Scheduling

Today
- Introduction to scheduling
- Classical algorithms
- Thread scheduling
- Evaluating scheduling
- OS example

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

- Bursts of 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

- Large number of short CPU bursts
- Small number of long CPU bursts
Multilevel scheduling

- Batch systems allow scheduling at 3 levels

- Look for mix of CPU- & I/O-bound jobs or Shortest job first

- These two determine the degree of multiprogramming

- Too many processes & not enough memory – swap somebody out; when there's room: which one to bring in?
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

![Diagram of process states](image)
Dispatcher

- Dispatcher module gives control of CPU to process selected by short-term scheduler
  - Switching context
  - Switching to user mode
  - Jumping to proper location in user program to restart it

- Dispatch latency – time for the dispatcher to stop one process & start another running
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 the CPU busy all time

- 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
  - 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 to complete)
    - Each I/O-bound process reads one block per sec!
FCFS scheduling

Order of arrival: P1, P2, P3

Gantt chart for schedule

<table>
<thead>
<tr>
<th></th>
<th>P_1</th>
<th>P_2</th>
<th>P_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
<td>27</td>
<td>30</td>
</tr>
</tbody>
</table>

Waiting times: P1 = 0; P2 = 24; P3 = 27
Average waiting time: \( \frac{0 + 24 + 27}{3} = 17 \)

Order of arrival: P_2, P_3, P_1

Gantt chart for schedule is

<table>
<thead>
<tr>
<th></th>
<th>P_2</th>
<th>P_3</th>
<th>P_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>6</td>
<td>30</td>
</tr>
</tbody>
</table>

Waiting times: P1 = 6; P2 = 0; P3 = 3
Average waiting time: \( \frac{6 + 0 + 3}{3} = 3 \)

Preemptive or not?
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
  \[(4a + 3b + 2c + d)/4\]

- **A preemptive variation – Shortest Remaining Time (or SRPT)**

<table>
<thead>
<tr>
<th>Job #</th>
<th>Finish time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
</tbody>
</table>

**Biggest contributor**

**Preemptive or not?**
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
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:

\[
\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n.
\]

Most recent information
Past history

Weight of history

CPU burst \( t_i \):
6 6 4 6 4 13 13 13 ...

"guess" \( \tau_i \):
10 8 6 6 5 9 11 12 ...

EECS 343 Operating Systems
Northwestern University
Examples of Exponential Averaging

- $\alpha = 0$
  
  - $\tau_{n+1} = \tau_n$
  
  - Recent history does not count

- $\alpha = 1$
  
  - $\tau_{n+1} = t_n$
  
  - Only the actual last CPU burst counts

- If we expand the formula, we get:

  \[
  \tau_{n+1} = \alpha t_n + (1 - \alpha) t_{n-1} + \ldots + (1 - \alpha)^i \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0
  \]

- Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.
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.

<table>
<thead>
<tr>
<th>process</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>6</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>1</td>
</tr>
<tr>
<td>P₄</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
Multiple (feedback) queues

- Multiple queues, allow processes to move between queues
- Example CTSS – Idea: separate processes based on CPU bursts
  - 7094 had only space for 1 process in memory (switch = swap)
  - Goals: low context switching cost & good response time
  - Priority classes: class $i$ gets $2^i$ quanta ($i: 0 \ldots$)
  - 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 → 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: $C = 50$ msec, $P = 100$msec (.5)
- P2: $C = 30$ msec, $P = 200$msec (.15)
- P3: $C = 100$ msec, $P = 500$msec (.2)
- P4: $C = 200$ msec, $P = 1000$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 periodically checks load in processors & pushes processes for balance
  - Pull migration – idle processor pulls processes from busy one
- 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

- Response time experienced by users of P2P services is dominated by downloading process.
  - >80% of all 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 by Qiao et al. @ 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
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5

/* Each thread begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread exit(0);
}

int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[NUM THREADS]; pthread_attr_t attr;

    pthread_attr_init(&attr); /* get the default attributes */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM); /* set the sched algo */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER); /* set the sched policy */

    for (i = 0; i < NUM_THREADS; i++) /* create the threads */
        pthread_create(&tid[i], &attr, runner, NULL);

    for (i = 0; i < NUM_THREADS; i++) /* now join on each thread */
        pthread_join(tid[i], NULL);
}
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
Algorithm evaluation

- First problem: criteria to be used in selection
  - E.g. Max CPU usage, but w/ max. response time of 1 sec.

- Evaluation forms
  - Analytic evaluation - deterministic modeling:
    - Given workload & algorithm → number or formula
    - Simple & fast, but workload specific
  - Queueing models
    - Computer system described as a network of servers
    - Load characterized by distributions
    - Applicable to limited number of algorithms – complicated math & questionable assumptions
  - Simulations
    - Distribution-driven or trace-based
  - Implementation
    - Highly accurate & equally expensive
Next time

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

- What really happened in Mars?
  http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html
OS examples – Linux

- Preemptive, priority-based scheduling
  - Two separate priority ranges mapping to a global priority scheme
    - Real-time [0,99] & nice [100,140]

- Two algorithms
  - Time-sharing
    - Prioritized credit-based – process w/ most credits is scheduled next
    - Credit subtracted when timer interrupt occurs
    - When credit = 0, another process chosen
    - When all processes have credit = 0, re-crediting occurs
      - Based on factors including priority and history
  - (Soft) Real-time
    - Static priority for RT tasks
    - Two classes
      - FCFS (2+ task w/ = priority RR) and RR (FCFS w/ quantum)
      - Highest priority process always runs first
OS examples – Linux (Ingo Molnar’s O(1))

- Perfect SMP scalability & improved SMP affinity
- O(1) scheduling – constant-time, regardless of # of running processes
  - One run queue per processor
  - Two priority arrays: Active (tasks w/ remaining quantum) & Expired
  - Each array includes 1 queue of runnable processes per priority level
  - Recalculation of task’s dynamic priority done when task has exhausted its time quantum & moved to expired
  - When active is empty – swap