• Due data of HW#4 is on Sunday.

• Due data of Exercise # 5 is on Friday.

• Two more office hours on this **weekend** for HW#4.

  Saturday    3-4 pm
  Sunday      6-7 pm
Chapter 5
Concurrency: Mutual Exclusion and Synchronization

Seventh Edition
By William Stallings
Modified by Rana Forsati for CSE 410
Objectives

• To introduce the critical-section problem

• Critical section solutions can be used to ensure the consistency of shared data

• To present both software and hardware solutions of the critical-section problem

Outline

• Background

• The Critical-Section Problem

• Peterson’s Solution

• Synchronization Hardware

• Semaphores

• Classic Problems of Synchronization

• Monitors
Multiple Processes

- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing
Difficulties of Concurrency

- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible
What is the output of the following code?

```cpp
g++ -pthread -w hello.cpp -o hello.exe

using namespace std;

#include <unistd.h>    // Symbolic Constants */
#include <sys/types.h>  // Primitive System Data Types */
#include <errno.h>      // Errors */
#include <stdio.h>      // Input/Output */
#include <stdlib.h>     // General Utilities */
#include <pthread.h>    // POSIX Threads */
#include <string.h>     // String handling */
#include <iostream>

void *Hello(void *tid)
{
    int *temp = (int *) tid;
    int i = *temp;
    cout << "Hello from thread:" << i << endl;
}

int main()
{
    pthread_t thread1, thread2;    // thread variables */
    int p0 = 0;
    int p1 = 1;
    pthread_create(&thread1, NULL, Hello, (void *) &p0);
    pthread_create(&thread1, NULL, Hello, (void *) &p1);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    /* exit */
    exit(0);
} 
```
Race Condition: Shared Data

What is the output of following code?

```cpp
g++ -w -pthread sum.cpp -o sum.exe
```

The sum is: 3000
The sum is: 2999
Race Condition

**PRODUCER (count++)**

register1 = count
register1 = register1 + 1
count = register1

**CONSUMER (count--)**

register2 = count
register2 = register2 - 1
count = register2

CPU

Main Memory

```
register1 = 5
register2 = 5
```

```
count = 5
```
Consider this execution interleaving with “count = 5” initially:

- **S0**: producer execute `register1 = count`  \{`register1 = 5`\}
- **S1**: producer execute `register1 = register1 + 1`  \{`register1 = 6`\}
- **S2**: consumer execute `register2 = count`  \{`register2 = 5`\}
- **S3**: consumer execute `register2 = register2 - 1`  \{`register2 = 4`\}
- **S4**: producer execute `count = register1`  \{`count = 6`\}
- **S5**: consumer execute `count = register2`  \{`count = 4`\}

Can we control the execution order?
Race Condition (I)

- **Concurrent** access to **shared data** may result in **data inconsistency**
- Multiple processes or threads **compete** for a **shared resource**

  - Occurs when multiple processes or threads read and write shared data so that the final result **depends on the order of execution of instructions in the multiple processes.**

  - The final result **depends** on the order of execution
    - the “loser” of the race is the process that updates last and will determine the final value of the variable
- Maintaining **data consistency** requires mechanisms to ensure the **orderly execution** of cooperating processes.

**Shared Data**

Can be a shared memory variable, a global variable in a multi-thread program or a file; or a kernel variable.

**Concurrent Threads or Processes**
Atomicity (I)

- The assignment of instruction

- Consider thread 1 executing
  - \( X = 689076 \)

- Thread two executing
  - \( X = 856903 \)

- Final value of \( X \) is either 689076 or 856903
  - It cannot be 659973 (obtained by taking alternate digits from each number)

**Atomicity** means the whole instruction can be done **at once** and cannot be interrupted.
Atomicity (II)

- Let initial value of $X$ be 689076
- Consider thread one executing
  - $Y = X$
- Thread two executing
  - $X = 856903$
- Final value of $Y$ in thread one is either 689076 OR 856903
  - It cannot be 659973 (obtained by taking alternate digits from each number)
Non Atomic Actions

- Increment instruction (x++ in C++)
- sum = sum + 1 is not atomic
  - It consists of
    1. register = sum // Read value from memory into a register
    2. register = register + 1 // Increment the register (atomic)
    3. Sum = register(temp + 1) // Save the value of register in variable

Some other thread/process could read the value of sum between statements 1 and 2
Producer Consumer Problem

- Also known as the bounded-buffer problem

- Make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

or

shared fixed-size buffer
Producer Consumer Problem: A Solution

- Below is a solution to the consumer-producer problem that fills all the slots of the shared buffer.

- Use an integer count to keep track of the number of full slots.

- Initially, count is set to 0. It is incremented by the producer after it puts a new item and is decremented by the consumer after it retrieves an item from the buffer.
Producer and Consumer

Code

while (true) {
    /* produce an item */
    nextProduced = ....

    while (count == BUFFER_SIZE)
        ; // do nothing

    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}

Consumer

while (true) {
    while (count == 0)
        ; // do nothing

    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;

    /* consume item nextConsumed */

}
A Possible Problem: Race Condition

• Assume we had 5 items in the buffer

• Then:
  – Assume producer has just produced a new item and put it into buffer and is about to increment the count.
  – Assume the consumer has just retrieved an item from buffer and is about the decrement the count.
  – Namely: Assume producer and consumer are now about to execute count++ and count– statements.
Race Condition

Non atomic manipulation of `count` shared variable:

- `count++` could be implemented as (atomic operations)
  
  ```
  register1 = count
  register1 = register1 + 1
  count = register1
  ```

- `count--` could be implemented as (atomic operations)
  
  ```
  register2 = count
  register2 = register2 - 1
  count = register2
  ```
Concurrent computing makes it hard to predict all possible executions. Concurrency raises inconsistency issues [No free lunch].

But concurrency is desirable for:

- Multiprogramming
  - Multiple processes within a uniprocessor system
- Multiprocessing
  - Multiple processes within a multiprocessor
- Distributed processing
  - Multiple processes on a multiple, distributed computer system, such as a cluster
The part of the program (process) that is accessing and changing shared data is called its **critical section**.

Assuming X and Y are shared data.

**Can you give an example?**
Program Lifetime and its Structure

• Considering a process:
  – It may be executing **critical section** code from time to time
  – It may be executing **non critical section** code (remainder section) other times.

• We should not allow more than one process to be in their critical regions where they are **manipulating** the same shared data (or resource).
Structuring Programs

- The general way to do that is:

```c
do {
    critical section
    remainder section
} while (TRUE)
```

The general structure of a program works on a variable shared with other programs.

*Entry section* will allow only one process to enter and execute critical section code.

```c
do {
    entry section
    critical section
    exit section
    remainder
} while (TRUE)
```
Design and management issues raised by the existence of concurrency:

The OS must:

- be able to keep track of various processes
- allocate and de-allocate resources for each active process
- protect the data and physical resources of each process against interference by other processes
- ensure that the processes and outputs are independent of the processing speed

This is the subject of this chapter.
Solution to Critical-Section Problem

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

2. **Progress** - If two or more processes are trying to enter their critical sections, one of them will eventually succeed // **freedom from deadlock**

3. **Bounded Waiting** - If a process is trying to enter its critical section, it will eventually succeed // **freedom from starvation**

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes
**Table 5.1 Some Key Terms Related to Concurrency**

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic operation</td>
<td>A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.</td>
</tr>
<tr>
<td>critical section</td>
<td>A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.</td>
</tr>
<tr>
<td>deadlock</td>
<td>A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.</td>
</tr>
<tr>
<td>livelock</td>
<td>A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.</td>
</tr>
<tr>
<td>mutual exclusion</td>
<td>The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.</td>
</tr>
<tr>
<td>race condition</td>
<td>A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.</td>
</tr>
<tr>
<td>starvation</td>
<td>A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.</td>
</tr>
</tbody>
</table>
Concurrent processes come into conflict when they are competing for use of the same resource.

- for example: I/O devices, memory, processor time, clock

In the case of competing processes three control problems must be faced:

- the need for mutual exclusion
- deadlock
- starvation
Mutual Exclusion

Figure 5.1 Illustration of Mutual Exclusion
Requirements for Mutual Exclusion

- Must be enforced: **Only one process at a time is allowed into its critical section**
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation
- A process must not be denied access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only
Starvation is a problem where a process is perpetually denied necessary resources to proceed its work.

Dining philosophers problem:

- Forks are placed between each pair of adjacent philosophers.
- Each philosopher must alternately think and eat.
- A philosopher can only eat spaghetti when he has both left and right forks.
- Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher.
- A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before getting both of them.

- **Question:** design a discipline of behavior (a concurrent algorithm) such that each philosopher will not starve
A **deadlock** is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.
Key Terms

• **Deadlock**
  – A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something

• **Livelock**
  – A situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.

