The scheduling problem:
- Have $k$ jobs ready to run
- Have $n \geq 1$ CPUs that can run them

Which jobs should we assign to which CPU(s)?
1 Textbook scheduling
2 Priority scheduling
When do we schedule CPU?

- Scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from new/waiting to ready
  4. Exits

- Non-preemptive schedules use 1 & 4 only
- Preemptive schedulers run at all four points
Scheduling criteria

- Why do we care?
  - What goals should we have for a scheduling algorithm?

- Throughput – # of processes that complete per unit time
  - Higher is better

- Turnaround time – time for each process to complete
  - Lower is better

- Response time – time from request to first response
  - I.e., time between waiting → ready transition and ready → running (e.g., key press to echo, not launch to exit)
  - Lower is better

- Above criteria are affected by secondary criteria
  - CPU utilization – fraction of time CPU doing productive work
  - Waiting time – time each process waits in ready queue
Scheduling criteria

- **Why do we care?**
  - What goals should we have for a scheduling algorithm?

- **Throughput** – # of processes that complete per unit time
  - Higher is better

- **Turnaround time** – time for each process to complete
  - Lower is better

- **Response time** – time from request to first response
  - I.e., time between *waiting*→*ready* transition and *ready*→*running* (e.g., key press to echo, not launch to exit)
  - Lower is better

- Above criteria are affected by secondary criteria
  - *CPU utilization* – fraction of time CPU doing productive work
  - *Waiting time* – time each process waits in ready queue
Example: FCFS Scheduling

- Run jobs in order that they arrive
  - Called “First-come first-served” (FCFS)
  - E.g., $P_1$ needs 24 sec, while $P_2$ and $P_3$ need 3.
  - $P_2, P_3$ arrived immediately after $P_1$, get:

- Dirt simple to implement—how good is it?
- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround Time: $P_1 : 24$, $P_2 : 27$, $P_3 : 30$
  - Average TT: $(24 + 27 + 30)/3 = 27$
- Can we do better?
FCFS continued

- Suppose we scheduled $P_2$, $P_3$, then $P_1$
  - Would get:

```
  P_2  P_3  P_1
  0   3    6    30
```

- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround time: $P_1 : 30$, $P_2 : 3$, $P_3 : 6$
  - Average TT: $(30 + 3 + 6)/3 = 13$ – much less than 27
- Lesson: scheduling algorithm can reduce TT
  - Minimizing waiting time can improve RT and TT
- What about throughput?
View CPU and I/O devices the same

- CPU is one of several devices needed by users’ jobs
  - CPU runs compute jobs, Disk drive runs disk jobs, etc.
  - With network, part of job may run on remote CPU

- Scheduling 1-CPU system with $n$ I/O devices like scheduling asymmetric $(n + 1)$-CPU multiprocessor
  - Result: all I/O devices + CPU busy $\Rightarrow$ n+1 fold speedup!

- Example: disk-bound grep + CPU-bound matrix multiply
  - Overlap them just right? throughput will be almost doubled
Bursts of computation & I/O

- Jobs contain I/O and computation
  - Bursts of computation
  - Then must wait for I/O

- To Maximize throughput
  - Must maximize CPU utilization
  - Also maximize I/O device utilization

- How to do?
  - Overlap I/O & computation from multiple jobs
  - *Means response time very important for I/O-intensive jobs:* I/O device will be idle until job gets small amount of CPU to issue next I/O request
Histogram of CPU-burst times

- What does this mean for FCFS?
FCFS Convoy effect

- CPU-bound jobs will hold CPU until exit or I/O (but I/O rare for CPU-bound thread)
  - long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization

- Example: one CPU-bound job, many I/O bound
  - CPU-bound job runs (I/O devices idle)
  - CPU-bound job blocks
  - I/O-bound job(s) run, quickly block on I/O
  - CPU-bound job runs again
  - I/O completes
  - CPU-bound job continues while I/O devices idle

- Simple hack: run process whose I/O completed
  - What is a potential problem?
• *Shortest-job first* (SJF) attempts to minimize TT
  - Schedule the job whose next CPU burst is the shortest
  - Misnomer unless “job” = one CPU burst with no I/O

• Two schemes:
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *Preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Known as the *Shortest-Remaining-Time-First* or SRTF)

• What does SJF optimize?
Shortest-job first (SJF) attempts to minimize TT
- Schedule the job whose next CPU burst is the shortest
- Misnomer unless “job” = one CPU burst with no I/O

Two schemes:
- Non-preemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
- Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Known as the Shortest-Remaining-Time-First or SRTF)

What does SJF optimize?
- Gives minimum average waiting time for a given set of processes
### Examples

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>$P_2$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$P_3$</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

- **Non-preemptive**

  ![Non-preemptive Diagram](chart)

- **Preemptive**

  ![Preemptive Diagram](chart)

- **Drawbacks?**
 SJF limitations

- Doesn’t always minimize average TT
  - Only minimizes waiting time
  - Example where turnaround time might be suboptimal?

