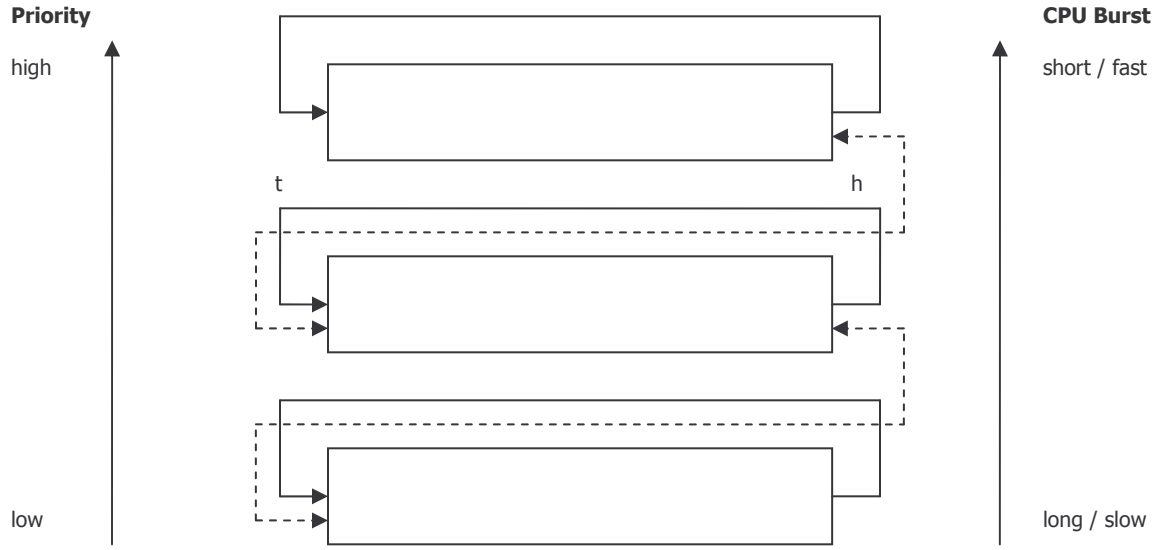
Preemption with Multilevel Queues

Arrival Preemption
Higher priority process arrives in a higher priority queue. The context switch is done immediately. Ex: I/O wait of a system process completes.

Time slice expiration
Used to move processes on and off of the CPU of the OS.
Weighted by priority

70% system
10% interactive
10% network
5% editing
5% batch

Multilevel-*feedback* queue scheduling
The ready queue is still segmented into multiple queues. Instead, they are simply ready queues of different priority. What makes them different is how processes move through them.



This is the most complex scheduling algorithm for a single-processor architecture, but it statistically gives the best performance.

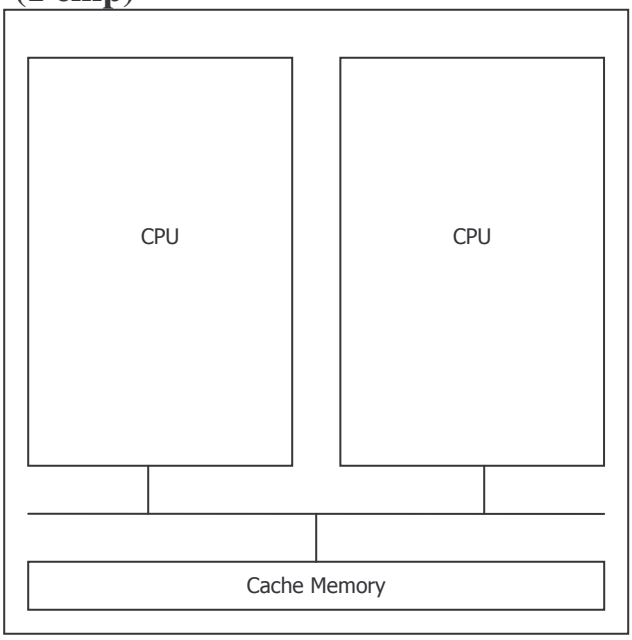
A short job completing does not delay a long job as long as a long job completing delays a short job.

