CPS221 Lecture: Scheduling

last revised 9/4/14

Objectives

1. To introduce basic scheduling concepts
2. To introduce the three levels of scheduling
3. To introduce various short-term scheduling algorithms

Materials:

1. Projectables of process in isolation; in competition with other processes
2. Projectable of process states
3. Projectable of types of scheduler
4. Scheduler demo program with parameter files FCFS.params, SJF.params, PreemptiveSJF.params, RR.params

I. Introduction

A. We have seen that a major task of an operating system is to manage a collection of threads of execution.

B. This raises the following issue: on a system with a single CPU (or on a multi-processor system with fewer CPU's than threads), how is CPU time divided among the different threads that are competing to use it?

1. The component of the operating system that addresses these issues is called the scheduler.

2. As we shall see, scheduling is often handled on several levels, with CPU scheduling being the lowest level. So we will want to discuss scheduling in general and CPU scheduling in particular.

3. Many of the basic concepts of scheduling were developed in an era when batch processing and/or timesharing dominated computing, so that each thread of execution was part of a different process that typically represented the needs of different users.

a) Thus, discussions of this subject have historically spoken in terms of "processes" rather than "threads" - and most writers (but not your text) still do.

b) However, the same principles are relevant whether we have different processes servicing different users or multiple processes performing different tasks on behalf of the same user or multiple threads that are part of a single task.

4. Actually, though we will focus on CPU scheduling, the principles we will discuss are relevant to any situation in which there is competition among users for some shared resource.

C. CPU scheduling views a process (or thread) as being in one of three states at any given time:

1. The states are:

a) Running - i.e. actually using a processor.

b) Ready - i.e. able to use a processor, but not running because there are not enough processors to go around.

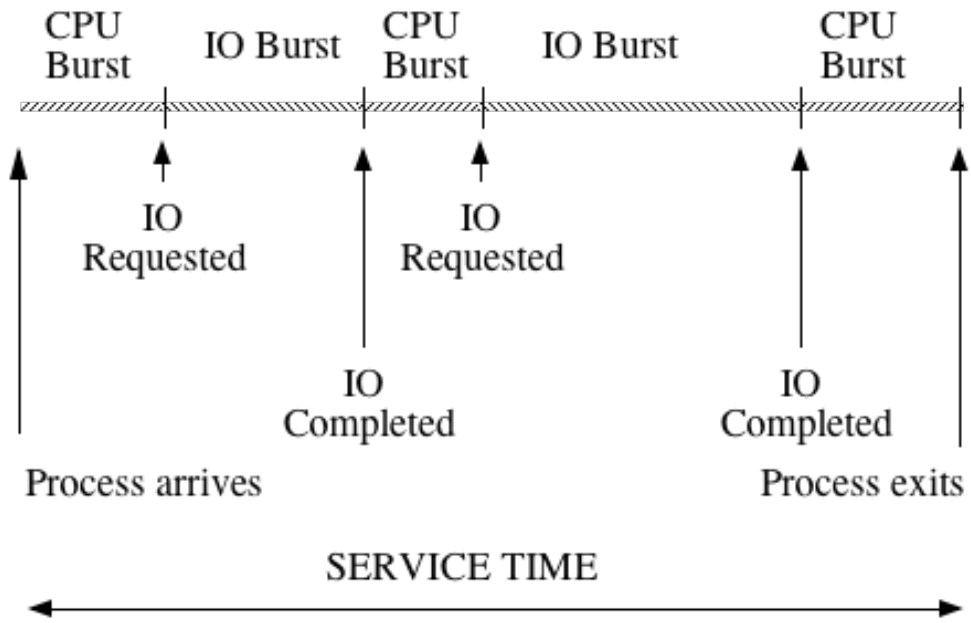
c) Blocked - i.e. unable to use a processor until some other event has occurred, such as completion of a disk operation or waiting for a message from some other process, or executing a sleep() operation.

Basic CPU scheduling strategies don't distinguish between different possible reasons why a process (or thread) may be blocked. Conventionally, we just call such situations "IO" (though the actual reason may be quite different, such as a sleep).

2. If a process (or thread) were executing in a context where there was no competition for resources, it would alternate between the running and the blocked state. Conventionally, we refer to these as "CPU bursts" and "IO bursts".

We can picture the behavior of a process (or thread) operating in a context in which there is no competition with other processes this way:

PROJECT Process in isolation



a) Observe that a process (or thread) always begins and ends with a CPU burst, with IO bursts and CPU bursts alternating in between.

(1) The last CPU burst ends with a termination operation (e.g. something like System.exit() in java).