• **Race condition**
  – A situation where multiple threads/processes read and write shared data item and the final result depends on the relative timing of their execution

• **Starvation**
  – A situation in which a runnable process is overlooked infinitely often by the scheduler; although it is able to proceed it is never chosen.
Types of Solutions for concurrency control

- Software solutions
  - algorithms who’s correctness does not rely on any other assumptions (see framework)

- Hardware solutions
  - rely on some special machine instructions

- Operation System solutions
  - provide some functions and data structures to the programmer

- Programming language
  - the programming language provides a mechanism (e.g., `java.util.concurrent`)
Mutual Exclusion: Software Solution

- Peterson’s algorithm
- Lamport’s bakery algorithm
- Tournament algorithm
Framework for Analysis of Solutions

• Each process executes at nonzero speed but no assumption on the relative speed of n processes
• General structure of a process:
  repeat
    enter critical section
    critical section
    exit critical section
    non-critical section
  forever
• Many CPUs may be present but memory hardware prevents simultaneous access (read or write but not both) to the same memory location
• No assumption about order of interleaved execution
• For solutions: we need to specify entry and exit sections
• What goes inside critical section/non-critical section depends upon the application (and generally not a concern for synchronization issues).
We consider first the case of 2 processes
- We will work along a series of *failing attempts* until establishing a solution
- Peterson’s algorithm (correct solution)

Then we generalize to n processes
- the Bakery algorithm
- the Tournament based implementation

Notation
- We start with 2 processes: P0 and P1
- When presenting process Pi, Pj always denotes the other process (i ≠ j)

Note: The mutual exclusion problem is quite tricky: in the 1960's many incorrect solutions were published
First idea: use two variables \texttt{flag[0]} and \texttt{flag[1]}; if \texttt{flag[i]} is true, it means that process \textit{Pi} intends to enter the critical section.

\begin{align*}
\texttt{flag[0]} &= \texttt{flag[1]} = \texttt{false} \\
\text{Process P0:} & \text{ repeat} \\
& \quad \text{while(\texttt{flag[1]}){};} \\
& \quad \texttt{flag[0]} := \texttt{true}; \\
& \quad \texttt{CS} \\
& \quad \texttt{flag[0]} := \texttt{false}; \\
& \quad \texttt{RS} \\
& \quad \texttt{forever} \\
\text{Process P1:} & \text{ repeat} \\
& \quad \text{while(\texttt{flag[0]}){};} \\
& \quad \texttt{flag[1]} := \texttt{true}; \\
& \quad \texttt{CS} \\
& \quad \texttt{flag[1]} := \texttt{false}; \\
& \quad \texttt{RS} \\
& \quad \texttt{forever}
\end{align*}
The solution attempt I **fails** to ensure mutual exclusion!

The two processes can end up in their critical sections at the same time, as demonstrated by the following execution sequence:

**flag[0] = false**  
**flag[1] = false**

<table>
<thead>
<tr>
<th>Step</th>
<th>Process</th>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P1</td>
<td>1</td>
<td>while(flag[0]){};</td>
</tr>
<tr>
<td>2</td>
<td>P0</td>
<td>1</td>
<td>while(flag[1]){};</td>
</tr>
<tr>
<td>3</td>
<td>P0</td>
<td>2</td>
<td>flag[0]:=true;</td>
</tr>
<tr>
<td>4</td>
<td>P1</td>
<td>2</td>
<td>flag[1]:=true;</td>
</tr>
<tr>
<td>5</td>
<td>P1</td>
<td>3</td>
<td>CS</td>
</tr>
<tr>
<td>6</td>
<td>P0</td>
<td>3</td>
<td>CS</td>
</tr>
</tbody>
</table>
Second idea: switch these statements (blue) around in solution I

Process P0:
repeat
  flag[0]:=true;
  while(flag[1]){}
  CS
  flag[0]:=false;
  RS
forever

Process P1:
repeat
  flag[1]:=true;
  while(flag[0]){}
  CS
  flag[1]:=false;
  RS
forever

The solution provides mutual exclusion
- However, the processes can deadlock:

**Process P0:**
```plaintext
repeat
1: flag[0]:=true;
2: while(flag[1]){};
3: CS
4: flag[0]:=false;
5: RS
forever
```

**Process P1:**
```plaintext
repeat
1: flag[1]:=true;
2: while(flag[0]){};
3: CS
4: flag[1]:=false;
5: RS
forever
```

<table>
<thead>
<tr>
<th>Step</th>
<th>Process</th>
<th>Line</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P0</td>
<td>1</td>
<td>flag[0]:=true;</td>
</tr>
<tr>
<td>2</td>
<td>P1</td>
<td>1</td>
<td>flag[1]:=true;</td>
</tr>
<tr>
<td>3</td>
<td>P1</td>
<td>2</td>
<td>while(flag[0]){};</td>
</tr>
<tr>
<td>4</td>
<td>P0</td>
<td>2</td>
<td>while(flag[1]){};</td>
</tr>
</tbody>
</table>
Third idea: let's try something new, namely a single variable **turn** that has value 1 if it's Pi's turn to enter the critical section.

```plaintext
turn = 0 OR 1
```

**Process P0:**
```plaintext
repeat
    while (turn != 0){};
    CS
    turn = 1;
    RS
forever
```

**Process P1:**
```plaintext
repeat
    while (turn != 1){};
    CS
    turn = 0;
    RS
forever
```
Solution Attempt III

- Solution attempt III satisfies mutual exclusion

- Solution attempt III is deadlock-free

- But, solution attempt III suffers from starvation
  - Recall: processes may terminate in non-critical section
  - A problematic case: variable turn = 1, P0 trying to enter critical section (although not its turn), P1 in non-critical section
  - If P1 terminates, turn will never be set to 0: P0 will starve
Peterson’s Algorithm

- The solution attempt I **fails** to ensure mutual exclusion!

- The solution attempt II **is not** deadlock-free!

- The solution attempt III suffers from **starvation**

Peterson’s algorithm combines the ideas of solution attempts II and III (proposed in 1981)
Peterson’s algorithm combines the ideas of solution attempts II and III

flag[0] = flag[1] = false
turn= 0 OR 1

- A flag[i] value of true indicates that the process i wants to enter the critical section.

- Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting turn to 0.
If both processes have set their flag to true, then the value of `turn` decides who may enter the critical section.

**Process P0:**
```plaintext
repeat
  flag[0]:=true;
  // 0 wants in
  turn:= 1;
  // 0 gives a chance to 1
  while
      (flag[1]&turn=1)
        CS
  flag[0]:=false;
  // 0 no longer wants in
 RS
forever
```

**Process P1:**
```plaintext
repeat
  flag[1]:=true;
  // 1 wants in
  turn:=0;
  // 1 gives a chance to 0
  while
      (flag[0]&turn=0)
        CS
  flag[1]:=false;
  // 1 no longer wants in
 RS
forever
```
Peterson’s Algorithm

• Two process solution
• Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
• The two processes share two variables:
  – int turn;
  – Boolean flag[2]
• The variable turn indicates whose turn it is to enter the critical section.
• The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process Pi is ready!
Peterson’s Algorithm

- Initialization:
  flag[0]:=flag[1]:=false turn:= 0 or 1
- Willingness to enter CS specified by flag[i]:=true
- If both processes attempt to enter their CS simultaneously, only one turn value will last
- Exit section: specifies that Pi is unwilling to enter CS

Process Pi:
repeat
  flag[i]:=true;
  // I want in
  turn:=j;
  // but I let the other in
  while (flag[j]&turn=j){{};
  CS
  flag[i]:=false;
  // I no longer want in
  RS
  forever
Two processes executing concurrently

**PROCESS i (0)**

```plaintext
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
    critical section.....
    flag[i] = FALSE;
    remainder section.....
} while (1)
```

**PROCESS j (1)**

```plaintext
do {
    flag[j] = TRUE;
    turn = i;
    while (flag[i] && turn == i);
    critical section.....
    flag[j] = FALSE;
    remainder section.....
} while (1)
```

**Shared Variables**
- flag[]
- turn
Algorithm for Process $P_i$

do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

    critical section

    flag[i] = FALSE;

    remainder section

} while (1)
Peterson’s Algorithm: proof of correctness

• Mutual exclusion is preserved since:
  – P0 and P1 are both in CS only if flag[0] = flag[1] = true and only if turn = i for each Pi (impossible)

• We now prove that the progress and bounded waiting requirements are satisfied:
  – Pi cannot enter CS only if stuck in while() with condition flag[j] = true and turn = j.
  – If Pj is not ready to enter CS then flag[j] = false and Pi can then enter its CS
Peterson’s Algorithm: proof of correctness (cont.)

– If Pj has set flag[j]=true and is in its while(), then either turn=i or turn=j

– If turn=i, then Pi enters CS. If turn=j then Pj enters CS but will then reset flag[j]=false on exit: allowing Pi to enter CS

– But if Pj has time to reset flag[j]=true, it must also set turn=i

– Since Pi does not change value of turn while stuck in while(), Pi will enter CS after at most one CS entry by Pj (bounded waiting)
The Critical Section Problem: Algorithm 3

flag  false  false
  i     j

process P_i

doi{

entry section
flag[i] = true;
turn = j;
while (flag[j] && turn == j);

critical section

exit section
flag[i] = false;

remainder section

} while (1);

process P_j

do{

entry section
flag[j] = true;
turn = i;
while (flag[i] && turn == i);

critical section

exit section
flag[j] = false;

remainder section

} while (1);

Proving Property 1: Mutual Exclusion must be preserved.
The Critical Section Problem: Algorithm 3

<table>
<thead>
<tr>
<th>flag</th>
<th>false</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td>j</td>
</tr>
<tr>
<td>process P_i</td>
<td></td>
<td>process P_j</td>
</tr>
</tbody>
</table>

```plaintext
do {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j);
  flag[i] = false;
} while (1);
```

```plaintext
do {
  flag[j] = true;
  turn = i;
  while (flag[i] && turn == i);
  flag[j] = false;
} while (1);
```

