorithms also became important. It has long been recognized that the precise 'arrangement of processes' is critical in reducing elapse time.

- "In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purposes of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation"

Ada Lovelace 1815-1852, generally considered as 'the first programmer' who worked on Charles Babbage's early mechanical general-purpose computer

- "In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering"

Extract from "Structured Programming with go to Statements" by Donald Knuth, renowned computer scientist and Professor Emeritus of the Art of Computer Programming at Stanford University.

- "The key to performance is elegance, not battalions of special cases" attributed to Jon Bentley and (Malcolm) Douglas McIlroy

Speed

The absolute speed of an algorithm for a given input can simply be measured as the duration of execution (or clock time) and the results can be averaged over several executions to eliminate possible random effects. Most modern processors operate in a multi-processing & multi-programming environment so consideration must be made for parallel processes occurring on the same physical machine, eliminating these as far as possible. For superscalar processors, the speed of a given algorithm can sometimes be improved through instruction-level parallelism within a single processor (but, for optimal results, the algorithm may require some adaptation to this environment to gain significant advantage ('speedup'), becoming, in effect, an entirely different algorithm). A relative measure of an algorithms performance can sometimes be gained from the total instruction path length which can be determined by a run time Instruction Set Simulator (where available). An estimate of the speed of an algorithm can be determined in various ways. The most common method uses time complexity to determine the Big-O of an algorithm. See Run-time analysis for estimating how fast a particular algorithm may be according to its type (example: lookup unsorted list, lookup sorted list etc.) and in terms of scalability - its dependence on 'size of input', processor power and other factors.

Memory

Often, it is possible to make an algorithm faster at the expense of memory. This might be the case whenever the result of an 'expensive' calculation is cached rather than recalculating it afresh each time. The additional memory requirement would, in this case, be considered additional overhead although, in many situations, the stored result occupies very little extra space and can often be held in pre-compiled static storage, reducing not just processing time but also allocation & deallocation of working memory. This is a very common method of improving speed, so much so that some programming languages often add special features to support it, such as C++'s 'mutable' keyword. The memory requirement of an algorithm is actually two separate but related things:-

- The memory taken up by the compiled executable code (the object code or binary file) itself (on disk or equivalent, depending on the hardware and language). This can often be reduced by preferring run-time decision making mechanisms (such as virtual functions and run-time type information) over certain compile-time decision making mechanisms (such as macro substitution and templates). This, however, comes at the cost of speed.
- Amount of temporary "dynamic memory" allocated during processing. For example, dynamically pre-caching results, as mentioned earlier, improves speed at the cost of this attribute. Even the depth of sub-routine calls can impact heavily on this cost and increase path length too, especially if there are 'heavy'

dynamic memory requirements for the particular functions invoked. The use of copied function parameters (rather than simply using pointers to earlier, already defined, and sometimes static values) actually doubles the memory requirement for this particular memory metric (as well as carrying its own processing overhead for the copying itself. This can be particularly relevant for quite 'lengthy' parameters such as html script, JavaScript source programmes or extensive freeform text such as letters or emails.

Rematerialization

It has been argued that Rematerialization (re-calculating) may occasionally be more efficient than holding results in cache. This is the somewhat non-intuitive belief that it can be faster to re-calculate from the input - even if the answer is already known - when it can be shown, in some special cases, to decrease "register pressure". Some optimizing compilers have the ability to decide when this is considered worthwhile based on a number of criteria such as complexity and no side effects, and works by keeping track of the expression used to compute each variable, using the concept of available expressions. This is most likely to be true when a calculation is very fast (such as addition or bitwise operations), while the amount of data which must be cached would be very large, resulting in inefficient storage. Small amounts of data can be stored very efficiently in registers or fast cache, while in most contemporary computers large amounts of data must be stored in slower memory or even slower hard drive storage, and thus the efficiency of storing data which can be computed quickly drops significantly.

Precomputation

Precomputing a complete range of results prior to compiling, or at the beginning of an algorithm's execution, can often increase algorithmic efficiency substantially. This becomes advantageous when one or more inputs is constrained to a small enough range that the results can be stored in a reasonably sized block of memory. Because memory access is essentially constant in time complexity (except for caching delays), any algorithm with a component which has worse than constant efficiency over a small input range can be improved by precomputing values. In some cases efficient approximation algorithms can be obtained by computing a discrete subset of values and interpolating for intermediate input values, since interpolation is also a linear operation.

Transmission Size

Data compression algorithms can be useful because they help reduce the consumption of expensive resources, such as hard disk space or transmission bandwidth. This however also comes at a cost - which is additional processing time to compress and subsequently decompress. Depending upon the speed of the data transfer, compression may reduce overall response times which, ultimately, equates to speed - even though processing within the computer itself takes longer. For audio, MP3 is a compression method used extensively in portable sound systems. The efficiency of a data compression algorithm relates to the compression factor and speed of achieving both compression and decompression. For the purpose of archiving an extensive database, it might

be considered worthwhile to achieve a very high compression ratio, since decompression is less likely to occur on the majority of the data.

Data Presentation

Output data can be presented to the end user in many ways - such as via punched tape or card, digital displays, local display monitors, remote computer monitors or printed. Each of these has its own inherent initial cost and, in some cases, an ongoing cost (e.g. refreshing an image from memory). As an example, the web site "Google" recently showed, as its logo, an image of the Vancouver olympics that is around 8K of gif image.

The normally displayed Google image is a PNG image of 28K (or 48K), yet the raw text string for "Google" occupies only 6 octets or 48 bits (4,778 or 8192 *times* less). This graphically illustrates that how data is presented can significantly effect the overall efficiency of transmission (and also the complete algorithm - since both GIF and PNG images require yet more processing). It is estimated by "Internet World Stats" that there were 1,733,993,741 internet users in 2009 and, to transmit this new image to each one of them, would require around 136,000 *billion* (10^9)octets of data to be transmitted - at least once - into their personal web cache. In "Computational Energy Cost of TCP", co-authors Bokyung Wang and Suresh Singh examine the energy costs for TCP and calculated, for their chosen example, a cost of 0.022 Joules per packet (of approx 1489 octets). On this basis, a total of around 2,000,000,000 joules (2 GJ) of energy might be expended by TCP elements alone to

display the new logo for all users for the first time. To maintain or re-display this image requires still more processing and consequential energy cost (in contrast to printed output for instance).

Encoding and Decoding Methods (Compared and Contrasted)

When data is encoded for any 'external' use, it is possible to do so in an almost unlimited variety of different formats that are sometimes conflicting. This content encoding (of the raw data) may be designed for:

- optimal readability – by humans
- optimal decoding speed – by other computer programmes
- optimal compression – for archiving or data transmission
- optimal compatibility – with "legacy" or other existing formats or programming languages
- optimal security – using encryption
  (For character level encoding, see the various encoding techniques such as EBCDIC or ASCII)

It is unlikely that all of these goals could be met with a single 'generic' encoding scheme and so a compromise will often be the desired goal and will often be compromised by the need for standardization and/or legacy and compatibility issues.

Encoding Efficiently

For data encoding whose destination is to be input for further computer processing, readability is not an issue –

since the receiving processors algorithm can decode the data to any other desirable form including human readable. From the perspective of algorithmic efficiency, minimizing subsequent decoding (with zero or minimal parsing) should take highest priority. The general rule of thumb is that any encoding system that 'understands' the underlying data structure - sufficiently to encode it in the first place - should be equally capable of easily encoding it in such a way that makes decoding it highly efficient. For variable length data with possibly omitted data values, for instance, this almost certainly means the utilization of declarative notation (i.e. including the length of the data item as a prefix to the data so that a de-limiter is not required and parsing completely eliminated). For keyword data items, tokenizing the key to an index (integer) after its first occurrence not only reduces subsequent data size but, furthermore, reduces future decoding overhead for the same items that are repeated. For more 'generic' encoding for efficient data compression see Arithmetic encoding and entropy encoding articles.

Historically, optimal encoding was not only worthwhile from an efficiency standpoint but was also common practise to conserve valuable memory, external storage and processor resources. Once validated a country name for example could be held as a shorter sequential country code which could then also act as an index for subsequent 'decoding', using this code as an entry number within a table or record number within a file. If the table or file contained fixed length entries, the code could easily be converted to an absolute memory address or disk address for fast retrieval. The ISBN system for identifying books is a good example

of a practical encoding method which also contains a built-in hierarchy. According to recent articles in New Scientist and Scientific American; "TODAY'S telecommunications networks could use one ten-thousandth of the power they presently consume if smarter data-coding techniques were used", according to Bell Labs, based in Murray Hill, New Jersey It recognizes that this is only a theoretical limit but nevertheless sets itself a more realistic, practical goal of a 1,000 fold reduction within 5 years with future, as yet unidentified, technological changes.

Examples of Several Common Encoding Methods

- Comma separated values (CSV - a list of data values separated by commas)
- Tab separated values (TSV) - a list of data values separated by 'tab' characters
- HyperText Markup Language (HTML) - the predominant markup language for web pages
- Extensible Markup Language (XML) - a generic framework for storing any amount of text or any data whose structure can be represented as a tree with at least one element - the root element.
- JavaScript Object Notation (JSON) - human-readable format for representing simple data structures

The last of these, (JSON) is apparently widely used for internet data transmission, primarily it seems because the data can be uploaded by a single JavaScript 'eval' statement - without the need to produce what otherwise would likely have been a more efficient purpose built encoder/decoder. There are in fact quite large amounts of repeated (and

therefore redundant data) in this particular format, and also in HTML and XML source, that could quite easily be eliminated. XML is recognized as a verbose form of encoding. Binary XML has been put forward as one method of reducing transfer and processing times for XML and, while there are several competing formats, none has been widely adopted by a standards organization or accepted as a de facto standard. It has also been criticized by Jimmy Zhang for essentially trying to solve the wrong problem There are a number of freely available products on the market that partially compress HTML files and perform some or all of the following:

- merge lines
- remove unnecessary whitespace characters
- remove unnecessary quotation marks. For example, BORDER=”0" will be replaced with BORDER=0)
- replace some tags with shorter ones (e.g. replace STRIKE tags with S, STRONG with B and EM with I)
- remove HTML comments (comments within scripts and styles are not removed)
- remove <!DOCTYPE..> tags
- remove named meta tags

The effect of this limited form of compression is to make the HTML code smaller and faster to load, but more difficult to read manually (so the original HTML code is usually retained for updating), but since it is predominantly meant to be processed only by a browser, this causes no problems. Despite these small improvements, HTML, which is the

predominant language for the web still remains a predominantly *source* distributed, interpreted markup language, with high redundancy.

Kolmogorov Complexity

The study of encoding techniques has been examined in depth in an area of computer science characterized by a method known as Kolmogorov complexity where a value known as ('K') is accepted as 'not a computable function'. The Kolmogorov complexity of any computable object is the length of the shortest programme that computes it and then halts. The invariance theorem shows that it is not really important which computer is used. Essentially this implies that there is no automated method that can produce an optimum result and is therefore characterized by a requirement for human ingenuity or Innovation. See also Algorithmic probability.

Effect of Programming Paradigms

The effect that different programming paradigms have on algorithmic efficiency is fiercely contested, with both supporters and antagonists for each new paradigm. Strong supporters of structured programming, such as Dijkstra for instance, who favour entirely goto-less programmes are met with conflicting evidence that appears to nullify its supposed benefits. The truth is, even if the structured code itself contains no gotos, the optimizing compiler that creates the binary code almost certainly generates them (and not necessarily in the most efficient way). Similarly, OOP protagonists who claim their paradigm is superior are met with opposition from strong sceptics such as Alexander

Stepanov who suggested that OOP provides a mathematically limited viewpoint and called it, "almost as much of a hoax as Artificial Intelligence" In the long term, benchmarks, using real-life examples, provide the only real hope of resolving such conflicts - at least in terms of run-time efficiency.

Optimization Techniques

The word optimize is normally used in relation to an existing algorithm/computer programme (i.e. to improve upon completed code). In this section it is used both in the context of existing programmes and also in the design and implementation of new algorithms, thereby avoiding the most common performance pitfalls. It is clearly wasteful to produce a working programme - at first using an algorithm that ignores all efficiency issues - only to then have to redesign or rewrite sections of it if found to offer poor performance. Optimization can be broadly categorized into two domains:-

- Environment specific - that are essentially worthwhile only on certain platforms or particular computer languages
- General techniques - that apply irrespective of platform

Environment Specific

Optimization of algorithms frequently depends on the properties of the machine the algorithm will be executed on as well as the language the algorithm is written in and chosen data types. For example, a programmer might optimize code for time efficiency in an application for home

computers (with sizable amounts of memory), but for code destined to be embedded in small, "memory-tight" devices, the programmer may have to accept that it will run more slowly, simply because of the restricted memory available for any potential software optimization. For a discussion of hardware performance, see article on Computer performance which covers such things as CPU clock speed, cycles per instruction and other relevant metrics. For a discussion on how the choice of particular instructions available on a specific machine effect efficiency, see later section 'Choice of instruction and data type'.

General Techniques

- Linear search such as unsorted table look-ups in particular can be very expensive in terms of execution time but can be reduced significantly through use of efficient techniques such as indexed arrays and binary searches. Using a simple linear search on first occurrence and using a cached result thereafter is an obvious compromise.
- Use of indexed programme branching, utilizing branch tables or "threaded code" to control programme flow, (rather than using multiple conditional IF statements or unoptimized CASE/SWITCH) can drastically reduce instruction path length, simultaneously reduce programme size and even also make a programme easier to read and more easily maintainable (in effect it becomes a 'decision table' rather than repetitive spaghetti code).
- Loop unrolling performed manually, or more usually by an optimizing compiler, can provide significant

savings in some instances. By processing 'blocks' of several array elements at a time, individually addressed, (for example, within a While loop), much pointer arithmetic and end of loop testing can be eliminated, resulting in decreased instruction path lengths. Other Loop optimizations are also possible.

Tunnel Vision

There are many techniques for improving algorithms, but focusing on a single favourite technique can lead to a "tunnel vision" mentality. For example, in this X86 assembly example, the author offers loop unrolling as a reasonable technique that provides some 40% improvements to his chosen example.

However, the same example would benefit significantly from both inlining and use of a trivial hash function. If they were implemented, either as alternative or complementary techniques, an even greater percentage gain might be expected. A combination of optimizations may provide ever increasing speed, but selection of the most easily implemented and most effective technique, from a large repertoire of such techniques, is desirable as a starting point.

Dependency Trees and Spreadsheets

Spreadsheets are a 'special case' of algorithms that self-optimize by virtue of their dependency trees that are inherent in the design of spreadsheets to reduce re-calculations when a cell changes. The results of earlier calculations are effectively cached within the workbook and only updated if another cells changed value effects it directly.

Table Lookup

Table lookups can make many algorithms more efficient, particularly when used to bypass computations with a high time complexity. However, if a wide input range is required, they can consume significant storage resources. In cases with a sparse valid input set, hash functions can be used to provide more efficient lookup access than a full table.

Hash Function Algorithms

A hash function is any well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. Hash functions are frequently used to speed up table lookups. The choice of a hashing function (to avoid a linear or brute force search) depends critically on the nature of the input data, and their probability distribution in the intended application.

Trivial Hash Function

Sometimes if the datum is small enough, a "trivial hash function" can be used to effectively provide constant time searches at almost zero cost. This is particularly relevant for single byte lookups (e.g. ASCII or EBCDIC characters)

Searching Strings

Searching for particular text strings (for instance "tags" or keywords) in long sequences of characters potentially generates lengthy instruction paths. This includes searching for delimiters in comma separated files or similar processing

which can be very simply and effectively eliminated (using declarative notation for instance). Several methods of reducing the cost for general searching have been examined and the "Boyer–Moore string search algorithm" (or Boyer–Moore–Horspool algorithm, a similar but modified version) is one solution that has been proven to give superior results to repetitive comparisons of the entire search string along the sequence.

Hot Spot Analyzers

Special system software products known as "performance analyzers" are often available from suppliers to help diagnose "hot spots" - during actual execution of computer programmes - using real or test data - they perform a Performance analysis under generally repeatable conditions. They can pinpoint sections of the programme that might benefit from specifically targeted programmer optimization without necessarily spending time optimizing the rest of the code. Using programme re-runs, a measure of relative improvement can then be determined to decide if the optimization was successful and by what amount. Instruction Set Simulators can be used as an alternative to measure the instruction path length at the machine code level between selected execution paths, or on the entire execution. Regardless of the type of tool used, the quantitative values obtained can be used in combination with anticipated reductions (for the targeted code) to estimate a relative or absolute overall saving. For example if 50% of the total execution time (or path length) is absorbed in a subroutine whose speed can be doubled by programmer optimization, an overall saving of around 25% might be expected (Amdahl

law). Efforts have been made at the University of California, Irvine to produce dynamic executable code using a combination of hot spot analysis and run-time programme trace tree. A JIT like dynamic compiler was built by Andreas Gal and others, "in which relevant (i.e., frequently executed) control flows are ...discovered lazily during execution"

Benchmarking & Competitive Algorithms

For new versions of software or to provide comparisons with competitive systems, benchmarks are sometimes used which assist with gauging an algorithms relative performance. If a new sort algorithm is produced for example it can be compared with its predecessors to ensure that at least it is efficient as before with known data - taking into consideration any functional improvements. Benchmarks can be used by customers when comparing various products from alternative suppliers to estimate which product will best suit their specific requirements in terms of functionality and performance. For example in the mainframe world certain proprietary sort products from independent software companies such as Syncsort compete with products from the major suppliers such as IBM for speed. Some benchmarks provide opportunities for producing an analysis comparing the relative speed of various compiled and interpreted languages for example and *The Computer Language Benchmarks Game* compares the performance of implementations of typical programming problems in several programming languages. (Even creating "do it yourself" benchmarks to get at least some appreciation of the relative performance of different programming languages, using a

variety of user specified criteria, is quite simple to produce as this "Nine language Performance roundup" by Christopher W. Cowell-Shah demonstrates by example)

Compiled Versus Interpreted Languages

A compiled algorithm will, in general, execute faster than the equivalent interpreted algorithm simply because some processing is required even at run time to 'understand' (i.e. interpret) the instructions to effect an execution. A compiled programme will normally output an object or machine code equivalent of the algorithm that has already been processed by the compiler into a form more readily executed by microcode or the hardware directly. The popular Perl language is an example of an interpreted language and benchmarks indicate that it executes approximately 24 times more slowly than compiled C.

Optimizing Compilers

Many compilers have features that attempt to optimize the code they generate, utilizing some of the techniques outlined in this study and others specific to the compilation itself. Loop optimization is often the focus of optimizing compilers because significant time is spent in programme loops and parallel processing opportunities can often be facilitated by automatic code re-structuring such as loop unrolling. Optimizing compilers are by no means perfect. There is no way that a compiler can guarantee that, for all programme source code, the fastest (or smallest) possible equivalent compiled programme is output; such a compiler is fundamentally impossible because it would solve the halting problem. Additionally, even optimizing compilers

generally have no access to runtime metrics to enable them to improve optimization through 'learning'.

Just-in-Time Compilers

'On-the-fly' processors known today as just-in-time or 'JIT' compilers combine features of interpreted languages with compiled languages and may also incorporate elements of optimization to a greater or lesser extent. Essentially the JIT compiler can compile small sections of source code statements (or bytecode) as they are newly encountered and (usually) retain the result for the next time the same source is processed. In addition, pre-compiled segments of code can be in-lined or called as dynamic functions that themselves perform equally fast as the equivalent 'custom' compiled function. Because the JIT processor also has access to run-time information (that a normal compiler can't have) it is also possible for it to optimize further executions depending upon the input and also perform other run-time introspective optimization as execution proceeds. A JIT processor may, or may not, incorporate self modifying code or its equivalent by creating 'fast path' routes through an algorithm. It may also use such techniques as dynamic Fractional cascading or any other similar runtime device based on collected actual runtime metrics. It is therefore entirely possible that a JIT compiler might (counter intuitively) execute even faster than an optimally 'optimized' compiled programme.

Self-Modifying Code

As mentioned above, just-in-time compilers often make extensive use of self-modifying code to generate the actual

machine instructions required to be executed. The technique can also be used to reduce instruction path lengths in application programmes where otherwise repetitive conditional tests might otherwise be required within the main programme flow. This can be particularly useful where a sub routine may have embedded debugging code that is either active (testing mode) or inactive (production mode) depending upon some input parameter. A simple solution using a form of dynamic dispatching is where the sub routine entry point is dynamically 'swapped' at initialization, depending upon the input parameter. Entry point A) includes the debugging code prologue and entry point B) excludes the prologue; thus eliminating all overhead except the initial 'test and swap' (even when test/debugging is selected, when the overhead is simply the test/debugging code itself).

Genetic Algorithm

In the world of performance related algorithms it is worth mentioning the role of genetic algorithms which compete using similar methods to the natural world in eliminating inferior algorithms in favour of more efficient versions.

Object Code Optimizers

Some proprietary programme optimizers such as the "COBOL Optimizer" developed by Capex Corporation in the mid 1970's for COBOL, actually took the unusual step of optimizing the Object code (or binary file) after normal compilation. This type of optimizer, recently sometimes referred to as a "post pass" optimizer or peephole optimizer, depended, in this case, upon knowledge of 'weaknesses' in the standard IBM COBOL compiler and actually replaced

(or patched) sections of the object code with more efficient code. A number of other suppliers have recently adopted the same approach.

Alignment of Data

Most processors execute faster if certain data values are aligned on word, doubleword or page boundaries. If possible design/specify structures to satisfy appropriate alignments. This avoids exceptions.

Locality of Reference

To reduce Cache miss exceptions by providing good spatial locality of reference, specify 'high frequency'/volative working storage data within defined structure(s) so that they are also allocated from contiguous sections of memory (rather than possibly scattered over many pages). Group closely related data values also 'physically' close together within these structures. Consider the possibility of creating an 'artificial' structure to group some otherwise unrelated, but nevertheless frequently referenced, items together.

Choice of Instruction or Data Type

Particularly in an Assembly language (although also applicable to HLL statements), the choice of a particular 'instruction' or data type, can have a large impact on execution efficiency. In general, instructions that process variables such as signed or unsigned 16-bit or 32-bit integers are faster than those that process floating point or packed decimal. Modern processors are even capable of executing multiple 'fixed point' instructions in parallel with the simultaneous execution of a floating point instruction. If the largest integer to be encountered can be accommodated by

the 'faster' data type, defining the variables as that type will result in faster execution - since even a non-optimizing compiler will, in-effect, be 'forced' to choose appropriate instructions that will execute faster than would have been the case with data types associated with 'slower' instructions. Assembler programmers (and optimizing compiler writers) can then also benefit from the ability to perform certain common types of arithmetic for instance - division by 2, 4, 8 etc. by performing the very much faster binary shift right operations (in this case by 1, 2 or 3 bits).

If the choice of input data type is not under the control of the programmer, although prior conversion (outside of a loop for instance) to a faster data type carries some overhead, it can often be worthwhile if the variable is then to be used as a loop counter, especially if the count could be quite a high value or there are many input values to process. As mentioned above, choice of individual assembler instructions (or even sometimes just their order of execution) on particular machines can effect the efficiency of an algorithm. See Assembly Optimization Tips for one quite numerous arcane list of various technical (and sometimes non-intuitive) considerations for choice of assembly instructions on different processors that also discusses the merits of each case.

Sometimes microcode or hardware quirks can result in unexpected performance differences between processors that assembler programmers can actively code for - or else specifically avoid if penalties result - something even the best optimizing compiler may not be designed to handle.

Data Granularity

The greater the granularity of data definitions (such as splitting a geographic address into separate street/city/ postal code fields) can have performance overhead implications during processing. Higher granularity in this example implies more procedure calls in Object-oriented programming and parallel computing environments since the additional objects are accessed via multiple method calls rather than perhaps one.

Subroutine Granularity

For structured programming and procedural programming in general, great emphasis is placed on designing programmes as a hierarchy of (or at least a set of) subroutines. For object oriented programming, the method call (a subroutine call) is the standard method of testing and setting all values in objects and so increasing data granularity consequently causes increased use of subroutines. The greater the granularity of subroutine usage, the larger the proportion of processing time devoted to the mechanism of the subroutine linkages themselves.The presence of a (called) subroutine in a programme contributes nothing extra to the functionality of the programme. The extent to which subroutines (and their consequent memory requirements) influences the overall performance of the complete algorithm depends to a large extent on the actual implementation. In assembly language programmes, the invocation of a subroutine need not involve much overhead, typically adding just a couple of machine instructions to the normal instruction path length, each one altering the control

flow either *to* the subroutine or returning *from* it (saving the state on a stack being optional, depending on the complexity of the subroutine and its requirement to reuse general purpose registers). In many cases, small subroutines that perform frequently used data transformations using 'general purpose' work areas can be accomplished without the need to save or restore any registers, including the return register.

By contrast, HLL programmes typically always invoke a 'standard' procedure call (the *calling convention*), which involves saving the programme state by default and usually allocating additional memory on the stack to save all registers and other relevant state data (the prologue and epilogue code). Recursion in a HLL programme can consequently consume significant overhead in both memory and execution time managing the stack to the required depth. Guy Steele pointed out in a 1977 paper that a *well-designed* programming language implementation *can* have very low overheads for procedural abstraction (but laments, in most implementations, that they seldom achieve this in practice - being "rather thoughtless or careless in this regard"). Steele concludes that "we should have a healthy respect for procedure calls" (because they are powerful) but he also cautioned "use them sparingly" See section Avoiding costs for discussion of how inlining subroutines can be used to improve performance. For the Java language, use of the "final" keyword can be used to force method inlining (resulting in elimination of the method call, no dynamic dispatch and the possibility to constant-fold the value - with no code executed at runtime)

Choice of Language / Mixed Languages

Some computer languages can execute algorithms more efficiently than others. As stated already, interpreted languages often perform less efficiently than compiled languages. In addition, where possible, 'high-use', and time-dependent sections of code may be written in a language such as assembler that can usually execute faster and make better use of resources on a particular platform than the equivalent HLL code on the same platform. This section of code can either be statically called or dynamically invoked (external function) or embedded within the higher level code (e.g. Assembler instructions embedded in a 'C' language program). The effect of higher levels of abstraction when using a HLL has been described as the *Abstraction penalty* Programmers who are familiar with assembler language (in addition to their chosen HLL) and are also familiar with the code that will be generated by the HLL, under known conditions, can sometimes use HLL language primitives of that language to generate code almost identical to assembler language. This is most likely to be possible only in languages that support pointers such as PL/1 or C. This is facilitated if the chosen HLL compiler provides an optional assembler listing as part of its printout so that various alternatives can be explored without also needing specialist knowledge of the compiler internals.

Software Validation Versus Hardware Validation

An optimization technique that was frequently taken advantage of on 'legacy' platforms was that of allowing the hardware (or microcode) to perform validation on numeric data fields such as those coded in (or converted to) packed

decimal (or packed BCD). The choice was to either spend processing time checking each field for a valid numeric content in the particular internal representation chosen or simply assume the data was correct and let the hardware detect the error upon execution. The choice was highly significant because to check for validity on multiple fields (for sometimes many millions of input records), it could occupy valuable computer resources. Since input data fields were in any case frequently built from the output of earlier computer processing, the actual probability of a field containing invalid data was exceedingly low and usually the result of some 'corruption'. The solution was to incorporate an 'event handler' for the hardware detected condition ('data exception)' that would intercept the occasional errant data field and either 'report, correct and continue' or, more usually, abort the run with a core dump to try to determine the reason for the bad data.

Similar event handlers are frequently utilized in today's web based applications to handle other exceptional conditions but repeatedly parsing data input, to ensure its validity before execution, has nevertheless become much more commonplace - partly because processors have become faster (and the perceived need for efficiency in this area less significant) but, predominantly - because data structures have become less 'formalized' (e.g. .csv and .tsv files) or uniquely identifiable (e.g. packed decimal). The potential savings using this type of technique may have therefore fallen into general dis-use as a consequence and therefore repeated data validations and repeated data conversions have become an accepted overhead. Ironically, one

consequence of this move to less formalized data structures is that a corruption of say, a numeric binary integer value, will not be detected at all by the hardware upon execution (for instance: is an ASCII hexadecimal value '20202020' a valid signed or unsigned binary value - or simply a string of blanks that has corrupted it?)

Algorithms for Vector & Superscalar Processors

Algorithms for vector processors are usually different than those for scalar processors since they can process multiple instructions and/or multiple data elements in parallel.

The process of converting an algorithm from a scalar to a vector process is known as vectorization and methods for automatically performing this transformation as efficiently as possible are constantly sought. There are intrinsic limitations for implementing instruction level parallelism in Superscalar processors but, in essence, the overhead in deciding for certain if particular instruction sequences can be processed in parallel can sometimes exceed the efficiency gain in so doing. The achievable reduction is governed primarily by the (somewhat obvious) law known as Amdahl's law, that essentially states that the improvement from parallel processing is determined by its slowest sequential component. Algorithms designed for this class of processor therefore require more care if they are not to unwittingly disrupt the potential gains.

Avoiding Costs

- Defining variables as integers for indexed arrays instead of floating point will result in faster execution.

- Defining structures whose structure length is a multiple of a power of 2 (2,4,8,16 etc.), will allow the compiler to calculate array indexes by shifting a binary index by 1, 2 or more bits to the left, instead of using a multiply instruction will result in faster execution. Adding an otherwise redundant short filler variable to 'pad out' the length of a structure element to say 8 bytes when otherwise it would have been 6 or 7 bytes may reduce overall processing time by a worthwhile amount for very large arrays. See for generated code differences for C as for example.
- Storage defined in terms of bits, when bytes would suffice, may inadvertently involve extremely long path lengths involving bitwise operations instead of more efficient single instruction 'multiple byte' copy instructions. (This does not apply to 'genuine' intentional bitwise operations - used for example instead of multiplication or division by powers of 2 or for TRUE/FALSE flags.)
- Unnecessary use of allocated dynamic storage when static storage would suffice, can increase the processing overhead substantially - both increasing memory requirements and the associated allocation/ deallocation path length overheads for each function call.
- Excessive use of function calls for very simple functions, rather than in-line statements, can also add substantially to instruction path lengths and stack/unstack overheads. For particularly time critical systems that are not also code size sensitive,

automatic or manual inline expansion can reduce path length by eliminating all the instructions that call the function and return from it. (A conceptually similar method, loop unrolling, eliminates the instructions required to set up and terminate a loop by, instead; repeating the instructions inside the loop multiple times. This of course eliminates the branch back instruction but may also increase the size of the binary file or, in the case of JIT built code, dynamic memory. Also, care must be taken with this method, that re-calculating addresses for each statement within an unwound indexed loop is not more expensive than incrementing pointers within the former loop would have been. If absolute indexes are used in the generated (or manually created) unwound code, rather than variables, the code created may actually be able to avoid generated pointer arithmetic instructions altogether, using offsets instead).

Memory Management

Whenever memory is *automatically* allocated (for example in HLL programmes, when calling a procedure or when issuing a system call), it is normally released (or 'freed'/ 'deallocated'/ 'deleted') automatically when it is no longer required - thus allowing it to be re-used for another purpose *immediately*. Some memory management can easily be accomplished by the compiler, as in this example. However, when memory is *explicitly* allocated (for example in OOP when "new" is specified for an object), releasing the memory is often left to an asynchronous 'garbage collector' which

does not necessarily release the memory at the earliest opportunity (as well as consuming some additional CPU resources deciding if it can be). The current trend nevertheless appears to be towards taking full advantage of this fully automated method, despite the tradeoff in efficiency - because it is claimed that it makes programming easier. Some functional languages are known as 'lazy functional languages' because of the significant use of garbage collection and can consume much more memory as a result.

- Array processing may simplify programming but use of separate statements to sum different elements of the same array(s) may produce code that is not easily optimized and that requires multiple passes of the arrays that might otherwise have been processed in a single pass. It may also duplicate data if array slicing is used, leading to increased memory usage and copying overhead.
- In OOP, if an object is known to be immutable, it can be copied simply by making a copy of a reference to it instead of copying the entire object. Because a reference (typically only the size of a pointer) is usually much smaller than the object itself, this results in memory savings and a boost in execution speed.

Readability, Trade Offs and Trends

One must be careful, in the pursuit of good coding style, not to over-emphasize efficiency. Frequently, a clean, readable and 'usable' design is much more important than a fast, efficient design that is hard to understand. There

are exceptions to this 'rule' (such as embedded systems, where space is tight, and processing power minimal) but these are rarer than one might expect.

However, increasingly, for many 'time critical' applications such as air line reservation systems, point-of-sale applications, ATMs (cash-point machines), Airline Guidance systems, Collision avoidance systems and numerous modern web based applications - operating in a real-time environment where speed of response is fundamental - there is little alternative.

Determining if Optimization is Worthwhile

The essential criteria for using optimized code are of course dependent upon the expected use of the algorithm. If it is a new algorithm and is going to be in use for many years and speed is relevant, it is worth spending some time designing the code to be as efficient as possible from the outset. If an existing algorithm is proving to be too slow or memory is becoming an issue, clearly something must be done to improve it. For the average application, or for one-off applications, avoiding inefficient coding techniques and encouraging the compiler to optimize where possible may be sufficient. One simple way (at least for mathematicians) to determine whether an optimization is worthwhile is as follows: Let the original time and space requirements (generally in Big-O notation) of the algorithm be O_1 and O_2. Let the new code require N_1 and N_2 time and space respectively. If $N_1N_2 < O_1O_2$, the optimization should be carried out. However, as mentioned above, this may not always be true.