- Can lead to unfairness or starvation

- In practice, can’t actually predict the future

- But can estimate CPU burst length based on past
  - Exponentially weighted average a good idea
  - $t_n$ actual length of process’s $n^{th}$ CPU burst
  - $\tau_{n+1}$ estimated length of proc’s $(n + 1)^{st}$
  - Choose parameter $\alpha$ where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
SJF limitations

- Doesn’t always minimize average TT
  - Only minimizes waiting time
  - Example where turnaround time might be suboptimal?
  - Overall longer job has shorter bursts

- Can lead to unfairness or starvation

- In practice, can’t actually predict the future

- But can estimate CPU burst length based on past
  - Exponentially weighted average a good idea
  - $t_n$ actual length of process’s $n^{th}$ CPU burst
  - $\tau_{n+1}$ estimated length of proc’s $(n+1)^{st}$
  - Choose parameter $\alpha$ where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
Exp. weighted average example

CPU burst ($t_i$)   6  4  6  4  13  13  13  13  ...  
"guess" ($\tau_i$)  10 8  6  6  5  9  11  12  ...
**Round robin (RR) scheduling**

- **Solution to fairness and starvation**
  - Preempt job after some time slice or *quantum*
  - When preempted, move to back of FIFO queue
  - (Most systems do some flavor of this)

- **Advantages:**
  - Fair allocation of CPU across jobs
  - Low average waiting time when job lengths vary
  - Good for responsiveness if small number of jobs

- **Disadvantages?**
RR disadvantages

- Varying sized jobs are good ... what about same-sized jobs?
- Assume 2 jobs of time=100 each:

```
P_1   P_2   P_1   P_2   P_1   P_2   ...   P_1   P_2
0     1     2     3     4     5     6     198    199    200
```

- Even if context switches were free...
  - What would average turnaround time be with RR?
  - How does that compare to FCFS?
RR disadvantages

- Varying sized jobs are good... what about same-sized jobs?
- Assume 2 jobs of time=100 each:
  
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>198</th>
<th>199</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>P₂</td>
<td>P₁</td>
<td>P₂</td>
<td>P₁</td>
<td>P₂</td>
<td></td>
<td>P₁</td>
<td>P₂</td>
<td></td>
</tr>
</tbody>
</table>

- Even if context switches were free...
  - What would average turnaround time be with RR? 199.5
  - How does that compare to FCFS? 150
Context switch costs

- What is the cost of a context switch?

- Brute CPU time cost in kernel:
  - Save and restore registers, etc.
  - Switch address spaces (expensive instructions)

- Indirect costs: cache, buffer cache, & TLB misses

- CPU cache

P1

P2
• What is the cost of a context switch?
• Brute CPU time cost in kernel
  - Save and restore registers, etc.
  - Switch address spaces (expensive instructions)
• Indirect costs: cache, buffer cache, & TLB misses

![Diagram showing CPU cache transitions between two processor contexts](image-url)
Context switch costs

- What is the cost of a context switch?
- Brute CPU time cost in kernel
  - Save and restore registers, etc.
  - Switch address spaces (expensive instructions)
- Indirect costs: cache, buffer cache, & TLB misses
Time quantum

- How to pick quantum?
  - Want much larger than context switch cost
  - Majority of bursts should be less than quantum
  - But not so large system reverts to FCFS

- Typical values: 1–100 msec
Turnaround time vs. quantum

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>3</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
</tr>
<tr>
<td>$P_4$</td>
<td>7</td>
</tr>
</tbody>
</table>
• Switching to swapped out process very expensive
  - Swapped out process has most memory pages on disk
  - Will have to fault them all in while running
  - One disk access costs $\sim 10\text{ms}$. On 1GHz machine, $10\text{ms} = 10$ million cycles!

