Concurrent Programming

Rafael Ramirez
Dep Tecnologia
Universitat Pompeu Fabra
Processes

- Process Concept
- Process Scheduling
- Operation on Processes
- Cooperating Processes
- Interprocess Communication
Process Concept

- Early systems allowed only one program to be executed at a time. This program had complete control of the system.
- Current computer systems allow multiple programs to be loaded into memory and to be executed *concurrently*.
- This resulted in the notion of *process* (which we’ll define as a program in execution).
- A system consists of a collection of processes: OS processes and user processes.
- All these processes can potentially execute concurrently with the CPU (or CPU’s) multiplexed among them.
- By switching the CPU among them the OS can make the computer more productive.
Brief Note on Concurrency

**Parallelism**: performance

**Concurrency**: reactive programming

**Note**
- Sequential languages may be executed in parallel
- Concurrent languages may be implemented sequentially only.

What is a concurrent program?

**Concurrent program**: a set of sequential programs (called processes) which are executed in *abstract parallelism*, i.e. not required a separate physical processor to execute each process.
An operating system executes a variety of programs:
- Batch system – jobs
- Time-shared systems – user programs or tasks

OS Textbooks use the terms *job* and *process* almost interchangeably.

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

A process includes:
- program counter
- stack
- data section
Process Concept (Cont.)

- Note: a process is NOT a program (program is a passive entity while process is an active entity).

- Two processes may be associated with the same program, e.g. user may invoke many copies of an editor program, each copy being a separate process.

- A process may spawn many processes as it runs.
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 process.
- **terminated**: The process has finished execution.
Diagram of Process State

- **new**
- **ready**
- **running**
- **waiting**
- **terminated**

- Admitted
- Interrupt
- Exit
- I/O or event completion
- Scheduler dispatch
- I/O or event wait
Process Control Block (PCB)

Information associated with each process.

- **Process state**: e.g. new, ready, running, etc.
- **Program counter**: indicates the next instruction to be executed by the process
- **CPU registers**: accumulators, stack pointers
- **CPU scheduling information**: includes process priority, etc.
- **Memory-management information**: value of the base and limit registers
- **Accounting information**: includes process #, amount of CPU used, etc.
- **I/O status information**: includes the I/O allocated to the process, open files, ...
## Process Control Block (PCB)

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Process State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Number</td>
<td></td>
</tr>
<tr>
<td>Program Counter</td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td></td>
</tr>
<tr>
<td>Memory Limits</td>
<td></td>
</tr>
<tr>
<td>List of Open Files</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
CPU Switch From Process to Process

<table>
<thead>
<tr>
<th>process $P_0$</th>
<th>operating system</th>
<th>process $P_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>executing</td>
<td>interrupt or system call</td>
<td>idle</td>
</tr>
<tr>
<td>idle</td>
<td>save state into PCB$_0$</td>
<td>executing</td>
</tr>
<tr>
<td>idle</td>
<td>interrupt or system call</td>
<td>idle</td>
</tr>
<tr>
<td>executing</td>
<td>save state into PCB$_1$</td>
<td>reload state from PCB$_0$</td>
</tr>
<tr>
<td>idle</td>
<td>reload state from PCB$_1$</td>
<td>idle</td>
</tr>
</tbody>
</table>
Process Scheduling

- Basically, it is choosing the order in which processes run.
- For a uniprocessor system, only one running process. If more than one process, the rest have to wait until CPU is free.
- Multiprogramming’s goal is to have process running at all times to maximise CPU utilisation.
- Time sharing goal is to switch the CPU among processes very frequently to minimise response time.
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.

- Context-switch time is overhead; the system does no useful work while switching.

- Time dependent on hardware support. (depends on memory speed, # of registers which must be copied, etc.)
Process Creation

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

- Resource sharing (CPU time, memory, I/O devices, etc.)
  - Parent and children share all resources.
  - Children share subset of parent’s resources.
  - Parent and child share no resources.

- Execution
  - Parent and children execute concurrently.
  - Parent waits until children terminate.
Process Creation (Cont.)

- **Address space**
  - Child duplicate of parent.
  - Child has a program loaded into it.