Implications for Algorithmic Efficiency

A recent report, published in December 2007, from Global Action Plan, a UK-based environmental organization found that computer servers are "at least as great a threat to the climate as SUVs or the global aviation industry" drawing attention to the carbon footprint of the IT industry in the UK. According to an Environmental Research Letters report published in September 2008, "Total power used by information technology equipment in data centers represented about 0.5% of world electricity consumption in 2005. When cooling and auxiliary infrastructure are included, that figure is about 1%. The total data center power demand in 2005 is equivalent (in capacity terms) to about seventeen 1000 MW power plants for the world." Some media reports claim that performing two Google searches from a desktop computer can generate about the same amount of carbon dioxide as boiling a kettle for a cup of tea, according to new research; however, the factual accuracy of this comparison is disputed, and the author of the study in question asserts that the two-searches-tea-kettle statistic is a misreading of his work.

Greentouch, a recently established consortium of leading Information and Communications Technology (ICT) industry, academic and non-governmental research experts, has set itself the mission of reducing reduce energy consumption per user by a factor of 1000 from current levels. "A thousand-fold reduction is roughly equivalent to being able to power the world's communications networks, including the Internet, for three years using the same amount of energy that it currently takes to run them for a single day". The first

meeting in February 2010 will establish the organization's five-year plan, first year deliverables and member roles and responsibilities. Intellectual property issues will be addressed and defined in the forum's initial planning meetings. The conditions for research and the results of that research will be high priority for discussion in the initial phase of the research forum's development. Computers having become increasingly more powerful over the past few decades, emphasis was on a 'brute force' mentality. This may have to be reconsidered in the light of these reports and more effort placed in future on reducing carbon footprints through optimization. It is a timely reminder that algorithmic efficiency is just another aspect of the more general thermodynamic efficiency. The genuine economic benefits of an optimized algorithm are, in any case, that more processing can be done for the same cost or that useful results can be shown in a more timely manner and ultimately, acted upon sooner.

Criticism of the Current State of Programming

- David May FRS a British computer scientist and currently Professor of Computer Science at University of Bristol and founder and CTO of XMOS Semiconductor, believes one of the problems is that there is a reliance on Moore's law to solve inefficiencies. He has advanced an 'alternative' to Moore's law (May's law) stated as follows:

Software efficiency halves every 18 months, compensating Moore's Law. In ubiquitous systems, halving the instructions executed can double the battery life and big data sets bring

big opportunities for better software and algorithms: Reducing the number of operations from N x N to N x log(N) has a dramatic effect when N is large... for N = 30 billion, this change is as good as 50 years of technology improvements

- Software author Adam N. Rosenburg in his blog "*The failure of the Digital computer*", has described the current state of programming as nearing the "Software event horizon", (alluding to the fictitious "*shoe event horizon*" described by Douglas Adams in his *Hitchhiker's Guide to the Galaxy* book). He estimates there has been a 70 dB factor loss of productivity or "99.99999 percent, of its ability to deliver the goods", since the 1980s - "When Arthur C. Clarke compared the reality of computing in 2001 to the computer HAL in his book 2001: A Space Odyssey, he pointed out how wonderfully small and powerful computers were but how disappointing computer programming had become".
- Conrad Weisert gives examples, some of which were published in ACM SIGPLAN (Special Interest Group on Programming Languages) Notices, December, 1995 in: "Atrocious Programming Thrives"

3

CPU/Process Scheduling

The assignment of physical processors to processes allows processors to accomplish work. The problem of determining when processors should be assigned and to which processes is called processor scheduling or CPU scheduling.

When more than one process is runable, the operating system must decide which one first. The part of the operating system concerned with this decision is called the scheduler, and algorithm it uses is called the scheduling algorithm.

Goals of Scheduling (Objectives)

Many objectives must be considered in the design of a scheduling discipline. In particular, a scheduler should consider fairness, efficiency, response time, turnaround time, throughput, *etc.,* Some of these goals depends on the system one is using for example batch system, interactive system or real-time system, *etc.* but there are also some goals that are desirable in all systems.

Goals

Fairness

Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU and no process can suffer indefinite postponement.

Note that giving equivalent or equal time is not fair. Think of *safety control* and *payroll* at a nuclear plant.

Policy Enforcement

The scheduler has to make sure that system's policy is enforced.

For example, if the local policy is safety then the *safety control processes*must be able to run whenever they want to, even if it means delay in *payroll processes*.

Efficiency

Scheduler should keep the system (or in particular CPU) busy cent percent of the time when possible. If the CPU and all the Input/Output devices can be kept running all the time, more work gets done per second than if some components are idle.

Response Time

A scheduler should minimize the response time for interactive user.

Turnaround

A scheduler should minimize the time batch users must wait for an output.

Throughput

A scheduler should maximize the number of jobs processed per unit time.

A little thought will show that some of these goals are contradictory.

It can be shown that any scheduling algorithm that favors some class of jobs hurts another class of jobs. The amount of CPU time available is finite, after all.

Preemptive Vs. Non-preemptive Scheduling

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

Nonpreemptive Scheduling

A scheduling discipline is non-preemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.

Following are some characteristics of non-preemptive scheduling

1. In non-preemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
2. In non-preemptive system, response times are more predictable because incoming high priority jobs can not displace waiting jobs.
3. In non-preemptive scheduling, a schedular executes jobs in the following two situations.
   - When a process switches from running state to the waiting state.
   - When a process terminates.

Preemptive Scheduling

A scheduling discipline is preemptive if, once a process has been given the CPU can taken away. The strategy of allowing processes that are logically runable to.

Inter Processes Communication

Caveat

The material in this topic uses the *os.fork()* function which is not available under Microsoft Windows. (One reason is that Windows, compared to Unix, is very slow to create new processes so that in practice you would never use fork() even if you could!

There is another solution to this problem - threading - that works on both Unix and Windows which we will cover in a later topic.) If you are using Windows I recommend that you read the concept sections at the top because the principles will be used in the next topic. But there is no point in typing in the examples as they will not work. If you are using any form of Unix, including MacOS X then it should all work as advertised.

What Does Inter Processes Communication Mean

As the name suggests *Inter-Process communications* or IPC, is the mechanism whereby one process can communicate, that is, exchange data, with another process. You will recall if you read the Using the Operating System topic that a process is an executing programme.

Also that one process can communicate to another using various means such as os.popen or the subprocess module's

features. While these techniques are very useful for communicating with other programmes they do not provide the fine grained control that's sometimes needed for larger scale applications.

In these applications it is quite common for several processes to be used, each performing a dedicated task and other processes requesting services from them.

As an example a web server might have one process for listening to web requests from browsers and serving up simple HTML but use another process for serving more complex data queries, and possibly yet another process for handling ftp requests and the like.

Each process is tuned to perform one job well. In addition this *architecture* allows the administrator to share the processing load over several processes so that if there are a lot of ftp requests a second ftp process can be started and the ftp requests distributed between the two processes.

Do I Really Care

While you might not expect to be writing such large scale applications it can still be a useful technique and the principles involved are important in many of the later topics we will discuss such as network programming.

Even quite small applications can benefit from this approach particularly where there could be many users accessing a common resource, such as a database. Also if you are reading data from various communications ports it is useful to have a process per network connection so that blockages in one network do not prevent the other networks from being read.

Clint/Server

You may have heard the term *client/server* being bandied around and this is a very common software architecture for business applications. Client/Server refers to every specific type of IPC arrangement whereby one process, the client, makes requests of another process, the server. The server never requests anything from the client. There can be many client processes accessing a single server. Servers can in turn be clients to other servers - this is known as N-Tier Client/Server computing.

So Client/Server computing always uses IPC but IPC is not always client/server based. It is quite possible to have a network of processes each sending messages to the others without and rigid demarcation between clients and servers, this is often called *peer to peer* computing, While it may sound like an attractive model it turns out to be quite difficult to manage beyond a very small number of processes and the more rigid client/server approach is generally easier to design and operate.

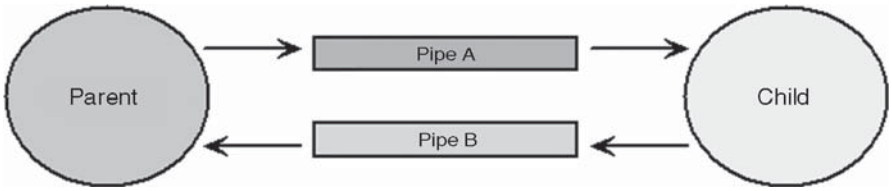
You may have noticed a similarity in terminology to Object Oriented Programming with the use of messages between processes being similar to the sending of messages between objects. It turns out that there is indeed a lot of synergy between an object model and an IPC model and some IPC architectures have developed that capitalize on these synergies. One such architecture is the *Common Object Request Broker Architecture* or CORBA. In this architecture we can register objects with a central *Object Request Broker* or *ORB* and messages sent to the object from anywhere in the architecture get routed to whichever process registered

the object. We will not be looking at CORBA in this tutorial but there are Python implementations of ORBs that you can investigate if you are interested. We're nearly ready to start writing some code. There are several different mechanisms for communicating between processes and we will consider two of them. The first IPC mechanism we are going to look at uses a mechanism called a *pipe* to exchange data between the two processes.

What is a Pipe

Conceptually a pipe can be thought of much like a hose-pipe, in that it is a conduit where we pour data in at one end and it flows out at the other. A pipe looks a lot like a file in that it is treated as a sequential data stream. Unlike a file, a pipe has two ends, so when we create a pipe we get two end points back in return. We can write to one end point and read from the other. Also unlike a file, there is no physical storage of data when we close a pipe, anything that was written in one end but not read out from the other end will be lost.

We can illustrate the use of pipes as conduits between processes in the following diagram:



Here we see two processes which I've called Parent and Child, for reasons that will become apparent shortly. The Parent can write to Pipe A. and read from Pipe B. The Child can read from Pipe A and write to Pipe B.

Note that each pipe can be used to send a request or return data depending upon which process is initiating the *transaction*.

OK, enough theory, let's roll up our sleeves, write some code and see how we can build an IPC mechanism using Python. The general idea is that we will create a parent process which will open two pipes. We then *fork* or *spawn* off a clone of the process, the *child*, which is an exact copy of the parent process, including the same two pipes. We can then use the pipes to communicate between parent and child.

The first thing to do is see how we use pipes to send and receive data. We create a pipe using the os.pipe() function which returns two file descriptors, one for each end of the pipe. We can then use the os.read/write functions to send data along the pipe.:

```
import os
# create the pipe
receive, transmit = os.pipe()
data = 'Here is a data string'
length = os.write(transmit, data)
# 1024 size buffer to ensure all data received
print 'The pipe contains:', os.read(receive, 1024)
```

Of course this is all happening within a single process so its not really very useful. But once we clone our process we can separate the reading and writing code and really start to achieve something. So how do we spawn our clone process?

Spawning Process

The mechanism used for spawning a child process is to use the os.fork() system call. This returns different values

depending on whether you are in the original, parent process or in the new child process. In the original process the returned value is the process ID or *pid* of the child process. If you are in the child process then the return value from fork is zero.

This means that in the code we will have an if statement that tests the fork() return value and if it is zero performs the child functions and if non zero does the parent function. To keep things manageable it is usually best to put those functions into separate modules and call the functions as required. In our example here we won't do that since the code is short and a single listing will suffice. (Again I stress that Microsoft Windows does not support the fork() function so the code will not work under Windows. Windows users will need to wait till the next topic to find out how to write client-server programmes, sorry, but complain to Microsoft not me!)

What we are going to do is create a child process that can perform a simple text formatting operation for us, it will return the value that we pass to it prefixed and postfixed with the phrase 'Ni'.

Here we go:

```
import os,signal

# create pipes

ServerReceive,ClientSend = os.pipe()

ClientReceive, ServerSend = os.pipe()

pid = os.fork()

if pid == 0: # in child

while True: # serve forever

data = os.read(ServerReceive,1024)
```

```
data = 'Ni!\n' + data + '\nNi!'
os.write(ServerSend,data)
else: # in parent
data = ['The Knights who say Ni!',
 'Appear in the film "Monty Python and the Holy Grail" ']
for line in data:
os.write(ClientSend,line)
print os.read(ClientReceive,1024)
# now terminate the child process
os.kill(pid,signal.SIGTERM)
```

Note that we use the pid received from fork to terminate the child process. If we failed to do this the child would become a background or *daemon* process running silently, forever waiting for data to appear on its input pipe. The actual termination is done using the os.kill() function which, despite the name, is actually used to send any *signal* to any process. Signal SIGTERM is the terminate signal as defined in the signal module. The full list is platform dependent and definitively documented in the documentation for the libc C library for your platform, but it is more easily obtained by using

```
>>> dir (signal)
```

at the Python interactive prompt or, on a Unix system, by typing:

```
$ kill -l
```

at the operating system prompt (Note, that's an 'ell' not a one). In the latter case you also get the numeric value which can be used in kill() directly but at the risk of losing platform independence.

Let's review what we have done: So far we have created pipes, and transmitted data along them. We have spawned a

child process, sent data to it, manipulated the data in the child process and sent data back to the parent. Finally we terminated the child process. That's really all you need for basic IPC, the complexity of the processing is simply a matter of writing more complex functions for the child. So lets see a real example in action. It's time to revisit the address book again.

Address Book Using

Back in the files topic we built a version of our address book using a dictionary. Let's reuse that example but this time we will build a client/server version. Notice that the original code broke one of the good practice rules, with User Interface code in my helper functions. If we were to try to use this code we would get messages from the child process mixed up with messages from the parent. We need to tweak the code slightly so that we can turn it into a reusable module. The main thing is to remove any print statements from the functions and pass the data in as arguments. We also want to return a result from each function.

Once we have done that we can import the code and access the helper functions without executing the main() function. The functions that we will make available are therefore:

- ReadBook(filename)
- SaveBook(book, filename)
- AddEntry(book, name, data)
- RemoveEntry(book, name)
- FindEntry(book, name)

The modified code looks like this:

```
def readBook(filename='addbook.dat'):
import os
```

```
book = {}
if os.path.exists(filename):
store = file(filename,'r')
for line in store:
 name,entry = line.strip().split(':')
 book[name] = entry
else:
 store = file(filename,'w') # create new empty file
store.close()
return book
def saveBook(book, filename = "addbook.dat"):
store = file(filename,'w')
for name,entry in book.items():
 line = "%s:%s" % (name,entry)
 store.write(line + '\n')
store.close()
def addEntry(book, name, data):
book[name] = data
return 'Added entry for ' + name
def removeEntry(book, name):
del(book[name])
return 'Deleted entry for ' + name
def findEntry(book, name):
if name in book.keys():
result = "%s: %s" % (name, book[name])
else: result = "Sorry, no entry for: " + name
return result
```

Note that I've ignored the user interface functions because we don't need them here but you might want to try making the necessary modifications to allow it to still function as a

standalone programme as well as serve as a module for the client/server version as an exercise.

Once you've fixed up the code and got it working as a standalone programme once more (or just saved the code above as address_srv.py, which is what I've done), we can proceed to writing our client/server code.

The client/server code will comprise our standard structure of creating pipes and forking the process. In the child process we will read the incoming pipe and interpret the data as a command followed by the arguments supplied and then call the relevant database function. In the parent, or client process, we will present the user with a menu and depending on the choice, request any needed additional data before sending the combined data string to the child or server process. The client will then read back the response and present it to the user.

The main programme looks like this:

```
import os, signal, address_srv
fromClient,toServer = os.pipe()
fromServer,toClient = os.pipe()
pid = os.fork()
if pid == 0:
addresses = address_srv.readBook()
while True:
s = os.read(fromClient,1024)
cmd,data = s.split(':')
if cmd == "add":
 details = data.split(',')
 name = details[0]
 entry = ','.join(details)
```

```
 s = address_srv.addEntry(addresses, name, entry)
 address_srv.saveBook(addresses)
elif cmd == "rem":
 s = address_srv.removeEntry(addresses, data)
 address_srv.saveBook(addresses)
elif cmd == "fnd":
 s = address_srv.findEntry(addresses, data)
else: s = "ERROR: Unrecognized command: " + cmd
os.write(toClient,s)
else:
menu = '''
1) Add Entry
2) Delete Entry
3) Find Entry
4) Quit
'''
while True:
print menu
try: choice = int(raw_input('Choose an option '))
except: continue
if choice == 1:
 name = raw_input('Enter the name: ')
 num = raw_input('Enter the House number: ')
 street= raw_input('Enter the Street name: ')
 town = raw_input('Enter the Town: ')
 phone = raw_input('Enter the Phone number: ')
 data = "%s,%s,%s,%s,%s" % (name,num,street,town,phone)
 cmd = "add:%s" % data
elif choice == 2:
 name = raw_input('Enter the name: ')
```

```
 cmd = 'rem:%s' % name
elif choice == 3:
 name = raw_input('Enter the name: ')
 cmd = 'fnd:%s' % name
elif choice == 4:
 break
else:
 print "Invalid choice, must be between 1 and 4."
 continue

os.write(toServer, cmd)
      print "\nRESULT: ", os.read(fromServer,1024)

os.kill(pid, signal.SIGTERM)
```

Obviously we could tidy that up a bit more by using some functions for the if/elif chains but the example is small enough for that not to be necessary. A couple of points to note are the use of break to stop the loop and continue to go round to the top of the loop again. We didn't discuss these in the looping topic but hopefully their use is obvious and if not the documentation describes them in more detail. An obvious extension to this exercise would be to use the database version of the address book as the server instead of the dictionary version. I'll leave that as an exercise for the keen students among you. The big snag with this form of client/server programming is that the server can only talk to the client that started it. It would be much better if the server could be started first and then multiple clients attach to it. That's exactly what we will do at the end of the next topic where we introduce networking.

