Process Scheduling

Goal

What characterizes a “fair internal policy?” Which process is given the CPU next? This is the province of *schedulers.*

Schedulers

Three kinds:

- *long-term scheduler* determines which jobs are admitted to the system for processing
  example: in a batch system, often more jobs are submitted than can be done at once, so some are spooled out to a mass storage device; the long-term scheduler selects the next one to be loaded into memory. So it controls the degree of multiprogramming, *i.e.*, the number of processes in memory.
- *short-term scheduler* determines which job in memory (*i.e.*, in the ready queue) goes next
- *medium-term scheduler*: at times jobs may have to be removed from the system temporarily; that is, too many jobs may be competing for memory. The removed process will be restarted where it left off later; called *swapping*. This scheduler decides who gets swapped out and in.

The long term scheduler is invoked relatively infrequently, but the short term one is invoked often — whenever *any* process returns control to the operating system. Hence the short-term scheduler must be very fast. (Context switching also must be very fast; typically, 10μs to 100μs. Many machines have special-purpose instructions, like the VAX LDCTX, for just this reason.)

The system should try to balance CPU-bound and I/O-bound jobs.

Scheduling Considerations and Overview

These choose which process goes next. Which one is used depends on what is wanted from the system; possible measures are:

- *throughput*; get the most work done in a given time
- *turnaround*; complete jobs as soon as possible after submission
- *response*; minimize the amount of time from submission to the first response (called the *response time*); this interval does *not* include the time to output the response
- *resource use*; keep each type of resource assigned to some process as much as possible, but avoid waiting too long for certain resources.
- *waiting time*; minimize the amount of time the process sits in the ready queue
- *consistency*; treat processes with given characteristics in a predictable manner that doesn't vary greatly over time.

In the process of scheduling, the processes being considered must be distinguished upon many parameters, among them

priority
anticipated resource need (including running time)
running time, resources used so far
interactive/non-interactive
frequency of I/O requests
time spent waiting for service

To demonstrate how algorithms work, we'll use this set of jobs:

	Arrival Time	Service Time
A	0	10
B	1	29
C	2	3
D	3	7
E	4	12

and measure 3 quantities:

- turnaround time: time the process is present in the system

$$T = \text{finish time} - \text{arrival time}$$

- waiting time: time the process is present and not running

$$W = T - \text{service time}$$

- response ratio (sometimes called the “penalty ratio”): the factor by which the processing rate is reduced, from the user's point of view:

$$R = \frac{T}{\text{service time}}$$

Characterization of Scheduling Algorithms

decision mode

This is *non-preemptive* if a process runs until it blocks or completes; at no time during its run will the operating system replace it with another job. It is *preemptive* if the operating system can interrupt the currently running process to start another one.

priority function

This is a mathematical function which assigns a priority to the process; the process with the highest (numerical) priority goes next. The function usually involves the service time so far a, the real time spent in the system so far r, and the total required service time t.

arbitration rule

If two processes have the same priority, this rule states how one of them is selected to run.

The Scheduling Algorithms

First Come, First Served (FCFS)

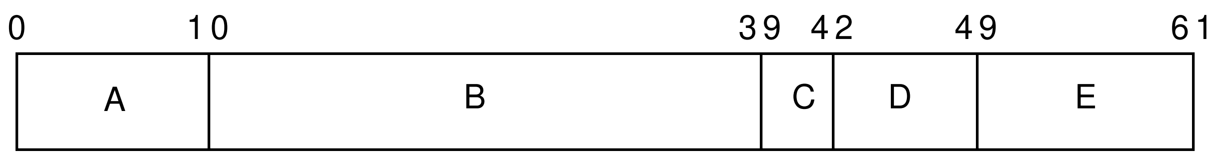
decision mode: non-preemptive
priority function: p(a, r, t) = r
arbitration rule: random

	service time	arrival time	start	finish	T	W	R
A	10	0	0	10	10	0	1.0
B	29	1	10	39	38	9	1.3
C	3	2	39	42	40	37	13.3
D	7	3	42	49	46	39	6.6
E	12	4	49	61	57	45	4.8
mean					38.2	26	5.4

A potential problem is when a short job follows a long one:

	service time	arrival time	start	finish	T	W	R
A'	1000	0	0	1000	1000	0	1.0
B'	1	1	1000	1001	1000	999	1000.0

Gantt Chart:



Basically, long processes love FCFS, but short ones seem to be much slower.