(2) All other CPU bursts end with an operation that starts an IO burst. When discussing scheduling, we conventionally refer to these as IO requests - even though the actual operation may be something else.

(3) All IO bursts end with an operation that makes the process (or thread) ready for another CPU burst. When discussing scheduling, we conventionally refer to these as IO completion - even though the actual operation may be something else.

b) We refer to the sum total of the time required for all the bursts as the "service time" of the process (or thread).

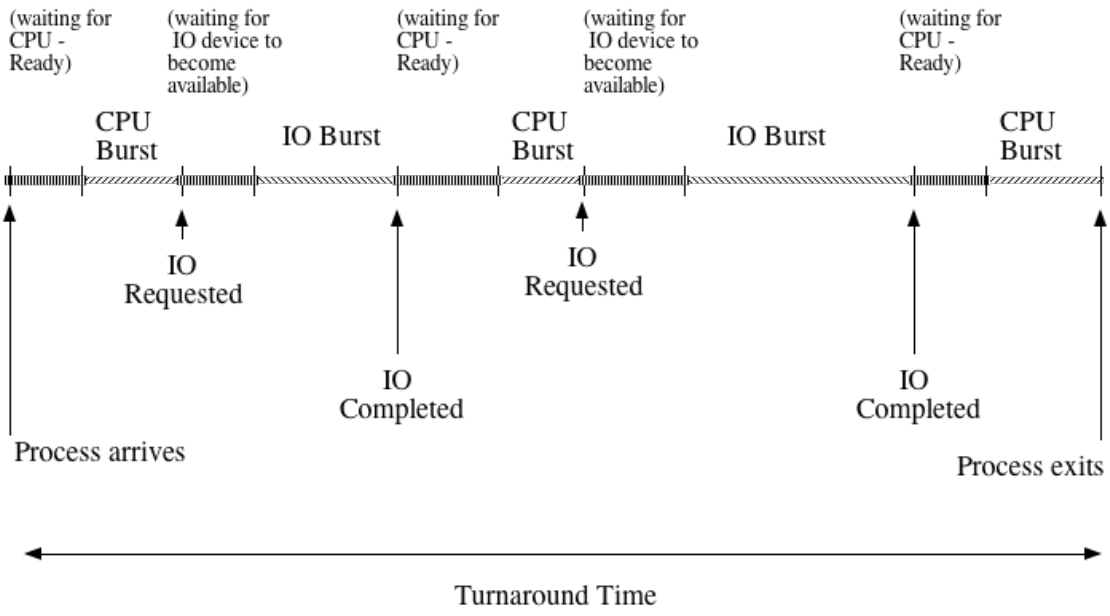
3. If the CPU bursts of a process (or thread) are typically shorter than the IO bursts, we say that the process/thread is IO Bound. If they are typically longer, we say that it is CPU Bound. This terminology reflects the fact that

   a) If it is IO bound, the service time is largely determined by the time spent doing IO. Improving the speed of the IO device could significantly reduce the service time, but improving the speed of the CPU might not.

   b) If it is CPU bound, the service time is largely determined by the time spent doing computation. Improving the speed of the CPU device could significantly reduce the service time, but improving the speed of the IO devices might not.

4. I f there are more processes (or threads) present on the system than can have their needs met by the available resources at any one time, then the actual situation will look like this

   PROJECT process in competition with other processes



a) Notice that it will often be the case that a process(or thread) that has requested an IO operation will have to wait until the device becomes available if some other process is using it. Of course, this entails a form of scheduling as well - but not the subject we are discussing here.

b) We refer to the total time between the time the process (or thread) starts and the time it terminates - including the waiting time - as its turnaround time.

D. Scheduling systems seek to achieve a balance between several factors.

1. Efficient utilization of resources. In general, we want to avoid having a resource sit idle when there is some process/thread that could use it.

   Example: Suppose that the system contains a mix of CPU bound and IO bound processes (or threads). In this case, if the CPU bound processes/threads are allowed to "hog" the CPU, then utilization of the IO devices will be low, since the primary processes that use them will have to spend a lot of time waiting for the CPU in order to be able to generate work for the IO devices to do.

2. Maximizing throughput. The throughput of a system is the number of processes (or threads) that actually complete in a period of time.

3. Minimizing average turnaround time.

4. Minimizing response time. The response time is the time between when an interactive users submits a request and when the system begins to respond to the request. (Response time is only an issue when the user is interactive).