- **UNIX examples**
  - **fork** system call creates new process (with same address space)
  - **execve** system call used after a **fork** to replace the process’ memory space with a new program.
Process Termination

- Process executes last statement and asks the operating system to terminate it (exit).
  - Output data from child to parent (via wait).
  - Process’ resources are deallocated by operating system.
- Parent may terminate execution of children processes (abort).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - Cascading termination.
Cooperating Processes

- *Independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* process can affect or be affected by the execution of another process.

Advantages of process cooperation

- Information sharing (we must provide for concurrent access to share resources)
- Computation speed-up (break task into subtasks; exec. concurrently. >1 CPU)
- Modularity (divide functions into separate processes)
- Convenience (even a user may have >1 tasks to work on at one time)
Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
  - *bounded-buffer* assumes that there is a fixed buffer size.
Bounded-Buffer – Shared-Memory Solution

Shared data

```pascal
var n;
type item = ...;
var buffer. array [0..n-1] of item;
in, out: 0..n-1; initially =0
```

Producer process

```pascal
repeat
...
produce an item in nextp
...
while in+1 mod n = out do no-op;
buffer [in] := nextp;
in := in+1 mod n;
until false;
```
Bounded-Buffer (Cont.)

- Consumer process

```
repeat
  while in = out do no-op;
  nextc := buffer[out];
  out := out+1 mod n;
  ...
  consume the item in nextc
  ...
until false;
```

- Solution is correct, but can only fill up n–1 buffer.
Threads

- A *thread* (or *lightweight process*) is a basic unit of CPU utilization; it consists of:
  - program counter
  - register set
  - stack space

- A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resources
    collectively know as a *task*.

- A traditional or *heavyweight* process is equal to a task with one thread
Threads (Cont.)

- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
  - Cooperation of multiple threads in same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.

- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.
- Kernel-supported threads (Mach and OS/2).
- User-level threads; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU).
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).
Multiple Threads within a Task
Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

- Java threads are managed by the JVM.
Interprocess Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- IPC facility provides two operations:
  - `send(message)` – message size fixed or variable
  - `receive(message)`
- If $P$ and $Q$ wish to communicate, they need to:
  - establish a `communication link` between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)
Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?
Direct Communication

Processes must name each other explicitly:

- **send** \((P, \text{message})\) – send a message to process P
- **receive** \((Q, \text{message})\) – receive a message from process Q

Properties of communication link

- Links are established automatically.
- A link is associated with exactly one pair of communicating processes.
- Between each pair there exists exactly one link.
- The link may be unidirectional, but is usually bi-directional.
Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports).
  - Each mailbox has a unique id.
  - Processes can communicate only if they share a mailbox.

- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link may be unidirectional or bi-directional.

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
Indirect Communication (Continued)

- Mailbox sharing
  - $P_1$, $P_2$, and $P_3$ share mailbox A.
  - $P_1$ sends; $P_2$ and $P_3$ receive.
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.
Buffering

Queue of messages attached to the link; implemented in one of three ways.

1. Zero capacity – 0 messages
   Sender must wait for receiver (rendezvous).
2. Bounded capacity – finite length of \( n \) messages
   Sender must wait if link full.
3. Unbounded capacity – infinite length
   Sender never waits.
Process Synchronization

- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Concurrent programming in Java
Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Shared-memory solution to bounded-buffer problem allows at most \( n - 1 \) items in buffer at the same time. A solution, where all \( N \) buffers are used is not simple.
  - Suppose that we modify the producer-consumer code by adding a variable \( \text{counter} \), initialized to 0 and incremented each time a new item is added to the buffer.
Bounded-Buffer

- **Shared data**
  ```
  type item = ... ;
  var buffer array [0..n-1] of item;
  in, out: 0..n-1;
  counter: 0..n;
  in, out, counter := 0;
  ```

- **Producer process**
  ```
  repeat
      ...
      produce an item in nextp
      ...
      while counter = n do no-op;
      buffer [in] := nextp;
      in := in + 1 mod n;
      counter := counter +1;
  until false;
  ```
Bounded-Buffer (Cont.)

