II. Processes, Threads and Scheduling

SGG9: 3.1-3.3, 4.1-4.3, 5.1-5.4
- Process concepts: context switch, scheduling queues, creation
- Multithreaded programming
- Process scheduling

TDIU11: Operating Systems

Ahmed Rezine, Linköping University

Process Concept

• Program is **passive** entity stored on disk (**executable file**), process is **active**. Consider multiple users executing the same program.

• Textbook uses the terms **job** and **process** almost interchangeably.

• **Process** – a program in execution; process execution must progress in sequential fashion.

• Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data: function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time
Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution
Process Control Block (PCB)

Information associated with each process (also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information – priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files
CPU Switch From Process to Process

- **Process $P_0$**
  - Executing
  - Interrupt or system call
  - Save state into PCB
  - ... (omitted)
  - Reload state from PCB
  - Idle

- **Operating System**
  - ... (omitted)

- **Process $P_1$**
  - Idle
  - Executing
  - Save state into PCB
  - ... (omitted)
  - Reload state from PCB
  - Idle
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for execution on CPU
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Job queue** – set of all processes in the system
  - **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues
- Queueing diagram
Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good **process mix**
Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**

- **Context** of a process represented in the PCB

- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch

- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once
Operations on Processes: Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes

- Generally, process identified and managed via a **process identifier (pid)**

- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources

- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
Operations on Processes: Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process’ memory space with a new program
Operations on Processes: 
Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process’ resources are deallocated by operating system

- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
Operations on Processes: Inter-process Communication

- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need inter-process communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing
Operations on Processes: Communications Models

(a) Message passing.

(b) shared memory.
Threads: Motivation

• Most modern applications are multithreaded
• Threads run within application
• Multiple tasks with the application can be implemented by separate threads
  – Update display
  – Fetch data
  – Spell checking
  – Answer a network request
• Process creation is heavy-weight while thread creation is light-weight
• Can simplify code, increase efficiency
• Kernels are generally multithreaded
Threads: Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – single threaded process can take advantage of only a processor in a multiprocessor architectures
Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include: *Dividing activities, Balance, Data splitting, Data dependency, Testing and debugging*

- **Parallelism** implies a system can perform more than one task simultaneously

- **Concurrency** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
Concurrent execution on single-core system:

<table>
<thead>
<tr>
<th>single core</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₁</th>
<th>...</th>
</tr>
</thead>
</table>

Parallelism on a multi-core system:

<table>
<thead>
<tr>
<th>core 1</th>
<th>T₁</th>
<th>T₃</th>
<th>T₁</th>
<th>T₃</th>
<th>T₁</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>core 2</td>
<td>T₂</td>
<td>T₄</td>
<td>T₂</td>
<td>T₄</td>
<td>T₂</td>
<td>...</td>
</tr>
</tbody>
</table>

| time |
Single and Multithreaded Processes

Single-threaded process

Multithreaded process

Diagram showing the differences between single-threaded and multithreaded processes.
Amdahl’s Law

• Identifies performance gains from adding additional cores to an application that has both serial and parallel components

• $S$ is serial portion

• $N$ processing cores

• That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times

• As $N$ approaches infinity, speedup approaches $1 / S$

Serial portion of an application has disproportionate effect on performance gained by adding additional cores
User Threads and Kernel Threads

• **User threads** - management done by user-level threads library

• Three primary thread libraries:
  – POSIX **Pthreads**
  – Windows threads
  – Java threads

• **Kernel threads** - Supported by the Kernel

• Examples – virtually all general purpose operating systems, including:
  – Windows
  – Solaris
  – Linux
  – Mac OS X
Multithreading Models: Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
Multithreading Models: One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
Multithreading Models: Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
Multithreading Models: Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
Scheduling: 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** followed by **I/O burst**
- CPU burst distribution is of main concern
Histogram of CPU-burst Times
CPU Scheduler

• **Short-term scheduler** selects among processes in ready queue

• 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**

• Any other scheduling is **preemptive**.
  – shared data, preemption during crucial OS activities
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 restart that program

• **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running
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: from submission to completion, including waiting to get to memory, ready queue, executing on CPU, I/O,...
- **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)
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:

- 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:

```
<p>| | | | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>P1</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 using length of previous CPU bursts, using exponential averaging

  1. \( t_n = \text{actual length of } n^{th} \text{ CPU burst} \)
  2. \( \tau_{n+1} = \text{predicted value for the next CPU} \)
  3. \( \alpha: 0 \leq \alpha \leq 1 \)
  4. Define: \( \tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n \)

• Results in: \( \tau_n = \alpha t_n + (1 - \alpha) \alpha t_n + \ldots + (1 - \alpha)^n \alpha t_1 + (1 - \alpha)^{n+1} \tau_0 \)

• Commonly, \( \alpha \) set to \( \frac{1}{2} \)

• Preemptive version called **shortest-remaining-time-first**
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 = \[\frac{(10-1)+(1-1)+(17-2)+(5-3)}{4} = \frac{26}{4} = 6.5 \text{ msec}\]
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer $\equiv$ highest priority)
  - Preemptive
  - Non-preemptive
- 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 = 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 $\Rightarrow$ FIFO
  - $q$ small $\Rightarrow q$ must be large with respect to context switch, 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:

```
P1  P2  P3  P1  P1  P1  P1  P1
0   4   7   10  14  18  22  26  30
```  

- 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
Time Quantum and Context Switch Time

- Process time = 10
- Quantum: 12
- Context switches: 0
- Quantum: 6
- Context switches: 1
- Quantum: 1
- Context switches: 9
Turnaround Time Varies With The Time Quantum

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

<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>
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; i.e., 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:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS

- Scheduling
  - A new job enters queue $Q_0$ 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 $Q_1$
  - At $Q_1$ job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue $Q_2$