5. Exhibiting desirable behavior

   a) Avoiding indefinite postponement. It should not be the case that biases in the design of the system result in some process (or thread) never being able to complete.

   b) Uniformity - the behavior of the system should be predictable. (e.g. interactive users tend to prefer a response time of several seconds to response times generally very low with occasional long delays.)

   c) Graceful degradation - in the face of excessive loads, the system response should deteriorate gradually, rather than coming to a sudden virtual standstill.

6. If the system is servicing real-time processes, it will also have to respect hard deadlines for these processes.

7. Schedulers also typically try to achieve some sort of fairness in the way they treat different users - though fairness might not be strict equality, but might rather be based on the system's priorities. (E.g. in a real time system it might well be undesirable for some notion of fairness to interfere with satisfying real-time deadlines.)

   Note that the above factors conflict with one another - so any scheduler represents a tradeoff between different considerations.

E. The scheduler works in cooperation with the interrupt system of the CPU. Most IO devices are capable of interrupting the CPU when they complete the work assigned to them - this causes the CPU to put aside the process it is working on and run a special interrupt handler.

1. The scheduler assigns the CPU to perform computation on behalf of a particular process (or thread within a process).

2. The CPU can be "borrowed" from its current process by an interrupt. This is under the control of the external devices, not the scheduler - though interrupts can be disabled for a short time if need be.

3. When a process (or thread) requests an IO operation, it becomes ineligible to use the CPU until the transfer is complete.

   a) This means that the scheduler will have to choose a new process (or a new thread within the same process) to use the CPU.

   b) Eligibility for the process (or thread) that requested the IO to use the CPU is restored when the device in question interrupts to indicate that the transfer is complete. Following the interrupt, the scheduler may be invoked to decide whether the CPU should go back to:

      (1) The process (or thread) that was running when the interrupt occurred.

      (2) The process (or thread) that requested the IO operation that caused the interrupt. This is called "IO preemption"

      (But on non-preemptive systems the CPU always goes back to the process (or thread) that was running.)

   c) Thus, the interrupt handlers have to interact in some way with the data structures used by the scheduler.

   d) Typically, a system also has a timer that interrupts the CPU at fixed intervals (e.g once per millisecond).

      (1) At each timer interrupt, the CPU's internal clock is updated by incrementing it.

      (2) A timer interrupt may also result in invoking the scheduler to give another process (or thread) a turn, to prevent a compute bound process/thread from "hogging" the CPU. We use the term "timer preemption" to describe the situation where the running process (or thread) is is forced to yield the CPU to some other process (or thread).

e) In a multiprogrammed operating system, the interrupt handling code of the device drivers together with the scheduler (or at least a portion of it) constitute the "kernel" of the operating system. Because the kernel routines must modify the scheduler data structures, it is common for them to run at least part of the time with interrupts disabled, so that an update is not interrupted in mid-stream.

F. For simplicity, in the remainder of our discussion we will speak in terms of scheduling processes. However, the same principles would apply to scheduling threads within a process if thread are implemented in the kernel (the one to one model) rather than in user mode (the many to one model).

II. **Basic Scheduling concepts**

A. Terminology

1. Recall that a process is a program in execution. From a scheduling point of view, a process constitutes a claim on the system resources.

   a) A process is defined by:

      (1) A portion of memory that contains the program that is being executed by the process, together with its data.

      Because the size of main memory is limited, a system may not be able to store all the code and data for all processes it is handling in memory.

      (a) In this case, the code and/or data of a process that does not appear to be needed immediately may be kept on disk to free up main memory for the needs of other programs. (This is referred to as virtual memory).

(b) If a process appears to be waiting for an event that is expected to be a long time coming (e.g. interactive input from a user who has not input anything to a given program for a while), it may be desirable to swap the entire program out to "swap space" on disk to free up memory space for currently running programs.

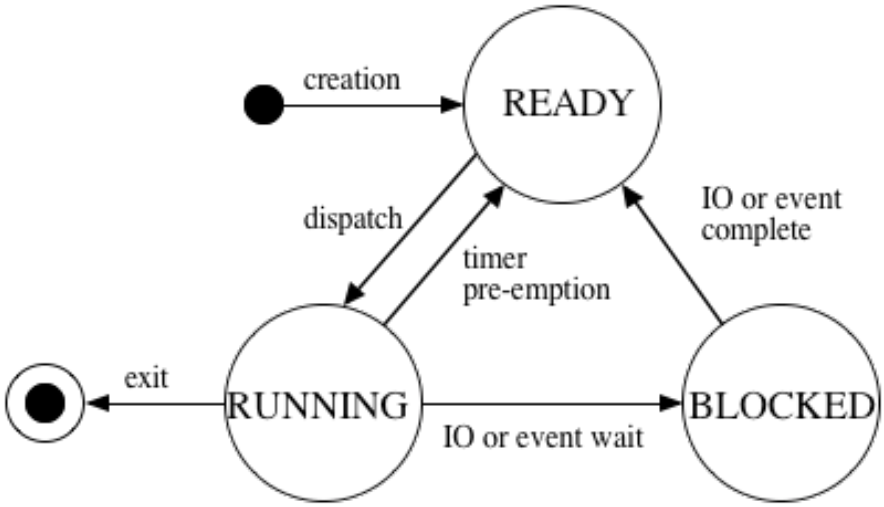
However, because it takes a significant amount of time to transfer information between main memory and disk, it is necessary that processes that are likely to be run in the near future (or at least the portion needed immediately) be resident in main memory.