• Context-switch-cost aware scheduling
  - Run in-core subset for “a while”
  - Then swap some between disk and memory

• How to pick subset? How to define “a while”? 
  - View as scheduling memory before scheduling CPU
  - Swapping in process is cost of memory “context switch”
  - So want “memory quantum” much larger than swapping cost
Outline

1 Textbook scheduling

2 Priority scheduling
Priority scheduling

- Associate a numeric priority with each process
  - E.g., smaller number means higher priority (Unix/BSD)
- Give CPU to the process with highest priority
  - Can be done preemptively or non-preemptively
- Note SJF is priority scheduling where priority is the predicted next CPU burst time
- Starvation – low priority processes may never execute
- Solution?
Priority scheduling

- Associate a numeric priority with each process
  - E.g., smaller number means higher priority (Unix/BSD)
- Give CPU to the process with highest priority
  - Can be done preemptively or non-preemptively
- Note SJF is priority scheduling where priority is the predicted next CPU burst time
- Starvation – low priority processes may never execute
- Solution?
  - Aging: increase a process’s priority as it waits
Multilevel feedback queues (BSD)

- Every runnable process on one of 32 run queues
  - Kernel runs process on highest-priority non-empty queue
    - Round-robbins among processes on same queue
- Process priorities dynamically computed
  - Processes moved between queues to reflect priority changes
  - If a process gets higher priority than running process, run it
- Idea: Favor interactive jobs that use less CPU
Process priority

- **p_nice** – user-settable weighting factor
- **p_estcpu** – per-process estimated CPU usage
  - Incremented whenever timer interrupt found process running
  - Decayed every second while process runnable

\[
p_{\text{estcpu}} \leftarrow \left( \frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right) p_{\text{estcpu}} + p_{\text{nice}}
\]

- Load is sampled average of length of run queue plus short-term sleep queue over last minute
- Run queue determined by \( p_{\text{usrpri}}/4 \)

\[
p_{\text{usrpri}} \leftarrow 50 + \left( \frac{p_{\text{estcpu}}}{4} \right) + 2 \cdot p_{\text{nice}}
\]

(value clipped if over 127)
Sleeping process increases priority

- $p_{estcpu}$ not updated while asleep
  - Instead $p_{slptime}$ keeps count of sleep time
- When process becomes runnable
  
  $p_{estcpu} \leftarrow \left( \frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1} \right)^{p_{slptime}} \times p_{estcpu}$
  
  - Approximates decay ignoring nice and past loads
- Previous description based on [McKusick] (The Design and Implementation of the 4.4BSD Operating System)
Thread scheduling

With thread library, have two scheduling decisions:

- **Local Scheduling** – Thread library decides which user thread to put onto an available kernel thread
- **Global Scheduling** – Kernel decides which kernel thread to run next

Can expose to the user

- E.g., `pthread_attr_setscope` allows two choices
  - `PTHREAD_SCOPE_SYSTEM` – thread scheduled like a process (effectively one kernel thread bound to user thread – Will return ENOTSUP in user-level pthreads implementation)
  - `PTHREAD_SCOPE_PROCESS` – thread scheduled within the current process (may have multiple user threads multiplexed onto kernel threads)
Thread dependencies

- Assume $H$ at high priority, $L$ at low priority
  - $L$ acquires lock $l$.
  - Scenario 1: $H$ tries to acquire $l$, fails, spins. $L$ never gets to run.
  - Scenario 2: $H$ tries to acquire $l$, fails, blocks. $M$ enters system at medium priority. $L$ never gets to run.
  - Both scenes are examples of priority inversion

- Scheduling = deciding who should make progress
  - A thread’s importance should increase with the importance of those that depend on it
  - Naïve priority schemes violate this
Priority donation

- Assume higher number = higher priority

Example 1: \(L\) (prio 2), \(M\) (prio 4), \(H\) (prio 8)
  - \(L\) holds lock \(l\)
  - \(M\) waits on \(l\), \(L\)’s priority raised to \(L_1 = \max(M, L) = 4\)
  - Then \(H\) waits on \(l\), \(L\)’s priority raised to \(\max(H, L_1) = 8\)

Example 2: Same \(L, M, H\) as above
  - \(L\) holds lock \(l\), \(M\) holds lock \(l_2\)
  - \(M\) waits on \(l\), \(L\)’s priority now \(L_1 = 4\) (as before)
  - Then \(H\) waits on \(l_2\). \(M\)’s priority goes to \(M_1 = \max(H, M) = 8\), and \(L\)’s priority raised to \(\max(M_1, L_1) = 8\)

Example 3: \(L\) (prio 2), \(M_1, \ldots, M_{1000}\) (all prio 4)
  - \(L\) has \(l\), and \(M_1, \ldots, M_{1000}\) all block on \(l\). \(L\)’s priority is \(\max(L, M_1, \ldots, M_{1000}) = 4\).