Process Scheduling

Using Operating System Scheduling on z/OS Systems

On z/OS operating system scheduling servers, SAS 9.2 supports the scheduling of flows that have a single time event as a trigger. The jobs in a flow are submitted to the Job Entry Subsystem (JES) for execution in an asynchronous manner.

How Operating System Scheduling Works on z/OS Systems

When you schedule a flow, Schedule Manager creates the following files:

- A REXX programme called RunFlow_*yyymmddhh mmssssss*.rex, which submits each batch job to run on the requested schedule.
- A file called *jobname*.status, which contains information about the status of the job.
- The following files, each of which invokes the REXX programme to log the job status.
- Logstart_*jobname*, which is executed as the first step of each job
- Logend_*jobname*, which is executed as the last step of the job if all previous steps have a return code of 0
- Logfail_*jobname*, which is executed as the last step of the job if any of the previous steps has a return code other than 0
- *Flowname*.cmd, which contains the statements that execute the programme for the flow. This file is passed to the at command to be executed when the time event trigger occurs.

- *Flowname*.log, which contains the log information for the flow.
- *Flowname*.status, which contains the current status of the flow.

The files are written to a subdirectory called *username*/*flowdefname* in the Control Directory that was specified in the operating system scheduling server definition, where *username* is the identity that scheduled the flow.

Manually Submitting a Flow for Execution on a z/OS Scheduling Server

Note: On z/OS, unexpected results can occur if you manually submit a flow without having selected Manually Schedule when you scheduled the flow. If you need to manually submit a flow that has been scheduled on a z/OS operating system scheduling server, follow these steps:

1. Change to the directory where the flow's files are located (/*outdir*/*username*/*flowdefname*), where *outdir* is the value of the Control Directory attribute in the operating system scheduling server definition.
2. At the USS prompt, type a command with the following syntax: RunFlow_*yyymmddhhmmssss*.rex <runonce|start> Specify runonce if you want to run the flow just once, or specify start if you want to resume a recurrence that was specified for the flow.

Canceling a Scheduled Flow on a z/OS Operating System Scheduling Server. If you need to cancel a flow that has been scheduled on a z/OS operating system scheduling server, follow these steps:

1. At the USS prompt, type at -l.

Note: Type a lowercase L, not the numeral 1.

A list of jobs appears, as in the following example:

1116622800.a Fri May 20 17:00:00 2005

1116734400.a Sun May 22 00:00:00 2005

1116820800.a Mon May 23 00:00:00 2005

1116620801.a Mon May 23 00:00:00 2005

1117029000.a Wed May 25 09:50:00 2005

2. For each job that you want to cancel, type the following command at the USS prompt: at -r *at-jobid*, where *jobid* is the first part of each line of the list of jobs (for example, at -r 1116622800.a).

4

Multiprocessor Scheduling

The multiprocessor scheduling problem can be described as the problem of fairly distributing a workload on a distributed computer system consisting of multiple processors with distributed local memories. Many researchers have worked on the problem of multiprocessor scheduling and various algorithms have been proposed. Proposed solution methods in the literature fall under two major categories: Heuristics and physical optimization algorithms. Heuristic procedures are mainly geometry based methods whereas physical optimization algorithms are derived from the natural sciences. In the case of heuristics, Hellstorn and Kanal (1992) proposed a solution for multiprocessor scheduling problem using an asymmetric mean-field neural network model.

In the case of physical optimization, Hou, Ansari and Ren (1993) proposed a genetic algorithm based solution and

Driessche and Piessens (1992) developed parallel versions of the genetic algorithm. Hwang and Xu (1993) proposed a simulated annealing algorithm for solving the multiprocessor scheduling problem. In this study, the behaviour of simulated annealing and genetic algorithms on the multiprocessor scheduling problem are examined under different situations. In addition, the modified-uniform simulated annealing algorithm (MSA), the enhanced simulated annealing algorithm (Enh_SA) (Loganantharaj, 1997) the multithread simulated annealing algorithm (Mul_SA), and multi-thread modified-uniform simulated annealing algorithm (Mul_MSA) are investigated. Simulation results are presented and compared.

Problem

The general problem of multiprocessor scheduling can be stated as scheduling a set of computational tasks onto a multiprocessor system so that a set of performance criteria will be optimized. The goal (or programme) assigned to the multicomputer is divided into several programme modules Mi.

The interrelationship among Mi can be specified by a programme graph after compilation. Each programme module, M*i* for *i*=1,2,É,m, corresponds to one node in the graph. The edge connections in the programme graph correspond to communication paths among programme modules. The module weight, W*i*, reflects the code segment length and size of the data set used in M*i*, giving an approximated estimate of the CPU cycles, needed to execute M*i*. The edge weight, e*ij*, reflects the expected communication cost between the two programme modules.

The assumptions made in this work are the following:

- The Multiprocessor is a distributed-memory message-passing multicomputer whose processors are connected by a static point-to-point interconnection network.
- The Routing used is a wormhole routing.
- The computation model used is a loosely synchronous computation model in which processors perform compute-communication cycles in SPMD (Single Programme, Multiple Data) scheme.
- The edges of the graph are bi-directional between any pair of modules Mi and Mj. *i.e.* e*ij* =e*ji*.
- Bi-directional links are assumed between all pairs of computer nodes in a multicomputer.
- The distance between any two computer nodes is defined as the number of hops between them.

Scheduling Methods

SA is a physical optimization technique which accepts a bad criteria in order to reach a global minimum (or optimal solution). SA starts with a randomly generated initial solution. In each iteration, the solution is randomly perturbed. If such a perturbation results in negative or zero change in the system energy, then it is accepted. To prevent premature convergence when trapped in a bad optimum, the simulated annealing accepts positive changes in energy with a certain temperature dependent probability. In order to apply the simulated annealing algorithm for solving the multiprocessor scheduling problem the following steps are followed:

- Give an initial value for the temperature T and assign a constant value for K (cooling schedule).

- Distribute the jobs (tasks) on the processors randomly.
- Evaluate the system weight, for this initial random generation, using the objective function OF so.
- Select two processors a and b, randomly (the selected processes a and b must be different). Then select (also randomly) two tasks, one from a and another one from b.
- Swap the selected tasks between the two processors a and b.
- Evaluate again the system weight, for this perturbation, using the objective function OFso.
- If the new system weight is smaller than the previous one, accept this new state and use it as a new starting point for the next perturbation.
- If not, generate a random number, RND, between 0 and 1. If RND< e[Previous system weight - new system weight]/T], accept the situation and use it as a new starting point for the next perturbation. If not, reject and keep the previous one as the starting point for the next perturbation.
- Repeat the steps starting from part 4 for n successive rejections (in this study n = 3).
- Change the temperature T as follows: T = T * K.
- Repeat the steps starting from part 4 for a new temperature until the temperature becomes less or equal to 1. The value of the cooling factor is problem-dependent and has to be determined experimentally.

Deadlock

A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only

another process in the set. In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process. None of the processes can run, none of them can release any resources, and none of them can be awakened. It is important to note that the number of processes and the number and kind of resources possessed and requested are unimportant. The resources may be either physical or logical. Examples of physical resources are Printers, Tape Drivers, Memory Space, and *CPU* Cycles. Examples of logical resources are Files, Semaphores, and Monitors.

The simplest example of deadlock is where process 1 has been allocated non-shareable resources *A*, say, a tap drive, and process 2 has be allocated non-sharable resource *B*, say, a printer. Now, if it turns out that process 1 needs resource *B* (printer) to proceed and process 2 needs resource *A* (the tape drive) to proceed and these are the only two processes in the system, each is blocked the other and all useful work in the system stops. This situation ifs termed deadlock. The system is in deadlock state because each process holds a resource being requested by the other process neither process is willing to release the resource it holds.

Preemptable and Nonpreemptable Resources

Resources come in two flavors: preemptable and nonpreemptable. A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource. On the other hand, a nonpreemptable resource is one that cannot be taken away from process (without causing ill effect). For example, *CD*

resources are not preemptable at an arbitrary moment. Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with.

Necessary and Sufficient Deadlock Conditions

Coffman identified four (4) conditions that must hold simultaneously for there to be a deadlock.

Mutual Exclusion Condition

The resources involved are non-shareable.

Explanation:

At least one resource (thread) must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

Hold and Wait Condition

Requesting process hold already, resources while waiting for requested resources.

Explanation:

There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

No-Preemptive Condition

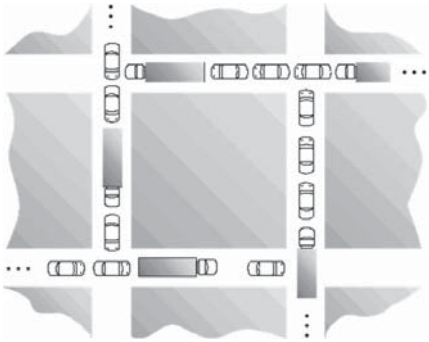
Resources already allocated to a process cannot be preempted.

Explanation:

Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.

Circular Wait Condition

The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list. As an example, consider the traffic deadlock in the following figure



Consider each section of the street as a resource.

1. Mutual exclusion condition applies, since only one vehicle can be on a section of the street at a time.
2. Hold-and-wait condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
3. No-preemptive condition applies, since a section of the street that is a section of the street that is occupied by a vehicle cannot be taken away from it.
4. Circular wait condition applies, since each vehicle is waiting on the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of street held by the next vehicle in the traffic.

The simple rule to avoid traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection. It is not possible to have a deadlock involving only one single process. The deadlock involves a circular "hold-and-wait" condition between two or more processes, so "one" process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread that is, each thread has access to the resources held by the process.

Dealing with Deadlock Problem

In general, there are four strategies of dealing with deadlock problem:

1. The Ostrich Approach

Just ignore the deadlock problem altogether.

2. Deadlock Detection and Recovery

Detect deadlock and, when it occurs, take steps to recover.

3. Deadlock Avoidance

Avoid deadlock by careful resource scheduling.

4. Deadlock Prevention

Prevent deadlock by resource scheduling so as to negate at least one of the four conditions.

Deadlock Prevention

Havender in his pioneering work showed that since all four of the conditions are necessary for deadlock to occur, it follows that deadlock might be prevented by denying any one of the conditions.

Elimination of "Mutual Exclusion" Condition

The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the tap drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

Elimination of "Hold and Wait" Condition

There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources.

This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks simply cannot occur. This strategy can lead to serious waste of resources. For example, a programme requiring ten tap drives must request and receive all ten derives before it begins executing. If the programme needs only one tap drive to begin execution and then does not need the remaining tap drives for several hours. Then substantial computer resources (9 tape drives) will sit idle for several

hours. This strategy can cause indefinite postponement (starvation). Since not all the required resources may become available at once.

Elimination of "No-preemption" Condition

The nonpreemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. Suppose a system does allow processes to hold resources while requesting additional resources. Consider what happens when a request cannot be satisfied. A process holds resources a second process may need in order to proceed while second process may hold the resources needed by the first process. This is a deadlock. This strategy require that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively.

High Cost: When a process release resources the process may lose all its work to that point. One serious consequence of this strategy is the possibility of indefinite postponement (starvation). A process might be held off indefinitely as it repeatedly requests and releases the same resources.

Elimination of "Circular Wait" Condition

The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and than forcing, all processes to request the resources in order (increasing or decreasing). This strategy impose a total

ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle. For example, provide a global numbering of all the resources, as shown.

1	≡	Card reader
2	≡	Printer
3	≡	Plotter
4	≡	Tape drive
5	≡	Card punch

Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a tape drive (order: 2, 4), but it may not request first a plotter and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

Deadlock Avoidance

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. This method differs from deadlock prevention, which guarantees that deadlock cannot occur by denying one of the necessary conditions of deadlock.

If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. Perhaps the most famous deadlock avoidance algorithm, due to Dijkstra, is the Banker's algorithm. So named because the process is analogous to that used by a banker in deciding if a loan can be safely made.

Banker's Algorithm

Customers	≡	Processes
Units	≡	Resources, say, tape drive
Banker	≡	Operating System

Customers	Used	Max	
A	0	6	Available Units = 10
B	0	5	
C	0	4	
D	0	7	

In the above figure, we see four customers each of whom has been granted a number of credit nits. The banker reserved only 10 units rather than 22 units to service them. At certain moment, the situation becomes

Customers	Used	Max	
A	1	6	Available Units = 2
B	1	5	
C	2	4	
D	4	7	

Safe State The key to a state being safe is that there is at least one way for all users to finish. In other analogy, the state of figure above is safe because with 2 units left, the banker can delay any request except *C's*, thus letting *C* finish and release all four resources. With four units in hand, the banker can let either *D* or *B* have the necessary units and so on.

Unsafe State Consider what would happen if a request from B for one more unit were granted in above figure.
We would have following situation

Customers	Used	Max	
A	1	6	Available Units = 1
B	2	5	
C	2	4	
D	4	7	

This is an unsafe state.

If all the customers namely *A, B, C,* and *D* asked for their maximum loans, then banker could not satisfy any of them and we would have a deadlock.

Important Note: It is important to note that an unsafe state does not imply the existence or even the eventual existence a deadlock. What an unsafe state does imply is simply that some unfortunate sequence of events might lead to a deadlock.

The Banker's algorithm is thus to consider each request as it occurs, and see if granting it leads to a safe state. If it does, the request is granted, otherwise, it postponed until later.

Haberman has shown that executing of the algorithm has complexity proportional to N^2 where N is the number of processes and since the algorithm is executed each time a resource request occurs, the overhead is significant.

Deadlock Detection

Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state a. Of course, the deadlock detection algorithm is only half of this strategy.

Once a deadlock is detected, there needs to be a way to recover several alternatives exists:

- Temporarily prevent resources from deadlocked processes.
- Back off a process to some check point allowing preemption of a needed resource and restarting the process at the checkpoint later.
- Successively kill processes until the system is deadlock free.

These methods are expensive in the sense that each iteration calls the detection algorithm until the system proves to be deadlock free. The complexity of algorithm is $O(N^2)$ where N is the number of proceeds. Another potential problem is starvation; same process killed repeatedly.

SYSTEM MODEL

A system consists of a finite number or resources to be distributed among a number of competing processes. The resources are partitioned into several types, each of which consists of some number of identical instances.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

- Request If the request cannot be granted immediately (for example, if the resource is a printer, the process can acquire the resource.
- Use: The process can operate on the resource (for, example, if the resource is a printer, the process can print on the printer).
- Release: The process releases the resource.

DEADLOCK CHARACTERIZATION

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

Mutual Exclusion

East one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource.

If another process requests that resource, the requesting process must be delayed until the resource has been released.

Hold and Wait

Tthere must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes. No preemption resources cannot be preempted; that is, the process holding it after that process has completed its task can release a resource only voluntarily by the process holding it, after that process has completed its task. Circular wait, there must exist a set {Po,P1,Pn} of waiting processes such that P0 is waiting for a resource that is held by P1,P1 is waiting for a resource that is held by P2,Pn-1 is waiting for a resource that held by Pn and Pn is waiting for a resource that is held by Po.

Necessary Conditions

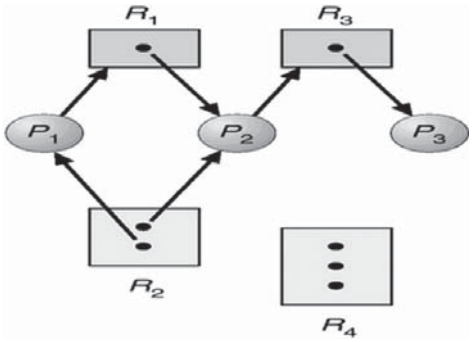
Mutual exclusion: At least one resource must be held in a non-sharable mode.

- Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by others
- No preemption: a resource can be released only voluntarily by the process holding it, after it has completed its task
- Circular wait: there exists a set {P0, P1, …, P0} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.