- Consumer process

  repeat
  
  while counter = 0 do no-op;
  nextc := buffer[out];
  out := out + 1 mod n;
  counter := counter – 1;
  ...
  consume the item in nextc
  ...
  until false;

- The statements:
  - counter := counter + 1;
  - counter := counter – 1;

must be executed atomically.
(Atomic operation means an operation that completes in its entirety without interruption).
The Critical-Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- Structure of process $P_i$
  
  ```
  repeat
    entry section
    critical section
    exit section
    reminder section
  until false;
  ```
Solution to Critical-Section Problem

1. Mutual Exclusion. If process $Pi$ is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress. If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. Bounded Waiting. A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed
   - No assumption concerning relative speed of the $n$ processes.
Initial Attempts to Solve Problem

- Only 2 processes, \( P_0 \) and \( P_1 \)
- General structure of process \( P_i \) (other process \( P_j \))

\[
\text{repeat} \\
\quad \text{entry section} \\
\quad \text{critical section} \\
\quad \text{exit section} \\
\quad \text{reminder section} \\
\text{until false;}
\]

- Processes may share some common variables to synchronize their actions. (no update to these vars in critical section)
Algorithm 1

Shared variables:
- **var** `turn: (0..1);`
  
  initially `turn = 0`
- `turn = i ⇒ P_i` can enter its critical section

Process `P_i`

repeat
  
  **while** `turn ≠ i` do `no-op;`

  critical section

  `turn := j;`

  reminder section

  **until** `false;`

Satisfies mutual exclusion, but not progress
Algorithm 1 (Cont.)

Process $P_0$

```
repeat
  while $\text{turn} \neq 0$ do no-op;
  $\text{critical section}$
  $\text{turn} := 1$;
  $\text{reminder section}$
until false;
```

Process $P_1$

```
repeat
  while $\text{turn} \neq 1$ do no-op;
  $\text{critical section}$
  $\text{turn} := 0$;
  $\text{reminder section}$
until false;
```

Suppose $P_0$ and $P_1$ in CS
⇒ $\text{turn}=0$ and $\text{turn}=1$ !!!
⇒ $P_0$ and $P_1$ cannot be in CS
⇒ Mutual exclusion satisfied

Suppose $P_1$ terminates in RS
⇒ $P_0$ will be blocked at while loop even if no process is in CS.
Algorithm 2

- Shared variables
  - var flag: array [0..1] of boolean;
    - initially flag [0] = flag [1] = false.
  - flag [i] = true \Rightarrow P_i ready to enter its critical section
- Process $P_i$

\[
\text{repeat}
\begin{align*}
  &\text{flag}[i] := \text{true}; \\
  &\text{while flag}[j] \text{ do no-op;} \\
  &\text{critical section} \\
  &\text{flag}[i] := \text{false}; \\
  &\text{remainder section} \\
\end{align*}
\text{until false};
\]

- Satisfies mutual exclusion, but not progress requirement.
Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process \( P_i \)

\[
\text{repeat} \\
\text{flag}[i] := \text{true}; \\
\text{turn} := j; \\
\text{while} (\text{flag}[j] \text{ and } \text{turn} = j) \text{ do no-op;} \\
\quad \text{critical section} \\
\text{flag}[i] := \text{false}; \\
\quad \text{remainder section} \\
\text{until} \text{false};
\]

- Meets all three requirements; solves the critical-section problem for two processes.
Bakery Algorithm (not examinable)

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.

- If processes \( P_i \) and \( P_j \) receive the same number, if \( i < j \), then \( P_i \) is served first; else \( P_j \) is served first.

- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
Semaphore

- Synchronization tool that does not require busy waiting.
- Semaphore $S$ – integer variable
- can only be accessed via two indivisible (atomic) operations

\[
\text{wait}(S) : \quad \text{while } S \leq 0 \quad \text{do no-op; } \\
S := S - 1;
\]

\[
\text{signal}(S) : \quad S := S + 1;
\]
Example: Critical Section of $n$ Processes

- Shared variables
  - `var mutex : semaphore`
  - Initially $mutex = 1$

- Process $P_i$

```plaintext
repeat
  `wait(mutex);`
  critical section
  `signal(mutex);`
  remainder section
until false;
```
Semaphore Implementation

- Define a semaphore as a record

  ```
  type semaphore = record
      value: integer
      L: list of process;
  end;
  ```

- Assume two simple operations:
  - block suspends the process that invokes it.
  - `wakeup(P)` resumes the execution of a blocked process `P`. 