Shortest Job Next (SJN), Shortest Job First (SJF), Shortest Process Next (SPN)

As an estimate of the total service time neded is required, this algorithm is usually used in batch systems.

decision mode: non-preemptive
priority function: $p(a, r, t) = -t$
arbitration rule: chronological or random

	service time	arrival time	start	finish	T	W	R
A	10	0	0	10	10	0	1.0
B	29	1	32	61	60	31	2.1
C	3	2	10	13	11	8	3.7
D	7	3	13	20	17	39	2.4
E	12	4	20	32	28	10	2.3
mean					25.2	17.6	2.3

Claim: Shortest Job First gives the smallest average turnaround time T out of all non-preemptive priority functions.

Proof: Suppose n jobs arrive at the same time, with $t_1 \le t_2 \le \ldots \le t_n$. Then $T(t_1) = t_1$, $T(t_2) = t_1 + t_2$, ..., hence the average turnaround time is

$$T_{av} = \sum_i i t_i$$

Now suppose t_a and t_b, $a < b$, are swapped. The new average turnaround time is:

$$T'_{av} = \frac{1}{n}(nt_1 + (n-1)t_2 + \ldots + (n-a+1)t_b + \ldots + (n-b+1)t_a + \ldots + t_n)$$

so

$$T'_{av} - T_{av} = \frac{1}{n}((n-a+1)t_b - (n-b+1)t_a + (n-a+1)t_a - (n-b+1)t_b)$$

and

$$T'_{av} - T_{av} = \frac{1}{n}(b-a)(t_b-t_a) \ge 0 \text{ because } b \ge a \text{ implies } t_b \ge t_a.$$

Problem: need to know service times into the future so you can run the process with the shortest next CPU burst. How does the short-term scheduler choose the next process to run? It can use a number of different ways:

- Most accurate is to run all ready processes, time the CPU bursts, and then schedule them (*snicker*)
- Characterize each process as CPU-bound or I/O-bound, and specify for each an “average service time needed” based upon timing processes over a period of time and averaging. Note that

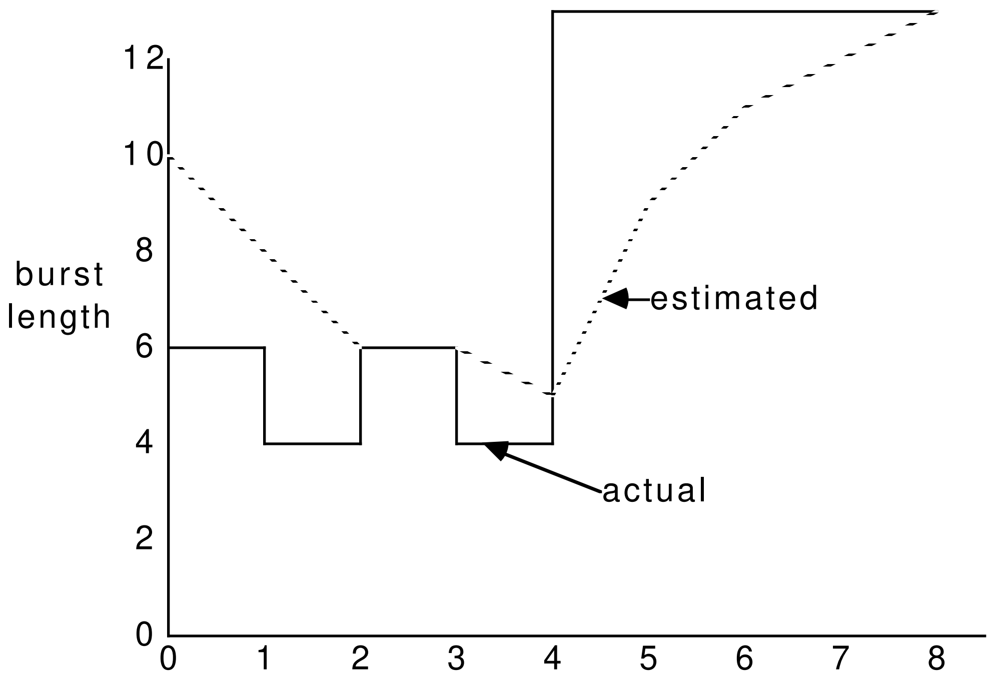
characteristics might change over a period of time; that is, a process might be CPU-bound for a time, then I/O-bound, then CPU-bound, etc.