  Deadlock can arise if four conditions hold simultaneously Resource-Allocation Graph.

- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - P = {P1, P2, …, Pn} (All processes in the system)
  - R = {R1, R2, …, Rm} (All resources in the system).
- Two kinds of edges
  - Request edge Rj®– directed edge P1
  - Assignment edge Pi®– directed edge Rj continuation
- Process
- Resource Type with 4 instances
- Pi requests instance of Rj
- Pi is holding an instance of Rj

Example of a Resource Allocation Graph



Resource Allocation Graph With A Cycle But No Deadlock.

5

Unix File System

Overview

In Unix, the operating system comprises of files which are arranged in a tree structure. We call files as everything which includes:

- Programmes
- Directories
- Images
- Texts
- Services etc.

Sometimes a file with a special attribute, but a file nevertheless.

Introduction

File system in Unix is a service which supports an abstract representation of the secondary storage to the Operating System. It organizes data logically for random access by the

Operating System. In Unix, the virtual file system provides the interface between the data representation by kernel to the user process and the data presentation to the kernel in memory along with the file and directory system cache. Due to such performance disparity among disk and CPU/memory, file system performance is a paramount point for any Operating System. The file structure in Unix looks as shown in figure below.

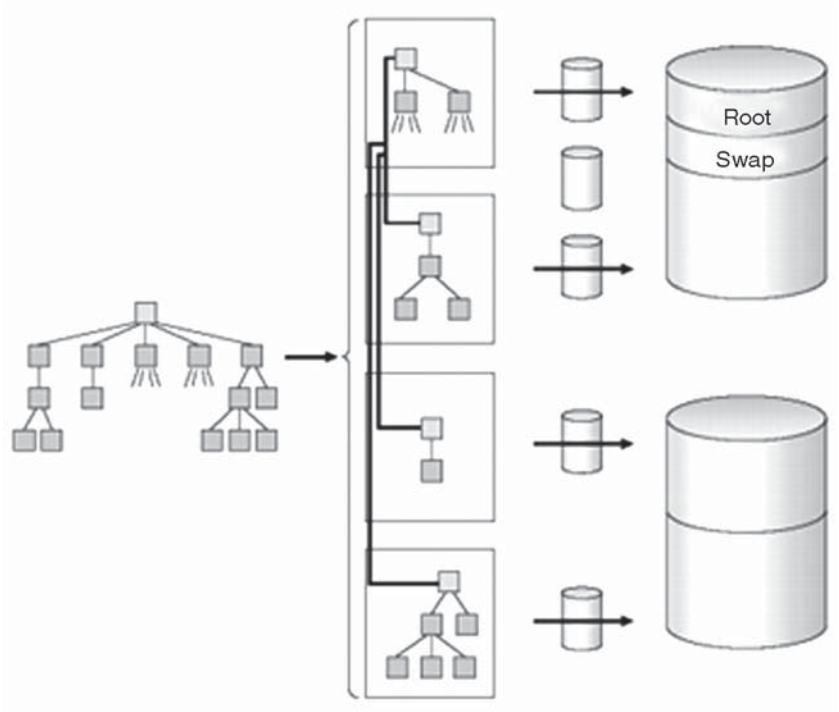


Fig. File structure in Unix.

Generally, UNIX File System has:

- File system is organized in tree structure.
- File tree can be arbitrarily deep.
- File name must NOT LONGER than 256 chars.
- Single path name must NOT LONGER than 1023 chars.

In Unix, the classical file system shown in fig comprises of:

- Sequentially from a predefined disk addresses (cylinder 0, sector 0):
  - Boot block (Master Boot Record)
  - Superblock
  - I-node hash-array
  - Data blocks
- Boot block, which is a hardware specific programme that automatically load UNIX at system startup time.
- Super block in Unix contains two lists:
  - A chain of free data block numbers
  - A chain of free i-node numbers.

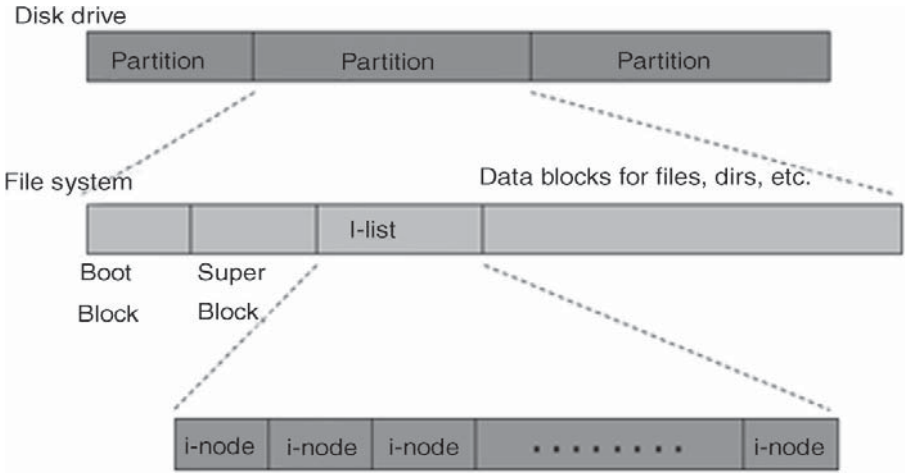


Fig. Classical File System.

File System in Unix carries storage devices. It is done through:

- Number of disks which is attached to the computer system
- Network disks by Storage Area Network
- Disks such as IDE and SATA which can access #surface, #track, #sector
- Smart disks such as SCSI, SAN, NAS which access #sector

– Sequential accessing.

In UNIX, there are certain special files called as directories which carry information about other files. We can say that a UNIX directory is a file whose data is an array or list of filename and i-node pairs which has:

– Owner, group owner, size, access permissions, *etc.*
– File operations on directories
– Directory in shape of an I-node type structure.
– Flag in the structure which shows its type.

Types of Files

Files in UNIX file system are arranged in multilevel hierarchy structure which is called as directory tree as shown in fig. The figure shows an arrangement of a family tree which shows how UNIX file system is organized. The structure looks like an inverted tree or root system of plant. In the figure, at the top, there is a single directory which is known as root shown by a/(slash). In this, all other files are descendents of root. The number of levels is random, though most UNIX systems share similar organizational features. It is observed that UNIX file system contains several different types of files such as.

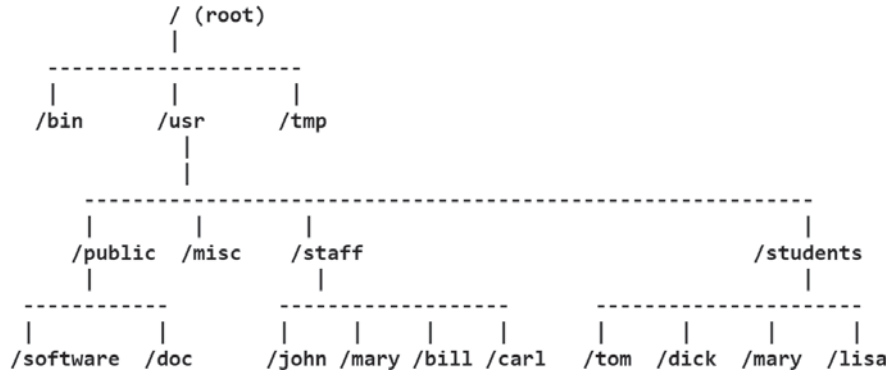


Fig. Example of Unix File System.

Ordinary Files

- Stores information such as text or image. This is the type of file that you usually work with.
- Always located within/under a directory file
- Do not contain other files.

Directories

- Branching points in the hierarchical tree
- Used to organize groups of files
- Contains ordinary files, special files or other directories
- Simply used for organizing files.
- In this, files are descendants of root directory, (named/) located at top of the tree.

Special Files

- It shows physical devices such as printer, tape drive or terminal for Input/Ouput operations
- In Unix, any device attached to system is considered as a file:
- As by default, a command treats your terminal as standard input file (stdin) for command input
- As your terminal is also treated as standard output file (stdout) for command output
- Carries two types of I/O command: character and block
- It is found under directories named/dev.

Pipes

– It is special type of file in UNIX which allows you to link your commands together.

– It is a temporary file which exists so as to keep data from one command till it is read by another.
– To pipe the output from one command into another command, we show:

```
who | wc –l
```

The above command will tell you how many users are presently logged into the system. Here the standard output from who command is a list of all users currently logged into the system. We see that this particular output is piped into the wc command assuming it as standard input. In this, -l option command shows the numbers of lines in standard input and gives the result on its standard output terminal.

The file structure in Unix comprises of three kinds of files:

- Byte sequence
- Record sequence
- Tree

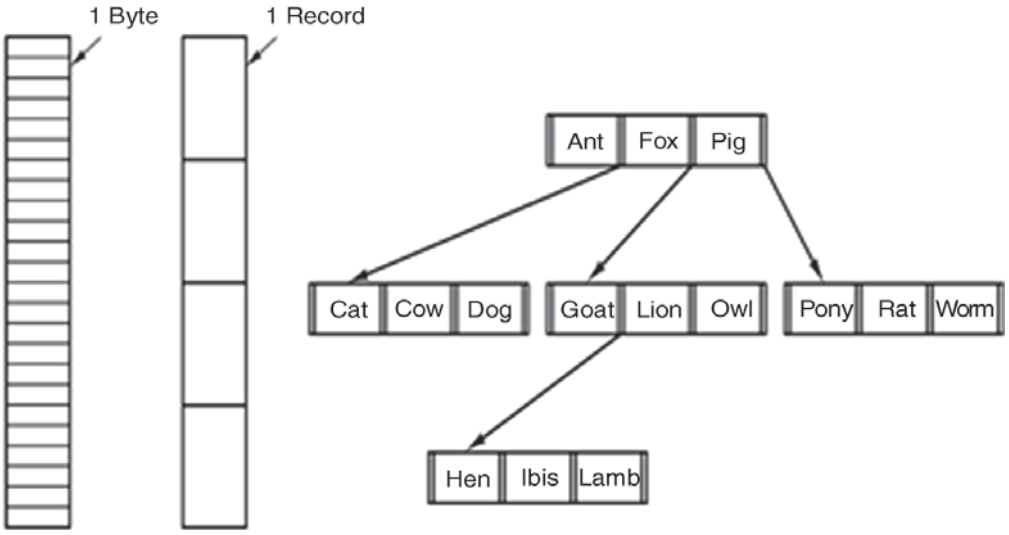


Fig. Structure of Unix File System.

The file type in Unix consist of

- Executable file
- Archive.

In Unix, files can be accessed through:

Sequential access, where:

- Reading of all bytes/records is done from starting
- It cannot jump around and could rewind or back up
- It is easy with mag tape medium.

Random access, where:

- Reading of bytes/records is done in any order
- Data base systems is required.

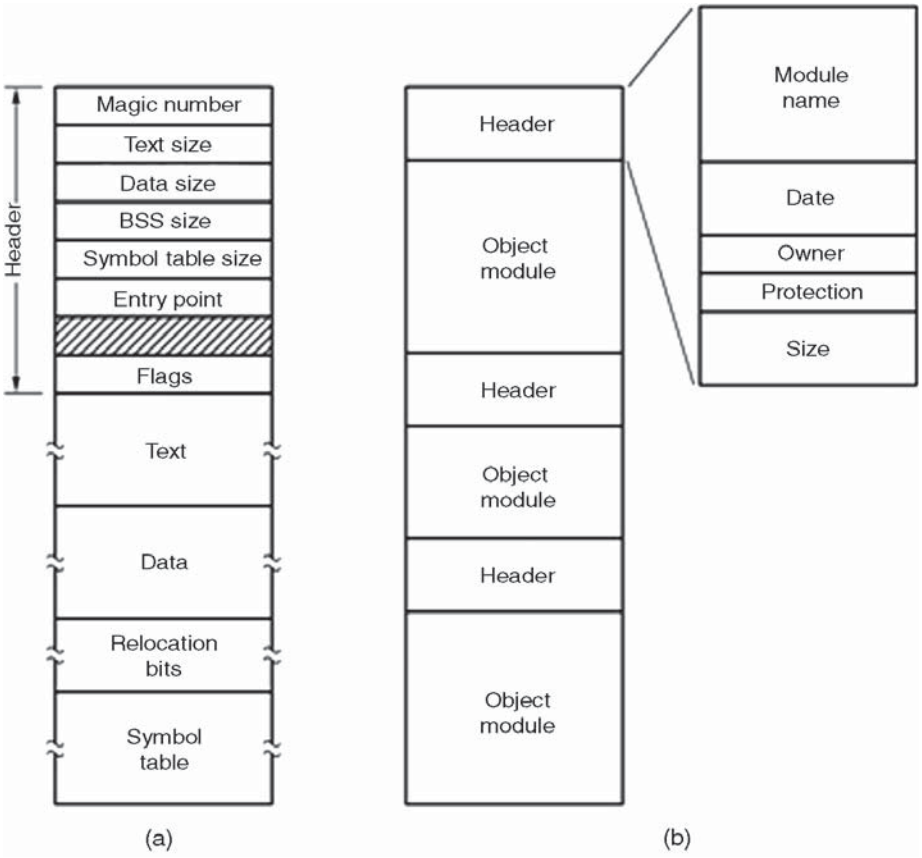


Fig. File type in Unix.

- Possibility of reading appears
- Initially read and then move file marker.

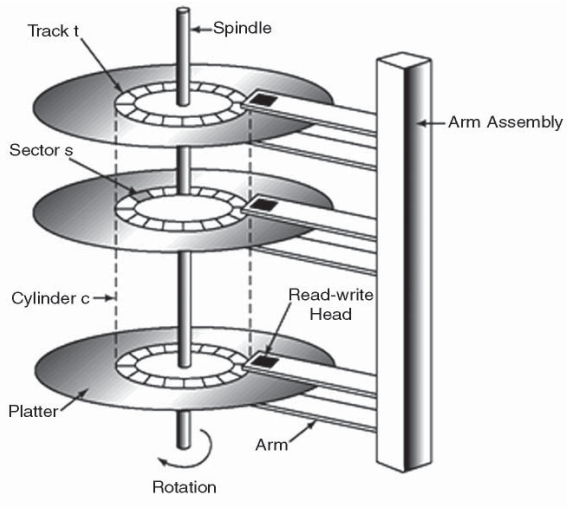


Fig. Disk Structure.

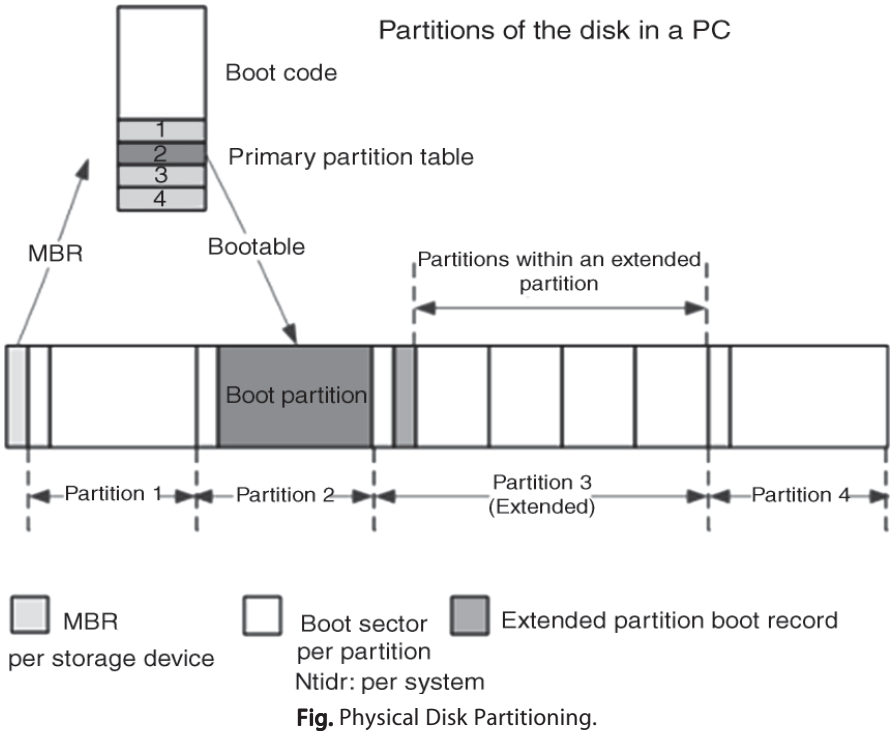


Fig. Physical Disk Partitioning.

Representation of Files

In Unix, all devices are shown by files called as special files which are present in/dev directory. So, device files and other files are named and accessed in same way.

In Unix:

- Regular file is an ordinary data file present in the disk.
- Block special file shows a device with characteristics similar to a disk.
- Character special file shows a device with characteristics same as keyboard.

