Introduction

When multiple tasks need to be done on a single CPU, we need to figure out a way to distribute the work done by the CPU across all of the tasks.

At any point there are **running, waiting, and blocked threads.** A processor's scheduling policy determines how and when threads transition between these states.

Scheduling Goals

There are three primary goals for an effective scheduling algorithm.

1. **Minimize Response Time:** reduce the amount of total elapsed time required to do any one job. For example, time might be measured from a user's keypress to the event being triggered. If a job has to wait for others to finish first, then response time increases.
2. **Maximize Throughput:** maximize the number of operations per second. This occurs when as much time is spend on the jobs themselves, rather than on scheduling/context switching.
3. **Maximize Fairness:** The concept of fairness is somewhat vague, but typically shorter jobs with more IO-bound operations should go first, and time should be somewhat equally distributed across jobs depending on priority.

No single scheduling policy can achieve every goal; there must be some tradeoffs. For example, minimizing response time results in more context switching, so throughput cannot also be maximized.

Vocabulary

Workload: the input to a scheduling algorithm, which includes the set of tasks to perform, when they arrive, and how long they will take.

Compute-bound tasks primarily use the CPU, whereas **IO-bound** tasks spend most of their time blocked by IO and only a small amount of time using the CPU.

Preemption is the process of interrupting a running thread to allow for the scheduler to decide which thread runs next.

Priority Inversion occurs when a higher priority thread is blocking on a resource (e.g. a lock) that a lower priority resource holds. This may cause **starvation** (when a thread waits indefinitely for another).

Priority donation attempts to solve priority inversion by swapping priorities of two threads when priority inversion occurs. This must be done recursively through all chains of dependencies, such that each thread ends with their priority being the max of all donated priorities and its own priority.

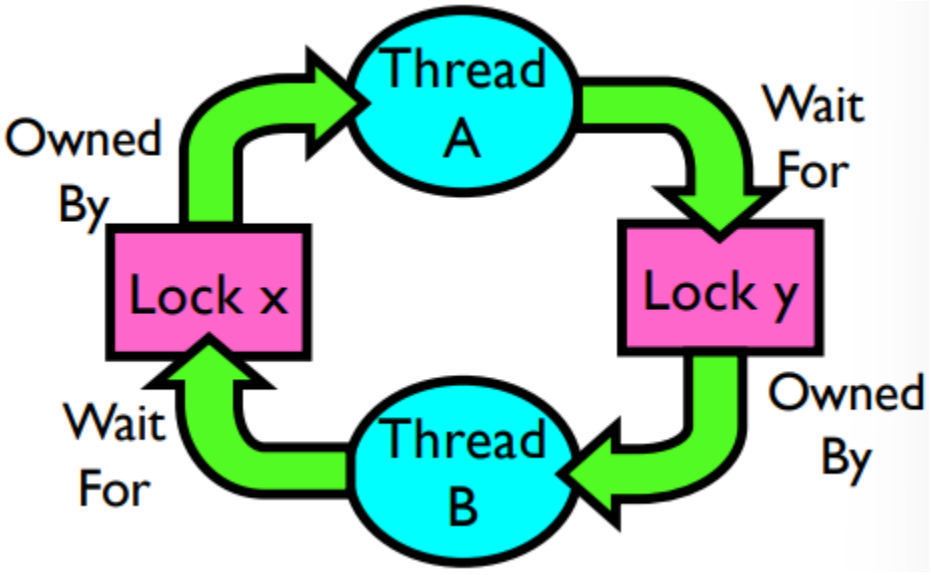
Deadlock is a specific type of starvation where two or more threads are circularly waiting for each other. For more info, see the section below.

Deadlock

As an example of deadlock, consider two threads with two locks, `x` and `y`:

```
// THREAD A     // THREAD B
x.Acquire();    y.Acquire();
y.Acquire();    x.Acquire();
y.Release();    x.Release();
x.Release();    y.Release();
```

If Thread A acquires `x` and Thread B acquires `y`, then neither can run the subsequent line because there is a circular wait:



There are **four requirements for deadlock to occur:**

1. **Mutual exclusion:** only one thread at a time can use a particular resource.
2. **Hold and wait:** A thread holding at least one resource is waiting to acquire additional resources from other threads.
3. **No preemption:** Resources are only released when a thread is finished with it.
4. **Circular wait:** There exists a set of threads such that T1T_1T1 is waiting for T2T_2T2, which is waiting for T3T_3T3, and so on, until TnT_nTn is waiting for T1T_1T1.