(2) The contents of the CPU registers used by the process (including the PC which determines which instruction in the program is to be executed next.

In the case of a running process, the values are actually present in the registers of the CPU. For ready processes, they are kept in the PCB of the process in main memory.

When a new process is run in place of the previous one, it is necessary to copy the CPU registers for the process that was running to its PCB, and to copy to the CPU registers the values stored in the PCB of the process that is about to run. This is referred to as a "context switch", and does require enough time to make it important to avoid unnecessary context switches.

(3) A state. From the standpoint of the scheduler, the state transitions of a process may be pictured as follows:



PROJECT

Note that the operation of giving a process the CPU (so it becomes the running process) is called “dispatching”; for this reason, the scheduler is sometimes called “the dispatcher”.

(4) We have already noted that the system keeps information about a process in a process control block (PCB). In addition to containing information about the process’s memory allocation, registers, and state, the PCB will also contain appropriate information needed by the scheduler; the exact nature of this information will vary depending on the scheduling algorithm being used, of course.

2. A scheduler uses a number of lists - called queues - to keep track of processes waiting to use various resources: the CPU, disks etc.

a) In the context of schedulers, we use the term "queue" in a broader sense than the way we defined it in Data Structures.

(1) An operating system scheduler queue may be managed using a FIFO discipline, but often uses some other ordering mechanism such as some sort of priority scheme.

(2) In fact, a scheduler queue may be a fairly complex data structure composed of a number of lists - we'll see examples of this shortly.

b) In the case of the CPU, the queue is called the ready list and contains all the ready processes that are not currently running.

c) In addition, queues may be associated with various IO devices that (such as disk) that may be accessed by more than one process at a time. (We will not discuss scheduling algorithms for these.)

B. Types of schedulers: A multiprogrammed system may include as many as three types of scheduler:

1. The long-term (high-level) scheduler admits new processes to the system.

   Long-term scheduling may be necessary because each process requires a portion of the available memory to contain its code and data.

   a) This is typically used on batch systems, but is less common on timeshared systems or single user systems.

   b) In a batch system, a long term scheduler may attempt to achieve a good balance (job mix) between compute-bound and IO bound jobs. Ideally, the job mix would include 1-2 compute bound jobs that can help maximize CPU usage, plus as many IO bound jobs as possible - preferably representing a good mixture of demands on the various shared IO devices.

      Note that CPU time is wasted whenever the ready list is empty - i.e. all jobs are in the wait state. Compute bound jobs are almost always ready and thus help ensure that little CPU time goes unused.

2. Medium-term (intermediate) scheduling is controls which processes are actually resident in memory, as opposed to being swapped out to disk.

   The need for swapping may arise in two ways:

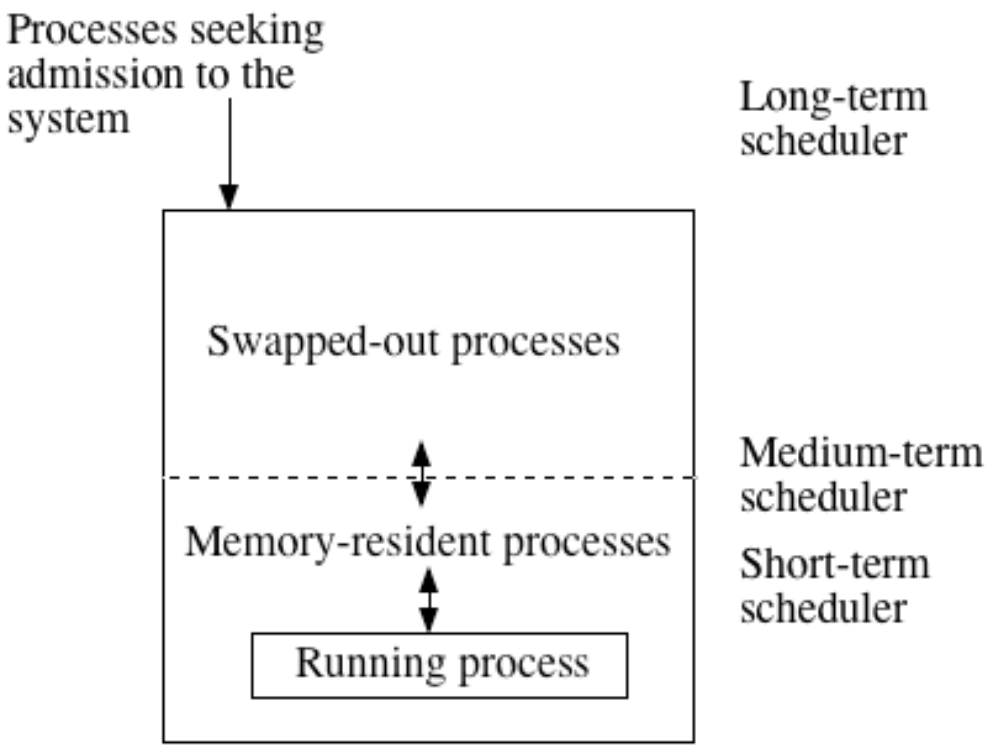
   a) In a batch system, the long-term scheduler may admit more users or in a time-shared systems more users may try to use the system than can all fit in memory, . However, since time-shared processes are characterized by bursts of activity interspersed with periods of idleness while the user thinks or takes a break, the medium-term scheduler can swap-out temporarily inactive processes. As soon as new input arrives from the user, the process can be swapped back in and another process swapped out.

   b) In any kind of system, a sudden increase in the memory requirements for one process can make it necessary to either swap out the process that wants to grow until space is available for it or swap out another process to make room for it.

   c) Note that the medium-term scheduler only swaps out processes when it has too. Ideally, swapping should only occur when the usage level on the system is very high. Often, the onset of swapping brings a noticeable decrease in system performance - hence the advice that one can often improve the performance of a sluggish system by buying more memory!

3. The short term scheduler determines the assignment of the CPU to ready processes. The rest of our discussion will focus on this kind of scheduling.

4. Summary:



PROJECT Types of scheduler

Note on frequency of execution:

a) A long-term scheduler is executed infrequently - only when a new job arrives. Thus, it can be fairly sophisticated. (Long-term schedulers are generally only found on batch systems)

b) A medium-term scheduler (if it exists) is executed more frequently. To minimize overhead, it cannot be too complex.

c) A short-term scheduler is executed very frequently - whenever any IO request occurs, and often when one completes, as well as possibly when a timer interrupt occurs Thus, it must be very fast. On some machines, it is common to find hardware support for certain scheduler functions to help out.

III. **Scheduling Algorithms**

A. Scheduling algorithms can be classified in two different ways:

1. First-come-first-served type algorithms vs priority algorithms.

   a) Under a first-come first-served scheme, a queue (in the operating systems sense) is indeed a queue in the data structures sense.

   b) As we shall see, it is often desirable to use some sort of scheme in which each process has a priority assigned to it. In a priority scheme, the highest priority ready process is selected; in the case of equal priorities, FCFS is used to resolve ties. (Note we can think of FCFS as the degenerate case of a priority scheme in which all processes have the same priority.)

      Priorities can be

      (1) Externally assigned

         (a) On the basis of the type of process:

            i) Timeshared systems may allow batch processes to run as background processes with lower priority. Thus, the system gives priority to interactive users, but uses otherwise unused CPU cycles for batch processes.

            ii) Real-time systems may allow non realtime processes to run, but give priority to real-time processes which may be idle for much of the time, but have get all the CPU resources they need when they need the CPU.

         (b) On the basis of payment for service. If computer time is being sold to outside users, multiple classes of service may be provided, with higher priority for higher billing.

This doesn't seem to be particularly relevant now, but could become much more relevant as "Cloud computing" catches on.

(c) On the basis of the user voluntarily choosing a lower priority.

SHOW UNIX man page for nice

(2) Internally computed by some algorithm that seeks to maximize one or more performance characteristics. (We'll discuss some of these below)

(3) Note that external priorities are typically static - i.e they are determined before a process is started - while internal priorities are typically dynamic - i.e. they are computed as the process is running, based on its behavior

(4) Also possible is some combination of external and internal priorities.