Implementation (Cont.)

- Semaphore operations now defined as

  \[
  \text{wait}(S): \quad S.value := S.value - 1; \\
  \text{if } S.value < 0 \\
  \quad \text{then begin} \\
  \quad \quad \text{add this process to } S.L; \\
  \quad \quad \text{block}; \\
  \quad \text{end;}
  \]

  \[
  \text{signal}(S): \quad S.value := S.value + 1; \\
  \text{if } S.value \leq 0 \\
  \quad \text{then begin} \\
  \quad \quad \text{remove a process } P \text{ from } S.L; \\
  \quad \quad \text{wakeup}(P); \\
  \quad \text{end;}
  \]
Semaphore as General Synchronization Tool

- Execute $B$ in $P_j$ only after $A$ executed in $P_i$
- Use semaphore $flag$ initialized to 0
- Code:

  $$
  \begin{align*}
  &P_i \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quarter
Deadlock and Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Let $S$ and $Q$ be two semaphores initialized to 1

  $P_0$
  \[
  \begin{align*}
  &\text{wait}(S); \\
  &\text{wait}(Q); \\
  &\ldots \\
  &\text{signal}(S); \\
  &\text{signal}(Q);
  \end{align*}
  \]

  $P_1$
  \[
  \begin{align*}
  &\text{wait}(Q); \\
  &\text{wait}(S); \\
  &\ldots \\
  &\text{signal}(Q); \\
  &\text{signal}(S);
  \end{align*}
  \]

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.
Two Types of Semaphores

- **Counting** semaphore – integer value can range over an unrestricted domain.

- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement.

- Can implement a counting semaphore $S$ as a binary semaphore.
Implementing $S$ as a Binary Semaphore

- Data structures:
  
  ```
  var S1: binary-semaphore;  (for mutex)
  S2: binary-semaphore;      (for waiting)
  C: integer;               (val of count $S$)
  ```

- Initialization:

  $S1 = 1$
  $S2 = 0$
  $C = \text{initial value of semaphore } S$
Implementing $S$ (Cont.)

**wait operation**

```plaintext
wait(S1);
C := C - 1;
if C < 0
then begin
   signal(S1);
   wait(S2);
end
else signal(S1);
```

**signal operation**

```plaintext
wait(S1);
C := C + 1;
if C ≤ 0 then signal(S2);
signal(S1);
```
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem
Bounded-Buffer Problem

- **Shared data**

```plaintext
type item = ...
var buffer = ...
    full, empty, mutex: semaphore;
    nextp, nextc: item;
    full := 0; empty := n; mutex := 1;
```
Bounded-Buffer Problem (Cont.)

- Producer process

```plaintext
repeat
    ...
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    ...
    signal(mutex);
    signal(full);
until false;
```
Bounded-Buffer Problem (Cont.)

- Consumer process

```plaintext
repeat
  wait(full)
  wait(mutex);
  ...
  remove an item from buffer to nextc
  ...
  signal(mutex);
  signal(empty);
  ...
  consume the item in nextc
  ...
until false;
```
Readers-Writers Problem

- Shared data

```plaintext
var mutex, wrt: semaphore (=1);
readcount : integer (=0);
```

- Writer process

```plaintext
wait(wrt);
...
writing is performed
...
signal(wrt);
```
Readers-Writers Problem (Cont.)

- Reader process

  ```
  wait(mutex);
  readcount := readcount + 1;
  if readcount = 1 then wait(wrt);
  signal(mutex);
  ...
  reading is performed
  ...
  wait(mutex);
  readcount := readcount - 1;
  if readcount = 0 then signal(wrt);
  signal(mutex):
  ```
Dining-Philosophers Problem

- Shared data

  `var chopstick: array [0..4] of semaphore;`  
  (=1 initially)
Dining-Philosophers Problem (Cont.)