To avoid deadlock, we can use **Banker's Algorithm** to simulate the allocation for resources, and decide if the current available resources is sufficient for completing the current state of the program.

- To detect safe state, we can pick any thread to simulate a run to completion (i.e. release all of its resources). If there are no threads available to choose that can be run with the current allocation, then the program is in an unsafe state.
- An *unsafe* state is different from a *deadlocked* state in that it is still possible to recover from one (i.e. a deadlock is sure to happen at some point at the future, but not now).
- The two resource requests that need to be checked for are `sema_down` and `lock_acquire`.

Scheduling Algorithms

The following policies are primarily concerned with uniprocessor scheduling (how to schedule tasks onto a single CPU).

First-In-First-Out (FIFO)

Also known as First-Come-First-Served (FCFS), this is a very simple scheduling algorithm that runs processes until completion, then chooses the next process on the queue and repeats.

The major drawback with FIFO is that the effectiveness depends on the order that jobs enter the queue: if the short jobs go first, then it works well; however, if long jobs take up all of the CPU time, average waiting time and completion time suffer significantly.

FCFS is prone to starvation because if a task at the front of the queue never yields, then tasks at the back will never run.

Round Robin (RR)

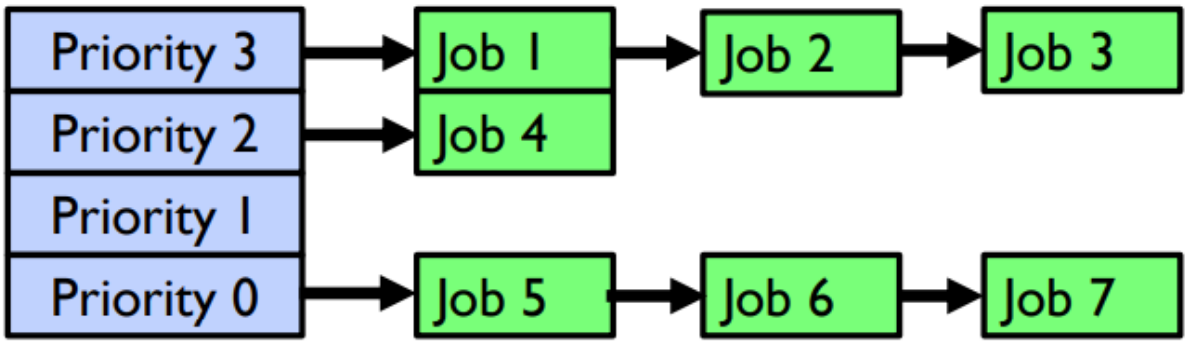
In Round Robin scheduling, each process gets a **quantum** of CPU time (typically 10-100ms). When the quantum expires, the process is preempted and moved to the back of the ready queue. The performance depends on the value of qqq (the time quantum):

- If qqq is infinite, Round Robin becomes FCFS (since jobs will always complete before the end of their quantum).
- If qqq is large, then response time decreases since jobs take a long time to get scheduled.
- If qqq is very small, then more time will be spent on context switching than executing jobs themselves, resulting in low throughput.

Round Robin is not prone to starvation because it is guaranteed that a process will wait at most $(N-1) * Q$ time to run, where N is the total number of processes and Q is the length of the quantum.

Strict Priority Scheduling

Strict Priority Scheduling allows for the incoming job queue to be split across multiple priority levels. Higher priority jobs are always run before lower priority jobs.



Individual queues of jobs with the same priority can be processed with another scheduling algorithm (for example, RR).

The major problem with strict priority scheduling is **deadlock:** if a lower priority task has a lock required by a high priority task, and some intermediate priority task is blocking the high priority task from running, then a circular wait dependency is created and the system cannot continue.

Another problem is **starvation** of lower priority jobs that may not ever run due to always being superceded by higher priority tasks.

Some solutions to deadlock and starvation:

- Dynamic priorities: shift priorities of tasks based on some heuristic
- Priority donation/inversion
- **Niceness:** In UNIX, each process has a nice value (-20 to 19). Higher nice-value tasks yield to other tasks more often, so tasks that suffer from starvation can be set as not-nice.