2. Non-preemptive vs preemptive schemes.

a) In a non-preemptive scheme, once a process has been granted the CPU, the simplest approach is allow the process to continue using the CPU until it voluntarily yields the CPU - eg by requesting an IO transfer. Of course, IO interrupts may steal the CPU from time to time; but after each interrupt, control passes back to the process that was running when it occurs. This is called a non-preemptive approach.

b) In a preemptive scheme, a running process may be forced to yield the CPU (thus returning to the ready list) by an external event rather than by its own action. Such external events can be either or both of the following kinds:

(1) Timer pre-emption. When a process is dispatched (given the CPU), some algorithms assign a time quantum which represents the amount of time the process is allowed to run without blocking. If the process still hasn't yielded the CPU of its own accord at the end of the time quantum, then it is pre-empted in favor of some other ready process.

(2) Priority pre-emption (in a scheme that uses priorities) When a higher priority process enters the system from outside or a higher-priority process that was in the blocked state becomes ready as the result of an IO interrupt that moves a formerly waiting process to the ready list, the currently running process may be preempted in favor of this higher priority process.

(3) While these two sorts of pre-emption is similar, it is helpful to distinguish them because some algorithms use one sort of preemption but not the other.

B. Probably the simplest scheduling algorithm is straight first-come first-served (FCFS).

1. This is a non-priority, non-preemptive algorithm. For reasons we shall soon see, it is rarely used for short-term scheduling, but it can be used for long-term scheduling and also provides a foundation for understanding other algorithms.

2. In the case of short term (CPU) scheduling, the basic idea is this

   a) a queue of ready processes is used - called the ready list.

   b) Whenever the CPU becomes available because the running process yields it by blocking or terminating, the first process on the ready list is removed and given the use of the CPU for as long as it needs it (i.e. until it blocks or terminates).

c) When a new process enters the system or a blocked process becomes ready, it is placed at the end of the ready list - to be given the CPU when its turn comes.

3. Evaluation

a) The obvious strength of this algorithm is its simplicity

b) But it has significant weaknesses.

(1) On a system that has a mixture of CPU-bound and IO-bound processes, this scheme would exhibit a bias toward one type of process. Which type, and why?

ASK

(a) It is biased in favor of CPU Bound processes, because CPU bound processes typically need few but long CPU bursts, while IO bound processes typically need many but short CPU bursts - so IO bound processes will have to wait for the CPU more often.

(b) DEMO: Scheduler demo with file FCFS.params

(2) A second problem - related to this - is that while this algorithm will keep the CPU fairly busy if system load is high, it tends not to make good use of IO devices. (Note in demo)

C. One way to address the problem of FCFS's bias in favor of CPU-bound processes is to use an internal-priority algorithm known as shortest job first (SJF).

1. The name comes from the fact that this algorithm was first used for long-term scheduling. In a batch environment, users might be required to give an upper bound on the <u>total</u> CPU time needed for a job. (If a job exceeded this upper bound, it would be terminated - something which tended to discourage low balling an estimate!)

2. For short-term scheduling, priority is based on inverse order of CPU burst length.

   Of course, this raises a question - how can CPU burst length be known?

3. The answer is that, while the time needed for a given CPU burst cannot be known ahead of time, it is possible to make a good estimate based on the past behavior of the process

   a) One approach is to simply keep track of the total CPU time used so far by a process, and the total number of CPU bursts received so far. Then the estimated time of the next burst is simply total time / number of bursts.

   b) A better approach, known as exponential averaging. is both simpler to calculate and takes into account the fact that burst lengths may vary over time as the process moves through various phases of activity, though not too widely from one burst to the next.

      (1) A process's estimated burst duration is a weighted average of:

         (a) The predicted duration of its last burst (which incorporates information about its long term history)

         (b) the actual duration of its last burst.

      (2) The formula that is used is

      $$t_{n+1 \text{ predicted}} = \alpha * t_{n \text{ actual}} + (1 - \alpha) * t_{n \text{ predicted}}$$

      (where alpha (α) is a parameter between 0 and 1 that determines the relative weight given to the two terms.)

(3) The name "exponential averaging" comes from the way this computation ends up treating the past history of the process. For example, using the above with $\alpha = .5$, we get

$$t_{n+1 \text{ predicted}} = \alpha * t_{n \text{ actual}} + (1 - \alpha) * t_{n \text{ predicted}}$$

$$= .5 * t_{n \text{ actual}} + .5 * t_{n \text{ predicted}}$$

$$= .5 * t_{n \text{ actual}} + .25 * t_{n-1 \text{ actual}} + .125 * t_{n-2 \text{ actual}} + \ldots$$

$$= \sum_{i=0}^{n} (.5)^{n-i+1} * t_{i \text{ actual}}$$

(4) Note that this can be easily implemented in software (without needing to actually keep a complete history of burst lengths). In each PCB we store the predicted value for the current burst. When the burst terminates, we can simply update the estimate. In the case where we use $\alpha = 0.5$, we can simply add in the actual value and then shift the result right one place = dividing by 2.

