Survey of Scheduling Algorithms

- From last time: jobs have different *states.*

- How do we choose among jobs that are ready to run?
  - The *scheduler* runs to decide which job is next.

- There are two major classes of scheduling systems
  - in preemptive systems, the scheduler can interrupt a job and force a context switch
  - in non-preemptive systems, the scheduler waits for the running job to explicitly (voluntarily) block

First Come First Serve

- Run jobs in order of arrival.
- Usually non-preemptive
  - No context switching at supermarket!
- Jobs treated equally, no starvation
  - Except possibly for infinitely long jobs
- The bad: average response time and turnaround time can be large
    - e.g., small jobs waiting behind long ones
    - results in high turnaround time

Round Robin

- Each process gets equal share of CPU
  - Must choose time slice (quantum) carefully!
  - Can give different processes different times to express priority.
- Reasonably fair, short jobs get through quickly.
- The bad:
  - If quantum too short, what happens?
  - Treats all jobs equally. . .Lots of jobs = degraded service.

Shortest Job First

- Run the job that has the shortest processing time.
- Optimal solution—provably shortest average flow time.
- The bad:
  - Requires future knowledge! (how do we know how much time is left?)
  - Long jobs wait.

Shortest Remaining Processing Time

- Shortest job first with preemption—can interrupt current job if a shorter one comes in.
- Improves SJF, minimizes flow time, response time, waiting time.
- The bad:
  - Same as SJF, with addition of time due to preemptions.

Shortest Elapsed Time

- Run job that has run the least so far.
- Gets short jobs out fast.
- Flow time variance low.
- The bad:
  - Extremely high overhead, especially if the processes are highly skewed.

Foreground / Background

- Two queues: foreground and background.
- Interactive processes go to foreground queue, cpu-bound processes go into background.
- Any scheduling technique can be used for the queues.
  - Typically involves timeslices.
- An *Adaptive Scheduling Algorithm*: changes priority of job depending on its behavior.

Multilevel Foreground / Background

- Like foreground/background, but has multiple queues.

- Jobs are assigned to queues based on any kind of criteria.
  - Priority
  - Runtime
  - Etc. . .

- Jobs can be put on any level.

Exponential Queue

- Also known as Multi-Level Feedback Queues.
- New process has high priority and short time slice.
  - Decrease priority and increase timeslice after process uses up quantum.
- Multiple queues representing different job types
  - batch, interactive, system, CPU-bound, etc.
- Queues have priorities
  - schedule jobs within a queue using RR
- Jobs move between queues based on execution history
  - “feedback”: switch from CPU-bound to interactive behavior

UNIX Scheduling

- Canonical scheduler uses a MLFQ
  - 4 classes spanning 128 priority levels
    - timesharing: first 60 priorities
    - system: next 40 priorities
    - real-time: next 60 priorities
  - priority scheduling across queues, RR within
    - process with highest priority always run first
    - processes with same priority scheduled RR
  - processes dynamically change priority
    - increases over time if process blocks before end of quantum
    - decreases if process uses entire quantum
- Priority: PUSER + PCPU + PNICE
- Goals:
  - reward interactive behavior over CPU hogs
    - interactive jobs typically have short bursts of CPU

Synchronization

- How do we control access to shared resources between concurrently running processes?
  - basic problem:
    - two concurrent threads are accessing a shared variable
    - if the variable is read/modified/written by both threads, then access to the variable must be controlled
    - otherwise, unexpected results may occur
- Must assume threads interleave executions arbitrarily and at different rates
  - scheduling is not under application writers' control
- We control cooperation using synchronization
  - enables us to restrict the interleaving of executions
- Vocab:
  - *Atomic operation* – an uninterruptable operation that happens either in its entirety, or not at all.
  - *Synchronization* – using atomic operations to ensure correct operation of cooperating processes.
  - *Mutual exclusion* – mechanisms to ensure only one process is doing certain things at on time.
  - *Critical section* – A sequence of operations only one process may be executing at a given time.