Inode

In UNIX, all the files description is stored in a structure known as inode. It carries information about:

- File-size
- Location
- Time of last access
- Time of last modification
- Permission

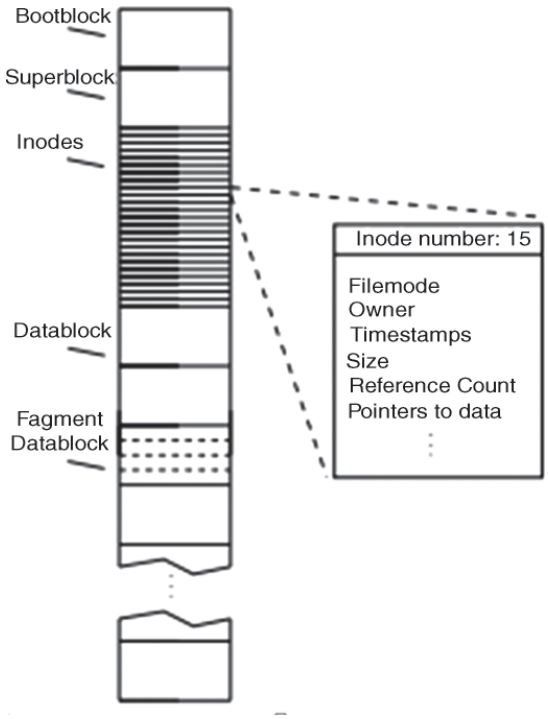


Fig. Representation of inode.

In this, directories are also shown as files. The descriptions about the file are done by inode pointers towards the data blocks of particular file as shown in figures.

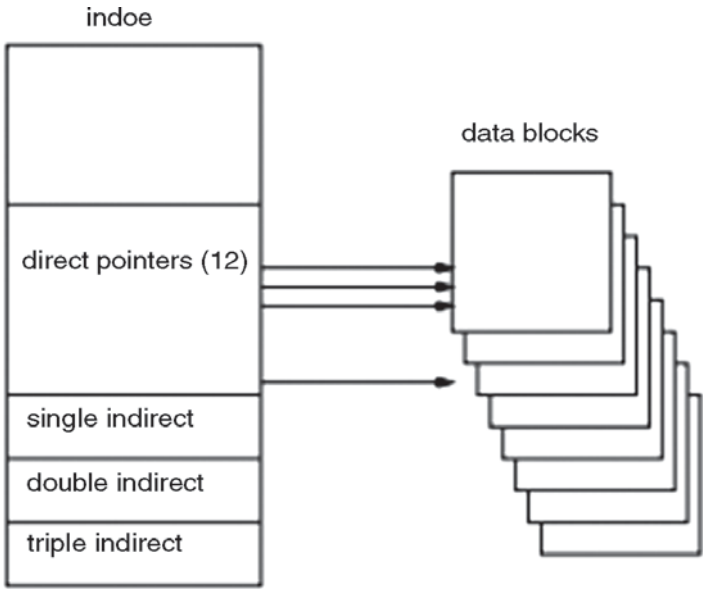


Fig. Direct Pointer.

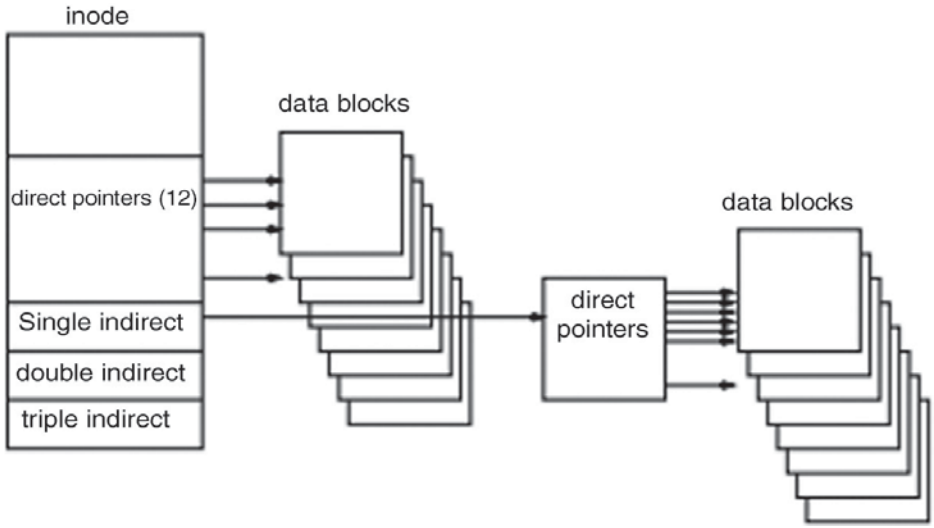


Fig. 2.Indirect pointer.

If the file is large, inode has indirect pointer to a block of pointers to additional data blocks. A data block carries file of size 8k.

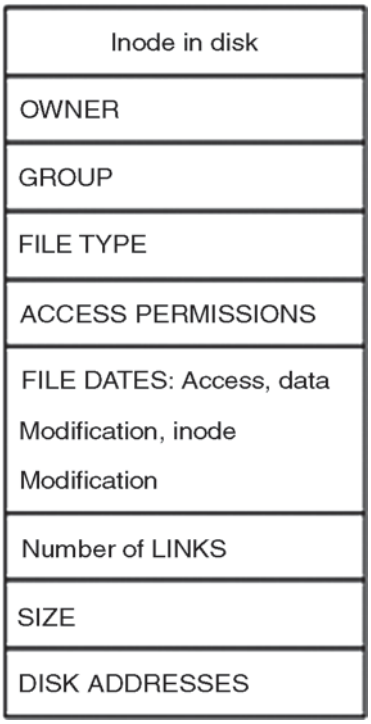


Fig. Representation of file information in inode.

Inode consists of the following fields as shown in fig:

- File owner identifier
- File type
- File access permissions
- File access times
- Number of links
- File size
- Location of the file data

Block Layout

In a block layout of inode, a file is associated:

- An inode of inode list
- Blocks of data area

These blocks serves as an information platforms for inode files. In the figure shown, using block of 1K and address of 4

bytes, the maximum size exist is 10K + 256K + 64M + 16G. It gives a slow access to heavy files.

Structure of Regular Files

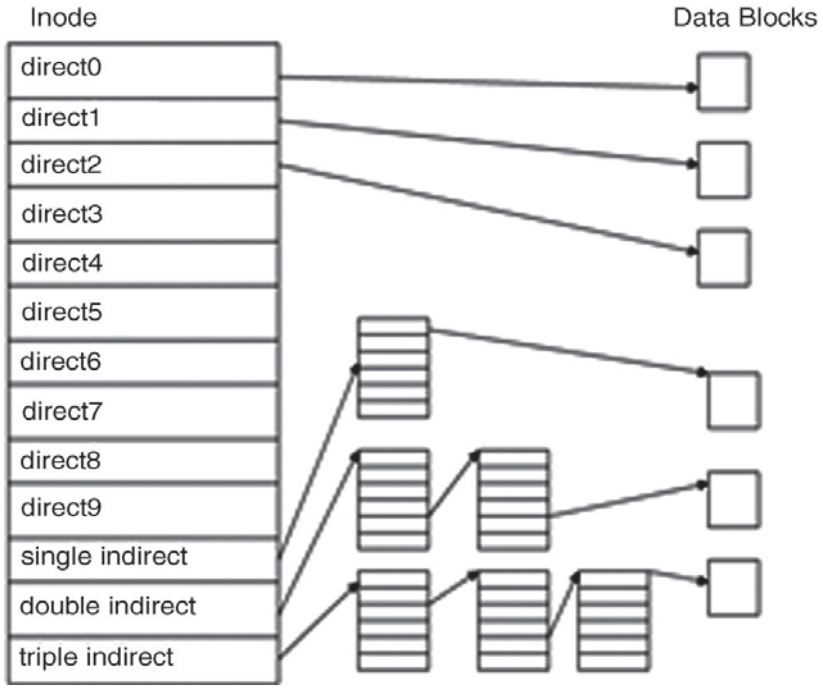


Fig. Regular file structure in Unix.

- Processes access data in a file by byte offset and view a file as a stream of bytes
- The kernel accesses the inode and converts the logical file block into the appropriate disk block
- Algorithm bmap
  - The kernel calculates logical block number in file from byte offset
  - The kernel calculates start byte in block for I/O
  - The kernel calculates number of bytes to copy to user

- The kernel checks if read-ahead is applicable, then marks inode
- The kernel determines level of indirection.

File Type and Permission

UNIX commands allow you to set permissions. File by file, allowing you to control who can read a file, write to a file or view a file on a Web page. Files uploaded to ones Unix account are automatically owned by him. Unless he give permission for other group members to edit or change a file, they cannot make modifications.

Every file or folder in UNIX has access permissions. There are three types of permissions:

- Read access
- Write access
- Execute access.

Permissions are defined for three types of users:

- The owner of the file
- The group that the owner belongs to
- Other users

Example: -rwxr—r—

0123456789

In above example:

- Symbol in 0 position is similar to symbol in-. It is either “d” if item is a directory or “l” if it is a link or “-” if the item is a regular file.
- Symbols in positions from 1 to 3 are rwx which has permission for owner of file.
- Symbols in positions from 4 to 6 shows r— having permissions for group.

- Symbols in positions from 7 to 9 shows r— having permissions for others.

r = Read access is allowed

w= Write access is allowed

x = Execute access is allowed

- = Replaces "r", "w" or "x" if according access type is denied.

Access permissions for files and folders mean different things from the user standpoint.

It is seen that every file in UNIX has following attributes:

- *Owner permissions*: The owner's permissions determine what actions the owner of the file can perform on the file.
- *Group permissions*: The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- *Other (world) permissions*: The permissions for others indicate what action all other users can perform on the file.

Directories

A directory is a file which consists of a number of records that contains the following fields:

- A pointer to the next record
- A number identifying an inode
- A number identifying the length of record
- A string that contains name of record which is commonly called as filename. (possibly some padding).

Directory in Unix file system is a sequence of lines or entries of variable length where each line contains an i-node number and a file name mapping: <filename, inode #>. In this, the directory data is stored as binary. Some earlier versions of UNIXs allow: od -c dir-name. Since directories are files, but

UNIX give the permission – rwx- which explains:

- r, lists directory contents
- w, add a file to the directory
- x, cd to the directory

In Unix, there exist: single level and two level directory system as shown:

Single level

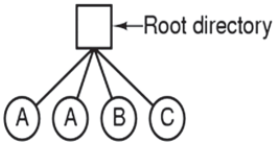


Fig. Single level directory

In the single level directory system, there are 4 files which are owned by 4 users A,B,C and D.

Two level

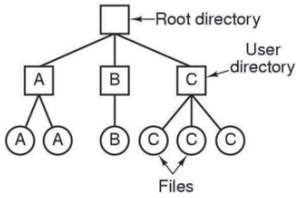


Fig. Two level directory system.

The figure shows a two level directory system in which, letters represents the owners of files and directories. Figure below shows a consolidated hierarchical directory systems.

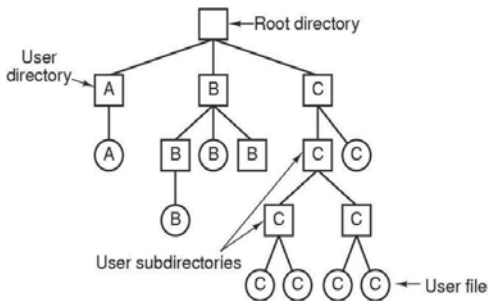


Fig. Hierarchical Directory Systems.

Working with Files and Directories

Creating a File

To create a file in Unix, use open system call with some arguments and tell it to create a file. While doing this, a free inode is found and is initialized. In this, an entry is created in the current directory which points to the inode. We see that the file was initially empty and carries no data blocks.

Creating a Directory

To create a directory in Unix, use either shell command mkdir or system call with same name. We see when a directory is created:

- A file is created
- An inode is allocated
- Is identified as a directory.

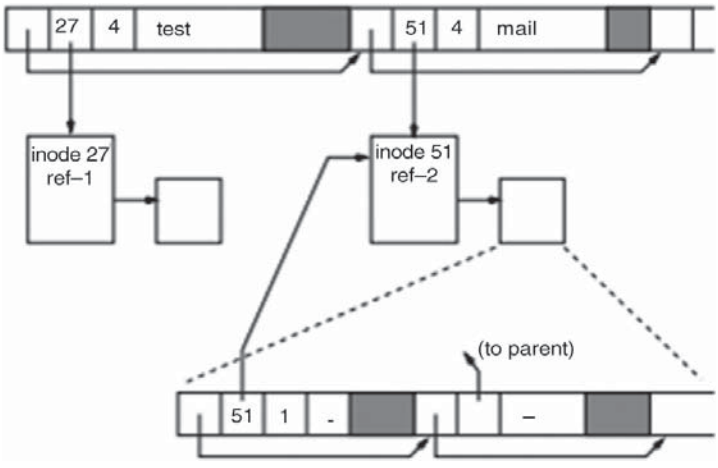


Fig. Creating a directory.

Finally, a link to inode is created in current directory and in new directory; following two entries are created such as:

- "." Which points to directory's own inode
- ".." Which points to parent's inode.

Removing Files

To remove a file in Unix, use shell command rm or system call unlink. While doing this, we see that:

- The directory entry is freed
- The record pointer of previous entry is reset
- The file reference counter is decreased by one
- If file reference counter reaches to zero, then data blocks and inode are freed.

Reading a Directory

To read a directory means to open it. Opening of directory is simply like seeing other file and read the data structures. To open a directory, it is easy to use the three standard functions such as;

- Opendir
- Readdir
- Closedir.

Reading an Inode

To read an inode, it is easy to read it with stat system calls. There is more than one stat function. Consider an example shown:

```
struct stat s;
d=opendir("nameofdirectory");
while (f=readdir(d)) {
//(use f)
stat(f->d_name, &s);
//(use s)
}
closedir(d);
```

Some of the common file management commands are describe below:

Command	Function
ls	List the contents of the current directory.
ls -	F List contents of current directory in terms of files, directories and exe.
ls -a	List the contents of the current directory, including hidden files.
ls -l	List the contents of the current directory in details.
pwd	Print the path of the current directory.
cd	Navigate to another directory.
Command	**Function**
mkdir	Make a directory.
rmdir	Remove an empty directory.
cp	Copy files.
mv	Move (rename) files.
rm	Remove files.
rm -i	Remove files with confirmation.
rm -r	Remove a directory. The contents of the directory is also removed.
rm -f	Remove files, overriding any confirmation.

6

Computer Operating Software

Earlier in 1960's, operating system serves as software which handles the hardware. Presently, we see operating system as set of programmes that create the hardware to work. Generally, operating system is set of programmes to facilitate controls of a computer.

There are different types of operating systems such as:

- UNIX,
- MS-DOS,
- MS-Windows,
- Windows/NT,
- VM

Over protecting of computer, engages software at numerous levels. We will distinguish kernel services, library services, as well as application-level services, all of which are division of an operating system. Processes run applications, which are related together by means of libraries that carry out

standard services. The kernel supports the development by providing a path to the peripheral devices. The kernel reacts to service calls as of the processes as well as interrupts from the devices. The centre of the operating system is the kernel, a organize programme with the purpose to function in restricted state, act in response to interrupts from external devices as well as service requests along with traps from processes. In order to run Computer hardware, we require an Operating System that will be able to recognize all hardware components and enable us to work on it.

Definition/Function

An operating system also known as OS is a software programme that enables the computer hardware to communicate and operate with the computer software.

Operating systems perform basic tasks:

- Recognizing input from the keyboard
- Sending output to Monitor
- keeping track of files and directories
- Controlling peripheral such as disk drives and printers.

The operating system is system software that is stored on the storage device such as hard disk, CD-ROM or floppy disk. When a computer is switched on, the operating system is transferred from the storage device into main memory through ROM.

An operating system controls and coordinates the operations of the computer system. It manages the computer hardware, controls the execution of application programmes and provides the set of services to the users. It acts as an interface between

user and the computer. The users interact with the operating system indirectly through application programme.

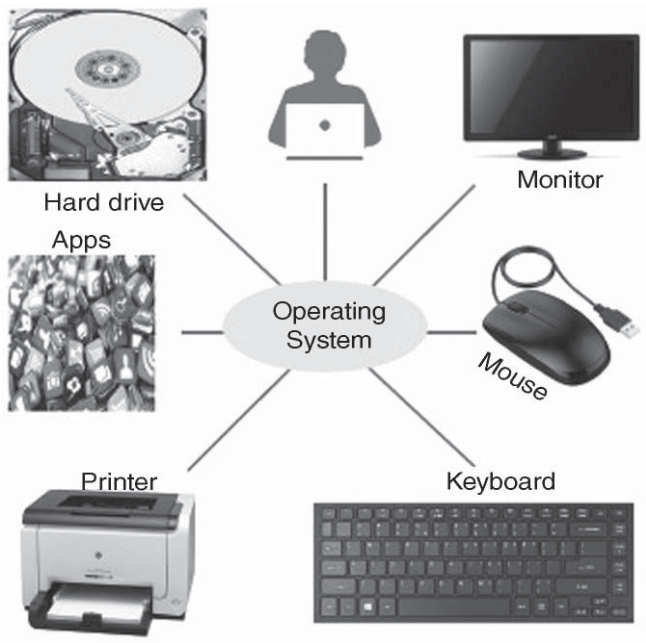


Fig. Operating System with Computer hardware.