4. Evaluation

a) It can be shown that SJF minimizes <u>average</u> turnaround time since moving a shorter job ahead of a longer job decreases the turnaround time for the shorter job more than it increases the turnaround time for the longer job.

b) However, in an environment where there are many IO bound processes, it becomes possible for IO bound processes to starve CPU bound processes.

DEMO: Scheduler with SJF.params

(Note: demo does not attempt to deal with waiting for IO devices - the average IO time presumably includes any waiting needed.)

5. It is also possible to create a preemptive variant of this scheme in which a blocked process using shorter burst lengths that becomes ready is allowed to preempt a process with longer burst lengths that is currently running.

   Demo: Scheduler with PreemptiveSJF.params - note slightly better performance

D. It is also possible to use a straight priority scheme, based on external priorities, with or without preemption.

1. When the time comes to choose a process to run, instead of choosing the first process to have arrived, we might choose the highest priority process, based on external priorities.

2. Of course, this runs into the problem of potentially starving low priority processes.

   a) There is a tale - possibly apocryphal - that when an early system that used priority scheduling was shut down after many years of operation, a low priority job was found that had been in the queue for several years but was never run!

   b) Whether this story is true or not, less extreme starvation of low priority processes is certainly a possibility.

   c) This problem is typically addressed by using some sort of aging scheme in which a low priority process's priority is increased gradually based on its waiting time, so that it will eventually be chosen to run. We will look at a particular variant of this shortly.)

3. Again, it is possible to create a preemptive version of this approach.

4. It is also possible to combine this approach with some adjustment of the external priority based on the behavior of the process - e.g.

a) A process's priority may be adjusted upward or downward based on its burst length behavior.

b) A process that has just been swapped into memory by the medium term scheduler may be given increased priority for access to the CPU and/or for protection against being swapped out, so as to not waste the investment involved in swapping it in.

c) A process that is using an otherwise under-utilized resource may be given increased priority.

E. An approach that is used quite widely is called round-robin (RR). In its basic form it is a non-priority scheme using just timer preemption.

1. This scheme is similar to FCFS, with one important difference: Whenever a process is given the CPU, it is allowed to keep it only for a fixed period of time called the time quantum. If a process does not voluntarily yield the CPU in this time (by becoming blocked or terminating), the CPU is taken away from it and given to the next ready process, while it is placed at the end of the ready list.

2. This algorithm tends to be more balanced in its treatment of IO bound and CPU bound processes.

   DEMO: Scheduler run with RR.params

3. While basic round robin uses just timer preemption, it can be extended to use external priorities and possibly priority preemption as well.

   a) When a process is chosen to run, the highest priority ready process is chosen, rather than the next in line as in FCFS.

b) When a blocked process becomes unblocked, it may be allowed to preempt the running process if the running process has lower priority. (This is in addition to timer preemption resulting from using up a time quantum).

c) Of course, one problem to be faced when external priorities are used is this: if a high priority process is pre-empted due to a time quantum expiration, then how is the next process to run selected? If we simply take the highest priority ready process, then we would simply select the process that was just pre-empted since, by assumption, if it was running then it was the highest priority running process!

(1) Often we find that the majority of the processes on a system may have the same priority, especially if priorities are externally assigned. In this case, we use FCFS within a priority; so the pre-empted process is placed behind all other processes of the same priority in the queue.

(2) The priority of a pre-empted process may be reduced in some way to enable another process to get a crack at the CPU.

F. An interesting approach that combines aspects of SJF and RR with a form of aging is called Highest Response Ratio Next (HRN).

1. HRN is based on something called the response ratio. The response ratio of a process is define as

response ratio = (time waiting + service time) / service time

a) Clearly, the response ratio grows with waiting time.

b) However, for shorter service times, the response ratio increases more rapidly with time waiting.

For example, the following table contrasts how response ratio grows with waiting time for two processes, one with service time 1 and one with service time 5

Waiting time	Response ratio service time 1	Response ratio service time 5
0	1	1
1	2	1.2
2	3	1.4
3	4	1.6
4	5	1.8
5	6	2
6	7	2.2
7	8	2.4
8	9	2.6
9	10	2.8
10	11	3

c) Thus, the response ratio can be used as a priority, with greater values meaning higher priority This ends up doing two things:

(1) It gives priority to processes needing shorter CPU bursts, as in SJF.

