Interactive Scheduling Algorithms Continued
  o Priority Scheduling
    ▪ Introduction
      • Round-robin assumes all processes are equal – often not the case
      • Assign a priority to each process, and always choose the process with the highest priority
    ▪ Problem – highest priority process could run indefinitely
      • Possible solution: decrease priority at each clock interrupt until it has lower priority
    ▪ Can be used to achieve system goals
      • Example – I/O bound processes
        o Spend most time waiting for I/O to finish
        o Should get CPU quickly to make the next I/O request
      • Assign priority of \( \frac{1}{f} \) to each process
        o \( f \) = fraction of the last quantum used
        • 1msec of 50msec quantum = 50
        • 25msec of 50msec = 2
        • 50msec of 50msec = 1
    ▪ Priority classes can also be used
      • Use priority scheduling among classes
      • Use round-robin within each class
      • Priorities must be adjusted so the lower priority classes won’t be starved
  o Multiple Queues
    ▪ Attempt to reduce context switches without hurting response time
    ▪ Set up a cascade of priority classes
      • Highest priority runs for one quantum
      • Second highest runs for 2
      • Third highest runs for 4
      • Etc.
    ▪ Processes that use up their quanta are moved down a priority class
      • Result: CPU-bound processes run longer, but less often
- Example: CPU-bound process that would run for 100 quanta continuously
  - 1, 2, 4, 8, 16, 32, 64 (only 37 required for last run)
  - Needs only 7 swaps (including initial load) as opposed to 100 for round robin
- Negative – long-running processes could be punished and not get the CPU often if there are a lot of new interactive processes
  - Shortest Process Next
    - Use shortest job first for interactive processes
    - Estimate the completion time of jobs based on past history
      - Aging
  - Guaranteed Scheduling – make quantifiable promises to processes
    - Example: If there are \( n \) users, each will get \( 1/n \) of the CPU power
    - Example: If there are \( n \) processes, each will get \( 1/n \) of the power
    - Requirements
      - Keep track of the amount of time the CPU has been run on the process
      - Keep track of the time the process has been active
    - Algorithm
      - Compute amount of time the process should have on the CPU (time since creation/\( n \))
      - Divide actual time on CPU by entitled time
      - Example: 0.5 means process has had half as much as entitled
      - Always run the process with the lowest ratio until another process has a small ratio
  - Problem – more complicated guarantees are difficult to implement
    - Lottery scheduling – simpler implementation but predictable results
      - Introduction
        - Give processes lottery tickets for CPU time
        - When scheduling, a lottery ticket is chosen at random and that process is given the CPU for a quantum.
      - Lottery tickets can be allocated in any way we want
        - Uniform allocation will result in guaranteed scheduling like above (on average)
          - Processes with \( f \) fraction of the tickets will get \( f \) fraction of the CPU
      - New processes that are assigned tickets will immediately have a chance to run
        - Highly responsive
      - Example – video server streaming content to clients at different frame rates
        - 10, 20, and 25 frames/sec required
        - Allocated that number of lottery tickets to each process
          - CPU will be divided up into that proportion automatically
Fair-Share Scheduling
- We've only considered cases where processes are independent (not based on owner)
  - Example: User 1 starts 9 processes and 2 starts 1 process
    - User 1 will get 90% of the CPU
- Fair-share scheduling allocates a certain percentage of the CPU to each user instead
  - User 1: Processes A, B, C, D
  - User 2: Process E
- Fair-share: A E B E C E D E A E...
- Other definitions of fairness can also be used here

Multiprocessor scheduling
- Introduction
  - Many new processors have multiple cores
  - Always share main memory, sometimes share cache as well
  - Scheduling before: “which thread to run” to
  - Scheduling after:
    - which threads to run
    - which CPU should each run on
  - Threads can either be unrelated or related
    - Timesharing system: all threads are independent
    - More common: multi-threaded process where the threads communicate frequently
- Assumptions
  - Multiple cores or CPUs that do not share cache
  - Individual threads are the scheduled elements (related and unrelated)