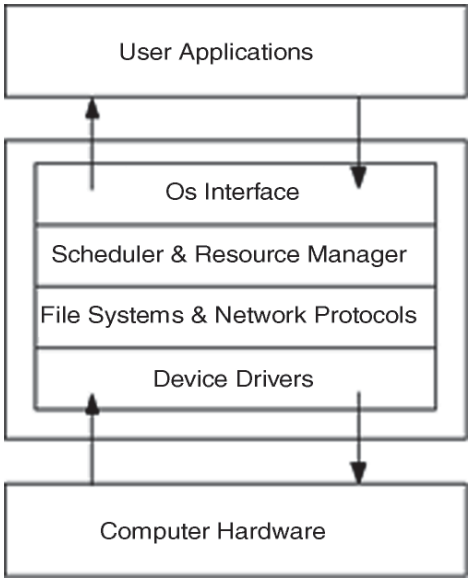


Fig. Position of Operating System.

The work of the operating system involves:

- Managing the processor.
- Managing Random Access Memory
- Managing Input/Output
- Managing execution of applications
- Managing Files
- Controlling Information management.

Basic Idea

The definition of an operating system is "the software that controls the hardware". However, today, due to microcode we need a better definition. We see an operating system as the programmes that make the hardware useable. In brief, an operating system is the set of programmes that controls a computer. Some examples of operating systems are UNIX, Mach, MS-DOS, MS-Windows, Windows/NT, Chicago, OS/2, MacOS, VMS, MVS, and VM.

Controlling the computer involves software at several levels. We will differentiate kernel services, library services, and application-level services, all of which are part of the operating system. Processes run Applications, which are linked together with libraries that perform standard services. The kernel supports the processes by providing a path to the peripheral devices. The kernel responds to service calls from the processes and interrupts from the devices. The core of the operating system is the kernel, a control programme that functions in privileged state (an execution context that allows all hardware instructions to be executed), reacting to interrupts from external devices and to service requests and traps from processes. Generally, the kernel is a permanent

resident of the computer. It creates and terminates processes and responds to their request for service.

Operating Systems are resource managers. The main resource is computer hardware in the form of processors, storage, input/output devices, communication devices, and data. Some of the operating system functions are: implementing the user interface, sharing hardware among users, allowing users to share data among themselves, preventing users from interfering with one another, scheduling resources among users, facilitating input/output, recovering from errors, accounting for resource usage, facilitating parallel operations, organizing data for secure and rapid access, and handling network communications.

Objectives

Modern Operating systems generally have following three major goals. Operating systems generally accomplish these goals by running processes in low privilege and providing service calls that invoke the operating system kernel in high-privilege state.

To Hide Details of Hardware by Creating Abstraction

An abstraction is software that hides lower level details and provides a set of higher-level functions. An operating system transforms the physical world of devices, instructions, memory, and time into virtual world that is the result of abstractions built by the operating system. There are several reasons for abstraction. First, the code needed to control peripheral devices is not standardized. Operating systems provide subroutines called device drivers that perform

operations on behalf of programmes for example, input/ output operations. Second, the operating system introduces new functions as it abstracts the hardware. For instance, operating system introduces the file abstraction so that programmes do not have to deal with disks. Third, the operating system transforms the computer hardware into multiple virtual computers, each belonging to a different programme. Each programme that is running is called a process. Each process views the hardware through the lens of abstraction. Fourth, the operating system can enforce security through abstraction.

Manage Resources

An operating system controls how processes (the active agents) may access resources (passive entities).

Provide a Pleasant and Effective User Interface

The user interacts with the operating systems through the user interface and usually interested in the "look and feel" of the operating system. The most important components of the user interface are the command interpreter, the file system, on-line help, and application integration. The recent trend has been towards increasingly integrated graphical user interfaces that encompass the activities of multiple processes on networks of computers.

One can view Operating Systems from two points of views: Resource manager and Extended machines. Form Resource manager point of view Operating Systems manage the different parts of the system efficiently and from extended machines point of view Operating Systems provide a virtual machine to users that is more convenient to use. The

structurally Operating Systems can be design as a monolithic system, a hierarchy of layers, a virtual machine system, an exokernel, or using the client-server model. The basic concepts of Operating Systems are processes, memory management, I/O management, the file systems, and security.

History of Operating Systems

Historically operating systems have been tightly related to the computer architecture, it is good idea to study the history of operating systems from the architecture of the computers on which they run. Operating systems have evolved through a number of distinct phases or generations which corresponds roughly to the decades.

The 1940's - First Generations

The earliest electronic digital computers had no operating systems. Machines of the time were so primitive that programmes were often entered one bit at time on rows of mechanical switches (plug boards). Programming languages were unknown (not even assembly languages). Operating systems were unheard of.

The 1950's - Second Generation

By the early 1950's, the routine had improved somewhat with the introduction of punch cards. The General Motors Research Laboratories implemented the first operating systems in early 1950's for their IBM 701. The system of the 50's generally ran one job at a time. These were called single-stream batch processing systems because programmes and data were submitted in groups or batches.

The 1960's - Third Generation

The systems of the 1960's were also batch processing systems, but they were able to take better advantage of the computer's resources by running several jobs at once. So operating systems designers developed the concept of multiprogramming in which several jobs are in main memory at once; a processor is switched from job to job as needed to keep several jobs advancing while keeping the peripheral devices in use.

For example, on the system with no multiprogramming, when the current job paused to wait for other I/O operation to complete, the CPU simply sat idle until the I/O finished. The solution for this problem that evolved was to partition memory into several pieces, with a different job in each partition. While one job was waiting for I/O to complete, another job could be using the CPU.

Another major feature in third-generation operating system was the technique called spooling (simultaneous peripheral operations on line). In spooling, a high-speed device like a disk interposed between a running programme and a low-speed device involved with the programme in input/output. Instead of writing directly to a printer, for example, outputs are written to the disk. Programmes can run to completion faster, and other programmes can be initiated sooner when the printer becomes available, the outputs may be printed.

Note that spooling technique is much like thread being spun to a spool so that it may be later be unwound as needed. Another feature present in this generation was time-sharing technique, a variant of multiprogramming technique, in which each user has an on-line (*i.e.*, directly connected)

terminal. Because the user is present and interacting with the computer, the computer system must respond quickly to user requests, otherwise user productivity could suffer. Time sharing systems were developed to multiprogram large number of simultaneous interactive users.

Fourth Generation

With the development of LSI (Large Scale Integration) circuits, chips, operating system entered in the system entered in the personal computer and the workstation age. Microprocessor technology evolved to the point that it become possible to build desktop computers as powerful as the mainframes of the 1970s. Two operating systems have dominated the personal computer scene: MS-DOS, written by Microsoft, Inc. for the IBM PC and other machines using the Intel 8088 CPU and its successors, and UNIX, which is dominant on the large personal computers using the Motorola 6899 CPU family.

Evolution of Operating System

Basic Idea of OS

In computing an operating system (OS) is the system software responsible for the direct control and management of hardware and basic system operations. Additionally, it provides a foundation upon which to run application software such as word processing programmes and web browsers.

Early computers lacked operating systems. A human operator would manually load and run programmes. When programmes were developed to load and run other programmes, it was natural to draw their name from the

human job they replaced. Today, the term is most often used colloquially to mean all the software which "comes with" a computer system before any applications are installed. The operating system ensures that other applications are able to use memory, input and output devices and have access to the file system. If multiple applications are running, the operating system schedules these such that all processes have sufficient processor time where possible and do not interfere with each other.

In general, the operating system is the first layer of software loaded into computer memory when it starts up. As the first software layer, all other software that gets loaded after it depends on this software to provide them with various common core services.

These common core services include, but are not limited to: disk access, memory management, task scheduling and user interfacing. Since these basic common services are assumed to be provided by the OS, there is no need to re-implement those same functions over and over again in every other piece of software that you may use. The portion of code that performs these core services is called the "kernel" of the operating system.

Operating system kernels had been evolved from libraries that provided the core services into unending programmes that control system resources because of the early needs of accounting for computer usage and then protecting those records. It is also noteworthy that some people use "kernel" to mean the core piece of the OS that deals most directly with the hardware, and have a slightly broader definition of "operating system". They would define "operating system" to

refer to the kernel plus some of the basic computer programmes and libraries that are necessary to use the kernel.

Modern Operating Systems

As of 2005, the major operating systems in widespread use on general-purpose computers have consolidated into two main families.

- Unix like family and
- Microsoft like family.

Mainframe computers and embedded systems use a variety of different operating systems, many with no direct connection to Windows or Unix. The Unix-like family is a more diverse group of operating systems, with several major sub-categories including SystemV, BSD and Linux. The name "Unix" is a trademark of The Open Group which licenses it for use to any operating system that has been shown to confirm to the definitions that they have cooperatively developed. The name is commonly used to refer to the large set of operating systems which resemble the original Unix.

Unix systems run on a wide variety of machine architectures. Unix systems are used heavily as server systems in business, as well as workstations in academic and engineering environments. Free Software Unix variants, such as Linux and BSD are increasingly popular, and have made inroads on the desktop market as well. Some proprietary Unix variants like HP's HP-UX and IBM's AIX are designed to run only on that vendor's proprietary hardware while others can run on the vendor's proprietary hardware and also on industry-standard PCs. Sun's formerly

proprietary Solaris (it is becoming open-source under the CDDL license) is one such versatile but true Unix (it can run on Sun's servers but also on smaller x86 systems). Apple's Mac OS X, a BSD variant, has replaced Apple's earlier (non-Unix) Mac OS in a small but dedicated market, becoming one of the most popular Unix systems in the process. The Microsoft-like family of operating systems originated as a graphical layer on top of the older MS-DOS environment for the IBM PC. Modern versions are based on the newer Windows NT core that first took shape in OS/2. Windows runs on 32 and 64-bit Intel and AMD computers.

Unix System V

System V, previously known as AT&T System V, was one of the versions of the Unix computer OS. It was originally developed by AT&T and first released in 1983. Four major versions of System V were released, termed Releases 1, 2, 3 and 4. System V Release 4, or SVR4, was the most successful version, and the source of several common Unix features, such as "SysV init scripts", used to control system startup and shutdown, and the *System V Interface Definition*(SVID), a standard defining how System V systems should work.

While AT&T sold their own hardware which ran System V, many customers ran a version from a reseller, based on AT&T's reference implementation. Popular SysV derivatives include Dell SVR4 and Bull SVR4. The most widely used versions of System V today are SCO Open Server, Based on System V Release 3, and Sun Microsystems Solaris Operating Environment and SCO Unix Ware, both based on System V Release 4.

System V was an enhancement over AT&T's first commercial UNIX called System III. Traditionally, System V has been considered one of the two major "flavors" of UNIX, the other being BSD. However, with the advent of UNIX implementations developed from neither code base, such as Linux and QNX, this generalization is not as accurate as it once was, and in any case standardization efforts such as POSIX are tending to reduce the differences between implementations.

There are five releases of SVR, namely:

1. SVR 1: The first version of System V was released in 1983. It introduced features such as the vi editor and cursors from the Berkley Software Distribution of UNIX developed at the University of California, Berkley (UCB). It also added support for inter-process comn using messages, semaphores and shared memory.
2. SVR 2: System V Release 2 was released in 1984. It added Unix shell functions and the SVID.
3. SVR 3: System V Release 3 was released in 1987. It included STREAMS, remote file sharing (RFS), shared libraries and the Transport Layer Interface(TLI).
4. SVR 4: System V Release 4.0 was announced on 1 Nov1989 and was released in 1990. A joint project of Unix Systems Labs and Sun Microsystems, it combined technology from Release 3 as well as 4.3 BSD, Xenix and Sun OS. TCP/IP and csh support from BSD. Network file system(NFS), memory mapped files, a new shared library system support from Sun OS. Other improvements were ksh, ANSI C, internationalization support, ABI and support for

standards such as POSIX, X/Open and SVID 3. SVR 4.1 added asynchronous I/O. SVR 4.2 added support for the Veritas file system, access control lists(ACLs), and dynamically loadable kernel modules.

5. SVR 5: Produced by the SCO group.

Berkley Software Distribution (BSD)

Berkeley Software Distribution (BSD) is the UNIX derivative distributed by the University of California, Berkeley starting in the 1970s. The name is also used collectively for the modern descendants of these distributions.

BSD pioneered many of the advances of modern computing. Berkeley's Unix was the first to include library support for the IP stacks, Berkeley *sockets*. By integrating sockets with the UNIX operating system file descriptors, users of their library found it almost as easy to read and write data across the network, as it was to put data on a disk. The AT&T laboratory eventually released their own STREAMS library, which incorporated much of the same functionality in a software stack with better architectural layers, but the already widely-distributed sockets library, together with the unfortunate omission of a function call for polling a set of open sockets (an equivalent of the select call in the Berkeley library), made it difficult to justify porting applications to the new API. Today, it continues to be used as technology testbed by academic organizations, as well as high-technology examples in a lot of commercial and free products. It is increasingly being used on embedded devices as well. The general quality of its source code design and clean writing, as well as its documentation, make such systems a heaven for programmers.

It is an interesting fact that BSD operating systems can run native software of several other operating systems on the same architecture, using binary compatibility. This, much faster than emulation, allows for instance to run applications intended for Linux on a BSD operating system at full speed. This makes BSD not only suitable for server environments, but also for workstation ones, considering the increasing availability of commercial or closed-source software for Linux. It also allows to migrate old commercial software which only used to run on commercial UNIX platforms to a modern BSD operating system, while retaining functionality of the previous system until it can fully be replaced by a better alternative.

Like AT&T Unix, the BSD kernel is monolithic, meaning that device drivers in the kernel run inprivileged mode, as part of the core of the operating system. Early versions of BSD were used to form Sun Microsystems' Sun OS, founding the first wave of popular Unix workstations.

GNU General Public License

The GNU General Public License (GNU GPL or simply GPL) is a free software license, originally written by Richard Stallman for the GNU project (a project to create a complete free software OS). It has since become the most popular license for free software (or "open source software"). The latest version of the license, version 2, was released in 1991.

The GPL grants the recipients of a computer programme the following rights, or "freedoms":

- The freedom to run the programme, for any purpose.
- The freedom to study how the programme works, and modify it. (Access to the source code is a precondition for this)

- The freedom to redistribute copies.
- The freedom to improve the programme, and release the improvements to the public. (Access to the source code is a precondition for this).

In contrast, the end user license that come with proprietary software rarely grant the end user any rights (other than the right to use the software, although it is debatable whether one requires a license for use *per se*), and may even attempt to restrict activities normally permitted by law, such as reverse engineering.

The primary difference between the GPL and more "permissive" free software licenses such as the BSD license is that the GPL seeks to ensure that the above freedoms are preserved in copies and in derivative works. It does this using a legal mechanism known as copyleft, invented by Stallman, which requires derivative works of GPL-licensed programmes to also be licensed under the GPL. In contrast, BSD-style licenses allow for derivative works to be redistributed as proprietary software.

By some measures, the GPL is the single most popular license for free and Open Source software. As of April 2004, the GPL accounted for nearly 75% of the 23,479 free-software projects listed on Freshmeat, and about 68% of the projects listed on SourceForge. (It should be noted that these two sites are owned by OSTG, a company that advocates Linux and the GPL).

The GPL does not give the licensee unlimited redistribution rights. The right to redistribute is granted only if the licensee includes the source code (or a legally-binding offer to provide the source code), including any modifications made.

Furthermore, the distributed copies, including the modifications, must also be licensed under the terms of the GPL.

This requirement is known as copyleft, and it gets its legal teeth from the fact that the programme is copyrighted. Because it is copyrighted, a licensee has no right to modify or redistribute it (barring fair use), except under the terms of the copyleft. One is only required to accept the terms of the GPL if one wishes to exercise rights normally restricted by copyright law, such as redistribution. Conversely, if one distributes copies of the work without abiding by the terms of the GPL (for instance, by keeping the source code secret), they can be sued by the original author under copyright law.

Many distributors of GPL'ed programmes bundle the source code with the executables. An alternative method of satisfying the copyleft is to provide a written offer to provide the source code on a physical medium (such as a CD) upon request. In practice, many GPL'ed programmes are distributed over the Internet, and the source code is made available over FTP. For Internet distribution, this complies with the license.