**critical section**
**critical section**

**remainder section**
**remainder section**

Each P_i enters its critical section only if either flag[j] == false or turn == i.
The Critical Section Problem: Algorithm 3

flag \(i\) \(true\) \(true\)  \(\) turn \(i\) 0

process \(P_i\)

\[
\begin{align*}
do & \{ \\
& \text{entry section} \\
& \text{flag}[i] = true; \\
& \text{turn} = j; \\
& \text{while} \ (\text{flag}[j] \&\& \text{turn} == j); \\
& \text{critical section} \\
& \text{flag}[i] = false; \\
& \text{remainder section} \\
& \} \text{ while } (1); \\
\end{align*}
\]

process \(P_j\)

\[
\begin{align*}
do & \{ \\
& \text{entry section} \\
& \text{flag}[j] = true; \\
& \text{turn} = i; \\
& \text{while} \ (\text{flag}[i] \&\& \text{turn} == i); \\
& \text{critical section} \\
& \text{flag}[j] = false; \\
& \text{remainder section} \\
& \} \text{ while } (1); \\
\end{align*}
\]

Also note that, if both processes can be executing in their critical sections at the same time, then \(\text{flag}[i] == \text{flag}[j] == true\)
The Critical Section Problem: Algorithm 3

<table>
<thead>
<tr>
<th>flag</th>
<th>true</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td></td>
</tr>
</tbody>
</table>

process $P_i$

```c
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
}
```

entry section

```c
while (1);
```

exit section

```c
flag[i] = false;
```

remainder section

process $P_j$

```c
do {
    flag[j] = true;
    turn = i;
    while (flag[i] && turn == i);
}
```

critical section

```c
while (1);
```

remainder section

These two observations imply that $P_i$ and $P_j$ could not have successfully executed their while statements at about the same time, since the value of turn can be either $i$ or $j$, but not both.
The Critical Section Problem: Algorithm 3

process $P_i$  
\[
\begin{align*}
\text{flag}[i] &= \text{true;} \\
\text{turn} &= j; \\
\text{while} &\ (\text{flag}[j] \land \text{turn} == j); \\
\text{exit section} &\text{ } \\
\text{flag}[i] &= \text{false;} \\
\text{remainder section} &\text{ } \\
\} \text{ while (1);}
\end{align*}
\]

process $P_j$  
\[
\begin{align*}
\text{flag}[j] &= \text{true;} \\
\text{turn} &= i; \\
\text{while} &\ (\text{flag}[i] \land \text{turn} == i); \\
\text{critical section} &\text{ } \\
\text{flag}[j] &= \text{false;} \\
\text{remainder section} &\text{ } \\
\} \text{ while (1);}
\end{align*}
\]

These two observations imply that $P_i$ and $P_j$ could not have successfully executed their while statements at about the same time, since the value of turn can be either $i$ or $j$, but not both.
The Critical Section Problem: Algorithm 3

<table>
<thead>
<tr>
<th>flag</th>
<th>true</th>
<th>true</th>
<th>turn</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td>j</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

process P_i

```plaintext
do {
  flag[i] = true;
  turn = j;
  while (flag[j]) && turn == j);
}
```  

**entry section**

**critical section**

**exit section**

flag[i] = false;

**remainder section**

process P_j

```plaintext
do {
  flag[j] = true;
  turn = i;
  while (flag[i]) && turn == i);
}
```  

**critical section**

**exit section**

flag[j] = false;

**remainder section**

```plaintext
} while (1);  
```

Hence, one of the processes, say P_j, must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("turn == j")
The Critical Section Problem: Algorithm 3

process $P_i$

\[
\text{do}\{
\text{flag}[i] = true; \\
\text{turn} = j; \\
\text{while} (\text{flag}[j] \land \text{turn} == j); \\
\text{flag}[i] = false;
\}\text{while} (1);
\]

\text{critical section}

process $P_j$

\[
\text{do}\{
\text{flag}[j] = true; \\
\text{turn} = i; \\
\text{while} (\text{flag}[i] \land \text{turn} == i); \\
\text{flag}[j] = false;
\}\text{while} (1);
\]

\text{remainder section}

\text{critical section}

However, since at that time, flag[j] == true and turn == j, and this condition will persist as long as $P_j$ is in its critical section, Mutual Exclusion is preserved.
The Critical Section Problem: Algorithm 3

<table>
<thead>
<tr>
<th>flag</th>
<th>false</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>process P_i</td>
<td>i</td>
<td>j</td>
</tr>
<tr>
<td>turn</td>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>

**process P_i**

```plaintext
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
}
```

**critical section**

```plaintext
exit section
flag[i] = false;
```

**remainder section**

```plaintext}
while (1);
```

**process P_i**

```plaintext
do {
    flag[j] = true;
    turn = i;
    while (flag[i] && turn == i);
}
```

**critical section**

```plaintext
exit section
flag[j] = false;
```

**remainder section**

```plaintext}
while (1);
```

Proving properties 2 and 3, Progress Requirement and Bounded-Waiting
The Critical Section Problem: Algorithm 3

<table>
<thead>
<tr>
<th>flag</th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>j</td>
<td></td>
</tr>
</tbody>
</table>

Process $P_i$

```c
if (flag[i] == true) {
    turn = j;
    while (flag[j] && turn == j);
    flag[i] = false;
} while (1);
```

**critical section**

**exit section**

```c
if (flag[j] == true) {
    turn = i;
    while (flag[i] && turn == i);
    flag[j] = false;
} while (1);
```

**remainder section**

A process $P_i$ can be prevented from entering the critical section only if it is stuck in the while loop with the condition $flag[j] == true$ and $turn == j$; this loop is the only one.
The Critical Section Problem: Algorithm 3

<table>
<thead>
<tr>
<th>flag</th>
<th>true</th>
<th>false</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td>j</td>
</tr>
</tbody>
</table>

process $P_i$

```c
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
}

entry section
```

```c
while (1);
```

```c
{critical section
flag[i] = false;

remainder section
}
```

process $P_j$

```c
do {
    flag[j] = true;
    turn = i;
    while (flag[i] && turn == i);
}

critical section
```

```c
while (1);
```

```c
{flag[j] = false;

remainder section
}
```

If $P_j$ is not ready to enter the critical section, then flag[j] == false and $P_i$ can enter its critical section.
The Critical Section Problem: Algorithm 3

<table>
<thead>
<tr>
<th>flag</th>
<th>true</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>process $P_i$</td>
<td>$i$</td>
<td>$j$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>flag</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>process $P_j$</td>
<td>$i</td>
</tr>
</tbody>
</table>

**entry section**

- $flag[i] = true$;
- $turn = j$;
- while ($flag[j] && turn == j$);

**critical section**

<table>
<thead>
<tr>
<th>flag</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>process $P_j$</td>
<td>$i$</td>
</tr>
</tbody>
</table>

- $flag[j] = true$;
- $turn = i$;
- while ($flag[i] && turn == i$);

**exit section**

- $flag[i] = false$;

**remainder section**

<table>
<thead>
<tr>
<th>flag</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>process $P_j$</td>
<td>$j$</td>
</tr>
</tbody>
</table>

- $flag[j] = false$;

**remainder section**

If $P_j$ has set $flag[j]$ to true and is also executing in its while statement, then either $turn == i$ or $turn == j$.
The Critical Section Problem: Algorithm 3

process $P_i$

**entry section**

```java
flag[i] = true;
turn = j;
while (flag[j] && turn == j);
```

**critical section**

```java
flag[j] = true;
turn = i;
while (flag[i] && turn == i);
```

**exit section**

```java
flag[i] = false;
```

**remainder section**

```java
flag[j] = false;
```

} while (1);

**remainder section**

} while (1);

If turn == i then $P_i$ will enter its critical section.
The Critical Section Problem: Algorithm 3

process $P_i$

\begin{align*}
\text{do } & \{ \\
\text{entry section} & \quad \text{flag}[i] = \text{true}; \\
& \quad \text{turn} = j; \\
& \quad \text{while} (\text{flag}[j] \&\& \text{turn} == j); \\
\text{critical section} & \quad \text{flag}[i] = \text{false}; \\
\text{exit section} & \quad \text{flag}[j] = \text{false}; \\
\text{remainder section} & \quad \text{flag}[j] = \text{false}; \\
\} \text{ while (1);}
\end{align*}

process $P_j$

\begin{align*}
\text{do } & \{ \\
\text{entry section} & \quad \text{flag}[j] = \text{true}; \\
& \quad \text{turn} = i; \\
& \quad \text{while} (\text{flag}[i] \&\& \text{turn} == i); \\
\text{critical section} & \quad \text{flag}[j] = \text{false}; \\
\text{exit section} & \quad \text{flag}[i] = \text{false}; \\
\text{remainder section} & \quad \text{flag}[j] = \text{false}; \\
\} \text{ while (1);}
\end{align*}

However, if turn == j and $P_j$ enters its critical section, once $P_j$ exits its critical section, it will reset flag[j] to false, allowing $P_i$ to enter its critical section.
The Critical Section Problem: Algorithm 3

If \( P_j \) returns to the start of its outer loop and resets flag \([j]\) to true, it must also set turn to "i" in order to get to its own while statement again.
Thus, since $P_i$ does not change the value of the variable turn while executing the while statement, $P_i$ will enter the critical section (Progress) after at most one entry by $P_j$ (Bounded Waiting).
A Tournament-Based Algorithm

- How can we use a two-thread algorithm to construct an algorithm for many threads?

- Each leaf node is a process (thread)

- Each internal node runs Peterson’s algorithm

Peterson’s Algorithm

leaf nodes: Threads

To Enter Critical Section
A Tournament-Based Algorithm

Reducing the number of shared memory accesses
A Tournament-Based Algorithm

- Each thread is progressing from the leaf to the root, where at each level of the tree it participates in a two thread mutual exclusion algorithm.

