

Outline for Today's Lecture

Objective for today: 2 parallel threads

- **Unix Process-oriented System Calls**
- Introduce the *scheduling policies* for choosing which process to run when.

Lecture context switch =

Timer interrupt goes off + Save current lecture state (ESC out of current powerpoint slideshow) + Load state of other lecture (choose powerpoint window) + Reset timer + Run new lecture (Click on slideshow)

Processor Scheduling

We've talked about *how*: mechanisms to support process abstraction (context switch, timer interrupts, queues of process descriptors)

We've talked about *why*: use of concurrent processes/threads in problem-solving.

Next issue: *policies* for choosing *which* process/thread, among all those ready to run, should be given the chance to run next.

Separation of Policy and Mechanism

“Why and What” vs. “How”

Objectives and strategies vs. data structures, hardware and software implementation issues.

Process abstraction vs. Process machinery

Policy and Mechanism

Scheduling policy answers the question:

Which process/thread, among all those ready to run, should be given the chance to run next?

Mechanisms are the tools for supporting the process/thread abstractions and affect *how* the scheduling policy can be implemented. (this is review)

- How the process or thread is represented to the system - process or thread control blocks.
- What happens on a context switch.
- When do we get the chance to make these scheduling decisions (timer interrupts, thread operations that yield or block, user program system calls)

CPU Scheduling Policy

The CPU scheduler makes a sequence of “moves” that determines the interleaving of threads.

- Programs use synchronization to prevent “bad moves”.
- ...but otherwise scheduling choices appear (to the program) to be *nondeterministic*.

The scheduler’s moves are dictated by a *scheduling policy*.



Scheduler Policy Goals & Metrics of Success

- *Response time* or latency (to minimize the average time between arrival to completion of requests)
How long does it take to do what I asked? (R) Arrival -> done.
- *Throughput* (to maximize productivity)
How many operations complete per unit of time? (X)
- *Utilization* (to maximize use of some device)
What percentage of time does the CPU (and each device) spend doing useful work? (U)
time-in-use / elapsed time
- *Fairness*
What does this mean? Divide the pie evenly? Guarantee low variance in response times? Freedom from starvation?
- *Meet deadlines and guarantee jitter-free periodic tasks*
real time systems (e.g. process control, continuous media)

Articulating Policies

Given some of the goals just mentioned, what kind of policies can you imagine?

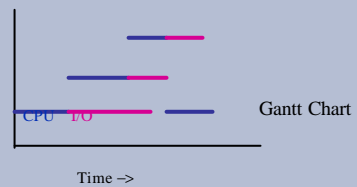
What information would you need to know in order to implement such a policy?

How would you get the information?

Multiprogramming and Utilization

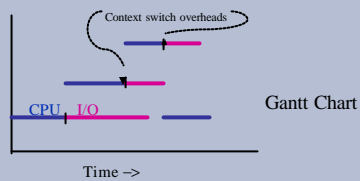
Early motivation: *Overlap* of computation and I/O

Determine *mix* and *multiprogramming level* with the goal of “covering” the idle times caused by waiting on I/O.



Multiprogramming and Utilization

Early motivation: *Overlap* of computation and I/O
Determine *mix* and *multiprogramming level* with the goal of “covering” the idle times caused by waiting on I/O.



Flavors

Long-term scheduling - which jobs get resources (e.g. get allocated memory) and the chance to compete for cycles (be on the ready queue).

Short-term scheduling or process scheduling - which of those gets the next slice of CPU time

Non-preemptive - the running process/thread has to explicitly give up control

Preemptive - interrupts cause scheduling opportunities to reevaluate who should be running now (is there a more “valuable” ready task?)

Preemption

Scheduling policies may be *preemptive* or *non-preemptive*.

- **Preemptive**: scheduler may unilaterally force a task to relinquish the processor before the task blocks, yields, or completes.
- **timeslicing** prevents jobs from monopolizing the CPU
 - Scheduler chooses a job and runs it for a *quantum* of CPU time.
 - A job executing longer than its quantum is forced to yield by scheduler code running from the clock interrupt handler.
- use preemption to honor priorities
 - Preempt a job if a higher priority job enters the *ready* state.

Priority

Some goals can be met by incorporating a notion of *priority* into a “base” scheduling discipline.

Each job in the ready pool has an associated *priority value*; the scheduler favors jobs with higher *priority values*.

External priority values:

- imposed on the system from outside
- reflect external preferences for particular users or tasks
 - “All jobs are equal, but some jobs are more equal than others.”
- **Example**: Unix **nice** system call to lower priority of a task.
- **Example**: Urgent tasks in a real-time process control system.

Manipulating Priorities

External priority (user rank, paid bribes)

Internal priorities - scheduler dynamically calculates and uses for queuing discipline. System adjusts priority values internally as an *implementation technique* within the scheduler.

