CPU Scheduler

- Selects from among the processes/threads that are ready to execute (in ready state), and allocates the CPU to one of them (puts in running state).
- CPU scheduling can be non-preemptive or pre-emptive
- Non-preemptive scheduling decisions may take place when a process changes state:
  1. switches from running to waiting state
  2. switches from running to ready state
  3. switches from waiting to ready
  4. terminates
- All other scheduling is preemptive
  - E.g. may be driven by an interrupt

Today's Topics

- CPU scheduling basics
- CPU scheduling algorithms

Preemptive and Non-Preemptive Scheduling

- Running
- Ready
- Blocked
- Terminated
- Exited

- Resource free, I/O completion interrupt (move to ready queue)
- Create
- Terminate (call scheduler)
- Block for resource (call scheduler)
- Yield, Interrupt (call scheduler)

- Scheduler dispatch
- Terminate (call scheduler)
### Scheduling Criteria

- **Assumptions made here**
  - One process per user and one thread per process
  - Processes are independent
- **Scheduling Goals**
  - Minimize response time (interactive) or turnaround time (batch)
    - Time from submission of job/operation to its completion
    - Job/operation could be a keystroke in an editor or running a big science simulation
  - Maximize throughput (operations/jobs per second)
    - Minimize overhead (e.g., context switching)
    - Use system resources efficiently (CPU, memory, disk, etc)
  - Fairness and proportionality
    - Share CPU in some equitable way, or that meets users’ expectations
    - Everyone makes some progress; no one starves

### Some Problem Cases in Scheduling

- **Scheduler completely blind about job types**
  - Little overlap between CPU and I/O
- **Optimization involves favoring jobs of type “A” over “B”**
  - Lots of A’s? B’s starve
- **Interactive process gets trapped behind others**
  - Response time bad for no good reason.
- **Priorities: A depends on B and A’s priority > B’s**
  - B never runs, so A doesn’t continue

### Scheduling Algorithms

- **Simplified view of scheduling:**
  - Save process state (to PCB)
  - Pick which process to run next
  - Dispatch process

### First-Come-First-Serve (FCFS) Policy

- **Schedule tasks in the order they arrive**
  - Run them until completion or they block or they yield
- **Example 1**
  - P1 = 24 sec, P2 = 3 sec, and P3 = 3 sec, submitted ‘same’ time in that order
  - Avg. response time = \((24+27+30)/3\) = 27. Avg. wait time = \((0+24+27)/3\) = 17
- **Example 2**
  - Same jobs but come in different order: P2, P3 and P1
  - Average response time = \((3 + 6 + 30) / 3 \approx 13 \text{ sec}\), avg wait time: 3 sec

- **FIFO pro: Simple. Con: Short jobs get stuck behind long ones**
Shortest Job First (SJF) Scheduling

- Whenever scheduling decision is to be made, schedule process with shortest remaining time to completion
  - Non-preemptive case: straightforward (if time can be estimated)
  - Preemptive case: if new process arrives with smaller remaining time, preempt running process and schedule new one
- Simple example
  - P1 = 6sec, P2 = 8sec, P3 = 7sec, P4 = 3sec
  - All arrive at the same time

- Can you do better than SRTCF in terms of average response time?
- Issues with this approach?

Example of non-preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (non-preemptive)

Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (preemptive)

Average waiting time = (9 + 1 + 0 +2)/4 = 3

Round Robin

- Similar to FCFS, but with a time slice for timer interrupt
  - Time-interrupted process is moved to end of queue
- FCFS for preemptive scheduling
- Real systems also have I/O interrupts in the mix

How do you choose time slice?
**FCFS vs. Round Robin**

- **Example**
  - 10 jobs and each takes 100 seconds
- **FCFS (non-preemptive scheduling)**
  - job 1: 100s, job2: 200s, ..., job10: 1000s
- **Round Robin (preemptive scheduling)**
  - time slice 1sec and no overhead
  - job1: 991s, job2: 992s, ..., job10: 1000s
- **Comparisons**
  - Round robin is much worse (avg turnaround time) for jobs about the same length
  - Both are fair, but RR is bad in the case where FIFO is optimal
  - But, e.g. for streaming video, RR is good, since everyone makes progress and gets a share "all the time"

**Resource Utilization Example**

- A, B, and C run forever (in this order)
  - A and B each uses 100% CPU forever
  - C is a CPU plus I/O job (1ms CPU + 10ms disk I/O)
- Time slice 100ms
  - A (100ms CPU), B (100ms CPU), C (1ms CPU + 10ms I/O), ...
- Time slice 1ms
  - A (1ms CPU), B (1ms CPU), C (1ms CPU), A (1ms CPU), B (1ms CPU), C(10ms I/O) || A, B, ..., A, B
- What do we learn from this example?