- As a thread advanced towards the root, it plays the role of thread 0 when it arrives from the left subtree, or of thread 1 when it arrives from the right subtree.
Analogy [Wikipedia]

- Each customer is given a unique number.
- Numbers increase by one as customers enter the store.
- A global counter displays the number of the customer that is currently being served.
- All other customers must wait in a queue until the baker finishes serving the current customer and the next number is displayed.
- When the customer is done shopping and has disposed of his or her number, the clerk increments the number, allowing the next customer to be served.
- That customer must draw another number from the numbering machine in order to shop again.
N-process Solution: Bakery Algorithm

- Before entering their CS, each Pi receives a number ().
- Holder of smallest number enter CS (like in bakeries, ice-cream stores...)
- When Pi and Pj receives same number:
  - if i<j then Pi is served first, else Pj is served first
- Pi resets its number to 0 in the exit section
- Notation:
  - \((a,b) < (c,d)\) if \(a < c\) or if \(a = c\) and \(b < d\)
  - \(\max(a_0,...a_k)\) is a number \(b\) such that
    - \(b \geq a_i\) for \(i=0,...,k\)
The Bakery Algorithm (cont.)

- **Shared data:**
  - choosing: array[0..n-1] of boolean;
    - initialized to false
  - number: array[0..n-1] of integer;
    - initialized to 0

- **Correctness relies on the following fact:**
  - If Pi is in CS and Pk has already chosen its number[k] != 0, then (number[i],i) < (number[k],k)
  - but the proof is somewhat tricky...
Process Pi:
repeat
  choosing[i]:=true;
  number[i]:=max(number[0]..number[n-1])+1;
  choosing[i]:=false;
  for j:=0 to n-1 do {
    while (choosing[j]) {}
    while (number[j]!=0
      and (number[j],j)<(number[i],i)){}
  }
  CS
  number[i]:=0;
  RS
forever
Drawbacks of Software Solutions

- Processes that are requesting to enter in their critical section are **busy waiting** (consuming processor time needlessly)

- If **Critical Sections are long**, it would be more efficient to block processes that are waiting...
Mutual Exclusion: Hardware Solution
Mutual Exclusion: Hardware Support

- We can use some **hardware support** (if available) for protecting critical section code
  
  - 1) Disable interrupts?
    - Sometimes only (kernel)
    - Not on multiprocessors
  
  - 2) Special machines instructions and **lock variables**
    - Test and Set
    - Swap
Interrupt Disabling (I)

- Consider a uniprogramming system
- A process runs until it invokes an operating system service or until it is interrupted

```plaintext
Process Pi:
repeat
    Disable interrupts;
    Critical Section
    Enable interrupts;
    Remainder Section
forever
```
Interrupt Disabling (II)

- **Interrupt Disabling**
  - Uniprocessor system
  - Disabling interrupts guarantees mutual exclusion

- **Disadvantages:**
  - The efficiency of execution could be noticeably degraded: i.e., processor is limited in its ability to interleave programs
  - This approach will not work in a multiprocessor architecture
  - Disabling interrupts on one processor will not guarantee mutual exclusion
Hardware Solutions: Special Machine Instructions

- Normally, access to a memory location excludes other access to that same location.

- Extension: designers have proposed machines instructions that perform two actions atomically (indivisible) on the same memory location (ex: reading and writing).

- The execution of such an instruction is mutually exclusive (even with multiple CPUs).

- They can be used simply to provide mutual exclusion but need more complex algorithms for satisfying the three requirements of the CS problem.
• Some possible atomic (non-interruptable) machine instructions:
  – **TestAndSet** instruction (TSL):
    *test memory word and set value*

  – **Swap** instruction (EXCH instruction in Intel x86 arch):
    *swap contents of two memory words*

• They can be executed **atomically** in a **multi-processor environment** as well (one CPU at a time executes the instruction: it involves memory access; memory is shared)
The test-and-set instruction

- **Is a machine/assembly** instruction.
  - But here we provide definition of it using a high level language code.

```cpp
bool testset(int& i) {
    if (i==0) {
        i=1;
        return true;
    } else {
        return false;
    }
}
```

- **A C++ description of test-and-set:**

```cpp
Process Pi:
repeat
    repeat{}
    until testset(b);
    CS
    b:=0;
    RS
forever
```

- **An algorithm that uses testset for Mutual Exclusion:**
- Shared variable b is initialized to 0
- Only the first Pi who sets b enter CS
The test-and-set instruction

- To use it, we need to program in assembly language. i.e., entry section code should be programmed in assembly.

Solution

We use a shared Boolean variable lock, initialized to false.

```c
do {
    while ( TestAndSet (&lock ) )
    ; // do nothing

    // critical section
    lock = FALSE;

    // remainder section
}
} while (TRUE);
```
The test-and-set instruction

- Mutual exclusion is preserved: if Pi enter CS, the other Pj are busy waiting
- Problem: still using busy waiting
- When Pi exit CS, the selection of the Pj who will enter CS is arbitrary: no bounded waiting. Hence starvation is possible

- Processors (ex: Pentium) often provide an atomic xchg(a,b) instruction that swaps the content of a and b.
- But xchg(a,b) suffers from the same drawbacks as test-and-set
Using `xchg` for mutual exclusion

- Is a machine/assembly instruction. Intel 80x86 architecture has an XCHG instruction
  - But here we provide definition of it using a high level language code.

- Shared variable `b` is initialized to 0
- Each Pi has a local variable `k`
- The only Pi that can enter CS is the one who finds `b=0`
- This Pi excludes all the other Pj by setting `b` to 1

```
Process Pi:
repeat
  k:=1
  repeat xchg(k,b) until k=0;
  CS
  b:=0;
  RS
forever
```
Using xchg for mutual exclusion

- Is a machine/assembly instruction. Intel 80x86 architecture has an XCHG instruction
  - But here we provide definition of it using a high level language code.

```c
void xchg (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
Using xchg for mutual exclusion

- Need to program in assembly to use. Hence Entry section code should be programmed in assembly

Solution
We use a shared Boolean variable lock initialized to FALSE. Each process also has a local Boolean variable key

do {
    key = TRUE;
    while ( key == TRUE)
        xchg (&lock, &key);
    //    critical section
    lock = FALSE;
    //      remainder section
}  while (TRUE);

We use a shared Boolean variable lock initialized to FALSE. Each process also has a local Boolean variable key.
Special Machine Instruction: Advantages

- Applicable to any number of processes on either a single processor or multiple processors sharing main memory
- Simple and easy to verify
- It can be used to support multiple critical sections; each critical section can be defined by its own variable
Special Machine Instruction: Disadvantages

- **Busy-waiting is employed**, thus while a process is waiting for access to a critical section it continues to consume processor time.
- **Starvation is possible** when a process leaves a critical section and more than one process is waiting.
- **Deadlock is possible** if a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region.
bool testset(int& i) {
    if (i==0) {
        i=1;
        return true;
    } else {
        return false;
    }
}
Suppose there is a resource that can be used by up to 2 processes at a time. The method `AccessResource` provides access to this resource. To ensure that only two processes can access resource at a time,

Solve this problem using testset instruction.

```cpp
bool testset(int& i) {
    if (i==0) {
        i=1;
        return true;
    } else
    { return false;
    }
}
```

```cpp
while( !(testset(a) || testset(b)) ) {}
AccessResource();
if (b == 1)
b = 0;
else
a = 0;
```
### Common Concurrency Mechanisms

<table>
<thead>
<tr>
<th><strong>Semaphore</strong></th>
<th>An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a <strong>counting semaphore</strong> or a <strong>general semaphore</strong>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Binary Semaphore</strong></td>
<td>A semaphore that takes on only the values 0 and 1.</td>
</tr>
<tr>
<td><strong>Mutex</strong></td>
<td>Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).</td>
</tr>
<tr>
<td><strong>Condition Variable</strong></td>
<td>A data type that is used to block a process or thread until a particular condition is true.</td>
</tr>
<tr>
<td><strong>Monitor</strong></td>
<td>A programming language construct that encapsulates variables, access procedures and initialization code within an abstract data type. The monitor’s variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are <strong>critical sections</strong>. A monitor may have a queue of processes that are waiting to access it.</td>
</tr>
<tr>
<td><strong>Event Flags</strong></td>
<td>A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).</td>
</tr>
<tr>
<td><strong>Mailboxes/Messages</strong></td>
<td>A means for two processes to exchange information and that may be used for synchronization.</td>
</tr>
<tr>
<td><strong>Spinlocks</strong></td>
<td>Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.</td>
</tr>
</tbody>
</table>
Semaphore
Semaphore

- Synchronization tool (provided by the OS) that do not require busy waiting
- Semaphore $S$: integer variable
- A semaphore $S$ is an integer variable that, apart from initialization, can only be modified through two atomic and mutually exclusive operations:
  - Wait ($S$)
  - Signal ($S$)

- If a process is waiting for a signal, it is suspended until that signal is sent
- Less complicated entry and exit sections when semaphores are used
Semaphore Abstraction

- Define semaphore as a record (structure):

```cpp
class semaphore
{
public:
    semaphore(int) // constructor
    atomic semWait(semaphore S);
    atomic semSignal(semaphore S);

private:
    int count;
    queue: list of process
}
```

There is no way to inspect or manipulate semaphores other than these three operations.

Atomic methods! Guaranteed by OS.
Semaphore

Atomic Operations on semaphore S:

1) May be initialized to a nonnegative integer value

2) The **Wait** operation decrements the value: if the value of S becomes negative ($S < 0$), the process executing `semWait` is blocked; otherwise continues execution

4) The **Signal** operation increments the value: if the value of S becomes less than equal zero ($S <= 0$), one blocked process, if any, is unblocked
Semaphore Operations

**semWait(S):**

```
S.count--;
if (S.count < 0) {
    block this process
    place this process in S.queue
}
```

**semSignal(S):**