(2) It incorporates aging to avoid SJF's potential starvation problem.

2. HRN can also use RR-style time quantums. In this case, if a process uses up its time quantum, response ratios can be recalculated before deciding whether to give it more CPU time.

a) The running process's response ratio will have decreased because it has received more service time.

b) But the response ratio of all other processes will have increased because they have experience additional waiting time.

DEMO

3. HRN can also use external priorities - so that the priority of a process reflects some combination of an external priority and its calculated response ratio.

IV. **Time quantum selection**

A. We have noted that many scheduling systems use some form of timer interruption to keep a single process from hogging the CPU. A key design question is the length of the time quantum to be used.

1. The main reason why this is an issue is that switching from running one process to running another process - known as context switching - can entail significant use of CPU time.

   a) The actual actions that need to occur - saving one process's set of registers in the PCB and loading another's - takes some time.

   b) But the biggest impact may come from the memory system, due to the need to replace entries in cache memory and possibly even in main memory.

2. If the time quantum is of magnitude comparable to the time needed for context switching, then the overhead of context switching will require an unduly high share of the processor cycles.

   DEMO: Scheduler with RR.params - first with time lost to context change initial value (0), then 1, then 5, then 10

3. If the time quantum is too long, then response time on a time shared system will begin to degenerate, and IO device usage will drop on any kind of system as IO bound processes are unable to get the CPU to start a new IO operation when the current one completes because a compute bound job has it.

B. One textbook author suggests that one picture an operating system as having a dial on the front labeled "q" (for quantum). Suppose the dial is initially set at 0.

1. No one would get any work done - as soon as a process gets the CPU, it has to yield it.

2. As the dial is slowly turned up, system response would begin to improve. At low settings, context changing overhead might still occupy a high percentage of the overall CPU time, so user processes would be slowed down. But this overhead would decrease as the quantum is increased.

3. As "q" is increased, more and more of the processes on the system would be able to complete a CPU burst without being interrupted by the timer.

4. However, at some point the response time would begin to deteriorate again. This would be due to an occasional compute-bound process taking a large slice of time while all the other processes wait.

5. Assuming there is a heavily compute bound process on the system, further increases in "q" beyond the optimal point would continue to degrade response time.

6. Note, however, that if we measure throughput in terms of CPU usage, it is possible that throughput might continue to increase even after response time has begun to decrease, since less and less of the CPU time goes to overhead. However, eventually a point would be reached where even throughput would decay; when a compute bound process does yield, other IO bound processes might not be able to take up the slack before themselves having to yield the CPU for IO. This might lead to the CPU sitting idle while all processes wait on IO.

C. A good rule of thumb would seem to be that the majority of processes should be able to complete a CPU burst without timer interruption. (One text suggests around 80% as a rule of thumb. This might be

higher in a highly-interactive situation, or lower in a situation where there is much heavy computation going on.)

D. It is also possible to relate time quantum to priority. A compute bound process that could benefit from a long time quantum might be given a much longer than usual quantum, but also a lower priority. This would mean that it gets CPU bursts less often, but gets longer bursts each time, resulting in the same overall average CPU use with less overhead.

   The following builds on this idea to create a scheme known as multi-level feedback queues. (MLFQ)

   1. The ready list consists of an array of FIFO queues of decreasing priority.

   2. There is a time quantum associated with each queue.

      a) For the highest priority queue, time quantum is some basic value.

      b) For each successive queue, the quantum is double the quantum for the next highest priority queue. Thus, if there are n queues, the lowest priority queue has quantum (quantum of the highest priority queue) * 2^{n-1}.

   3. A newly arrived process is placed at the rear of the highest priority queue.

   4. Whenever the dispatcher needs to choose a process to run, it starts looking at the highest priority queue. If the dispatcher finds the highest level queue to be empty, it looks at the next queue down - and keeps looking until it finds a non-empty queue. At that point, it dispatches the front process in that queue with time quantum equal to the value associated with that queue.

   5. Processes move between queues based on their performance.

a) A process that completes its CPU activity before the quantum expires will be moved up a level when it next becomes ready (unless it is already in the highest queue).

b) A job that fails to complete its CPU burst within the time quantum is moved down to the next queue, until it reaches the bottom, where it remains.

6. A process can be preempted by another process if

a) A blocked process associated with a queue of higher priority becomes ready

b) In the case of quantum expiration, if there is another process in the queue the process was in previously (since it has now gone down a level)

7. Some sort of aging scheme may also be applied to keep processes from getting "stuck" at the lowest level with no chance to run.