**Virtual Round Robin**

- I/O bound processes go to auxiliary queue (instead of ready queue) to get scheduled
- Aux queue is FIFO
- Aux queue has preference over ready queue

**Priority Scheduling**

- Not all processes are equal, so rank them
- **The method**
  - Assign each process a priority
  - Run the process with highest priority in the ready queue first
  - Adjust priority dynamically (I/O wait raises the priority, reduce priority as process runs)
- Why adjusting priorities dynamically
  - T1 at priority 4, T2 at priority 1 and T2 holds lock L
- Scenario
  - T1 tries to acquire L, fails, blocks.
  - T3 enters system at priority 3.
  - T2 never gets to run, and T1 is never unblocked
Multi-level Feedback Queues (MFQ)

- Round-robin queues, each with different priority
- Higher priority queues have shorter time slices
- Jobs start at highest priority queue
- If timeout expires, drop one level
- If timeout doesn’t expire, stay or push up one level
- What does this method do?

<table>
<thead>
<tr>
<th>Priority</th>
<th>Time slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Lottery Scheduling

- Motivations
  - SJF does well with average response time, but is unfair (long jobs can be starved)
  - Need a way to give everybody some chance of running
- Lottery method
  - Give each job a number of tickets
  - Randomly pick a winning ticket
  - To approximate SJF, give short jobs more tickets
  - To avoid starvation, give each job at least one ticket
  - Cooperative processes can exchange tickets

Multiprocessor and Cluster

- Multiprocessor architecture
  - Cache coherence
  - Single OS
- Cluster or multicompiler
  - Distributed memory
  - An OS in each box

Multiprocessor/Cluster Scheduling

- Design issue
  - Process/thread to processor assignment
- Gang scheduling (co-scheduling)
  - Threads of the same process will run together
  - Processes of the same application run together
- Dedicated processor assignment
  - Threads will be running on specific processors to completion
  - Is this a good idea?
Real-Time Scheduling

- Two types of real-time
  - Hard deadline
    - Must meet, otherwise can cause fatal error
  - Soft Deadline
    - Meet most of the time, but not mandatory
- Admission control
  - Take a real-time process only if the system can guarantee the "real-time" behavior of all processes.
  - Assume periodic processes. The jobs are schedulable, if the following holds:
    \[ \sum \frac{C_i}{T_i} \leq 1 \]
    where \( C_i \) = computation time, and \( T_i \) = period

Rate Monotonic Scheduling (Liu & Layland 73)

- Assumptions
  - Each periodic process must complete within its period
  - No process is dependent on any other process
  - A process needs same amount of CPU time on each burst
  - Non-periodic processes have no deadlines
  - Process preemption occurs instantaneously (no overhead)
- Main ideas of RMS
  - Assign each process a fixed priority = frequency of occurrence
  - Run the process with highest priority
- Example
  - P1 runs every 30ms gets priority 33 (33 times/sec)
  - P2 runs every 50ms gets priority 20 (20 times/sec)

Earliest Deadline Scheduling

- Assumptions
  - When a process needs CPU time, it announces its deadline
  - No need to be periodic process
  - CPU time needed may vary
- Main idea of EDS
  - Sort ready processes by their deadlines
  - Run the first process on the list (earliest deadline first)
  - When a new process is ready, it preempts the current one if its deadline is closer
- Example
  - P1 needs to finish by 30sec, P2 by 40sec and P3 by 50sec
  - P1 goes first
  - More in MOS 7.4.4

BSD 4.3 Scheduling with Multi-Queue

- "1 sec" preemption
  - Preempt if a process doesn’t block or complete within 1 sec
- Priority is recomputed every second
  - \( P_i = \text{base} + (\text{CPU}_{i-1}) / 2 + \text{nice} \), where \( \text{CPU}_i = (U_i + \text{CPU}_{i-1}) / 2 \)
  - Base is the base priority of the process
  - \( U_i \) is process utilization in interval \( i \)
- Priorities
  - Swapper
  - Block I/O device control
  - File operations
  - Character I/O device control
  - User processes
Linux Scheduling

- Time-sharing scheduling
  - Each process has a priority and # of credits
  - Process with the most credits will run next
  - I/O event increases credits
  - A timer interrupt causes a process to lose a credit, until zero credits reached at which time process is interrupted
  - If no process has credits, then the kernel issues credits to all processes: credits = credits/2 + priority

- Real-time scheduling
  - Soft real-time (really just higher priority threads: FIFO or RR)
  - Kernel cannot be preempted by user code

Windows Scheduling

- Classes and priorities
  - Real time: 16 static priorities
  - Variable: 16 variable priorities, start at a base priority
    - If a process has used up its quantum, lower its priority
    - If a process waits for an I/O event, raise its priority

- Priority-driven scheduler
  - For real-time class, do round robin within each priority
  - For variable class, do multiple queue

- Multiprocessor scheduling
  - For N processors, run N-1 highest priority threads on N-1 processors and run remaining threads on a single processor
  - A thread will wait for processors in its affinity set, if there are other threads available (for variable priorities)

Summary

- Best algorithms may depend on your primary goals
  - FIFO simple, optimal avg response time for tasks of equal size, but can be poor avg response time if tasks vary a lot in size
  - SJF gives the minimal average response time, but can be not great in variance of response times
  - RR has very poor avg response time for equal size tasks, but is close to SJF for variable size tasks
  - Small time slice is important for improving I/O utilization
  - If tasks have mix of processing and I/O, do well under SJF but can do poorly under RR
  - Priority and its variations are used in most systems
  - Lottery scheduling is flexible
  - Multi-queue can achieve a good balance
  - Admission control is important in real-time scheduling