```
S.count++;
if (S.count <= 0) {
    remove a process P from S.queue
    place this process P on ready list
}
```

S.count must be initialized to a value (depending on application)
Mutual Exclusion with Semaphores

**Critical Section of n Processes**

- Shared data:
  
  ```
  semaphore mutex; /* initially mutex = 1 */
  ```

- Process $P_i$:
  ```
  do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
  } while (1);
  ```

**Issues to think about**

- Why count initialized to 1
- What would happen if set to 0
- What if set to 2

**To allow $n$ processes into CS?**

- Initialize mutex to 1
- Then only 1 process is allowed into CS (mutual exclusion)
Mutual Exclusion with Semaphores

- To allow \( n \) processes into CS, we initialize \( S \).count to \( n \)
- \( \text{semaphore } s = n \)

Process \( P_i \):

```
repeat
    semWait(S);
    CS
    semSignal(S);
    RS
forever
```
Process Synchronization with Semaphores

Question:

- Suppose we have 2 processes: P1 and P2
- Where P1 contains code $B_1$ and P2 contains code $B_2$
- We want to ensure code block $B_1$ in P1 executes before code block $B_2$ in P2?

Use semaphores to synchronize the order of execution of the two processes?
Process Synchronization with Semaphores

Question:

- We have 2 processes: P1 and P2
- Code block B1 in P1 needs to be performed before code block B2 in P2

Solution:

- Define a semaphore “synch”
- Initialize synch to 0

- Semaphore synch=0; //initial value of semaphore
- Process P1:
  - B1; //execute B1
  - Signal(synch);
- Process P2:
  - Wait(synch);
  - B2; //execute B2
Semaphores Example

Question:

• Suppose We have 3 processes: P1, P2 and P3
• where P1 prints A and P2 prints B and P3 prints C
• We want to ensure to have ABC in output?
Semaphores Example

- Semaphore x, y=0 and z=1; //initial value of semaphore
- Process P1:
  - Wait (z)
  - Print “A”
  - Signal(x);
- Process P2:
  - Wait(x);
  - Print “B”
  - Signal(y);
- Process P3:
  - Wait(y);
  - Print “C”
  - Signal(z);
Assume we have a resource that has 5 instances.
A process need to use one instance.
We can allow at most 5 process concurrently using these 5 resource instances. Another process (processes) that want the resource need to block.
How can we code those processes?

Solution:
Assume we have a resource that has 5 instances.
A process that needs that type of resource will need to use one instance.
We can allow at most 5 process concurrently using these 5 resource instances. Another process (processes) that want the resource need to block. How can we code those processes?

**Solution:**

one of the processes creates and initializes a semaphore to 5.

```c
semaphore x = 5; // semaphore to access resource
```

wait (x);
...
....use one instance of the resource...
...
signal (x);

Each process has to be coded in this manner.
Mutual Exclusion with Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>start</td>
<td>start</td>
<td>start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>critical section</td>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>do remainder</td>
<td>do remainder</td>
<td>do remainder</td>
</tr>
</tbody>
</table>

Process A begins execution.
**Mutual Exclusion with Semaphores**

Mutual Exclusion Achieved Through Semaphores

**Mutex Value** | **Mutex List**
---|---
1 | 

**wait (mutex)**
{ value = value - 1;
  if (value < 0)
  { add process to list;
    block process;
  }
}

**signal (mutex)**
{ value = value + 1;
  if (value <= 0)
  { remove process from list;
    wake up process;
  }
}

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Process A executes the wait (mutex) instruction.
## Mutual Exclusion with Semaphores

### Flowchart:

- **Process A**
  - start
  - wait (mutex);
  - critical section
  - signal (mutex);
  - do remainder

- **Process B**
  - start
  - wait (mutex);
  - critical section
  - signal (mutex);
  - do remainder

- **Process C**
  - start
  - wait (mutex);
  - critical section
  - signal (mutex);
  - do remainder

### Code Snippet:

```c
wait (mutex) {
    value = value - 1;
    if (value < 0) {
        add process to list;
        block process;
    }
}
```

```c
signal (mutex) {
    value = value + 1;
    if (value <= 0) {
        remove process from list;
        wake up process;
    }
}
```

### Text:

The Mutex Value is decremented by 1.
**Mutual Exclusion with Semaphores**

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**wait (mutex)**

```plaintext
{ value = value - 1;
  if (value < 0)
    { add process to list;
      block process;
    }
}
```

**signal (mutex)**

```plaintext
{ value = value + 1;
  if (value <= 0)
    { remove process from list;
      wake up process;
    }
}
```

**Process A**

- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**

- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**

- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Comparison is made to determine if a process is already in its critical section.
Mutual Exclusion with Semaphores

Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>wait (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>{ value = value - 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (value &lt; 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ add process to list;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>block process;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signal (mutex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ value = value + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (value &lt;= 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ remove process from list;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake up process;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>start</td>
<td>start</td>
</tr>
<tr>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
</tr>
<tr>
<td>do remainder</td>
<td>do remainder</td>
<td>do remainder</td>
</tr>
</tbody>
</table>

Test fails, decision statement & wait (mutex) are exited.
### Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>wait (mutex)</th>
<th>signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>{ value = value - 1; if (value &lt; 0) { add process to list; block process; } }</td>
<td>{ value = value + 1; if (value &lt;= 0) { remove process from list; wake up process; } }</td>
</tr>
</tbody>
</table>

### Process A
- **start**
- wait (mutex);
- **critical section**
- signal (mutex);
- do remainder

### Process B
- **start**
- wait (mutex);
- **critical section**
- signal (mutex);
- do remainder

### Process C
- **start**
- wait (mutex);
- **critical section**
- signal (mutex);
- do remainder

Process A is allowed to enter its critical section.
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**wait (mutex)**
```
{  value = value - 1;
    if (value < 0)
    {  add process to list;
        block process;
    }
}
```

**signal (mutex)**
```
{  value = value + 1;
    if (value <= 0)
    {  remove process from list;
        wake up process;
    }
}
```

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

*Context Switch by CPU Scheduler, Process B begins execution.*
### Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Wait (mutex)</th>
</tr>
</thead>
</table>
| 0           |            | {
|             |            |   value = value - 1;
|             |            |   if (value < 0) {
|             |            |     add process to list;
|             |            |     block process;
|             |            |   }
|             |            | }

<table>
<thead>
<tr>
<th></th>
<th>Signal (mutex)</th>
</tr>
</thead>
</table>
|          | {
|          |   value = value + 1;
|          |   if (value <= 0) {
|          |     remove process from list;
|          |     wake up process;
|          | }
|          | }

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>start</td>
<td>start</td>
</tr>
<tr>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
</tr>
<tr>
<td>do remainder</td>
<td>do remainder</td>
<td>do remainder</td>
</tr>
</tbody>
</table>

Context Switch by CPU Scheduler, Process B begins execution.
# Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Wait (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td>{ value = value - 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (value &lt; 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ add process to list;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>block process;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

signal (mutex)

{ value = value + 1;

if (value <= 0)

{ remove process from list;

wake up process;

} | |

Process A  
start  
wait (mutex);  
critical section  
signal (mutex);  
do remainder

Process B  
start  
wait (mutex);  
critical section  
signal (mutex);  
do remainder

Process C  
start  
wait (mutex);  
critical section  
signal (mutex);  
do remainder

The Mutex Value is decremented by 1.
### Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
wait (mutex)
{  value = value - 1;
   if (value < 0)
   {  add process to list;
      block process;
   }
}
```

```plaintext
signal (mutex)
{  value = value + 1;
   if (value <= 0)
   {  remove process from list;
      wake up process;
   }
}
```

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>start wait (mutex); critical section</td>
<td>start wait (mutex); critical section</td>
<td>start wait (mutex); critical section</td>
</tr>
<tr>
<td>signal (mutex); do remainder</td>
<td>signal (mutex); do remainder</td>
<td>signal (mutex); do remainder</td>
</tr>
</tbody>
</table>

Comparison is made to determine if a process is already in its critical section.
## Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>B</td>
</tr>
</tbody>
</table>

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Test passes, Process B is added to the Mutex List.
### Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Code</th>
</tr>
</thead>
</table>
| -1          | B          | \[
|             |            | wait (mutex)  
|             |            | \{ value = value - 1; \}  
|             |            | \{ if (value < 0) \}  
|             |            | \{ add process to list; block process; \}  
|             |            | \}  
|             |            | \}  
|             |            | \]  

\[
| signal (mutex) \{ value = value + 1; \} \{ if (value <= 0) \} \{ remove process from list; wake up process; \} \} \]  

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B is blocked.**
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>B</td>
</tr>
</tbody>
</table>

```
wait (mutex)
{ value = value - 1;
  if (value < 0)
    { add process to list;
      block process;
    }
}

signal (mutex)
{ value = value + 1;
  if (value <= 0)
    { remove process from list;
      wake up process;
    }
}
```

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>start</td>
<td>start</td>
</tr>
<tr>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
</tr>
<tr>
<td>do remainder</td>
<td>do remainder</td>
<td>do remainder</td>
</tr>
</tbody>
</table>

Context Switch, Process C starts execution.
## Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Wait (mutex)</th>
<th>Signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>B</td>
<td>{ value = value - 1; if (value &lt; 0) { add process to list; block process; } }</td>
<td>{ value = value + 1; if (value &lt;= 0) { remove process from list; wake up process; } }</td>
</tr>
</tbody>
</table>

### Process A
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

### Process B
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

### Process C
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C executes the wait (mutex) instruction.**
# Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