- Compute the expected time of the next CPU-burst as an exponential average of previous CPU-bursts of the process. Let t_n be the length of the n-th CPU burst, and t_{n+1} the expected length of the next burst; then

$$t_{n+1} = a t_n + (1-a) t_n$$

where a is a parameter indicating how much to count past history (usually chosen around $\frac{1}{2}$)

$a = 1$ the estimate is simply the length of the last CPU burst
$a = 0$ the estimate is the initial estimate holds



t_i		6	4	6	4	13	13	13	13
t_i	10	8	6	6	5	9	11	12	13

Comparing exponential estimation with actual values: $a = 1/2$

SPN is better than FCFS for short jobs, but long jobs may have to wait for some time for service.

The long-term scheduler can simply use the job's time limit as specified by the user; this motivates users to be realistic in their limits, as:

- limits too low: job aborts with a “time limit exceeded”.
- limits too high: the turnaround time may be very long.

Shortest Remaining Time (SRT), Preemptive Shortest Process Next (PSPN)

This is like SPN, but preemptive.

decision mode: preemptive (at arrival)
priority function: p(a, r, t) = a–t
arbitration rule: chronological or random

	service time	arrival time	start	finish	T	W	R
A	10	0	0, 12	2, 20	20	10	2.0
B	29	1	32	61	60	31	2.1
C	3	2	2	5	3	0	1.0
D	7	3	5	12	9	2	1.3
E	12	4	20	32	28	16	2.3
mean					24	11.8	1.74

Miscellaneous:

- Whenever a new job comes in, check the remaining service time on the current job.
- For all but the longest jobs, SRT better than SJF
- The response ratio is good (low)
- Waiting time is also quite low for most processes.

Highest Response Ratio Next (HRRN, HRN)

This tries to level out bias towards long or short jobs

decision mode: non-preemptive

priority function: p(a, r, t) = a/c

arbitration rule: random or FIFO

	service time	arrival time	start	finish	T	W	R
A	10	0	0	10	20	10	2.0
B	29	1	32	61	60	31	2.1
C	3	2	2	5	3	0	1.0
D	7	3	5	12	9	2	1.3
E	12	4	20	32	28	16	2.3
mean					25.2	13	2.3

Why? Here are the response ratios as each process completes:

time	A	B	C	D	E
10		$\frac{29+9}{29}=1.3$	$\mathbf{\frac{3+8}{3}=3.7}$	$\frac{7+7}{7}=2.0$	$\frac{12+6}{12}=1.5$
13		$\frac{29+12}{29}=1.4$		$\mathbf{\frac{7+10}{7}=2.4}$	$\frac{12+9}{12}=1.8$
20		$\frac{29+19}{29}=1.7$			$\mathbf{\frac{12+16}{12}=2.3}$
32		$\mathbf{\frac{29+31}{29}=2.1}$			

The ratio used is actually

$$\frac{\text{estimated service time + waiting time so far}}{\text{estimated service time}}$$

The idea behind this method is to get the mean response ratio low, so if a job has a high response ratio, it should be run at once to reduce the mean.