Another Example

- What if two people on the same account withdraw simultaneously from ATM machines?
  - Two processes on the bank mainframe start simultaneously:

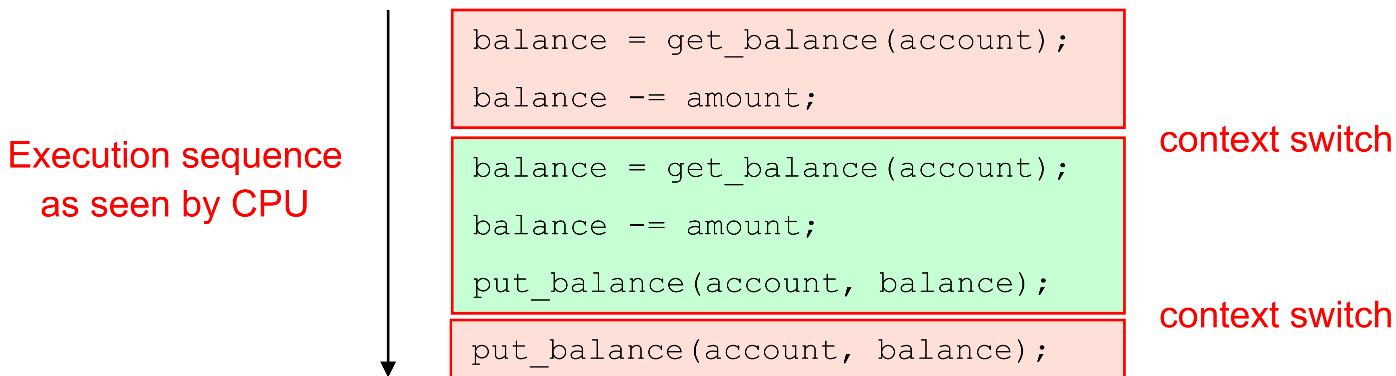
```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

```
int withdraw(account, amount) {
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  return balance;
}
```

- Is this a problem?

Another Example 2

- The problem is that the execution of the two threads can be interleaved, assuming preemptive scheduling:



- What's the account balance after this sequence?
  - who's happy, the bank or you? ;)

What is the problem?

- We want to use mutual exclusion to synchronize access to shared resources
- Code that uses mutual exclusion to synchronize its execution is called a critical section
  - only one thread at a time can execute in the critical section
  - all other threads are forced to wait on entry
  - when a thread leaves a critical section, another can enter
- Critical sections have the following requirements
  - mutual exclusion
    - at most one thread is in the critical section
  - progress
    - if thread T is outside the critical section, then T cannot prevent thread S from entering the critical section
  - bounded waiting (no starvation)
    - if thread T is waiting on the critical section, then T will eventually enter the critical section
      - assumes threads eventually leave critical sections
  - performance
    - the overhead of entering and exiting the critical section is small with respect to the work being done within it

How do we solve it?

- Locks
  - very primitive, minimal semantics; used to build others
- Semaphores
  - basic, easy to get the hang of, hard to program with
- Monitors
  - high level, requires language support, implicit operations
  - easy to program with; Java "`synchronized()`" as example
- Messages
  - simple model of communication and synchronization based on (atomic) transfer of data across a channel
  - direct application to distributed systems

Locks

- A lock is a object (in memory) that provides the following two operations:
  - acquire( ): a thread calls this before entering a critical section
  - release( ): a thread calls this after leaving a critical section
- Pair up calls to acquire( ) and release( )
  - between acquire( ) and release( ), the thread holds the lock
  - acquire( ) does not return until the caller holds the lock
    - at most one thread can hold a lock at a time (usually)
  - so: what can happen if the calls aren't paired?
- Two basic flavors of locks
  - spinlock
  - blocking (a.k.a. "mutex")

Example with Locks

- Does this work?

```
int withdraw(account, amount) {
  acquire(lock);
  balance = get_balance(account);
  balance -= amount;
  put_balance(account, balance);
  release(lock);
  return balance;
}
```

critical section

```
acquire(lock)
balance = get_balance(account);
balance -= amount;
```

```
acquire(lock)
```

```
put_balance(account, balance);
release(lock);
```

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
release(lock);
```