- **wait (mutex)**
  - `value = value - 1;`
  - `if (value < 0)`
    - `add process to list;`
    - `block process;`
  - `}

- **signal (mutex)**
  - `value = value + 1;`
  - `if (value <= 0)`
    - `remove process from list;`
    - `wake up process;`
  - `}

---

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

---

*The Mutex Value is decremented by 1.*
**Mutual Exclusion Achieved Through Semaphores**

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Wait (mutex)</th>
<th>Signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>B</td>
<td>{ value = value - 1; if (value &lt; 0) { add process to list; block process; } }</td>
<td>{ value = value + 1; if (value &lt;= 0) { remove process from list; wake up process; } }</td>
</tr>
</tbody>
</table>

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Comparison is made to determine if a process is already in its critical section.
## Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Wait (mutex)</th>
<th>Signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>B, C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ value = value - 1; if (value &lt; 0) { add process to list; block process; } }</td>
<td>{ value = value + 1; if (value &lt;= 0) { remove process from list; wake up process; } }</td>
</tr>
</tbody>
</table>

### Process A
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

### Process B
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

### Process C
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

*Test passes, Process C is added to the Mutex List.*
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>B C</td>
</tr>
</tbody>
</table>

```plaintext
wait (mutex)
{ value = value - 1;
  if (value < 0)
    { add process to list;
      block process;
    }
}
```

```plaintext
signal (mutex)
{ value = value + 1;
  if (value <= 0)
    { remove process from list;
      wake up process;
    }
}
```

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>start</td>
<td>start</td>
</tr>
<tr>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
</tr>
<tr>
<td>do remainder</td>
<td>do remainder</td>
<td>do remainder</td>
</tr>
</tbody>
</table>

Process C is blocked.
## Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>wait (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>B C</td>
<td>{ value = value - 1; if (value &lt; 0) { add process to list; block process; } }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signal (mutex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ value = value + 1; if (value &lt;= 0) { remove process from list; wake up process; } }</td>
</tr>
</tbody>
</table>

### Process A
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

### Process B
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

### Process C
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**BLOCKED**

Context Switch, Process A resumes execution.
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Wait (mutex)</th>
<th>Signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>B</td>
<td>{ value = value - 1; if (value &lt; 0) { add process to list; block process; } }</td>
<td>{ value = value + 1; if (value &lt;= 0) { remove process from list; wake up process; } }</td>
</tr>
</tbody>
</table>

Process A
start
wait (mutex);
critical section
signal (mutex);
do remainder

Process B
start
wait (mutex);
critical section
signal (mutex);
do remainder

Process C
start
wait (mutex);
critical section
signal (mutex);
do remainder

Process A leaves its critical section, and executes the signal (mutex) instruction.
## Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Code Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>B C</td>
<td>wait (mutex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ value = value - 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\quad if (value &lt; 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\quad \quad { add process to list;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\quad \quad \quad block process;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signal (mutex)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ value = value + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\quad if (value &lt;= 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\quad \quad { remove process from list;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\quad \quad \quad wake up process;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}}</td>
</tr>
</tbody>
</table>

### Process A
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

### Process B
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

### Process C
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

The Mutex Value is incremented by 1.
### Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Wait (mutex)</th>
<th>Signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>B C</td>
<td><code>{ value = value - 1;</code></td>
<td><code>{ value = value + 1;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (value &lt; 0)</td>
<td>if (value &lt;= 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ add process to list;</td>
<td>{ remove process from list;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>block process;</td>
<td>wake up process;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

#### Process A
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

#### Process B
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

#### Process C
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

A comparison is made to determine if any processes are waiting in the Mutex List.
### Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Wait (mutex)</th>
<th>Signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>C</td>
<td>{ value = value - 1; if (value &lt; 0) { add process to list; block process; } }</td>
<td>{ value = value + 1; if (value &lt;= 0) { remove process from list; wake up process; } }</td>
</tr>
</tbody>
</table>

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Test passes, the first process in the Mutex List is removed and is woken up.
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>wait (mutex)</th>
<th>signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>C</td>
<td>{ value = value - 1; if (value &lt; 0) { add process to list; block process; } }</td>
<td>{ value = value + 1; if (value &lt;= 0) { remove process from list; wake up process; } }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>start</td>
<td>start</td>
</tr>
<tr>
<td>wait (mutex); critical section</td>
<td>wait (mutex); critical section</td>
<td>wait (mutex); critical section</td>
</tr>
<tr>
<td>signal (mutex); do remainder</td>
<td>signal (mutex); do remainder</td>
<td>signal (mutex); do remainder</td>
</tr>
</tbody>
</table>

Context Switch, Process B resumes execution & enters its critical section.
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Wait (mutex)</th>
<th>Signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ value = value - 1;</td>
<td>{ value = value + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (value &lt; 0)</td>
<td>if (value &lt;= 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ add process to list;</td>
<td>{ remove process from list;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>block process;</td>
<td>wake up process;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

Process A
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Process B
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Process C
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Process B exits its critical section and executes the signal (mutex) instruction.
## Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
<td>start</td>
<td>start</td>
<td>start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>critical section</td>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>do remainder</td>
<td>do remainder</td>
<td>do remainder</td>
</tr>
</tbody>
</table>

The Mutex Value is incremented by 1.

- **wait (mutex)**
  
  ```
  { value = value - 1;
  if (value < 0)
    { add process to list;
      block process;
    }
  }
  ```

- **signal (mutex)**
  
  ```
  { value = value + 1;
  if (value <= 0)
    { remove process from list;
      wake up process;
    }
  }
  ```
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
</tr>
</tbody>
</table>

```
wait (mutex)
{  value = value - 1;
   if (value < 0)
      {  add process to list;
          block process;
      }
}
```

```
signal (mutex)
{  value = value + 1;
   if (value <= 0)
      {  remove process from list;
          wake up process;
      }
}
```

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

A comparison is made to determine if any processes are waiting in the Mutex List.
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>wait (mutex)</th>
<th>signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>{ value = value - 1;</td>
<td>{ value = value + 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (value &lt; 0)</td>
<td>if (value &lt;= 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ add process to list;</td>
<td>{ remove process from list;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>block process;</td>
<td>wake up process;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Test passes, the first process in the Mutex List is removed from the list and woken up.
## Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Code</th>
</tr>
</thead>
</table>
| 0           |            | ```
wait (mutex)
{ value = value - 1;
   if (value < 0)
     { add process to list;
       block process;
     }
}
``` |
|             |            | ```
signal (mutex)
{ value = value + 1;
   if (value <= 0)
     { remove process from list;
      wake up process;
     }
}
``` |

### Process A
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

### Process B
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

### Process C
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

*Context Switch, Process C resumes execution and enters its critical section.*
### Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>wait (mutex)</th>
<th>signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td><code>{ value = value - 1;</code></td>
<td><code>{ value = value + 1;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>if (value &lt; 0) { add process to list;</code></td>
<td><code>if (value &lt;= 0) { remove process from list;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>block process;</code></td>
<td><code>wake up process;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Process C exits its critical section and executes the signal (mutex) instruction.
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>start</td>
<td>start</td>
<td>start</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
<td>wait (mutex);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>critical section</td>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
<td>signal (mutex);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>do remainder</td>
<td>do remainder</td>
<td>do remainder</td>
</tr>
</tbody>
</table>

wait (mutex)
{ value = value - 1;
  if (value < 0)
  { add process to list;
    block process;
  }
}
signal (mutex)
{ value = value + 1;
  if (value <= 0)
  { remove process from list;
    wake up process;
  }
}

The Mutex Value is incremented by 1.
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>Wait (mutex)</th>
<th>Signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>wait (mutex)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{} value = value - 1;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (value &lt; 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{} add process to list;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>block process;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>signal (mutex)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{} value = value + 1;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (value &lt;= 0)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>{} remove process from list;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake up process;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Process A
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Process B
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Process C
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Test fails, there are no processes waiting in the Mutex List.
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>wait (mutex)</th>
<th>signal (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>{ value = value - 1; if (value &lt; 0) { add process to list; block process; } }</td>
<td>{ value = value + 1; if (value &lt;= 0) { remove process from list; wake up process; } }</td>
</tr>
</tbody>
</table>

**Process A**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process B**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

**Process C**
- start
- wait (mutex);
- critical section
- signal (mutex);
- do remainder

Process A is allowed to enter its critical section.
Mutual Exclusion Achieved Through Semaphores

<table>
<thead>
<tr>
<th>Mutex Value</th>
<th>Mutex List</th>
<th>wait (mutex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>{ value = value - 1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (value &lt; 0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ add process to list;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>block process;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>

signal (mutex)
{ value = value + 1;
if (value <= 0)
{ remove process from list;
wake up process;
} |
|

Process A
start
wait (mutex);
critical section
signal (mutex);
do remainder

Process B
start
wait (mutex);
critical section
signal (mutex);
do remainder

Process C
start
wait (mutex);
critical section
signal (mutex);
do remainder

gac procees with remainder of code.

As you can see, Semaphores can be used to achieve mutual exclusion, even when a context switch could possibly take place during execution of a critical section.
Strong/Weak Semaphores

- To avoid busy waiting: when a process has to wait, it will be put in a blocked queue of processes waiting for the same event.
- The Signal operation removes one process from the queue and puts it in the list of ready processes.

*A queue is used to hold processes waiting on the semaphore.*
The question arises of the order in which processes are removed from such a queue.