- Philosopher $i$:

  ```
  repeat
  wait(chopstick[i])
  wait(chopstick[(i+1) mod 5])
  ...
  eat
  ...
  signal(chopstick[i]);
  signal(chopstick[(i+1) mod 5]);
  ...
  think
  ...
  until false;
  ```
Synchronization Mechanisms

- Shared variables
  "while turn=1 do no-op;"
- Synch. Hardware
  "while test-and-set(lock) do no-op;"
- Semaphores
  "wait(S); CS; signal(S);"
- Monitors (like Java)

Airline database

User 1

User N

User 2
Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```plaintext
type monitor-name = monitor
  variable declarations
  procedure entry P1 :(...);
    begin ... end;
  procedure entry P2(...);
    begin ... end;
  ...
  procedure entry Pn (...);
    begin...end;
  begin
    initialization code
  end
```
Monitors(Cont.)

- To allow a process to wait within the monitor, a condition variable must be declared, as
  
  `var x, y: condition`

- Condition variable can only be used with the operations `wait` and `signal`.
  
  - The operation
    
    `x.wait;`
  
    means that the process invoking this operation is suspended until another process invokes
  
    `x.signal;`

  - The `x.signal` operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.
Schematic View of a monitor

- Shared data
- Operations
- Initialization code
- Entry queue
Monitor with condition variables

- Shared data
- Queues associated with $x$, $y$ conditions
- Operations
- Initialization code

Entry queue
Dining Philosophers Example

`type dining-philosophers = monitor
var state : array [0..4] of : (thinking, hungry, eating);
var self : array [0..4] of condition;
procedure entry pickup (i: 0..4);
  begin
    state[i] := hungry,
    test (i);
    if state[i] ≠ eating then self[i], wait,
  end;

procedure entry putdown (i: 0..4);
  begin
    state[i] := thinking;
    test (i+4 mod 5);
    test (i+1 mod 5);
  end;`

`Phil(I):
  dp.pickup(I);
  EAT
  dp.putdown(I);`
procedure test(k: 0..4);
begin
  if state[(k+4) mod 5] ≠ eating
    and state[k] = hungry
    and state[(k+1) mod 5] ≠ eating
  then begin
    state[k] := eating;
    self[k].signal;
  end;
end;

begin
  for i := 0 to 4
    do state[i] := thinking;
end.
Monitor Implementation Using Semaphors

- Variables
  
  ```
  var mutex: semaphore (init = 1)  (mutex in monitor)
  next: semaphore (init = 0)      (for susp. inside monitor)
  next-count: integer (init = 0)  (# of susp. proc. in next)
  ```

- Each external procedure \( F \) will be replaced by
  ```
  wait(mutex);
  ...
  body of \( F \),
  ...
  if next-count > 0
  then signal(next)
  else signal(mutex);
  ```

- Mutual exclusion within a monitor is ensured.
For each condition variable $x$, we have:

```plaintext
var x-sem: semaphore (init = 0)
    x-count: integer (init = 0)
```

The operation $x$.wait can be implemented as:

```plaintext
x-count := x-count + 1;
if next-count >0
    then signal(next)
    else signal(mutex);
wait(x-sem);
x-count := x-count – 1;
```
The operation x.signal can be implemented as:

```plaintext
if x-count > 0
    then begin
        next-count := next-count + 1;
        signal(x-sem);
        wait(next);
        next-count := next-count - 1;
    end;
```
Monitor Implementation (Cont.)

*Conditional-wait* construct: \( x.wait(c); \)

- \( c \) – integer expression evaluated when the wait operation is executed.
- value of \( c \) (*priority number*) stored with the name of the process that is suspended.
- when \( x.signal \) is executed, process with smallest associated priority number is resumed next.

- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.
Concurrent programming in Java

- Class **Thread** has several constructors. The constructor:
  - **Public Thread( String threadname )**
    Constructs a **Thread** object with name **threadname**.
  - **Public Thread()**
    Constructs a **Thread** whose name is **Thread-** concatenated with a number (e.g. **Thread-1**)
- The code of a thread is placed in its **run** method.
- The run method can be overridden in a subclass of **Thread**.
- Program launches a thread execution by calling the thread **start** method.
- Which in turn calls the **run** method.
- Caller executes concurrently with the launched thread.
Locking

- Methods or parts of methods can be declared `synchronized`.
- Each object has a lock associated with it.
- Threads can hold the lock of one or more objects.
- Only one thread can hold the lock of an object at any time.
- To execute synchronized code of an object, a thread needs to hold the lock of this object.
Wait and Notify

- In synchronized code, an object can invoke `wait()`. In that case the current thread suspends until another thread notifies the object. Invoking `wait()` releases the lock of the object.

- In synchronized code, an object can invoke `notify()`. In that case one thread, who is waiting for the lock of this object, is resumed. Notification is asynchronous; the waiting thread needs to re-obtain the object’s lock.

- Variants: `sleep(_), notifyAll()`
CPU Scheduling (briefly)

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Real-Time 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 distribution (large # of short CPU bursts, small # of long CPU bursts)

