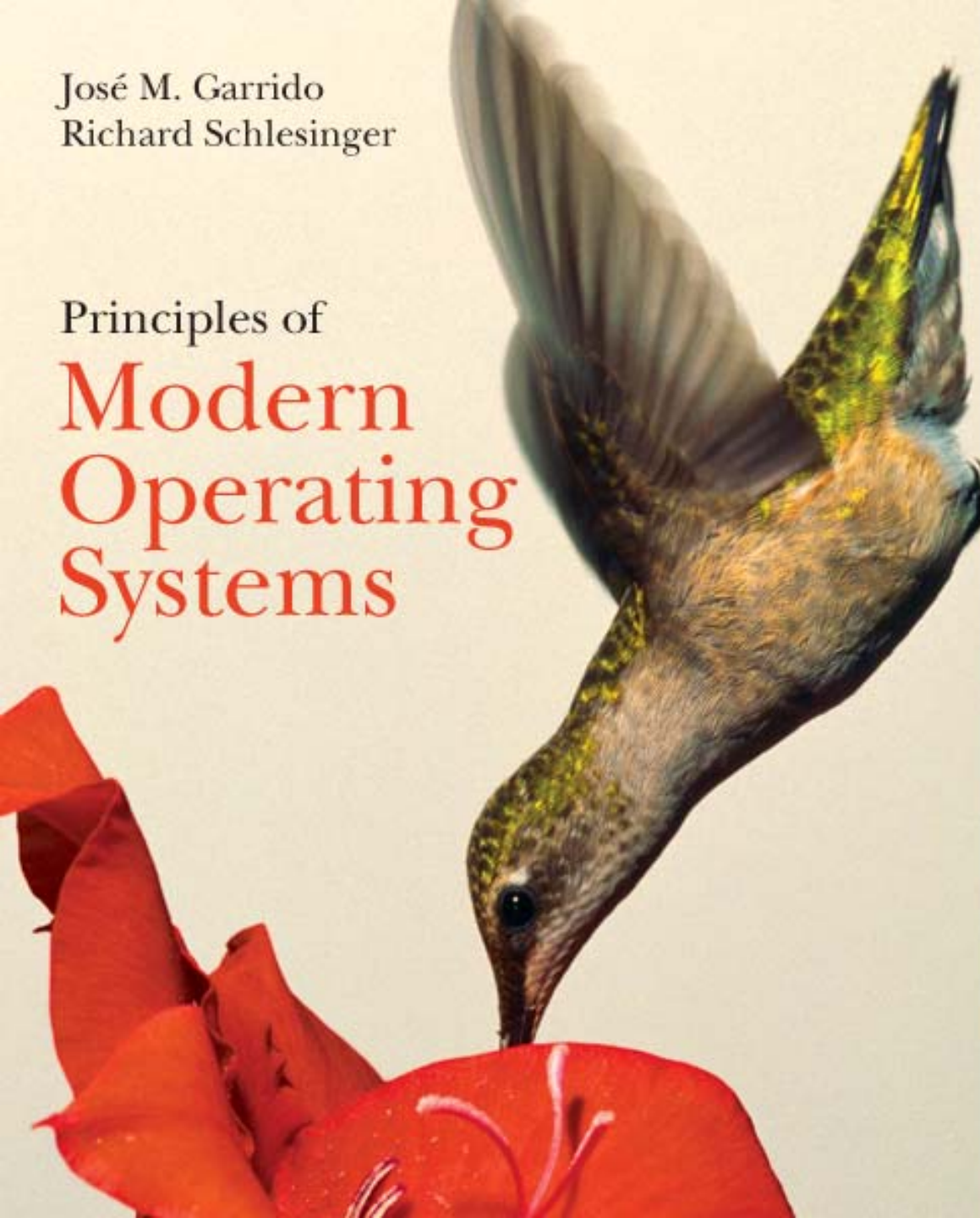


José M. Garrido  
Richard Schlesinger

Principles of  
**Modern  
Operating  
Systems**



# Chapter 6

## Synchronization Principles

# Synchronization as a Concept

---

- The coordination of the activities of the processes
  - Processes compete for resources
  - Processes interfere with each other
  - Processes cooperate

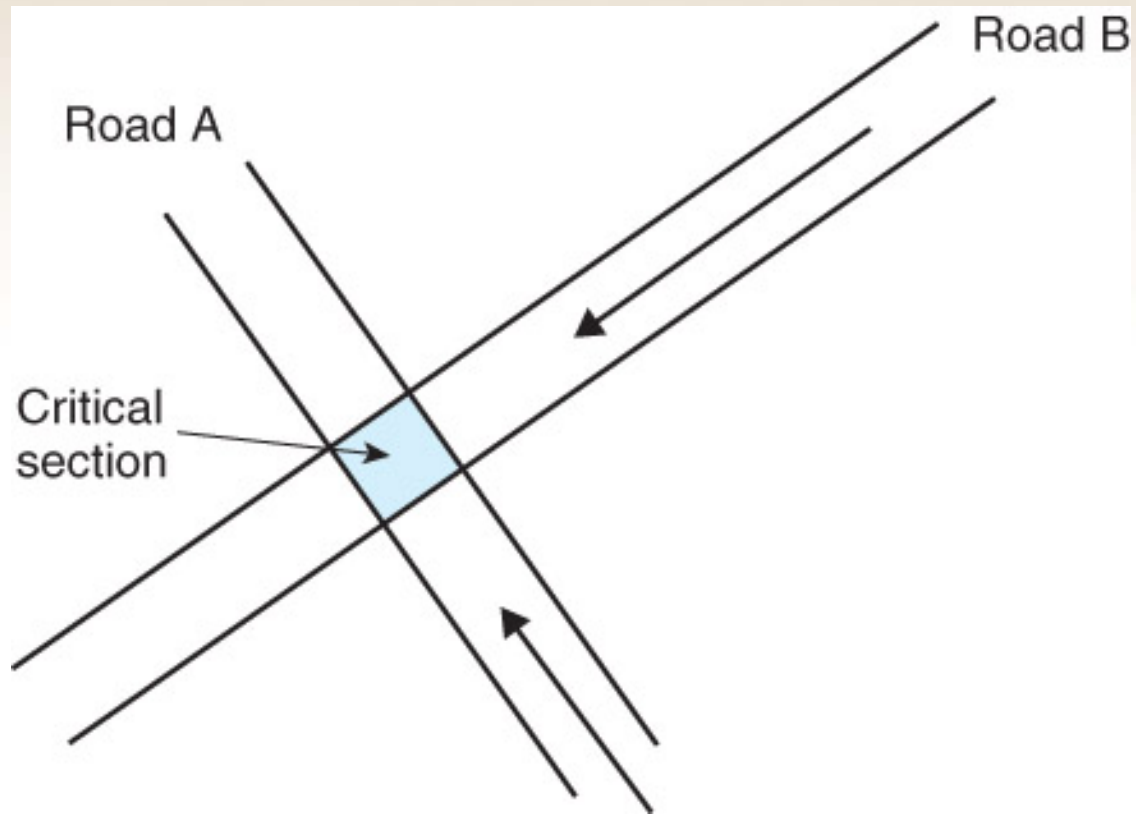
# Race Condition

---

- No synchronization
- Two or more processes access and manipulate the same data item together
- The outcome of the execution depends on the “speed” of the processes and the particular order in which each process accesses the shared data item
- No data integrity, results are generally incorrect

# Example of a Critical Section

---



# Road Intersection

---

- Two vehicles, one moving on Road A and the other moving on Road B are approaching the intersection
- If the two vehicles reach the intersection at the same time, there will be a collision, which is an undesirable event.
- The road intersection is a critical section for both roads because it is part of Road A and also part of Road B, but only one vehicle should reach the intersection at any given time.
- Therefore, mutual exclusion should be applied on the road intersection, the critical section.

# Mutual Exclusion

---

- A synchronization principle needed when a group of processes share a resource
- Each process should access the resource in a mutual exclusive manner
  - This means only one process at a time can access a shared resource
  - This is the most basic synchronization principle

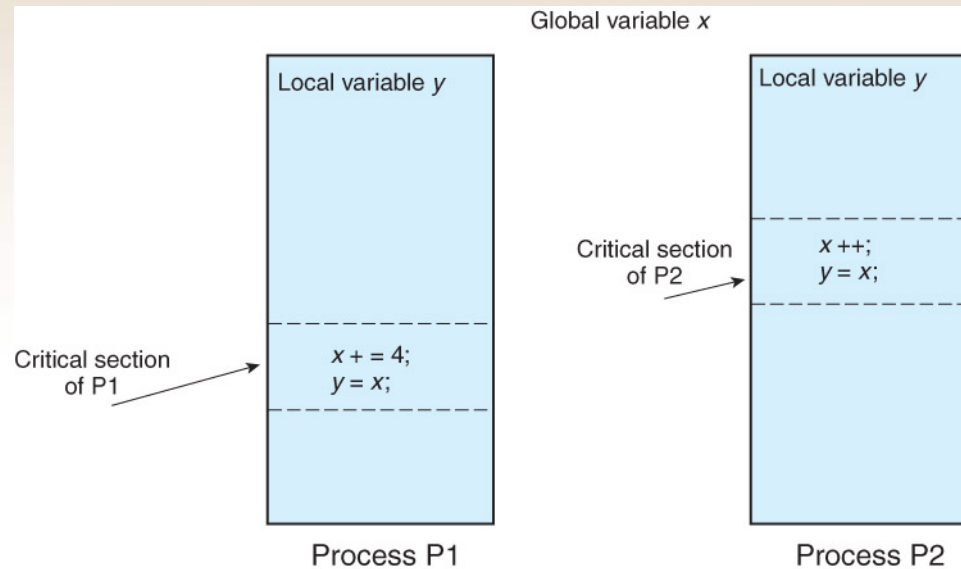
# Critical Section

---

A critical section is a portion of code in a process, in which the process accesses a shared resource

# Critical Sections in Two Processes

---



# Mutual Exclusion and Critical Sections

---

- Coordinate the group of processes that access shared resources such that only one process can execute its critical section at any time
- During this time interval, the other processes are excluded from executing their critical sections

# The Critical Section Problem

---

- The critical section protocol must be followed by all processes that attempt to access a shared resource
- The critical section protocol must satisfy the following requirements:
  - Mutual exclusion
  - Progress
  - Bounded waiting

# Example of Critical Sections

---

- Shared resource: Printer buffer
  - Two process:
    - Producer:
      1. produces a character,
      2. places the character in buffer.
    - Consumer:
      1. removes a character from the buffer,
      2. consumes the character.
- CRITICAL SECTION
-

# Definition of a Critical Section Protocol

---

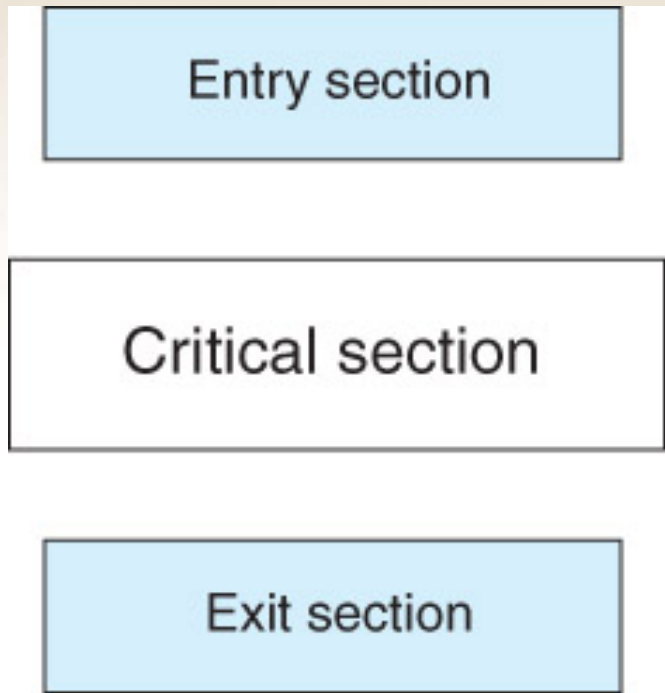
## 1. Entry section

check if any other process is executing its C.S., if so then the current process must wait otherwise set flags and proceed

