Scheduling

CS 3113
Behavior of a Process

• 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 followed by I/O burst

• When scheduling a process, the CPU burst distribution is our main concern
A Typical Distribution of CPU-burst Times
CPU Scheduler

- Short-term 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 nonpreemptive.
- Cases 2 and 3 require process preemption.
Dispatcher

• Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  • switching context
  • switching to user mode
  • jumping to the proper location in the user program to continue executing that program

• Dispatch latency: time it takes for the dispatcher to stop one process and start another running
Scheduling Criteria

A variety of metrics are possible …
Scheduling Criteria

A variety of metrics are possible:

- **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
Possibilities for 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 at time zero: $P_1$, $P_2$, $P_3$

The Gantt Chart for the schedule is:

- Waiting time for each: ????
- Average waiting 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 at time zero: $P_1, P_2, P_3$

The Gantt Chart for the schedule is:

• Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
• Average waiting time: $(0 + 24 + 27)/3 = 17$
Suppose that the processes arrive in the order: $P_2, P_3, P_1$

- The Gantt chart for the schedule is:

```
  P2 | P3 | P1
  0  3  6  30
```

- Waiting time for all: ???
- Average waiting time: ???
FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order: $P_2, P_3, P_1$

- The Gantt chart for the schedule is:

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

- Waiting time for all: $P_1 = 6; P_2 = 0; P_3 = 3$
- 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 programmer to tell us
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 = ?????
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 = $(3 + 16 + 9 + 0) / 4 = 7$
Estimating 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. \( t_n = \) 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: \( \tau_{n+1} = \alpha \ t_n + (1 - \alpha) \tau_n \).

• Commonly, \( \alpha \) set to \( \frac{1}{2} \)
• Preemptive version called shortest-remaining-time-first
Example Burst Length Predictions

<table>
<thead>
<tr>
<th>CPU burst ($t_i$)</th>
<th>6</th>
<th>4</th>
<th>6</th>
<th>4</th>
<th>13</th>
<th>13</th>
<th>13</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;guess&quot; ($\tau_i$)</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Example Cases of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:
  \[
  \tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots
  \]
  \[
  + (1 - \alpha)\alpha t_{n-j} + \ldots
  \]
  \[
  + (1 - \alpha)^{n+1} \tau_0
  \]

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

• Average waiting time = ??? msec
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

• Average waiting time ??? msec
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

```
0 1 5 10 17 26
P_1 P_2 P_4 P_1 P_3
```

• Average waiting time = \([(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
  - In Unix: smallest integer \( \equiv \) highest priority
  - Two versions:
    - Preemptive
    - Nonpreemptive

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

- Problem \( \equiv \) Starvation – low priority processes may never execute

- Solution \( \equiv \) 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 = ??? msec
### 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 = ??? msec
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) Scheduling

• 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.
Round Robin (RR) Scheduling

• Timer interrupts every quantum to schedule next process

• Performance
  • $q$ large $\Rightarrow$ Reduces to FIFO
  • $q$ small $\Rightarrow$ All jobs must use multiple quanta to complete
    • $q$ must be large with respect to context switch time, otherwise 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>$P_1$</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>

• The Gantt chart is:
Example of RR with Time Quantum = 4

<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>

• The Gantt chart is:
Round Robin Notes

• 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 usec
Multilevel Queues

• Ready queue is partitioned into separate queues, e.g.:
  • foreground (interactive)
  • background (batch)

• Process permanently in a given queue

• Each queue has its own scheduling algorithm. E.g.:
  • Foreground: RR
  • Background: FCFS
Multilevel Queues

Scheduling possibilities between the queues:

• Fixed priority scheduling
  • 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. For example:
  • 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
  - Called: Aging

- Multilevel-feedback-queue 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: Multilevel Feedback Queue

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR with time quantum 16 milliseconds
  - $Q_2$ – FCFS

- Scheduling
  - A new job enters queue $Q_0$
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue $Q_1$
  - At $Q_1$ job receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue $Q_2$
Thread Scheduling

Distinction between user-level and kernel-level threads

- Many-to-one and many-to-many models: the user-space thread library schedules user-level threads to run on a light-weight process (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

Andrew H. Fagg: Introduction to Operating Systems
Scheduling within the Pthread Library

• API allows the program to specify either PCS or SCS during thread creation
  • PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  • PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
• Options can be limited by OS: Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire about the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i],&attr, runner, NULL);
/* 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)
{
    /* do some work ... */
    pthread_exit(0);
}

Pthread Scheduling API
Multiple-Processor Scheduling

CPU scheduling more complex when multiple CPUs are available

- **Homogeneous processors** within a multiprocessor: all processors are the same
- **Asymmetric multiprocessing**: only one processor accesses the system data structures for scheduling, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)**: each processor is self-scheduling
  - All processes in common ready queue, or
  - Each processor has its own private queue of ready processes
- **Processor affinity**: process has affinity for processor on which it is currently running
  - soft affinity
  - hard affinity
  - Variations including processor sets
Multiple-Processor Scheduling: Load Balancing

If SMP, need to keep all CPUs loaded for efficiency

• **Load balancing** attempts to keep workload evenly distributed

• **Push migration**: periodic task checks load on a processor, and if the CPU is overloaded, pushes tasks to other CPUs

• **Pull migration**: idle CPU pulls waiting task from busy processor
NUMA and CPU Scheduling

NUMA: Non-Uniform Memory Allocation
• Each processor has its own memory: fast access
• Can still access memory of other processors, but much slower
• This is our motivation for processor 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
Multithreaded Multicore System

- C: compute cycle
- M: memory stall cycle

Thread 0: C M C M C M C C

Thread 1: C M C M C M C C

Time progression