Round Robin (RR) with Quantum q

This is especially designed for time sharing; the quantum is typically $\frac{1}{60} \le q \le 1$ seconds.

decision mode: preemptive (at quantum)
priority function: $p(a, r, t) = c$
arbitration rule: cyclic

In this example the quantum is 5:

	service time	arrival time	start	finish	T	W	R
A	10	0	...	28	28	18	2.8
B	29	1	...	61	60	31	2.1
C	3	2	...	13	11	8	3.7
D	7	3	...	35	32	25	4.6
E	12	4	...	47	43	31	3.5
mean					34.8	22.6	3.3

Why? Here is what things look like:

time	0	5	10	13	18	23	28	33	35	40	45	47	52	57	61
proc.	A	B	C	D	E	A	B	D	E	B	E	B	B	B	
rem	5	24	0	2	7	0	19	0	2	14	0	9	4	0	

(here, "proc" is the process starting at the indicated time, and "rem" the remaining time after the quantum is complete.)

- As each process is preempted, it moves to the rear of the queue
- All new arrivals come in at the rear of the queue
- As $q \infty 0$, every process thinks it is getting constant service from a processor that is slower in proportion to the number of competing processes; this is called *processor sharing*. This scheme is used in hardware in CDC6600 to implement 10 peripheral processors with one set of hardware (*i.e.*, processor) and 10 sets of registers; the processor does 1 instruction for one set of registers, then goes on to the next set. (This turns out to be not much slower than a real processor.)

Variants:

- Round Robin, but adjust quantum periodically.
  example: after every process switch, the quantum becomes q/n, where n is the number of processes in the ready list
  - few ready processes means that each gets a long quantum, minimizing process switches.
  - a lot of ready processes means that this algorithm gives more processes a shot at the CPU over a fixed period of time, at the price of more process switching
  - processes needing a small amount of CPU time get a quantum fairly soon, and hence *may* finish sooner.

- Round Robin, but give the current process an extra quantum when a new process arrives

  This reduces process switching in proportion to the number of processes arriving.

Selfish Round Robin (SRR) with Parameters a and b and Quantum q

This is like round robin, except newly-arrived jobs move into the *arrived queue* (or *new queue*), and assigned an initial priority of 0. Associated with the *accepted* queue is a single priority. The priority of all processes in the new queue increases at rate *a*; that of all processes in the accepted queue increases at rate *b*. When the priority of a process in the new queue is equal to the priority of the accepted queue, the process moves to the accepted queue.

decision mode: preemptive (at quantum)

priority function: here, W is the time that a process must wait before entering the accepted queue.

$$p(r, W) = \begin{cases} br & r \leq W \\ bW + a(r-W) & r > W \end{cases}$$

arbitration rule: first in, first out

In this example the quantum is 1, and a = 3, b = 2:

	service time	arrival time	start	finish	T	W	R
A	10	0	...	...	27	17	2.7
B	29	1	...	...	60	31	2.1
C	3	2	...	...	15	12	5.0
D	7	3	...	...	33	26	4.7
E	12	4	...	...	44	32	3.7
mean					35.8	23.6	3.6

Why? Assuming that new process promotions from the new queue to the ready queue precede quanta expirations, we have:

time	job running	ready queue (at end of interval)	new queue (at end of interval)
0-1	A	A(2)	B(0)
1-2	A	A(4)	B(3), C(0)
2-3	A	B(6), A(6)	C(3), D(0)
3-4	B	A(8), B(8)	C(6), D(3), E(0)
4-5	A	B(10),A(10)	C(9), D(6), E(3)
5-6	B	A(12), C(12), B(12)	D(9), E(6)
6-7	A	C(14), B(14), A(14)	D(12), E(9)
7-8	C	B(16), A(16), C(16)	D(15), E(12)
8-9	B	A(18), C(18), D(18), B(18)	E(15)
9-10	A	B(20), C(20), D(20), A(20)	E(18)
10-11	B	C(22), D(22), A(22), B(22)	E(21)
11-12	C	D(24), A(24), B(24), E(24), C(24)	

... round robin from here on ...