Internal Priority

- Drop priority of jobs consuming more than their share
- Boost jobs that already hold resources that are in demand
e.g., *internal sleep primitive in Unix kernels*
- Boost jobs that have starved in the recent past
- Adaptive to observed behavior: typically a continuous, dynamic, readjustment in response to observed conditions and events

May be visible and controllable to other parts of the system

Priority reassigned if I/O bound (large unused portion of quantum) or if CPU bound (nothing left)

Keeping Your Priorities Straight

Priorities must be handled carefully when there are dependencies among tasks with different priorities.

- A task with priority P should never impede the progress of a task with priority $Q > P$.

This is called *priority inversion*, and it is to be avoided.

- The basic solution is some form of *priority inheritance*.

When a task with priority Q waits on some resource, the holder (with priority P) temporarily inherits priority Q if $Q > P$.

Inheritance may also be needed when tasks coordinate with IPC.

- Inheritance is useful to meet deadlines and preserve low-jitter execution, as well as to honor priorities.

Pitfalls: Mars Pathfinder Example

In July 1997, Pathfinder's computer reset itself several times during data collection and transmission from Mars.

- One of its processes failed to complete by a deadline, triggering the reset.

Priority Inversion Problem.

- A low priority process held a mutual exclusion semaphore on a shared data structure, but was preempted to let higher priority processes run.
- The higher priority process which failed to complete in time was blocked on this semaphore.
- Meanwhile a bunch of medium priority processes ran, until finally the deadline ran out. The low priority semaphore-holding process never got the chance to run again in that time to get to the point of releasing the semaphore
- *Priority inheritance* had not been enabled on semaphore.

Can you imagine debugging this?

Scheduling Algorithms

SJF - Shortest Job First (provably optimal in minimizing average response time, assuming we know service times in advance)

FIFO, FCFS

Round Robin

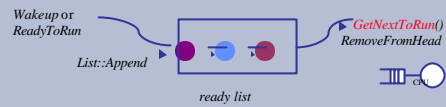
Multilevel Feedback Queuing

Priority Scheduling

A Simple Policy: FCFS

The most basic scheduling policy is *first-come-first-served*, also called *first-in-first-out* (FIFO).

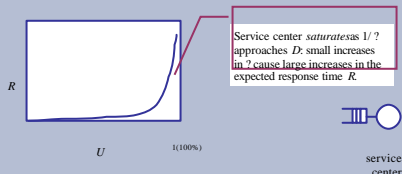
- FCFS is just like the checkout line at the QuickiMart.
 - Maintain a queue ordered by time of arrival.
 - GetNextToRun* selects from the front of the queue.
- FCFS with preemptive timeslicing is called *round robin*.



Behavior of FCFS Queues

Assume: stream of normal task arrivals with mean arrival rate λ . Tasks have normally distributed service demands with mean D .

Then: Utilization $U = \lambda D$ (Note: $0 \leq U < 1$)
Probability that service center is idle is $1 - U$.
"Intuitively", $R = D / (1 - U)$



Little's Law

For an unsaturated queue in steady state, queue length N and response time R are governed by:

Little's Law: $N = \lambda R$

While task T is in the system for R time units, λR new tasks arrive. During that time, N tasks depart (all tasks ahead of T). But in steady state, the flow in must balance the flow out.
(Note: this means that throughput $X = \lambda$.)

Little's Law gives response time $R = D / (1 - U)$.

Intuitively, each task T 's response time $R = D + DN$.
Substituting λR for N : $R = D + D \lambda R$
Substituting U for λD : $R = D + UR$
 $R - UR = D \rightarrow R(1 - U) = D \rightarrow R = D / (1 - U)$

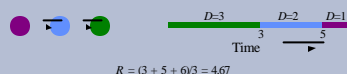
Why Little's Law Is Important

- Intuitive understanding of FCFS queue behavior.
 - Compute response time from demand parameters (λ, D).
 - Compute N : tells you how much storage is needed for the queue.
- Notion of a *saturated* service center. If $D=1, R=1/(1-\rho)$
 - Response times rise rapidly with load and are unbounded.
 - At 50% utilization, a 10% increase in load increases R by 10%.
 - At 90% utilization, a 10% increase in load increases R by 10x.
- Basis for predicting performance of *queuing networks*.
 - Cheap and easy "back of napkin" estimates of system performance based on observed behavior and proposed changes, e.g., *capacity planning*, "what if" questions.

Evaluating FCFS

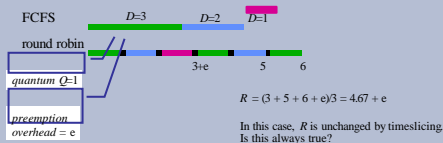
How well does FCFS achieve the goals of a scheduler?

- throughput*. FCFS is as good as any non-preemptive policy.
 - ...if the CPU is the only schedulable resource in the system.
- fairness*. FCFS is intuitively fair...sort of.
 - "The early bird gets the worm"...and everyone else is fed eventually.
- response time*. Long jobs keep everyone else waiting.



