Scheduling

Today
- Introduction to scheduling
- Classical algorithms

Next Time
- Process interaction & communication
Scheduling

- **Problem**
  - Several ready processes & much fewer CPUs

- **A choice has to be made**
  - By the *scheduler*, using a *scheduling algorithm*

- **Scheduling through time**
  - Early batch systems – Just run the next job in the tape
  - Early timesharing systems – Scarce CPU time so scheduling is critical
  - PCs – Commonly one active process so scheduling is easy; with fast & per-user CPU scheduling is not critical
  - Networked workstations & servers – All back again, multiple ready processes & expensive CS, scheduling is critical
Process behavior

- Burstsof CPU usage alternate with periods of I/O wait
  - A property key to scheduling
  - CPU-bound & I/O bound process
- As CPU gets faster – more I/O bound processes

![Histogram of CPU-burst times](image)

- Long CPU burst
- Waiting for I/O
- Short CPU burst

- Large number of short CPU bursts
- Small number of long CPU bursts
When to schedule?

- When?
  1. At process creation
  2. When a process exits
  3. When a process blocks on I/O, a semaphore, etc
  4. When an I/O interrupts occurs
  5. A fixed periods of time – Need a HW clock interrupting

- Preemptive and non-preemptive
  - No-preemptive: An allocated CPU is not released until the process terminates or switches to waiting
Environments and goals

- Different scheduling algorithms for different application areas
- Worth distinguishing
  - Batch
  - Interactive
  - Real-time
- All systems
  - Fairness – comparable processes getting comparable service
  - Policy enforcement – seeing that stated policy is carried out
  - Balance – keeping all parts of the system busy (mix pool of processes)
Environments and goals

- Batch systems
  - Throughput – max. jobs per hour
  - Turnaround time – min. time bet/ submission & termination
    - Waiting time – sum of periods spent waiting in ready queue
  - CPU utilization – keep CPU busy all time *(anything wrong?)*

- Interactive systems
  - Response time – respond to requests quickly (time to start responding)
  - Proportionality – meet users’ expectations

- Real-time system
  - Meeting deadlines – avoid losing data
  - Predictability – avoid quality degradation in multimedia systems

- Average, maximum, minimum or *variance*?
First-Come First-Served scheduling

- First-Come First-Served (FCFS)
  - Simplest, easy to implement, non-preemptive
  - Problem:
    - 1 CPU-bound process (burst of 1 sec.)
    - Many I/O-bound ones (needing to read 1000 records)
    - Each I/O-bound process reads one block per sec!

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

Change order of arrival ….

Average waiting time: 
\[
\frac{0 + 24 + 27}{3} = 17
\]
Shortest Job/Remaining Time First sched.

- Shortest-Job First
  - Assumption – total time needed (or length of next CPU burst) is known
  - Provably optimal
    First job finishes at time $a$
    Second job at time $a + b$
    ...

  Mean turnaround time
  $(4a + 3b + 2c + d)/4$

  Biggest contributor

Preemptive or not?

- A preemptive variation – Shortest Remaining Time (or SRPT)
SJF and SRT

- SJF Non-preemptive

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival</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>

avg. waiting time = \((0 + 6 + 3 + 7)/4 = 4\)

- SRT Preemptive

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival</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>

avg. waiting time = \((9 + 1 + 0 + 2)/4 = 3\)

EECS 343 Operating Systems
Northwestern University
Determining length of next CPU burst

- Can only estimate length
- Can be done using length of previous CPU bursts and exponential averaging

- $t_n =$ actual length of $n^{th}$ CPU burst
- $\tau_{n+1} =$ predicted value for the next CPU burst
- $\alpha$, $0 \leq \alpha \leq 1$
- Define:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n.$$
Priority scheduling

- SJF is a special case of priority-based scheduling
  - Priority = reverse of predicted next CPU burst
- Pick process with highest priority (lowest number)
- Problem
  - Starvation – low priority processes may never execute
- Solution:
  - Aging → increases priority (Unix’s nice)
  - Assigned maximum quantum

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

avg. waiting time = \( (6 + 0 + 16 + 18 + 1) / 5 = 8.2 \)
Round-robin scheduling

- Simple, fair, easy to implement, & widely-used
- Each process gets a fix quantum or time slice
- When quantum expires, if running preempt CPU
- With $n$ processes & quantum $q$, each one gets $1/n$ of the CPU time, no-one waits more than $(n-1)q$

$$q = 4$$

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>24</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
</tr>
</tbody>
</table>

avg. waiting time = $(6 + 4 + 7)/3 = 5.66$

Preemptive or not?
Quantum & Turnaround time

- **Length of quantum**
  - Too short – low CPU efficiency (*why?*)
  - Too long – low response time
    (*really long, what do you get?*)
  - Commonly ~ 50-100 msec.

