

Computer Operating Systems 214

Information Fact Sheet

Last Updated: February 7, 1997

Roberto Togneri

February 3, 2000

1 How to Use this Resource

This help sheet has been evolving for the past few years as a supplement to the lecture material and as such there will be a mixture of older help information with the new. Since the lecture material is regularly updated to more effectively tackle some of the problems highlighted here, don't be surprised if some of the help is not really all the helpful!

Unless contradictory to the lecture material, the older help information will be retained since it can be used as an alternative (perhaps more simplistic) explanation of some of the material covered in lectures.

2 Processes

2.1 CPU Scheduling

In multi-tasking operating systems there is more than one process or job that needs to be served by the CPU. Even in single task OS's like DOS the command shell (command.com) is a process as well as any TSR (Terminate and Stay Resident) programs like network drivers, graphics drivers, etc.; so there is more than one process in the system.

In sophisticated operating systems like UNIX the existence of CPU interrupt timer in the hardware allows pre-emptive scheduling of processes.

So what CPU scheduling strategies can we adopt?

First, we begin with a non-preemptive system (i.e. a process will run in the CPU until such time as it terminates or blocks on an I/O operation). To design a scheduling strategy we model a process as consisting of an alternating sequence of CPU burst + I/O burst + CPU burst + I/O burst + When the process enters the CPU it executes a CPU burst of specified duration and will block when

it reaches the I/O burst. Once the I/O burst block is complete the process is returned to the ready queue and waits to execute the next CPU burst and so on. Let t_i^j be the time to execute the i^{th} CPU burst for process j .

Let us assume that we are given t_i^j for all values of i and j . An optimal scheduling strategy is the *shortest-job-first (SJF)* where we select the next process to use the CPU to be the one with the smallest (or shortest) CPU burst time t_i^j across the j processes in the queue (less if any are still blocked).

Problems with SJF?

1. We do not know in advance the CPU burst times of a process, not unless all processes have been preprocessed (e.g. profiled) to derive the CPU burst times. In general this is impossible and specifically this will be impossible for processes which are input dependent (e.g. the CPU burst time of, say, a loop depends on an input variable value which is not known until the process actually starts executing).
2. As it stands jobs arriving and leaving the ready queue will affect the optimality of the scheduling. If shorter CPU burst processes enter the ready queue later an existing long CPU burst process in the running state will use the CPU until it completes its CPU burst time. A better average turnaround time would have been possible by waiting to schedule the shorter CPU burst processes or by somehow interrupting the running process if a shorter CPU burst process arrives in the ready queue.

The first problem can be solved by trying to estimate or predict the next CPU burst. Although there are many statistical and algebraic approaches to the general problem of time series prediction such techniques are either too simple to work or use up too much CPU time themselves (which is overhead) when they are sophisticated enough to work!

The second problem cannot be solved by delaying a process until shorter CPU burst processes enter the queue (since we don't know when such processes will arrive!) but we can allow a running process to be pre-empted hence:

shortest-remaining-time-first (pre-emptive SJF). In pre-emptive SJF if a process arrives in the ready queue with a shorter CPU burst than the remaining CPU burst time of the current running process the current running process is interrupted and placed back in the ready queue and the shortest CPU burst process is selected to run.

Problems with pre-emptive SJF?

1. Even more overheads! Pre-emption introduces context switching overheads. Furthermore the CPU burst times have to be resorted everytime a process enters the ready queue.

2. Short CPU burst processes can starve a long CPU burst process. Simply assume a steady stream of short CPU burst processes arriving in the ready queue, a longer CPU burst process will never get a chance to enter the running state.

So where do we stand? Not on very firm ground that's for sure! Solution? Go for a nice simple strategy which is easy to implement and does not make any assumption on CPU burst times:

Round Robin or RR scheduling. True RR is not optimal in any sense of the word but it is fair, allows priority control of processes easily and has minimal overheads. For specific applications and where optimal scheduling is imperative RR is not a good idea. However for general use commercial OS's RR is ideal. In RR scheduling each process is allowed a unit quantum of time to use the CPU. If the process reaches a blocked I/O state, all is fine, it goes in the blocked queue and the next process at the head of the ready queue enters the running state. If the process reaches the end of its allotted quantum of time, the CPU is interrupted and the process is placed at the tail of the ready queue.

With RR priority scheduling is also possible. Typically this is done by using different priority ready queues. There are various ways a queue can be made to have higher priority over another queue. One way is for lower priority queues to only become active when there are no processes in any of the higher priority queues. A fairer approach is to change the RR scheduling so the higher priority queues are scheduled more often than lower priority queues.

Let Q1, Q2 and Q3 be three priority queues with Q1 having the highest priority. Then with RR we select processes from the various queues as follows: Q1Q1Q1Q2Q2Q2Q3Q1Q1Q1... Thus processes in Q1 access the CPU 3 times more often than processes in Q3.

The next problem is priority allocation. Processes will need to move up and down the queues depending on how we define the characteristics of low and high priority processes. Typically we design so that interactive and I/O bound processes have higher priority than CPU bound processes. Apart from anything else I/O processes more often than not are blocked (so they don't use the CPU that much anyway) and an interactive response is highly critical on such processes acquiring the CPU immediately after becoming unblocked. On the otherhand CPU bound processes are usually background processes and they can access the CPU less often.

One possible scheme for dynamic priority allocation is to define different quantum times in the different queues with lower priority queues having longer quantum times (e.g Q1 has $q=1$, Q2 has $q=2$, etc.). If a process does not get blocked by the time it gets interrupted it is moved down a queue. Obviously this will move CPU bound process down the queues and keep I/O bound processes in the higher priority queues. A reverse process can be used to move processes up the queues; if a process blocks before its quantum expires it can be moved up a

queue. It must be remembered that with longer quantum times lower priority queues will need to be scheduled very much less often than higher priority queues. For instance: Q1Q1Q1Q1Q2... implies that Q1 processes with $q=1$ access the CPU 2 times more often than Q2 processes with $q=2$.

Of course when it comes to implementing the priority scheduling software techniques like linked lists, process structures, etc. will be used instead to simulate the above procedure rather than using actual queues.

2.2 Summary

In a multitasking system where there are many processes, the CPU scheduling system typically has processes in one of the following states:

- **Ready** queue waiting to be *dispatched* to the CPU.
- **Running** in the CPU (one process only can be in this state).
- **Blocked** queue waiting for *wakeup* after completion of blocking event.

With *pre-emptive* scheduling a process running in the CPU will be interrupted by a timer after an interval of time (the *quantum*). With *non pre-emptive* scheduling a process must run completion or become blocked before other processes are allowed to enter the Running state. Pre-emptive scheduling requires special CPU interrupt facilities. When the CPU switches to another process it performs a *context switch*. Each process is associated with a *Process Control Block* or PCB containing important state information for the process. The respective process PCB's need to be saved and loaded with each context switch and this constitutes an overhead.

Other issues covered:

- Operations on Processes: *create/fork, exit, suspend, resume, signal, changing priority using "nice", sending and receiving messages*
- Interrupt Processing
- High-level scheduling and low-level scheduling

How does the scheduler select processes from the Ready queue? Must satisfy the following criteria:

- High CPU utilization
- High Throughput
- Short Turnaround time (mainly for *batch* processes)

- Short Waiting time
- Fast Response times (mainly for *interactive* processes)

Processes are modelled as an alternating sequence of CPU bursts and I/O bursts. When selecting processes to enter the Running state we are interested in the duration of the CPU burst (low-level scheduler) or the duration of the process/job (high-level scheduler). A first order estimate of the next CPU burst can be made based on the past history.

Scheduling strategies based on knowing/estimating CPU bursts:

- *non-preemptive Shortest Job First* or simply *Shortest Job First* or SJF.
- *preemptive Shortest Job First* or *Shortest Remaining Time First*. Uses arrival time of processes.

Problem with SJF is the need to know the next CPU burst. A more direct scheduling strategy is the *Round-Robin* or RR scheduling which gives each process the same quantum of time at the CPU.

RR can also be configured for process priority scheduling since it is usually desired to give interactive mainly I/O-bound processes higher priority (for fast response time) and CPU-bound processes lower priority.

Implementing priority scheduling with RR:

- Give each each process in ready queue a priority number and select process with highest priority to run next. However need a mechanism to adjust priority values, usually based on past CPU usage.
- Use different quantum values for each process. However need a mechanism to adjust quantum times.
- Use multiple-queues with different quantum values and a policy to move processes between queues based on whether their allocated quantum has been used.

Other methods or heuristics for determining the next process from the Ready queue to be scheduled (e.g. for allocating process priorities, etc.) include:

- The lowest $\frac{CPU_{user}}{CPU_{real}}$ ratio is selected next.
- The highest $\frac{waitingtime+servicetime}{servicetime}$ ratio is selected next (*Highest-Response-Ratio-Next (HRRN)*).

3 Memory Management

3.1 Process Scheduling and Paging

It is important to remember that both process scheduling and paging occur together. So what are the dynamics of a system with process scheduling and paging? Let us assume that processes are scheduled on a round-robin basis and that the system is using a 2-level page table with TLB access. The best way to examine this is via the scenario method. Here are some scenarios involving the interaction between two processes A and B. Remember that the actual behaviour is highly implementation dependent (both the MMU hardware and operating system software):

- Scenario 1:
 1. Process A is running and Process is in the ready queue.
 2. Process A uses up its quantum and enters the ready queue.
 3. What happens to process A's pages? Nothing!
 4. What happens to process A's page tables? Probably nothing since each process has its own set of page tables.
 5. And what happens to the TLB? Well this is tricky since there is only one TLB, depending on the hardware MMU, the TLB may be flushed or simply marked so that all entries are invalidated. An alternative approach is to store a process ID with each entry of the TLB so that it contains the active page table entries for all processes. This makes sense for larger TLBs which are more costly to flush and are large enough to hold more than one process's active page table entries. An even more interesting solution is to store the TLB entries in the PCB for each process, thereby load the TLB immediately with the original entries and minimise the initial page faults. However this could be expensive and is not often implemented.
 6. As process B executes some of the pages from process A may be released and written back to the swap depending on the global page replacement policy in place. However the working set for process A should be remain intact as long as it returns to the ready queue asap.
- Scenario 2:
 1. Process A is running and Process is in the ready queue.
 2. Process A experiences a page fault and both a TLB and page table miss.
 3. Is process A placed on the blocked queue while it is waiting for the page to be read in from the disk? Assume yes.