```
P1
  CPU
  I/O
  CPU
  I/O
  ...
```
Alternating Sequence of CPU And I/O Bursts

- load store
- add store
- read from file

- wait for I/O
- store increment index
- write to file

- wait for I/O
- load store
- add store
- read from file

- wait for I/O

- CPU burst
- I/O burst
- CPU burst
- I/O burst
CPU Scheduler

- Also known as *short term scheduler*.
- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- 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*.
- All other scheduling is *preemptive*. 
CPU Scheduler (Cont.)

- Note1: Ready queue not necessarily FIFO
- Note2: Records in queues are generally PCBs
Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this includes:

- switching context
- 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.

Dispatcher should be as fast as possible (since it is invoked during every context switch).
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
First-Come, First-Served (FCFS) Scheduling

- Extremely Simple
- Implemented (easily) by a FIFO queue
- However, average waiting time often long.
First-Come, First-Served (FCFS) Scheduling

Example:

<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>0</td>
<td>24</td>
<td>27</td>
<td>30</td>
</tr>
</tbody>
</table>

- 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:

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>30</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
- Nonpreemptive, troublesome for time sharing systems.
Shortest-Job-First (SJF) Scheduling

- More appropriate name: *shortest next CPU burst*.
- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>6</td>
</tr>
<tr>
<td>P₂</td>
<td>8</td>
</tr>
<tr>
<td>P₃</td>
<td>7</td>
</tr>
<tr>
<td>P₄</td>
<td>3</td>
</tr>
</tbody>
</table>

- SJF (non-preemptive)

Average waiting time = (3 + 16 + 9 + 0)/4 = 7
Example of Non-Preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- SJF (non-preemptive)

\[
\text{Average waiting time} = \frac{(0 + 6 + 3 + 7)}{4} = 4
\]
Example of Preemptive SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_1 )</td>
<td>0.0</td>
<td>7</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>2.0</td>
<td>4</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>4.0</td>
<td>1</td>
</tr>
<tr>
<td>( P_4 )</td>
<td>5.0</td>
<td>4</td>
</tr>
</tbody>
</table>

- **SJF (preemptive)**

- Average waiting time = \( \frac{(9 + 1 + 0 + 2)}{4} = 3 \)
The real difficulty with the SJF algorithm is knowing the length of the next CPU request.

In short term scheduling, there is no way to know the length of the next CPU burst. However, we may predict its value.

By approximating the next CPU burst we can pick the process with the shortest predicted CPU burst.
Priority Scheduling

- A priority number (integer) is associated with each process.
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
  - preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem ≡ Starvation – low priority processes may never execute.
- Solution ≡ Aging – as time progresses increase the priority of the process.
- Priorities can be defined internally (mem.req., # open files) or externally (money payed for computer use, etc.)
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available.
  - *Homogeneous processors* within a multiprocessor.
  - *Load sharing* (whenever several identical processors are available)
  - *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.
Real-Time Scheduling

- **Hard real-time** systems – required to complete a critical task within a guaranteed amount of time.

- **Soft real-time** computing – requires that critical processes receive priority over less fortunate ones.

- HRT: generally a process is *submitted with the amount of time* in which it needs to complete or perform IO. Scheduler either admits the process and guarantees it will complete on time, or rejects the request.

- SRT: critical processes receive *priority* (priority scheduling). This may result in longer delays or even starvation.