Shortest Job First (SJF)

Also known as Shortest Remaning Time First (SRTF).

If we somehow know how long each job will take, we can schedule the task with the least remaining work first to minimize the average response time.

While this is optimal in terms of response time, in practice it is difficult to know exactly how long something will take.

Additionally, SJF makes starvation more likely (since short tasks may depend on long tasks), and context switches are more frequent.

Prediction

Although we cannot know exactly how long a job will take, we can predict approximately how long it is:

- Program behavior is usually predictable, so if a program was IO bound or had some certain scheduling patterns in the past, it should continue to have these patterns.
- Burst length: use an estimator function to aggregate past CPU bursts from this program.

Lottery Scheduling

Another alternative scheduling algorithm is lottery scheduling, where each job is given some number of lottery tickets.

On each time slice, randomly pick a winning job to run. This ensures that CPU time is proportional to the number of tickets given to each job.

In order to approximate SJF, short running jobs get more tickets than long running jobs.

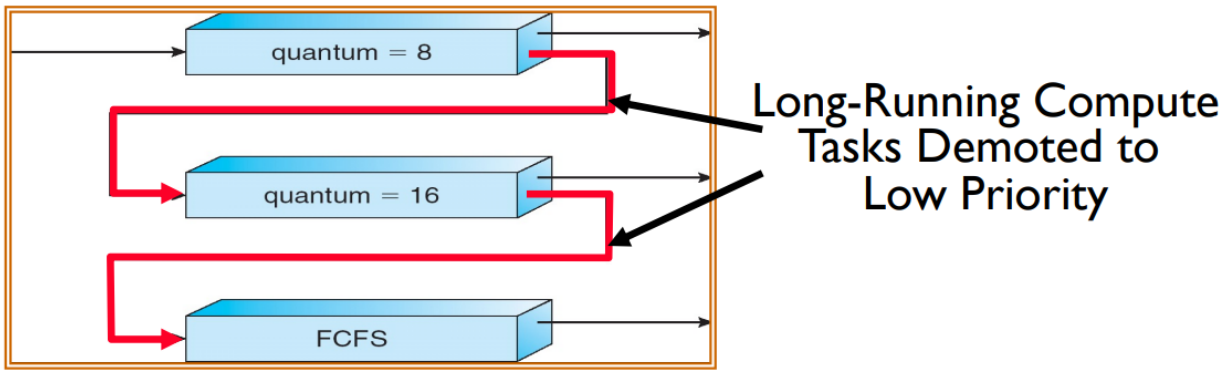
Multi-Level Feedback Scheduling

Many modern scheduling algorithms are based on a **multi-level feedback queue scheduler (MLFQS).**

Each queue has a different priority (typically for foreground vs. background tasks), and every queue has its own scheduling algorithm.

When jobs enter the MLFQ, they start in the highest priority queue. If they do not complete before a timeout, then they drop one level; otherwise, they stay at the same level or move up one level if available.

This is a good approximation of SRTF since long CPU-bound tasks will drop to the bottom of the queue.



Multi-Core Scheduling

Algorithmically, multi-core scheduling is not all too different from single core scheduling.

Mos commercial operating systems use a per-processor data structure, in which each CPU has its own task queue.

To maximize cache reuse, **affinity scheduling** can be used to prioritize scheduling yielded tasks onto the same CPU.

Rebalancing occurs when the length of a CPU's queue is long enough to justify moving the task to another CPU.

Spinlocks can be used for multiprocessing when disabling interrupts are insufficient. (Other threads on other CPUs can still run concurrently even if one CPU has interrupts disabled.)

```
int value = 0;
Acquire() { while (testAndSet(&value)) {}; // busy wait }
Release() { value = 0; }
```

Spinlocks are inefficient for long periods of waiting, but for brief (context switch level of speed) moments we can use this to ensure locks are acquired successfully before continuing.

Gang Scheduling

When multiple threads need to work together for a task, it makes sense to scheduling them together to make spin waiting more efficient.

Choosing the Right Scheduler

I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness (Wait Time to Get CPU)	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

Source: https://notes.bencuan.me/cs162/chapter-6-scheduling/