Timesharing – scheduling independent threads
- Simple approach
  - Maintain single system-wide structure of ready threads
  - When a CPU becomes available, pick the next highest priority thread
- **Advantages**
  - Single structure time-shares the CPUs
  - Automatic load balancing (no CPU stays idle with work left to do)
- **Disadvantages**
  - Contention for scheduling structure as # CPUs grow
  - Context switching overhead (same as uniprocessor)
  - Ignores any benefits of running a thread on the same CPU, which might have a significant number of cache entries remaining

- **Two-level scheduling – let threads run on the same CPU if possible**
  - **Algorithm**
    - Each CPU maintains its own scheduling structure
    - When thread is created, assign it to the CPU with the lowest load at the moment (top level)
    - Scheduling is done at the CPU level, according to priority or some other metric
    - If a CPU becomes idle, it takes a thread from another CPU
  - **Advantages**
    - Threads will often stay on the same CPU (cache benefits)
    - Load is still roughly evenly distributed
    - Contention for scheduling data structure is reduced
  - **Disadvantages**
    - More complicated to implement
    - Doesn’t take advantage of relationships between threads

- **Example**
  - Two threads belong to A and two threads belong to B
  - The threads of A need to communicate frequently, but get scheduled not at the same time
- **Message + Reply** takes 200ms (very slow)
- **Space Sharing** – scheduling related threads at the same time
  - Examples: parallel make, web crawler, most any multi-threaded application where threads communicate with each other
  - Simple approach
    - Algorithm
      - Assume all related threads are created at once
      - Scheduler assigns each thread its own dedicated CPU
        - Nonmultiprogrammed
        - Thread holds onto CPU until it terminates
        - I/O blocking doesn’t cause thread to lose CPU
      - If enough CPUs aren’t available, threads must wait
      - First-come, first-served used on the group level
    - Result – threads are statically partitioned, each one running a thread

![CPU partition diagram]

**Advantages**
- Eliminates context switching overhead
- Cache benefits are guaranteed
- Communicating threads can easily cooperate
- Simple to implement

**Disadvantages**
- I/O bound threads will waste huge portions of CPU cycles
- **Gang Scheduling** – combining timesharing and space sharing
  - Algorithm
    - Divide time into discrete quanta and schedule CPUs synchronously
      - When a quantum expires, schedule all CPUs again
      - No scheduling is done in the quantum (CPU idles if blocked)
    - Groups (gangs) of related threads are scheduled as a group
      - All members run simultaneously on a different timeshared CPU
Gang members start and end their time slices together

- **Example**
  - 6 CPUs and 5 processes (A through E) with 24 ready threads

<table>
<thead>
<tr>
<th>Time slot</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A₀</td>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>A₄</td>
<td>A₅</td>
</tr>
<tr>
<td>1</td>
<td>B₀</td>
<td>B₁</td>
<td>B₂</td>
<td>C₀</td>
<td>C₁</td>
<td>C₂</td>
</tr>
<tr>
<td>2</td>
<td>D₀</td>
<td>D₁</td>
<td>D₂</td>
<td>D₃</td>
<td>D₄</td>
<td>E₀</td>
</tr>
<tr>
<td>3</td>
<td>E₁</td>
<td>E₂</td>
<td>E₃</td>
<td>E₄</td>
<td>E₅</td>
<td>E₆</td>
</tr>
<tr>
<td>4</td>
<td>A₀</td>
<td>A₁</td>
<td>A₂</td>
<td>A₃</td>
<td>A₄</td>
<td>A₅</td>
</tr>
<tr>
<td>5</td>
<td>B₀</td>
<td>B₁</td>
<td>B₂</td>
<td>C₀</td>
<td>C₁</td>
<td>C₂</td>
</tr>
<tr>
<td>6</td>
<td>D₀</td>
<td>D₁</td>
<td>D₂</td>
<td>D₃</td>
<td>D₄</td>
<td>E₀</td>
</tr>
<tr>
<td>7</td>
<td>E₁</td>
<td>E₂</td>
<td>E₃</td>
<td>E₄</td>
<td>E₅</td>
<td>E₆</td>
</tr>
</tbody>
</table>

- **Advantages**
  - Reduces possibility of threads getting out of sync and communication being delayed
  - Less CPU cycles are wasted on idle threads because the CPUs are multiprogrammed

- **Disadvantages**
  - Requires scheduling synchronization between CPUs
  - Context switching overhead
  - Again requires thread affinity to take advantage of cache contents