**Strong Semaphores**
- the process that has been blocked the longest is released from the queue first (FIFO)

**Weak Semaphores**
- the order in which processes are removed from the queue is not specified

**Note:** strong semaphores guarantee freedom from starvation, while weak semaphores do not
Semaphore Observations

• When $S\.count \geq 0$: the number of processes that can execute $\text{semWait}(S)$ \textbf{without being blocked} $= S\.count$
• When $S\.count < 0$: the \textbf{number of processes waiting} on $S \mid S\.count\mid$, to be signaled (unblocked)
• Atomicity and mutual exclusion: no 2 processes can be in $\text{semWait}(S)$ and $\text{semSignal}(S)$ (on the same $S$) at the same time (even with multiple CPUs)
• Hence the blocks of code defining $\text{semWait}(S)$ and $\text{semSignal}(S)$ are, in fact, critical sections
Users of semaphore rely on the fact that semWait and semSignal are executed atomically.

OS needs to ensure that semWait and semSignal are executed atomically
  - How can the OS do that?
  - Is this similar to critical section problem?
    - Yes: count and queue are shared variables by all processes that use the same semaphore
    - We need to make sure that only one accesses it at a time.
    - Wonderful: the same critical section problem again.
  - Properties of wait and signal as critical sections themselves
    - Are they long or short?
Two Types of Semaphores

- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks**
  - Binary semaphores provides mutual exclusion; can be used for the critical section problem.
  - Binary semaphores can be used for process synchronization.

- **Counting** semaphore – integer value can range over an unrestricted domain
  - Can be used for other synchronization problems; for example for resource allocation.
Binary Semaphore

- The semaphores we have studied are called **counting** (or integer) semaphores/general semaphores

- We have also **binary semaphores**
  - similar to counting semaphores except that “count” is *Boolean* valued
  - counting semaphores can be implemented by binary semaphores… [Problem 5.17 in textbook]
  - generally more difficult to use than counting semaphores (eg: they cannot be initialized to an integer $k > 1$)
Binary Semaphore

\[
\text{waitB}(S): \\
\text{\hspace{1cm} if (S.value = 1) \{ \\
\text{\hspace{1.5cm} S.value := 0; \\
\text{\hspace{1.5cm}} \} else \{ \\
\text{\hspace{2.5cm} block this process \\
\text{\hspace{2.5cm} place this process in S.queue \\
\text{\hspace{1.5cm}}} \}
\]
\]

\[
\text{signalB}(S): \\
\text{\hspace{1cm} if (S.queue is empty) \{ \\
\text{\hspace{1.5cm} S.value := 1; \\
\text{\hspace{1.5cm}} \} else \{ \\
\text{\hspace{2.5cm} remove a process P from S.queue \\
\text{\hspace{2.5cm} place this process P on ready list \\
\text{\hspace{1.5cm}}} \}
\]
\]
We got semaphores?

- What can we do with them?
  - Can we solve critical section problem?
  - Can we solve classical problems of synchronization?

- Mutual Exclusion
- Process Synchronization
- Producer Consumer Problem
- The Dining Philosophers Problem
Producer/Consumer Problem

- N buffers, each can hold one item
- Producer takes an empty buffer and produces a full buffer
- Consumer takes a full buffer and produces an empty buffer
- Can consume only when number \( n \) of (consumable) items is at least 1
- Can produce only when number of empty spaces is at least 1
- only 1 process at a time can access the buffer
Producer/Consumer Problem

General Situation:
- one or more producers are generating data and placing these in a buffer
- a single consumer is taking items out of the buffer one at time
- only one producer or consumer may access the buffer at any one time

The Problem:
- ensure that the producer can’t add data into full buffer
- consumer can’t remove data from an empty buffer
- they can not access to buffer concurrently

- Ex1: a **print program** produces characters that are **consumed** by a printer
- Ex2: an assembler produces object modules that are consumed by a loader
we need a **binary semaphore S** to have mutual exclusion on **buffer access**: only 1 process at a time can access the buffer.

Semaphore S initialized to the value 1

We need two **counting semaphores** full and empty to keep **track of the number of full and empty buffers**.

Semaphore full to synchronize producer and consumer on the number of consumable items.

Semaphore empty to synchronize producer and consumer on the number of empty spaces.

Semaphore **full** initialized to the value 0.

Semaphore **empty** initialized to the value n (buffer size).
Why do we need counting semaphores? Why do we need two of them? Consider the following:

**Producer process**

```plaintext
do {
    //need code here to keep track of number of empty buffers
    wait (mutex);
    // obtain empty buffer and add next item, creating a full buffer
    signal (mutex);
    } while (TRUE);
```

**Consumer process**

```plaintext
do {
    wait (mutex);
    // remove item from full buffer, create an empty buffer
    signal (mutex);
    ......
    } while (TRUE);
```
we add an empty counting semaphore initialized to n, then a producer calling wait(EMPTY) will keep decrementing until 0, replacing n empty buffers with n full ones. If the producer tries to produce any more, wait(EMPTY) blocks producer until more empty buffers are available.
this is the proper behavior that we want!

**Producer process**

```plaintext
do {
    wait (empty);
    wait (mutex);
    // obtain empty buffer and add next item, creating a full buffer
    signal (mutex);
} while (TRUE);
```

**Consumer process**

```plaintext
do {
    wait (mutex);
    // remove item from full buffer, create an empty buffer
    signal (mutex);
    ...... 
} while (TRUE);
```
We also need to add signal(EMPTY), so that when the consumer is done reading, more empty buffers are produced.
Unfortunately, this solution does not prevent a consumer from reading even when there are no buffers to read.
For example, if the first consumer reads before the first producer executes, then this solution will not work.

Producer process

```
do {
    wait (empty);
    wait (mutex);
    // obtain empty buffer and add next item, creating a full buffer
    signal (mutex);
} while (TRUE);
```

Consumer process

```
do {
    wait (mutex);
    // remove item from full buffer, create an empty buffer
    signal (mutex);
    signal (empty);
} while (TRUE);
```
Producer/Consumer Problem

So add a second counting semaphore full, initially set to 0
Verify for yourself that this solution won’t deadlock, synchronizes properly with mutual exclusion, prevents a producer from writing to a full buffer, and prevents a consumer from reading from an empty buffer.

Produce process

do {

    wait (empty);

    wait (mutex);

    // obtain empty buffer and add
    // next item, creating a full buffer

    signal (mutex);

    signal (empty);
}

Consumer process

do {

    wait (full);

    wait (mutex);

    // remove item from full buffer,
    // create an empty buffer

    signal (mutex);

    signal (empty);
}

} while (TRUE);
The Dining Philosophers Problem

- N philosophers seated around a circular table
- There is one chopstick between each philosopher
- A philosopher must pick up its two nearest chopsticks in order to eat
- A philosopher must pick up first one chopstick, then the second one, not both at once
- Chopsticks are like resources.
- While a philosopher is holding a chopstick, another one cannot have it.

Devise an algorithm for allocating these limited resources (chopsticks) among several processes (philosophers) in a manner that is deadlock-free, and starvation-free?
The Dining Philosophers Problem

• Is not a real problem

• But lots of real resource allocation problems look like this. If we can solve this problem effectively and efficiently, we can also solve the real problems.

• From a satisfactory solution:
  
  – We want to have **concurrency**: two philosophers that are not sitting next to each other on the table should be able to eat **concurrently**.

  – We don’t want **deadlock**: waiting for each other indefinitely.

  – We don’t want **starvation**: no philosopher waits forever.
The Dining Philosophers Problem

Semaphore chopstick [5] initialized to 1

do {
    wait ( chopstick[i] );
    wait ( chopStick[ (i + 1) % 5 ] );

    // eat
    signal ( chopstick[i] );
    signal (chopstick[ (i + 1) % 5 ] );

    // think
} while (TRUE);

This solution provides concurrency but may result in deadlock.
Each philosopher is a process

One semaphore per chopstick:
  - chopstick: array[0..4] of semaphores
  - Initialization: chopstick[i].count:=1 for i:=0..4

Process Pi:
repeat
  think;
  //obtain the two chopsticks to my immediate right and left
  wait(chopstick[i]);
  wait(chopstick[i+1 mod 5]);
  eat;
  /// release both chopsticks
  signal(chopstick[i+1 mod 5]);
  signal(chopstick[i]);
forever

Guarantees that no two neighbors eat simultaneously, i.e. a chopstick can only be used by one of its two neighboring philosophers.

Deadlock if each philosopher start by picking his left chopstick!
The Dining Philosophers Problem: First Attempt

- Unfortunately, the previous “solution” can result in **deadlock**

- each philosopher grabs its right chopstick first
- causes each semaphore’s value to decrement to 0

- each philosopher then tries to grab its left chopstick
- each semaphore’s value is already 0, so each process will block on the left chopstick’s semaphore

- These processes will never be able to resume by themselves

- we have deadlock!
Some deadlock-free solutions:

- allow at most 4 philosophers at the same table when there are 5 resources [solution in the next slide]

- allow a philosopher to pick up chopsticks only if both are free. This requires protection of critical sections to test if both chopsticks are free before grabbing them.
- we’ll see this solution next using monitors
The Dining Philosophers Problem: Correct Solution

- A solution: admit only 4 philosophers at a time that tries to eat
- Then 1 philosopher can always eat when the other 3 are holding 1 fork
- Hence, we can use another semaphore T that would limit at 4 the numb. of philosophers “sitting at the table”
- Initialize: T.count:=4

Process Pi:
repeat
    think;
    wait(T);
    wait(fork[i]);
    wait(fork[i+1 mod 5]);
    eat;
    signal(fork[i+1 mod 5]);
    signal(fork[i]);
    signal(T);
forever
Kernel Implementing wait and signal

- Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time. (note: they can call but they can not execute)

- The wait and signal operations must be atomic. The integer value is updated. No two process should update at the same time. How can the kernel ensure that? It can NOT use semaphores to implement semaphores.