Preemptive FCFS: Round Robin

- Preemptive timeslicing is one way to improve fairness of FCFS.
- If job does not block or exit, force an involuntary context switch after each quantum Q of CPU time.
 - Preempted job goes back to the tail of the ready list.
 - With infinitesimal Q round robin is called *processor sharing*.



Evaluating Round Robin



- Response time*. RR reduces response time for short jobs.
 - For a given load, a job's wait time is proportional to its D .
- Fairness*. RR reduces variance in wait times.
 - But: RR forces jobs to wait for other jobs that arrived later.
- Throughput*. RR imposes extra context switch overhead.
 - CPU is only $Q/(Q+e)$ as fast as it was before.
 - Degrades to FCFS with large Q .

Q is typically 1-100 milliseconds.
 e is 1-5 μ s in 1998.

What can happen if a thread is listed twice in the list?

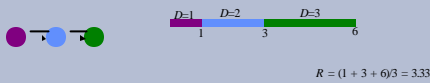
Minimizing Response Time: SJF

Shortest Job First (SJF) is provably optimal if the goal is to minimize R .

Example: express lanes at the MegaMart

Idea: get short jobs out of the way quickly to minimize the number of jobs waiting while a long job runs.

Intuition: longest jobs do the least possible damage to the wait times of their competitors.



SJF

In preemptive case, *shortest remaining time first*.

In practice, we have to predict the CPU service times (computation time until next blocking).

Favors interactive jobs, needing response, & repeatedly doing user interaction

Favors jobs experiencing I/O bursts - soon to block, get devices busy, get out of CPU's way

Focus is on an *average* performance measure, some long jobs may starve under heavy load/ constant arrival of new short jobs.

Behavior of SJF Scheduling

- With SJF, best-case R is not affected by the number of tasks in the system.

Shortest jobs budge to the front of the line.

- Worst-case R is unbounded, just like FCFS.

Since the queue is not "fair", starvation exists - the longest jobs are repeatedly denied the CPU resource while other more recent jobs continue to be fed.

- SJF sacrifices fairness to lower *average* response time.

SJF in Practice

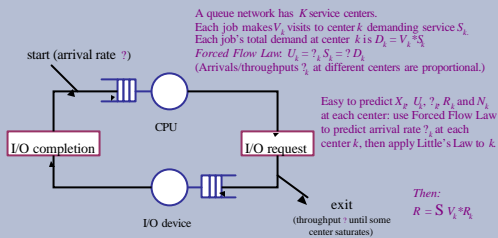
Pure SJF is impractical: scheduler cannot predict D values.

However, SJF has value in real systems:

- Many applications execute a sequence of short CPU bursts with I/O in between.
- E.g., *interactive* jobs block repeatedly to accept user input.
Goal: deliver the best response time to the user.
- E.g., jobs may go through periods of I/O-intensive activity.
Goal: request next I/O operation ASAP to keep devices busy and deliver the best overall throughput.
- Use *adaptive internal priority* to incorporate SJF into RR.
Weather report strategy: predict future D from the recent past.

Considering I/O

In real systems, overall system performance is determined by the interactions of multiple service centers.



Digression: Bottlenecks

It is easy to see that the maximum throughput X of a system is reached as $1/?_k$ approaches D_k for service center k with the highest demand D_k .

k is called the **bottleneck center**

Overall system throughput is limited by $?_k$ when U_k approaches 1.



This job is *I/O bound*. How much will performance improve if we double the speed of the CPU?
 Is it worth it?

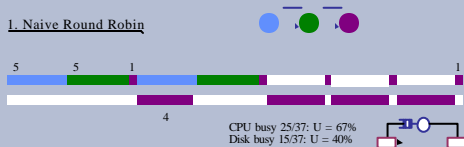
To improve performance, always attack the bottleneck center!



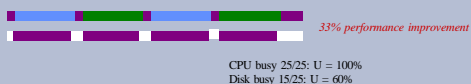
Demands are evenly balanced. Will multiprogramming improve system throughput in this case?

Two Schedules for CPU/Disk

1. Naive Round Robin



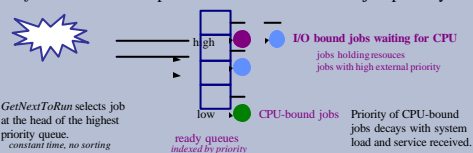
2. Round Robin with SJF



Multilevel Feedback Queue

Many systems (e.g., Unix variants) implement priority and incorporate SJF by using a *multilevel feedback queue*.

- multilevel.** Separate queue for each of N priority levels.
 Use RR on each queue; look at queue $i-1$ only if queue i is empty.
- feedback.** Factor previous behavior into new job priority.



Beyond “Ordinary” Uniprocessors

Multiprocessors

- Co-scheduling and gang scheduling
- Hungry puppy task scheduling
- Load balancing

Networks of Workstations

- Harvesting Idle Resources - remote execution and process migration

Laptops and mobile computers

- Power management to extend battery life, scaling processor speed/voltage to tasks at hand, sleep and idle modes.