What does this angry bird have to do with scheduling?

Chapter 5: CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Operating Systems Examples
- Algorithm Evaluation
Basic Concepts

- Maximum CPU utilization obtained with multiprogramming

- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait

- CPU burst distribution

Alternating CPU and I/O Bursts
**CPU Scheduler**

- Selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- 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 waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is non-preemptive
- All other scheduling is preemptive
- Preemptive schedule needs to consider:
  - Access to shared data
  - Preemption while in kernel mode
  - Interrupts occurring during crucial OS activities
Dispatcher

- Module that gives control of the CPU to the process selected by the short-term scheduler

- Responsible for:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

- Dispatch latency – time it takes for the dispatcher to stop one process and start running another

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible

- Throughput – # of processes that complete their execution per time unit

- Turnaround time – amount of time to execute a particular process

- Waiting time – amount of time a process has been waiting in the ready queue

- Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

First-Come, First-Served (FCFS) Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>24</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>3</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose that the processes arrive in the order: \( P_1, P_2, P_3 \)
The Gantt Chart for the schedule is:

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

- Waiting time for \( P_1 = 0; P_2 = 24; P_3 = 27 \)
- Average waiting time: \( (0 + 24 + 27)/3 = 17 \)
FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order: $P_2, P_3, P_1$
- The Gantt chart for the schedule is:

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>$P_2$</th>
<th>$P_3$</th>
<th>$P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- Convoy effect - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time

- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>6</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
</tr>
<tr>
<td>$P_3$</td>
<td>7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>3</td>
</tr>
</tbody>
</table>

- SJF scheduling chart

- Average waiting time = $(3 + 16 + 9 + 0) / 4 = 7$

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging
  1. $\tau_n$ = actual length of $n^{th}$ CPU burst
  2. $\tau_{n+1}$ = predicted value for the next CPU burst
  3. $\alpha$, $0 \leq \alpha \leq 1$
  4. Define: $\tau_{n+1} = \alpha \tau_n + (1-\alpha)\tau_n$

- Commonly, $\alpha = \frac{1}{2}$
- Preemptive version called shortest-remaining-time-first
Prediction of Next CPU Burst

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:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha \tau_{n-1} + \ldots$$

$$+ (1 - \alpha)^2 \alpha \tau_{n-2} + \ldots$$

$$+ (1 - \alpha)^n \tau_0$$

- Since $\alpha, (1 - \alpha) \leq 1$, each successive term has less weight than its predecessor
Example of Shortest-remaining-time-first

Now we add the concepts of varying arrival times and preemption to the analysis.

<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>8</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Preemptive SJF Gantt Chart

<table>
<thead>
<tr>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_4 )</th>
<th>( P_1 )</th>
<th>( P_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>17</td>
</tr>
</tbody>
</table>

Average waiting time = \[ \frac{(10-1)+(1-1)+(17-2)+(5-3)}{4} \] = 26/4 = 6.5 msec

Priority Scheduling

A priority number (integer) is associated with each process.

- The CPU is allocated to the process with the highest priority (smallest integer \( \rightarrow \) highest priority)
  - Preemptive
  - Nonpreemptive

- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

- Starvation – low priority processes may never execute

- Aging – as time progresses increase the priority of the process
Example of Priority Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>$P_2$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$P_3$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>$P_5$</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

- Priority scheduling Gantt Chart

- Average waiting time = 8.2 msec

Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum $q$), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

- If there are $n$ processes in the ready queue and the time quantum is $q$, then each process gets $1/n$ of the CPU time in chunks of at most $q$ time units at once. No process waits more than $(n-1)q$ time units.

- Timer interrupts every quantum to schedule next process

- Performance
  - $q$ large -> FIFO
  - $q$ small -> overhead is too high
Example of RR with Time Quantum = 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>

- The Gantt chart is:

- Typically, higher average turnaround than SJF, but better response
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 μsec

Time Quantum and Context Switch Time

<table>
<thead>
<tr>
<th>process time = 10</th>
<th>quantum</th>
<th>context switches</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>
### Turnaround Time vs Time Quantum

80% of CPU bursts should be shorter than $q$

Large $q$ makes RR $\rightarrow$ FCFS

### Back to the question

It's a Round Robin!
Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - foreground (interactive)
  - background (batch)
- Process permanently in a given queue

- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS

- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes
  - e.g., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling

highest priority

- system processes
- interactive processes
- interactive editing processes
- batch processes
- student processes

lowest priority
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q0 – RR with time quantum 8 milliseconds
  - Q1 – RR time quantum 16 milliseconds
  - Q2 – FCFS
- Scheduling
  - A new job enters queue Q0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q1
  - At Q1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q2
Multilevel Feedback Queues

quantum = 8

quantum = 16

FCFS

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as process-contention scope (PCS) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system
Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```c
#include <pthread.h>
#include <stdio.h>

#define NUM THREADS 5
int main(int argc, char *argv[])
{
    int i;
    pthread t_tid[NUM THREADS];
    pthread attr_t_attr;
    /* get the default attributes */
    pthread attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread attr_setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM THREADS; i++)
    {
        pthread_create(&tid[i],&attr,runner,NULL);
    }
}
Pthread Scheduling API

/* now join on each thread */
for (i = 0; i < NUM THREADS; i++)
    pthread join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread exit(0);
}

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing
- Symmetric multiprocessing (SMP) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- Processor affinity – process has affinity for processor on which it is currently running
  - soft affinity
  - hard affinity
  - Variations including processor sets
NUMA and CPU Scheduling

- Note that memory-placement algorithms can also consider affinity.

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip.
- Faster and consumes less power.
- Multiple threads per core also growing:
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens.
Virtualization and Scheduling

- Virtualization software schedules multiple guests onto CPU(s)

- Each guest doing its own scheduling
  - Not knowing it doesn’t own the CPUs
  - Can result in poor response time
  - Can effect time-of-day clocks in guests

- Can undo good scheduling algorithm efforts of guests
Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- Deterministic modeling
  - Type of analytic evaluation
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload

Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc
Little’s Formula

- $n =$ average queue length
- $W =$ average waiting time in queue
- $\lambda =$ average arrival rate into queue
- Little’s law – in steady state, processes leaving queue must equal processes arriving, thus $n = \lambda \times W$
  - Valid for any scheduling algorithm and arrival distribution
- For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing models limited
- Simulations more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems
Evaluation of CPU Schedulers by Simulation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary