CPU Scheduling
CPU Scheduling 101

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 Goals

• *response time* or latency
  How long does it take to do what I asked? \((R)\)

• **throughput**
  How many operations complete per unit of time? \((X)\)

  *Utilization*: what percentage of time does the CPU (and each device) spend doing useful work? \((U)\)

• **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*
  predictability
Outline

1. the CPU scheduling problem, and goals of the scheduler
2. scheduler “add-ons” used in many CPU schedulers.
   • priority (internal vs. external)
   • preemption
3. fundamental scheduling disciplines
   • FCFS: first-come-first-served
   • SJF: shortest-job-first
4. practical CPU scheduling
   multilevel feedback queues: using internal priority to create a hybrid of FIFO and SJF.
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.
Internal Priority

*Internal priority*: system adjusts priority values internally as an *implementation technique* within the scheduler.

- improve fairness, resource utilization, freedom from starvation
- 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
- typically a continuous, dynamic, readjustment in response to observed conditions and events
  - may be visible and controllable to other parts of the system
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.
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*.

```
Wakeup or ReadyToRun
List::Append

ready list

GetNextToRun()
RemoveFromHead
```

CPU
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.

\[
R = \frac{3 + 5 + 6}{3} = 4.67
\]
Behavior of FCFS Queues

Assume: stream of task arrivals with mean arrival rate \( ? \).

\[ \text{Poisson distribution: exponentially distributed inter-arrival gap.} \]

At any time, average time to next arrival is \( 1/ ? \).

Tasks have normally distributed service demands with mean \( D \), i.e., each task requires \( D \) units of time at the service center to complete.

Then: Utilization \( U = ?D \) (Note: \( 0 <= U <= 1 \))

Probability that service center is busy is \( U \), idle is \( 1-U \).

“Intuitively”, \( R = D/(1-U) \)
**Little’s Law**

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

**Little’s Law: $N = ?R$.**

While task $T$ is in the system for $R$ time units, $?R$ 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 = ?)*

Little’s Law just says that the average, steady-state number of jobs or items queued for service in the system is the *delay/bandwidth product* familiar from networking.
Response Time and Utilization

Little’s Law gives response time as:

\[ R = \frac{D}{1 - U} \]

Each task’s response time is \( R = D + DN \).
You have to wait for everyone before you to get service.

Substituting \( R \) for \( N \) (Little’s Law): \( R = D + D \cdot R \)
Substituting \( U \) for \( D \) (by definition): \( R = D + UR \)
\[ R - UR = D \]
\[ R(1 - U) = D \]
\[ R = \frac{D}{1 - U} \]
Why Little’s Law Is Important

1. Intuitive understanding of FCFS queue behavior.
   
   Compute response time from demand parameters (? , D).
   
   Compute N: tells you how much storage is needed for the queue.

2. Notion of a saturated service center.
   
   If D=1: \( R = \frac{1}{1 - ?} \)
   
   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.

   
   Cheap and easy “back of napkin” estimates of system
   performance based on observed behavior and proposed
   changes, e.g., capacity planning, “what if” questions.
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.

**FCFS-RTC**

$D=3$  $D=2$  $D=1$

round robin

$3+e$  $5$  $6$

**quantum $Q=1$**

**preemption overhead = $e$**

$$R = (3 + 5 + 6 + e)/3 = 4.67 + e$$

In this case, $R$ is unchanged by timeslicing.

Is this always true?
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-RTC with large $Q$.

<table>
<thead>
<tr>
<th>$D=5$</th>
<th>$D=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R = (5+6)/2 = 5.5$</td>
<td>$R = (2+6 + e)/2 = 4 + e$</td>
</tr>
</tbody>
</table>

$Q$ is typically 5-100 milliseconds; $e$ is on the order of μs
Digression: RR and System Throughput II

On a multiprocessor, RR may improve throughput under light load:

• *The scenario*: three salmon steaks must cook for 5 minutes per side, but there’s only room for two steaks on the hibachi.
  30 minutes worth of grill time needed: steaks 1, 2, 3 with sides A and B.
• *FCFS-RTC*: steaks 1 and 2 for 10 minutes, steak 3 for 10 minutes.
  Completes in 20 minutes with grill utilization a measly 75%.
• *RR*: 1A and 2A...flip...1B and 3A...flip...2B and 3B.
  Completes in three quanta (15 minutes) with 100% utilization.
• RR may speed up parallel programs if their inherent parallelism is poorly matched to the real parallelism.
  E.g., 17 threads execute for $N$ time units on 16 processors.
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.

$$R = \frac{1 + 3 + 6}{3} = 3.33$$
Behavior of SJF Scheduling

Little’s Law *does not hold* if the scheduler considers *a priori* knowledge of service demands, as in SJF.