The copyleft only applies when a person seeks to redistribute the programme. One is allowed to make private modified versions, without any obligation to divulge the modifications as long as the modified software is not distributed to anyone else. Note that the copyleft only applies to the software and not to its output (unless that output is itself a derivative work of the programme); for example, a web portal running a modified GPL content management system is not required to distribute its changes to the underlying software. (It has been suggested that this be changed for version 3 of the GPL).

Free BSD and Linux

Almost all code in Free BSD is under the BSD license (one notable exception being the compiler, gcc). The BSD license puts very few restrictions on what can be done with code placed under it. Essentially, the only restrictions are that the user must attribute the previous contributors (*i.e.* the user can't claim it was all his work), the user cannot claim that the previous contributors endorse the user's product, and the user cannot hold the contributors liable for any mistakes in the code. After meeting those restrictions, essentially anything else can be done with the code, including distributing closed-source modified versions.

The Linux kernel and much of the utilities commonly distributed with it are under the GNU General Public License (GPL). The GPL allows free use of the software licensed under it under essentially the same restrictions as the BSDL with the additional requirement that if modified code is distributed, then the changes must be made available in source code for all to use. Generally, Linux is less centralized than Free BSD. Linux by itself is only a kernel. To function as an operating system, other utilities are required. These other utilities are gathered from various sources and collected together with the kernel by various groups in distributions. Kernel and system utilities are developed independently and merged together to form an operating system. This means that the kernel has one version, and all the other utilities in the operating system have others. Free BSD is more centralized. The kernel and basic system utilities are developed, versioned, and distributed together. Other programmes, such as X and web browsers, can be brought in from elsewhere, but the

basic system comes from one source and is designed specifically for the Free BSD operating system. Being versioned together in the same CVS tree is an advantage. Changes must consider all affected parts, not just the particular part being changed. This leads to a more cohesive, polished system. In fact, the concept of a kernel version different from the rest of the system does not really exist in Free BSD.

The two systems share much of the same functionality. They are often able to run programmes coded for the other system. When a complete desktop environment, such as GNOME or KDE is running, the two systems are often difficult to tell apart. Free BSD can also run Linux programmes due to a very lightweight Linux subsystem, which is capable of running even commercial Linux software.

Parts of Operating System

Resident Part

It is called as kernel that contains critical functions. It is loaded inside the main memory during the booting. It performs various functions residing in the main memory.

Non-resident Part

This part of operating system is loaded into main memory when required. It includes:

- Disk Operating System (DOS) developed by Microsoft.
- Operating System 2 (QS/2) developed by IBM.
- XENIX or ZENIX developed by Microsoft.
- WTNOWS developed by Microsoft
- WINDOWS- NE.

Evolution of operating system

Initially, the computer utilises batch operating systems where batches of jobs are run without taking a break. These programmes are punched into cards where the processing was performed by copied into tape. After finishing the first job, the computer would soon start with the next job on the tape. Professional operators when interacted with computer found that users drop such jobs and finally returned to hold the result soon after running of particular job. It was quiet difficult for users as expensive computer were made to involve in such type of processing of jobs.

During late 1960s, invent of time sharing operating systems led to replacement of batch systems. Users when involved directly by way of printing terminal found that Western Electric Teletype shown was ok. With this time sharing OS, many users shared the computer and then spent only a fraction of second on every job before moving to the next job. It is found that a fast computer will work for many user's jobs at the same time thereby making the illusion that they were full attentive while receiving such jobs.

Printing terminals found that the programmes were set of characters or command line user interfaces (CLI) where user had to type responses in order to typed commands which led to scrolled down the instructions on paper. During mid 1970, the personal computers allows pockets and Altair 8800 were initially used for commercial purposes for an individuals. In the start of 1975, the Altair was sold to hobbyists in kit form. It was without the operating system because it has only toggle switches and light emitting diodes which serves as input and output.

After sometimes, people started connected terminals and floppy disk drives to Altairs. During the year 1976, Digital Research introduced CP/M operating system for such Computer.

CP/M and later on DOS had CLIs which were similar to time shared operating systems where computer was only for a particular user. With the success of Apple Macintosh in 1984, the particular system pushed the state of hardware art which were restricted to small with black and white display.

As hardware continued to develop, many colour Macs were under developed position and soon Microsoft introduced Windows as its GUI operating system.

It was found that the Macintosh operating system was based on decades of research on graphically-oriented personal computer operating systems and applications. Computer applications today require a single machine to perform many operations and the applications may compete for the resources of the machine. This demands a high degree of coordination which can be handled by system software known as an operating system.

Internal Parts

The internal part of the OS is often called the kernel which comprises of:

- File Manager
- Device Drivers
- Memory Manager
- Scheduler
- Dispatcher.

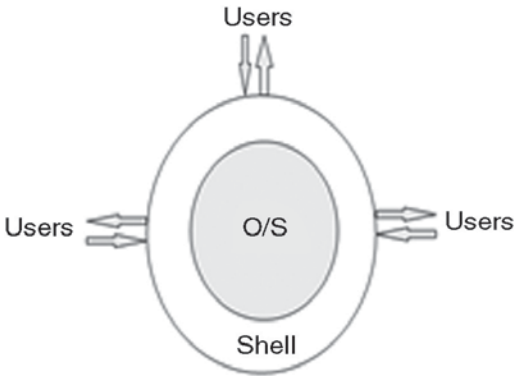


Fig. Interface of OS.

Operating System and Function

The operating system is the core software component of your computer. It performs many functions and is, in very basic terms, an interface between your computer and the outside world.. The operating system provides an interface to these parts using what is referred to as “drivers”. This is why sometimes when you install a new printer or other piece of hardware, your system will ask you to install more software called a driver.

Purpose of a Driver

A driver is a specially written programme which understands the operation of the device it interfaces to, such as a printer, video card, sound card or CD ROM drive. It translates commands from the operating system or user into commands understood by the component computer part it interfaces with. It also translates responses from the component computer part back to responses that can be understood by the operating system, application programme, or user. The below diagram gives a graphical depiction of the interfaces between the operating system and the computer component.

Operating System Functions

The operating system provides for several other functions including:

- System tools (programmes) used to monitor computer performance, debug problems, or maintain parts of the system.
- A set of libraries or functions which programmes may use to perform specific tasks especially relating to interfacing with computer system components.

The operating system makes these interfacing functions along with its other functions operate smoothly and these functions are mostly transparent to the user.

Operating System Concerns

As mentioned previously, an operating system is a computer programme. Operating systems are written by human programmers who make mistakes. Therefore there can be errors in the code even though there may be some testing before the product is released. Some companies have better software quality control and testing than others so you may notice varying levels of quality from operating system to operating system.

Errors in operating systems cause three main types of problems:

1. System crashes and instabilities - These can happen due to a software bug typically in the operating system, although computer programmes being run on the operating system can make the system more unstable or may even crash the system by themselves. This varies depending on the type of operating system. A system crash is the act of a system freezing and

becoming unresponsive which would cause the user to need to reboot.

2. Security flaws - Some software errors leave a door open for the system to be broken into by unauthorized intruders. As these flaws are discovered, unauthorized intruders may try to use these to gain illegal access to your system. Patching these flaws often will help keep your computer system secure. How this is done will be explained later.
3. Sometimes errors in the operating system will cause the computer not to work correctly with some peripheral devices such as printers.

Operating System Types

There are many types of operating systems. The most common is the Microsoft suite of operating systems.

They include from most recent to the oldest:

- *Windows XP Professional Edition*: A version used by many businesses on workstations. It has the ability to become a member of a corporate domain.
- *Windows XP Home Edition*: A lower cost version of Windows XP which is for home use only and should not be used at a business.
- *Windows 2000*: A better version of the Windows NT operating system which works well both at home and as a workstation at a business. It includes technologies which allow hardware to be automatically detected and other enhancements over Windows NT.
- *Windows ME*: A upgraded version from windows 98 but it has been historically plagued with programming errors which may be frustrating for home users.

- *Windows 98*: This was produced in two main versions. The first Windows 98 version was plagued with programming errors but the Windows 98 Second Edition which came out later was much better with many errors resolved.
- *Windows NT*: A version of Windows made specifically for businesses offering better control over workstation capabilities to help network administrators.
- *Windows 95*: The first version of Windows after the older Windows 3.x versions offering a better interface and better library functions for programmes.

Operating system structure-monolithic layered

The means of operating system architecture usually follows the leave-taking of particular principle which guides to re-structure the operating system mainly into relatively independent parts that can easily provide basic independent features by keeping complicated designs in manageable conditions.

Apart from controlling complexity, the architecture of operating system influences key features that are in terms of robustness or efficiency as:

- The OS receives importance which allows to work if not then protected resources like physical devices or application memory. With such importance, the various related parts of OS or OS as a whole will be both accidental and malicious privileges misuse gets lowered.

- By breaking OS into different parts will led to adverse effect on efficiency as overhead linked with communication among individual parts gets exacerbated when coupled with hardware mechanisms.

Monolithic Systems

Aboriginal concept of the operating system arrangement brings about no definite accommodation for the discriminating nature of the operating system. Furthermore the concept follows the separation of concern; no action is acted to limit the blessings granted to the single parts of the operating system. The complete operating system acts with maximum approvals. The communication overhead inside the basic operating system is identical as communication overhead in many other software, which are considered relatively low.

It is seen that CP/M and DOS are examples of monolithic operating systems that share common address space with certain applications. It is found in CP/M, 16 bit address space will begins with system variables along with application area additionally ends with 3 parts of O/S which are known as:

- CCP or Console Command Processor
- BDOS or Basic Disk Operating System
- BIOS or Basic Input/Output System.

If we see that in a DOS Operating System, there exists a 20 bit address space that begins with an array of interrupt vectors along with system variables that are followed by local DOS and its application area which will end with memory block utilised by video card and BIOS as shown in fig.

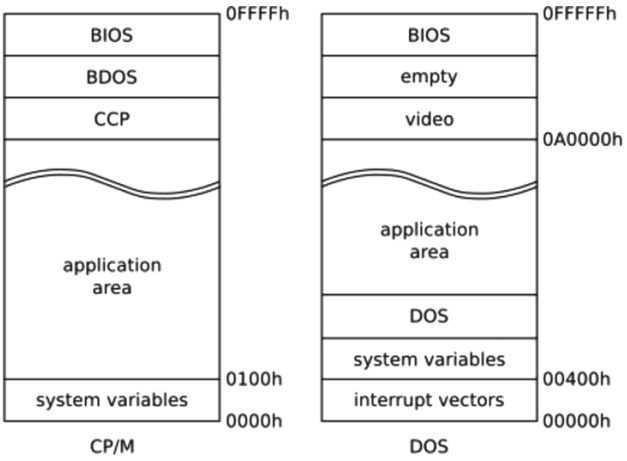


Fig. Monolithic Operating Systems.

Virtual machine and Client server

Virtual Machine

A virtual machine (VM) abides an operating system OS or conduct environment that is embedded on software which copies consecrated hardware. The end user embraces the equivalent experience on a virtual machine as they would acquire on dedicated hardware. Individualized software designated a hypervisor copies the PC client or server's CPU, memory, hard disk, network as well as other hardware resources collectively, allowing virtual appliances to participate the resources. The hypervisor can copy integral virtual hardware platforms that are occasional from each other, assigning virtual machines to run Linux as well as Windows server operating systems on the identical underlying physical aggregation.

Virtualization conserves costs by depreciating the need for physical hardware systems. Virtual machines additional desirably use hardware, which lowers the quantities of hardware as well as associated maintenance costs, along

with reduces power furthermore cooling demand. They also allay management due to virtual hardware does not collapse. Administrators can take advantage of virtual circumstances to simplify backups, disaster recovery, new deployments as well as elementary system administration tasks.

Virtual machines do not constrain distinguished hypervisor-specific hardware. Virtualization appears although require more bandwidth, storage along with processing capacity than a conventional server or desktop if the physical hardware is going to host multiple running virtual machines. VMs can easily move, be copied and reassigned between host servers to optimize hardware resource utilization. Because VMs on a physical host can consume unequal resource quantities, IT professionals must balance VMs with available resources.

Client Server

Client/server is a programme relationship in which one programme (the client) requests a service or resource from another programme (the server). It is seen that in client/server model, the programmes are used by single computer only. It serves as an important concept for networking. Here, the client makes a connection with the server through local area network (LAN) or wide-area network (WAN) like Internet. After clearing the client's request, the connection gets terminated. In this case, Web browser serves as a client programme which further appeals for a service from the server. The service and resource of the server will show the delivery of such Web page.

Computer assignments in which the server accomplishes a request created by a client are very customary furthermore

the client/server model has serve one of the main concepts of network computing. Most business approaches facilitate the client/server model as appears acts the Internet's core programme, TCP/IP. For exemplary, when you examine a bank account from the computer, a client approximation in computer overtures a request to a server programme at the bank which in twist forward a approach to its own client programme and conveys a request to a database server at another bank computer. Once the account balance sheet has been acquired from the database, it is acknowledged back to the bank data client, who in turn applies it back to the client in his/her personal computer that displays the information.

Both client programmes as well as server programmes are usual constituent of a larger programme or application. On account of multiple client programmes participate the services of the equivalent server programme, a special server identified a daemon may be charged due to anticipate client requests. In marketing, the client/server had been once used to differentiate allocated computing by personal computers (PCs) from the monolithic, concentrated computing model exercised by mainframes. This differentiation has largely evaporated, although, as mainframes along with their applications possess additionally turned to the client/server model further become part of network computing.

Types of Operating System

Introduction

There are abundant Operating Systems those monopolize be constructed for functioning the performances which are

demanded by the user. There are many different types of Operating Systems which acquire the ability to behave the entreaties acquired from different approach. The Operating system can behave in a unique operation and furthermore multiple movements at duration, so there are many categories of operating systems those are arranged by utilizing their working mechanisms.

There are many types of operating system such as:

- Serial Processing
- Batch Processing
- Multi-Programming
- Real Time System
- Distributed Operating System
- Multiprocessing
- Parallel operating systems.

Real Time Systems

There appears additionally an operating system which is comprehended as Real Time Processing System in which duration is already adjusted. Indicates duration to show the after-effects after acquiring has adjusted by the Processor or CPU. Real Time System is exercised at those areas in which we binds higher along with well-timed return. Such categories of approaches are exercised in reservation, so when we discriminate the demand, the CPU will conduct at that duration.

There are two types of Real Time System:

- Hard Real Time System: In Hard Real Time System, time is fixed and we can't change any moments of time of processing as the CPU will process the data as we enter it.

- Soft Real Time System: In Soft Real Time System, some moments can be change as after giving the command to CPU, the CPU will perform the operation after certain microseconds.

Multi-user System

As we comprehend that in case of Batch Processing System, there are results many jobs by the System. The System foremost compose a batch furthermore and will accomplish all jobs which gets saved in the Batch. Also, the innermost difficulty is that if a mechanism or jobs needs an input as well as output operation, in such case, it is not achievable and there will be the wastage of the duration when composing the batch processing as CPU will remain idle during the particular time.

Although with the help of multi programming we can achieve multiple programmes on the system at a duration as besides in multi-programming, the CPU determination never gets idle, so with the help of Multi-Programming we can achieve ample algorithms on system when functioning with programme and can acknowledge the supplement or other programme for sprinting extra CPU that will at that time behaves as secondary programme following the completion of original programme. Also in this, we can further differentiate our input means which a user can additionally interact with the system.

The multi-programming operating systems never utilize many cards on account of approach that is accessed on the spot by the user. Since the Operating System also utilizes the process of allocation and de-allocation of the memory

which shows providing memory space to all the running and all waiting processes. There must be the proper management of all the running jobs.

Distributed System

Distributed means data is stored and processed on multiple locations. When a data is stored on to the multiple computers, those are placed in different locations. Distributed in terms of network means, network collections of computers connected with each other.

If you want to take some data from other computer, then we uses the distributed processing system, as we can also insert and remove the data from one location to another location. In this, data is shared between many users, and we can also access all the Input and Output Devices by Multiple Users.

Other Types

There are other worthwhile types of operating systems not made by Microsoft. The greatest problem with these operating systems lies in the fact that not as many application programmes are written for them. However if you can get the type of application programmes you are looking for, one of the systems listed below may be a good choice.

- *Unix*: A system that has been around for many years and it is very stable. It is primary used to be a server rather than a workstation and should not be used by anyone who does not understand the system. It can be difficult to learn. Unix must normally run an a computer made by the same company that produces the software.

- *Linux*: Linux is similar to Unix in operation but it is free. It also should not be used by anyone who does not understand the system and can be difficult to learn.
- *Apple MacIntosh*: Most recent versions are based on Unix but it has a good graphical interface so it is both stable (does not crash often or have as many software problems as other systems may have) and easy to learn. One drawback to this system is that it can only be run on Apple produced hardware.