Processes migrate on CPU burst time (exponential average)

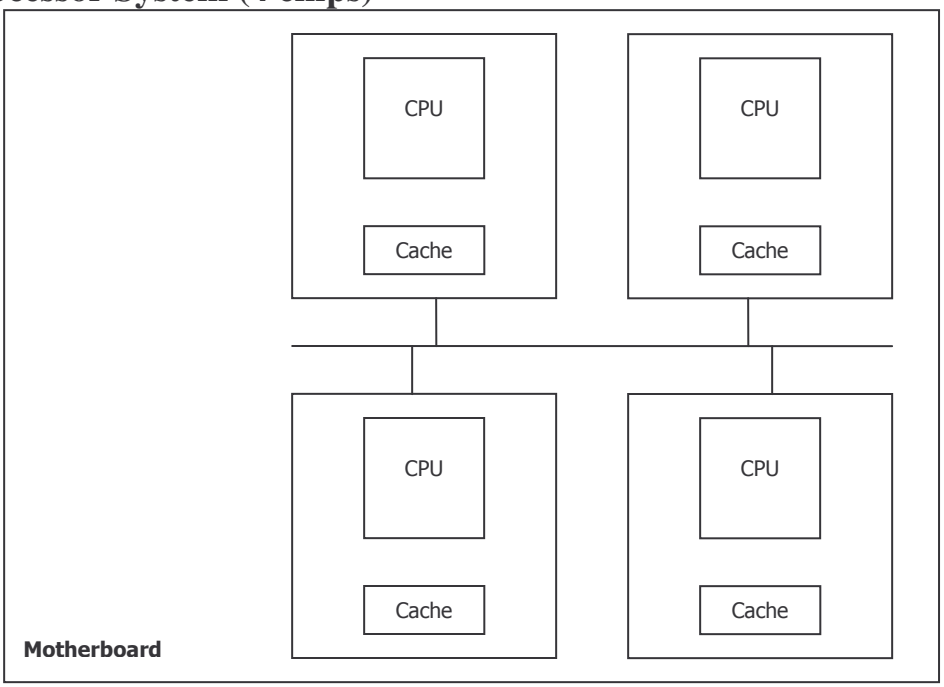
Issues

1. How many queues?
    a. Just by increasing the number of queues, you increase the complexity of your exponential process migration technique
2. How do you age?
3. How is demotion done?
    a. Exponential average, number of I/O waits, etc.
4. How is promotion done?
5. Where does a new process go?
    a. If using exponential average, throw in the high priority queue and measure its time.

Duo-core Processor (1 chip)



Quad-Processor System (4 chips)



Multiprocessor (MP) Systems

1. Asymmetric multiprocessor system (AMP)
    a. Central CPU runs the OS and schedules processes onto other CPUs
2. Symmetric multiprocessor system (SMP)
    a. All processors are self-scheduling – they have their own ready queues
    b. Adopted as the standard by the industry
    c. Support provided in Windows XP, Linux, Mac OS

Homogeneous MP System

All processors are identical.
Industry standard

Heterogeneous MP System

Scheduling behavior is much more difficult because the processors do not have the same capabilities.

Cache

Principle of data locality

Programmers tend to make variables, use the intensely, and then throw them out. The cache stores these variables that are in use. The cache memory saves the loss in performance that occurs when accessing main memory.

This problem is called **Cache Coherency**. The automatic flush and load process is essential, as data will be lost without it.

Processor affinity

If we don't invalidate the processor's cache on a context switch, the processor has lost its variables and we have dangling variables that are in the wrong cache.

Processors have an affinity (like, preference) of a specific processor. They want to stay on their initial processor to avid cache-miss. We either guarantee that the process goes on that initial processor or we must invalidate the cache.

Processes may request processor affinity by using a system call.

Two ways to guarantee that you get put back on the same processor:

1. **Soft Affinity = no guarantee**
   OS does not guarantee it, but will try really, really hard to not migrate the process.

   The goal is to balance the load of the processes between the available processors. The load balancing problem is more important than affinity, so that will often take priority.

2. **Hard Affinity = no migration**
   OS **guarantees** that the process will not migrate to another processor. It does not necessarily have to flush and reload the cache. This provides better potential for performance because the flush and reloading of the cache is avoided.

Lab Notes

Context, scheduler, power, reset

Instead of:

```
LDAA  PACTL
ANDA  #%11111101
ORAA  #%00000001
STAA  PACTL
```

Try:

```
LDX   #PACTL
BCLR  0,x   #%00000010
BSET  0,x   #%00000001
```