- Implementation of these operations in kernel becomes the critical section problem where the wait and signal code are placed in the critical section. How can ensure two processes will not execute at the same time in wait or signal?
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
  - Note that applications may spend lots of time in critical sections and therefore busy waiting is not a good solution for applications. But, for short kernel critical sections, it may be acceptable in multi-CPU systems.
Implementation of Semaphores

- Can be implemented in hardware or software
- Software schemes such as Peterson’s algorithms can be used
- Use one of the hardware-supported schemes for mutual exclusion
Implementation of Semaphores

```c
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow inter-
        rupts*/;
    }
    else
    allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count<= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

(a) Compare and Swap Instruction

```c
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
    /* do nothing */;
    s.count--;  
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set 
        s.flag to 0) */;
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
    /* do nothing */;
    s.count++;  
    if (s.count<= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

(b) Interrupts
A Semaphore Is Like a Bouncer

- Imagine a set of waiting threads, lined up in a queue

- A semaphore is like a bouncer at the front of the lineup. He only allows threads to proceed when *instructed* to do so.

Of course, the bouncer needs to keep track of this number, which is why all semaphores maintain an integer counter.


![Diagram](image-url)
A Semaphore Is Like a Bouncer

Now, what happens if some thread calls signal before there are any threads waiting in line?

No problem: As soon as the next thread arrives in the lineup, the bouncer will let it pass directly through. And if signal is called, say, 3 times on an empty lineup, the bouncer will let the next 3 threads to arrive pass directly through.

Stolen from: http://preshing.com/20150316/semaphores-are-surprisingly-versatile/ :-}
A Semaphore Is Like a Bouncer

The beauty of this strategy is that if \texttt{wait} is called some number of times, and \texttt{signal} is called some number of times, the outcome is always the same.
Problem with Semaphores

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes.
- But `Wait(S)` and `Signal(S)` are scattered among several processes. Hence, difficult to understand their effects.
- Semaphores can result in deadlock due to programming errors.
  - Forgot to add a `signal()` or `wait()`, or misordered them, or duplicated them.
- To reduce these errors, introduce high-level synchronization, e.g., monitors with condition variables, that essentially automates insertion of `signal` and `wait` for you.
Monitors
Monitors

★ Declare a monitor as follows (looks somewhat like a C++ class):
★ A monitor ensures that only 1 **process/thread** at a time **can be active** within a monitor
★ simplifies programming, no need to explicitly synchronize.

```plaintext
monitor monitor-name
{
    // shared local variable
    function P1 (...)
    {
        ....
    }

    function Pn (...)
    {
        ....
    }

    Initialization code (...)
}
```

**Implicitly**, the monitor defines a mutex-lock semaphore `mutex=1`;
Each function’s critical code is surrounded as follows:
```plaintext
function P1(...) {
    wait(mutex)
    //critical code
    signal(mutex)
}
```
A monitor is a software module containing:

- one or more procedures
- an initialization code
- local (or private) shared data variables
Schematic view of a Monitor
Example

```c
monitor sharedcounter
{
  int counter;
  function add() { counter++;}
  function sub() { counter--;}
  init() { counter=0; }
}
```

- If two processes want to access this shared counter monitor, then access is mutually exclusive and only one process at a time can modify the value of counter.

- If a write process calls sharedcounter.add(), then it has exclusive access to modifying counter until it leaves add(). No other process, e.g. a read process, can come in and call sharedcounter.sub() to decrement counter while the write process is still in the monitor.
Monitors

- In the previous shared counter example, a writer \textbf{process may be interacting} with a reader process via a \textbf{bounded buffer}

- like the solution to the producer/consumer problem, the writer should \textbf{signal} blocked reader processes when there are \textbf{no longer elements in the buffer}

- monitors alone don’t provide this signaling synchronization capability

- In general, there may be times \textbf{when one process wishes to signal another process} based on a condition, much like semaphores.

- Thus, monitors alone are insufficient.

- \textbf{Augment monitors with condition variables}
A condition variable x in a monitor allows two main operations on itself:

**x.wait():**
- blocks execution of the calling process on condition (variable) until another process calls x.signal()
  - the process can resume execution only if another process executes csignal(a)

**x.signal():**
- resume execution of exactly 1 suspended process on condition (variable)
  - If several process exists: choose first one
  - If no such process exists: do nothing
Subtle difference between Condition Variables and Semaphores

- Condition variables are not semaphores. They are different even though they look similar.

- Semaphores have memory, signal(): will increment the semaphore counter, even if no one has called wait().

- Condition variables do not have memory: if no one is waiting for a signal(), this signal() is not saved

  - A condition variable does not count: have no associated integer.

  - A signal on a condition variable x is lost (not saved for future use) if there is no process waiting (blocked) on the condition variable x.

  - The wait() operation on a condition variable x will always cause the caller of wait to block.

  - The signal() operation on a condition variable will wake up a sleeping process on the condition variable, if any. It has no effect if there is nobody sleeping.
Condition Variables: Example

- Assume we have 5 instances of the resource.
- 5 processes can use the resource simultaneously.
- We want to implement a monitor that will implement two functions:
  - acquire()
  - release()
- that can be called by a process before and after using a resource.
A process will be coded like the following:

Allocate MA; // resource allocation monitor

MA.acquire();

//use the resource …

MA.release();

....
monitor Allocate
{
    int count = 5;  // we initialize count to 5.
    condition c;

    void acquire ()
    {
        if (count == 0)
            c.wait();
        count--;
    }

    void release () {
        count++;
        c.signal();
    }
}
Monitor for Producer/Consumer Problem

- Two types of processes:
  - producers
  - consumers

- If these procedures are correct, synchronization will be correct for all participating processes

A process will be coded like the following:

```
Producer i:
  repeat
    produce v;
    append(v);
  forever

Consumer j:
  repeat
    take(v);
    consume v;
  forever
```
Monitor for the bounded P/C problem

- **Local data**: monitor needs to hold the buffer:
  - buffer: array[0..n-1] of items;

- **Procedures**: append(.) and take(.) are procedures within the monitor: are the only means by which P/C can access the buffer

- **Conditions**: Needs two condition variables:
  - C1: csignal(C1) indicates that the buffer is not full
  - C2: csignal(C2) indicates that the buffer is not empty
    - The book uses more descriptive names notfull and notempty respectively. But students often think that you are checking whether a queue is full or empty. They are just NAMES of variables. To avoid this, I am just using C1 and C2.

- **Needs buffer pointers and counts**:
  - nextin: points to next item to be appended
  - nextout: points to next item to be taken
  - count: holds the number of items in buffer
Monitor boundedbuffer
{
  buffer: array[0..n-1] of items;
  nextin:=0, nextout:=0, count:=0: integer;
  C1, C2: condition;

append(v):
  if (count == n) cwait(C1);
  buffer[nextin]:= v;
  nextin:= nextin+1 mod n;
  count++;
  csignal(C2); //indicates that the buffer is not empty

take(v):
  if (count=0) cwait(C2);
  v:= buffer[nextout];
  nextout:= nextout+1 mod n;
  count--; 
  csignal(C1); //indicates that the buffer in not full
}
Monitor for the bounded P/C problem

monitor ProducerConsumer {
    int nfull = 0;
    cond has_empty, has_full;

    producer() {
        if (nfull == N)
            wait (has_empty);
        ... // fill a slot
        ++ nfull;
        signal (has_full);
    }

    consumer() {
        if (nfull == 0)
            wait (has_full);
        ... // empty a slot
        -- nfull;
        signal (has_empty);
    }
};

- **Two condition variables**
  - has_empty: buffer has at least one empty slot
  - has_full: buffer has at least one full slot

- **nfull**: number of filled slots
  - Need to do our own counting for condition variables
Monitor in Java: A Synchronized Counter

To make a method synchronized, simply add the `synchronized` keyword to its declaration:

```java
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```
Summary

**Messages**
- Useful for the enforcement of mutual exclusion discipline

**Operating system themes are:**
- Multiprogramming, multiprocessing, distributed processing
- Fundamental to these themes is concurrency
- Issues of conflict resolution and cooperation arise

**Mutual Exclusion**
- Condition in which there is a set of concurrent processes, only one of which is able to access a given resource or perform a given function at any time
- One approach involves the use of special purpose machine instructions

**Semaphores**
- Used for signaling among processes and can be readily used to enforce a mutual exclusion discipline
Structure of a Monitor

- Awaiting processes are either in the entrance queue or in a condition queue.
- A process puts itself into condition queue $cn$ by issuing `cwait(cn)`.
- `csignal(cn)` brings into the monitor 1 process in condition $cn$ queue.
- `csignal(cn)` blocks the calling process and puts it in the urgent queue (unless `csignal` is the last operation of the monitor procedure).
Monitors

- Monitor is a software module
- Provides equivalent functionality to that of semaphore, but easier to control
- As high-level synchronization constructs, monitors are found in high-level programming languages like Java and C#
- The OS may implement monitors using semaphores and mutexlocks
Monitors

- **Characteristics:**
  - local variables accessible only by monitor's procedures
  - a process enters the monitor by invoking one of its procedures
  - only ONE process can be **active** in the monitor at any one time [By enforcing the discipline of one process at a time, the monitor is able to provide a mutual exclusion facility.]

  - A person using the monitors is guaranteed of this.

First two are similar to object oriented programming
Monitor Characteristics

- By enforcing the discipline of one process at a time, the monitor is able to provide a mutual exclusion facility.

Local data variables are accessible only by the monitor’s procedures and not by any external procedure.

Process enters monitor by invoking one of its procedures.

Only one process may be executing in the monitor at a time.