4. Process B now enters the running state. Following our discussion in Scenario 1 then most of the pages and page tables for process A will be retained in memory, since there are other older or lower priority processes whose pages can be replaced sooner.
5. When the kernel retrieves the page for process A, then process A is placed in the ready queue. The page table entries for process A are updated with the new page frame information.

You may wonder whether the following situation can occur: Process A page faults, then process B page faults and the net result is that process A's pages are released causing process A to page fault again and so on.

The answer is to remember that the page replacement policy will replace the LRU or NRU pages which will belong to processes which are idle so it is unlikely that the recently running process A will have its active pages released. Furthermore, if enough pages are available to process A and B to cover their working set size then the number of page faults should stabilise. Similarly, if the working set changes or even increases in size either pages from idle process will be released or the no longer active pages of process A or B will be released, making way for the new pages or the increased working set size.

Of course the bottomline is that there is enough physical memory for the working set of all processes. If not then the above scenario where all processes are page faulting will occur very quickly with noticeable degradation in performance and a very busy disk and a very full blocked I/O queue! pages will be swapped to disk and the new pages read in and thereby maintain the working set for both processes.

3.2 Too many Paging Policies!

Some confusion may arise from the seemingly fragmented approach on the various policies and practices on paging. So what is the story?

The real problem is that page faults are fatal in terms of system degradation if the total number is high or process turnaround time if the per process page faults are relatively high. The bottomline is the maintain enough active pages in physical memory for the working set size of each process. If a process has too few pages then it will experience too many page faults. If a process has too many pages then other process will begin to experience too many page faults. Furthermore, the working set size is dynamic, so a process will need to grow and contract its usage of physical memory with the minimum possible number of page faults. How can this be done?

One consideration is how pages should be made available to processes. The simplest and most obvious way is to simply make pages available when it is required or demanded by the process (*demand paging*). The only problem is that this forces a page fault if pages are not yet in physical memory. One solution to

this is to use *anticipatory paging* where extra pages are loaded (or pre-fetched) so that when they are needed the process will not page fault. The main problem of course is how to predict which pages a process needs. At best the user may indicate this with special directives to the compiler or the compiler may embed special load instructions from the analysis of the source code, otherwise the operating system can do little. Consequently, demand paging is usually used.

Another consideration is which, when and how pages should be released or swapped to disk when there is a page fault. The *which* depends on the page replacement policy: LRU, NRU or modified FIFO and whether a local policy (i.e. can only swap out pages from the same process faulting) or global (i.e. can swap any pages from the system). Usually most systems will implement a global LRU or global NRU page replacement strategies. The next question is *when*. Usually page replacement occurs the moment a page fault mandates it (demand page replacement?). And finally *how*? The simple solution is to treat all page faults as major page faults, that is:

- invoke the requisite page replacement policy action when a page fault occurs,
- figure out which page needs to be replaced,
- then write the page back to disk (if it is a modified page) and
- then load the required page from the swap area.

Alternatively a pagedaemon or equivalent kernel function can periodically inspect the system activity and based on the page replacement policy itself or a more sophisticated policy like Page Fault Frequency (PFF) page replacement select pages that can be released. When the system selects pages to be released this usually means:

- If the page is modified it is written back to the swap area, and
- the page is marked as a free page and added to list of free pages.

Thus the system maintains a list of free pages. When a page fault occurs things are now much simpler (and less costly):

- select the page frame from the head of the list of free pages,
- and load the required page from the swap area.

With this last approach based on maintaining a free list of page frames it becomes obvious that most often there is a successful TLB lookup and only rarely will a major page fault occur:

- the TLB lookup first has to fail (TLB miss),
- then the main page table lookup has to fail (page fault),
- and finally the free list has to be empty (major page fault),

3.3 Number of Page Faults versus the Page Size

Slide 20 seems to indicate that the number of page faults increase with larger page sizes. Is this true? In reality the page fault rate will depend on whether the total working set size of all processes exceeds the available physical memory. Obviously since the average wasted memory is $p/2$ (where p is the page size) the wastage increases with page size and reduces the available physical memory, increasing the number of page faults. On the otherhand with smaller pages size processes have more pages and hence more page faults will occur by default! Mathematically minded students may like to theorize about and perhaps come up with an expression for the average number of page faults as a function of the page size.