![Diagram showing average turnaround time and quantum sizes](image)

<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>

Largest quantums don’t imply shortest turnaround times
Combining algorithms

- In practice, any real system uses some hybrid approach, with elements of each algorithm

- **Multilevel queue**
  - Ready queue partitioned into separate queues
  - Each queue has its own scheduling algorithm
  - Scheduling must be done between the queues
    - Fixed priority scheduling; (i.e., foreground first); starvation?
    - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes

![Diagram of multilevel queue](image-url)
Multiple (feedback) queues

- Multiple queues, allow processes to move between queues
- Example CTSS – Idea: separate processes based on CPU bursts
  - IBM 7094 had space for 1 process in memory (switch = swap)
  - Goals: low context switching cost & good response time
  - Priority classes: class $i$ gets $2^i$ quantas
  - Scheduler executes first all processes in queue 0; if empty, all in queue 1, …
  - If process uses all its quanta → move to next lower queue (leave I/O-bound & interact. processes in high-priority queue)
  - What about process with long start but interactive after that?

Carriage-return hit → promote process to top class
Some other algorithms

- Guaranteed sched. - e.g. proportional to # processes
  - Priority = amount used / amount promised
  - Lower ratio $\rightarrow$ higher priority

- Lottery scheduling – simple & predictable
  - Each process gets lottery tickets for resources (CPU time)
  - Scheduling – lottery, i.e. randomly pick a ticket
  - Priority – more tickets means higher chance
  - Processes may exchange tickets

- Fair-Share scheduling
  - Schedule aware of ownership
  - Owners get a % of CPU, processes are picked to enforce it
Real-time scheduling

- Different categories
  - *Hard RT* – not on time ~ not at all
  - *Soft RT* – important to meet guarantees but not critical

- Scheduling can be static or dynamic

- Schedulable real-time system
  - \( m \) periodic events
  - event \( i \) occurs within period \( P_i \) and requires \( C_i \) seconds

Then the load can only be handled if

\[
\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1
\]

- P1: \( C = 50 \text{ msec}, P = 100\text{msec} \) (.5)
- P2: \( C = 30 \text{ msec}, P = 200\text{msec} \) (.15)
- P3: \( C = 100 \text{ msec}, P = 500\text{msec} \) (.2)
- P4: \( C = 200 \text{ msec}, P = 1000\text{msec} \) (.2)
Multiple-processor scheduling

- Scheduling more complex w/ multiple CPUs
- Asymmetric/symmetric (SMP) multiprocessing
  - Supported by most OSs (common or independent ready queues)
- Processor affinity – benefits of past history in a processor
- Load balancing – keep workload evenly distributed
  - Push migration – specific task pushes processes for balance
  - Pull migration – idle processor asks for/pulls work
- Symmetric multithreading (hyperthreading or SMT)
  - Multiple logical processors on a physical one
  - Each w/ own architecture state, supported by hardware
  - Shouldn’t require OS to know about it (but could benefit from)
Scheduling the server-side of P2P systems

- P2P users’ response is dominated by download
  - >80% download requests in Kazaa are rejected due to capacity saturation at server peers
  - >50% of all requests for large objects (>100MB) take more than one day & ~20% take over one week to complete

- Most implementations use FCFS or PS
  - Apply SRPT!
- Work from Northwestern

Mean response time of object download as a function of system load.
Thread scheduling

- Now add threads – user or kernel level?
- User-level (process-contention scope)
  - Context switch is cheaper
  - You can have an application-specific scheduler at user level
  - Kernel doesn’t know of your threads
- Kernel-level (system-contention scope)
  - Any scheduling of threads is possible (since the kernel knows of all)
  - Switching threads inside same process is cheaper than switching processes
Policy vs. mechanism

- Separate what is done from how it is done
  - Think of parent process with multiple children
  - Parent process may know the relative importance of children (if, for example, each one has a different task)

- None of the algorithms presented takes the parent process input for scheduling

- Scheduling algorithm parameterized
  - Mechanism in the kernel

- Parameters filled in by user processes
  - Policy set by user process
  - Parent controls scheduling w/o doing it
Next time

- Process synchronization
  - Race condition & critical regions
  - Software and hardware solutions
  - Review of classical synchronization problems
  - …

- What really happened in Mars?
  [Link](http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html)