- 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.
  
  The queue is not “fair”, this is *starvation*: 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.

A queue network has \( K \) service centers.
Each job makes \( V_k \) visits to center \( k \) demanding service \( S_k \).
Each job’s total demand at center \( k \) is \( D_k = V_k * S_k \).

**Forced Flow Law:** \( U_k = V_k S_k = D_k \)
(Arrivals/throughputs \( V_k \) at different centers are proportional.)

Easy to predict \( X_k \), \( U_k \), \( V_k \), \( R_k \) and \( N_k \) at each center: use Forced Flow Law to predict arrival rate \( V_k \) at each center \( k \), then apply Little’s Law to \( k \).

Then:
\[
R = S \cdot V_k \cdot R_k
\]
**Digression: Bottlenecks**

It is easy to see that the maximum throughput $X$ of a system is reached as $1/\?\approx \frac{1}{D_k}$ 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.

**Example 1:**

<table>
<thead>
<tr>
<th>CPU</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S_0 = 1$

$S_1 = 4$

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!*

---

**Example 2:**

<table>
<thead>
<tr>
<th>CPU</th>
<th>I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$S_0 = 4$

$S_1 = 4$

Demands are evenly balanced. Will multiprogramming improve system throughput in this case?
Two Schedules for CPU/Disk

1. Naive Round Robin

CPU busy 25/37: U = 67%
Disk busy 15/37: U = 40%

2. Round Robin with SJF

CPU busy 25/25: U = 100%
Disk busy 15/25: U = 60%

33% performance improvement
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.

*GetNextToRun* selects job at the head of the highest priority queue. 
*constant time, no sorting*

**I/O bound jobs waiting for CPU**
- jobs holding resources
- jobs with high external priority

**CPU-bound jobs**

Priority of CPU-bound jobs decays with system load and service received.

**ready queues** indexed by priority
Seeing the Future

What if the system has knowledge of future demands on the scheduler?

- Task arrival times
- Task service demands
- Task deadlines (soft/hard real-time, or QoS)

Applications may have this knowledge:

- Servers: request classes, protocol-defined exchanges
- Regular, periodic tasks
- Aperiodic events with known processing times

How can the scheduler exploit this information?

- Push policy up, or push knowledge down?
Rialto

Real-time schedulers must support regular, periodic execution of tasks (e.g., continuous media).

Microsoft’s Rialto scheduler [Jones97] supports an external interface for:

• **CPU Reservations**
  
  “I need to execute for $X$ out of every $Y$ units.”

  Scheduler exercises *admission control* at reservation time: application must handle failure of a reservation request.

• **Time Constraints**

  “Run this before my *deadline* at time $T$; it won’t take more than $X$ time units.”
A Rialto Schedule

Rialto schedules constrained tasks according to a static task graph.

- For each *base period*, pick a path from root to a leaf.
  
  At each visited node, execute associated task for specified time $t$.

- Visit subsequent leaves in subsequent base periods.

- Modify the schedule only at request time.

![Diagram of a Rialto Schedule](image)
Rialto Questions

How do apps know how long actions will take?
- Sources of variability
- What if $X$ isn’t enough? How does the app find out?
- What if $X$ is longer than needed?

How should the app respond if an allocation request fails?

Could we use this approach for other resources?

What about high-priority aperiodic tasks, or variable execution times?

What about execution in the kernel (e.g., net processing)?
More Rialto Questions

Are reservations and constraints “fair”?
Reservations are not guaranteed for activities that block.
   Why not? What problems does this cause?
Why is constraint inheritance useful/necessary?
Why are constraints “better” than the Unix way to improve interactive response (e.g., Figure 5.5).
What if unforeseen circumstances force the scheduler to break a promise?
Compare to proportional share or fixed-share reservations.
Lottery Scheduling

Lottery scheduling [Waldspurger96] is another scheduling technique.

Elegant probabilistic approach proportional share allocation, but does not guarantee deadlines.

- Give \( W_p \) “lottery tickets” to each process \( p \).
- \( GetNextToRun \) selects “winning ticket” randomly.
  
  If \( SW_p = N \), then each process gets CPU share \( W_p/N \) ...
  ...
  ...probabilistically, and over a sufficiently long time interval.

- \( \textbf{Flexible} \): tickets are transferable to allow application-level adjustment of CPU shares.

- Simple, clean, fast.

Random choices are often a simple and efficient way to produce the desired overall behavior (probabilistically).