How a and b are chosen greatly affects the way the algorithm works:

- a = b £ p(r,W) = ar, which is the ssame as the FCFS algorithm.

- b = 0 £ p(r,W) = 0, which is the same as Round Robin.
- b $\geq$ a £ accept no new processes until all accepted processes complete

This algorithm gives better service to processes executing for a while as opposed to new ones.

Multilevel Feedback Queues (MLF, MLFB) with n different priority levels each of priority T_p

Processes start out in the uppermost level. After getting T_0 units of CPU time, it drops to the next lower level, and after units of CPU time at that level, it drops down again ..., until it reaches the lowest level. If it blocks or otherwise leaves the scheduling system, and later returns, it may reenter the feedback queues at another queue (for example, the top one).

decision mode: preemptive (at quantum)

priority function: $p(a) = n - i$, where i satisfies both $0 \le i < n$ and $T_0(2^i—1) \le a < T_0(2^{i+1}—1)$, assuming that $T_p = 2^pT_0$

arbitration rule: cyclic or chronological within queues

In this example the quantum is 1, n = 3, $T_0 = 2$, and $T_p = 2^pT_0$:

	service time	arrival time	start	finish	T	W	R
A	10	0	...	...	38	28	3.8
B	29	1	...	...	60	31	2.1
C	3	2	...	...	11	8	3.7
D	7	3	...	...	27	20	3.9
E	12	4	...	...	40	28	3.3
mean					35.2	23.3	3.4

This algorithm favors short processes by giving them more of the CPU.

It is also adaptive, in that it responds to the changing behavior of the system it controls.

Variants

- MLFB with round robin for all but the lowest level, and thatr first come first serve (but preemption possible, of course):

	service time	arrival time	start	finish	T	W	R
A	10	0	...	...	25	15	2.5
B	29	1	...	...	49	20	2.5
C	3	2	...	...	11	8	1.4
D	7	3	...	...	50	43	1.2
E	12	4	...	...	57	45	1.3
mean					38.4	26.2	1.8

Comparison of Scheduling Methods

These policies were simulated assuming an exponential distribution of arrival and service rates (with parameters of $a = 0.8$, $b = 1.0$), and using 50,000 processes (the first 100 of which were ignored, to get to a steady state). The processes were categorized into percentiles based on service time required; each percentile had about 500 processes.

External Priority Methods

These adjust priority based on some external factors, and are quite common when users pay based upon their computer use.
Examples:

- round robin, where the quantum is set independently for each process, based on the external priority of process (*i.e.*, the more you pay, the bigger the quantum.)
- *Worst Service Next*: after each quantum, compute a "suffering function" (based on how long the process had to wait, how many times it has been preempted, how much the user is paying, and/or the amount of time and resources used). The process with the greatest suffering gets the next quantum.
- The user buys a response ratio guarantee; the suffering function used takes into account the difference between the guaranteed response ratio and the actual response ratio at the moment.
- *Deadline Scheduling*: each process specifies how much service it needs and by what real time it must be finished. The algorithm tries not to run jobs that cannot meet their deadline.
- *Fair-Share Scheduling*: allocate blocks of CPU time to a particular set of processes, usually by splitting user processes into groups; within each group, use a standard schedule, but allocate the CPU proportionately to each group
  example: All processes are infinite loops; 1 process in group 1, 2 in group 2, 3 in group 3, and 4 in group 4
  regular scheduler: each process gets 10%
  fair share scheduler: each group gets 25%; processes in sgroup share equally
  example: This uses UNIX internal, not external, priorities. Here, 3 processes: process A in one group; processes B and C in another group. The internal priority function is:
  $$\text{priority} = \frac{\text{recent CPU usage}}{2} + \frac{\text{group CPU usage}}{2} + \text{threshhold}$$
  (with the threshold being 60 for user processes). A decay function decrements the current CPU usage of processes not run; this has the effect of raising their priority. The function is:
  $$\text{decay of CPU usage} = \frac{\text{CPU usage}}{2}$$
  example of the UNIX Fair Share Scheduler: Here, the quantum is 1 second. Note that the higher the priority, the lower the integer representing that priority.

