CPU Scheduling

- Have many threads on ready list
  - need to choose one to execute
- When does kernel get to choose?
  - when a running thread blocks
  - when a running thread is preempted
  - when a running thread terminates
  - possibly, when a waiting thread moves to ready list
  - possibly, on any entry to the kernel

CPU bound vs. I/O bound Threads

- CPU bound
  - tend to have long CPU bursts
  - example: matrix multiplication
- I/O bound
  - tend to have short CPU bursts
  - example: netscape

Possible Goals of Scheduling

- Maximize throughput
  - number of threads completed per unit time
- Minimize turnaround time
  - how long does my thread take to execute?
- Minimize response time
  - amount of time until thread sees *some* result
- Fairness
- Predictability

These goals conflict! Which goals to optimize depend on system.

FCFS (First Come First Served)

- First thread to request CPU gets it
- Non-preemptive: run until done or blocked

- Characteristics:
  - Simple
  - Easy to implement
  - Short jobs get stuck behind large jobs

Round Robin

- Each thread gets a quantum (“time slice”)
  - when it’s up, thread gets preempted and put on tail of ready queue
- If quantum very large:
  - Round Robin behaves as does FCFS
- If quantum very small:
  - Spend all time context switching (hurts throughput)
- OS might try to spend 1% of time switching

Shortest Job First/Shortest Remaining Time First

- Idea: get short jobs out of the system ASAP
- Threads that take the least time execute first
- SJF nonpreemptive, STRF premptive

- Characteristics:
  - Optimal for avg. turnaround time and throughput
  - Unfair
  - Need to predict the future -- how?
  - If all threads take same time, SJF == FCFS

Priority Scheduling

- Priority associated with each thread
  - run highest priority thread next
- SJF is special case of priority scheduling
- Characteristics:
  - Unfair
  - How to compute priorities in general?

Multilevel Feedback Queues

- Use past to predict future
- Have some number of queues
  - threads move between queues, depending on their behavior
- Design parameters
  - how many queues
  - scheduling algorithm for each queue
  - when to move
- Variant of this used in UNIX

UNIX 4.4 Scheduling

- Use multilevel queuing
  - Desire: good response time for interactive jobs w/o starving compute-bound jobs
  - Uses pre-emption
  - Adjusts priority dynamically by moving jobs between queues

UNIX 4.4 Scheduling

- 128 priority ranges
  - 0-49 are kernel mode, 50-127 are user mode
- 32 run queues
  - Divide priority by 4
- Each process:
  - Has an entry in its process descriptor for its CPU utilization as well as its priority
    - CPU utilization incremented every tick process is running

UNIX 4.4 Scheduling

- Formulas:
  - New priority = 50 + estimatedUtilization/4
    - Lower priority is better!
  - Running process: New estimatedUtilization = DecayRunnable(estimatedUtilization)
    - Accounts for (hopes) process closer to terminating!
  - Sleeping process: new estimatedUtilization = DecaySleep(estimatedUtilization)
    - This decays much faster (exponentially)
- This means CPU bound jobs are pushed to lower priority queues, in general

Lottery Scheduling

- Need some fairness, but still want good average turnaround time
  - SJF unfair, other methods have poor turnaround times
  - Feedback queues try to be best of both worlds:
    - But, they are generally ad-hoc (ex: busy CPU)
- Instead, give each job lottery tickets
  - pick a winner when quantum expires
  - behaves well when load changes

Multiprocessor Scheduling

- Multiple CPUs, common main memory
  - can execute many threads at once
- Simple solution:
  - use one ready list (in shared memory)
  - grab first thread on list
  - need mutual exclusion between processors
  - problem: memory effects (more later)

Real-Time Scheduling

- Hard real time
  - must execute thread in specific time, or reject it
  - requires different kind of OS
- Soft real time
  - less restrictive -- critical threads get priority
  - need: dispatch latency to be small
    - hard if no preemption in system calls (UNIX)
    - may need to make kernel preemptible
    - need to avoid priority inversion

Deadlock

- Several threads in system
- Several *resources*
  - example: printer, CPU, disk
- Standard mode of operation
  - request resource (wait if necessary)
  - use resource
  - release resource
- Can lead to deadlock

Necessary (but not sufficient) conditions for deadlock

- Mutual exclusion
  - some resources are nonsharable, e.g. printer
- Hold and wait
  - some thread holds a resource waiting for another
- No preemption
  - cannot take resource away from thread
- Circular waiting
  - ex: 2 threads hold resources A and B, each waiting for other resource

Dining Philosophers cont. (note: on this slide '+' is modulo)

```
Philosopher(int j) {
    P(fork[j]);
    P(fork[j+1]);
    eat
    V(fork[j]);
    V(fork[j+1]);
}

    Can Deadlock!
```

```
Philosopher(int j) {
    if (j != 0)
      P(fork[j]); P(fork[j+1])
      eat
      V(fork[j]); V(fork[j+1])
    else
      P(fork[1]); P(fork[0])
      eat
      V(fork[1]); V(fork[0]
}
```

Resource Allocation Graph

- 2 types of nodes
  - threads
  - resources -- can have multiple instances
- 2 types of edges (both directed)
  - from thread to resource
    - indicates thread wants that resource
  - from resource to thread
    - indicates thread has that resource

Cycle may indicate deadlock

What to do about deadlock -- choose one

- Prevent it
  - ensure one of four conditions does not hold
  - read book for details
- Avoid it
  - only satisfy safe requests
- Allow it, and roll back
  - may be time consuming and/or hard
- Reboot

Avoiding Deadlock

- Definition: Safe State
  - a state where there exists some sequence in which resources can be allocated and released such that deadlock does not occur
- Threads must declare max. resource needs
- Don't allow OS to go from safe state to unsafe state

Banker's Algorithm (Dijkstra) [1 resource only]

- Each thread declares max number
- Data structures:
  - Available: how many of resource are available
  - Max: array of max demand per thread
  - Allocation: array of number allocated per thread
  - Need: Max - Allocation

Note: Requires each thread to declare max number of resource (not possible in general)

Basic idea behind Banker's Algorithm

- Have an algorithm to determine whether a state is safe or not
- When a thread wants a resource
  - if the request is too large, error
  - if the request cannot be satisfied, wait
  - if the request can be satisfied
    - move to the new state
    - run safety algorithm
    - if safe, allocate; else, wait

Banker's Algorithm [multiple resources]

- Just a generalization of single resource
- Data structures:
  - Available: array with count of each resource
  - Max: 2-d array of max demand per thread
  - Allocation: 2-d array of allocated resources per thread
  - Need: 2-d array (Max - Allocation)