A runs for 1 second
decay applied to CPU and group CPU usage; A's new priority is $60 + \frac{30}{2} + \frac{30}{2} = 90$. As B and C now have higher priority, one of them (say, B) goes next.

B runs for 1 second
decay applied to CPU and group CPU usage; A's new priority is $60 + \frac{15}{2} + \frac{15}{2} = 74$, B's new priority is $60 + \frac{30}{2} + \frac{30}{2} = 90$, and C's new priority is $60 + \frac{0}{2} + \frac{30}{2} = 75$; A has the highest priority, so it runs next.

A runs for 1 second.
decay applied to CPU and group CPU usage; A's new priority is $60 + \frac{37}{2} + \frac{37}{2} = 96$, B's new priority is $60 + \frac{15}{2} + \frac{15}{2} = 74$, and C's new priority is $60 + \frac{0}{2} + \frac{15}{2} = 67$; C has the highest priority, so it runs next.

C runs for 1 second.
decay applied to CPU and group CPU usage; A's new priority is $60 + \frac{18}{2} + \frac{18}{2} = 96$, B's new priority is $60 + \frac{7}{2} + \frac{37}{2} = 81$, and C's new priority is $60 + \frac{30}{2} + \frac{37}{2} = 93$; A has the highest priority, so it runs next.

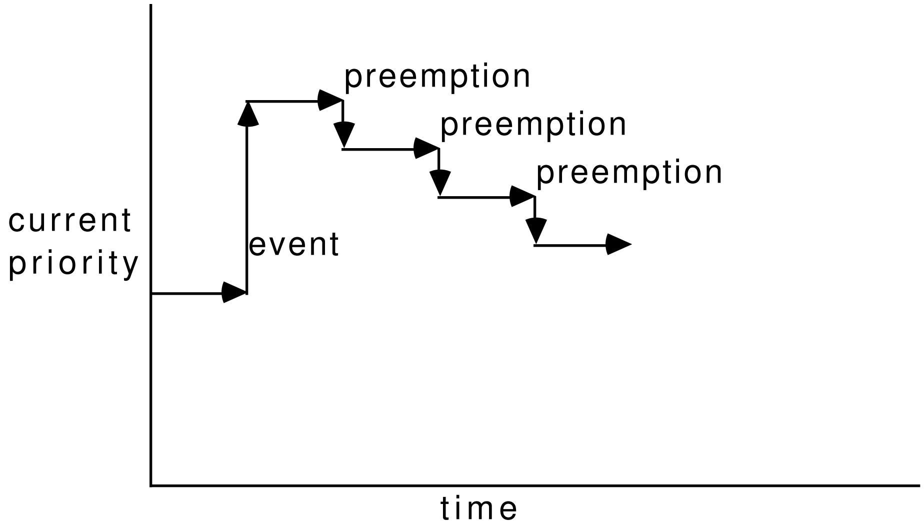
Hence the order of running is A B A C A B A C …, with A getting 50% of the CPU and B and C together getting 50%.

example: VAX/VMS scheduler

This scheduler has 32 priority levels: levels 31 to 16 are for real-time processes, and levels 15 to 0 for regular processes. Real-time processes have fixed priority throughout their lifetime; but the priority of regular processes is dynamic:

- at process creation, a *base* priority assigned; this is the process' minimum priority
- the current priority of the process is altered by a system events, each of which has an associated *increment*, *i.e.*, terminal read increment > terminal write increment > disk I/O

  When awakened due to a system event, the appropriate increment is added to the current priority value; on preemption due to quantum expiration, the current priority drops by 1.



Processes are dispatched by their current priority.
This scheme is like a MLFB scheme, with two differences:

- processes need not start at the highest level; and
- quanta are associated with each *process*, not level