## 2. Critical Section

## 3. Exit section

clear flags to allow any waiting process to enter its critical section.



Entry section

Critical section

Exit section

# Synchronization Solution

---

- Hardware
- Software

# Types of Synchronization

---

- Mutual exclusion
- Execution ordering
- Direct process cooperation

# Semaphores

---

- A semaphore is similar to a traffic light
- A software synchronization tool that can be used in the critical section problem solution
- It is an abstract data type implemented as a class provided by the operating system

# Semaphores Objects

---

Are objects that, must be initialized and can be manipulated only with two atomic operations: *wait* and *signal*.

# Semaphore Class

---

```
class Semaphore {
    private int sem;
    private Pqueue sem_q; // semaphore
    queue
    public Semaphore ( int initval);
    public wait (); // P ()
    public signal (); // V ()
} // end of class Semaphore
```

# Types of Semaphores

---

- Binary semaphore: the value of the integer attribute, *sem*, is either 0 or 1
- Counting semaphore: the value of the attribute can take any integer value  $> 0$

# Using Semaphores

---

- Include synchronization mechanisms that regulate access to a shared resource, used in:
  - the entry section
  - exit section
- Are used to allow a process to execute its critical section when no other process is already in its critical section, thus providing exclusive access to a shared resource

# Creating a Semaphore Object

---

```
// declaration of object reference
variables
Semaphore mutex;
// create object and initialize its
integer
// attribute
mutex = new Semaphore (1);
```

# Critical Section Protocol

---

- A semaphore object referenced by *mutex*, is used here with the two operations: *wait* and *signal*.

```
mutex.wait();
```

```
Critical section
```

```
mutex.signal();
```

# Implementation Of Semaphores

---

- Busy – Waiting (not practical or useful)
- Sleep and Wakeup (or Suspend and Reactivate) system functions

# Processes Synchronized by Semaphores

---

- A waiting process is blocked and waken up later.
- Processes waiting on a semaphore are kept in the semaphore queue.
- The *wait* and *signal* operations are implemented using the system calls *sleep()* and *wakeup()*.

# Semaphore Methods Implementation

---

```
wait() { disable the interrupt system
        if sem > 0 then
            sem := sem-1;
        else {add process into waiting queue;
              sleep(); // suspend process }
        enable the interrupt system
    }
signal() { disable interrupt system
          sem:= sem + 1;
          if processes in the queue then {
              remove process p from waiting queue;
              wakeup(p); // reactivate process }
          enable the interrupt system
    }
```

# Execution Ordering

---

- In addition to mutual exclusion, semaphores can also be used for execution ordering
- In this type of synchronization the processes exchange synchronization signals in order to coordinate the order of executions

# Example in Execution Ordering

---

- Assume two processes  $P1$  and  $P2$  need to synchronize their executions in the following order: in  $P1$ , `write(x)` must be executed before  $P2$  executes `read(x)`.
- This synchronization problem can be solved with semaphores
- The following is a solution that uses Semaphore `exord`, initialized to zero:

# Code of Processes P1 and P2

---

```
// process P1
```

```
...
```

```
write(x);
```

```
exord.signal();
```

```
...
```

```
// process P2
```

```
...
```

```
exord.wait();
```

```
read(x);
```

```
...
```

# Classical Synchronization Problems

---

1. Bounded-buffer problem – also called producer/consumer problem
2. Readers-writers problem
3. Dining philosophers

# Classical Synchronization Problems

---

- See simulations on the CD

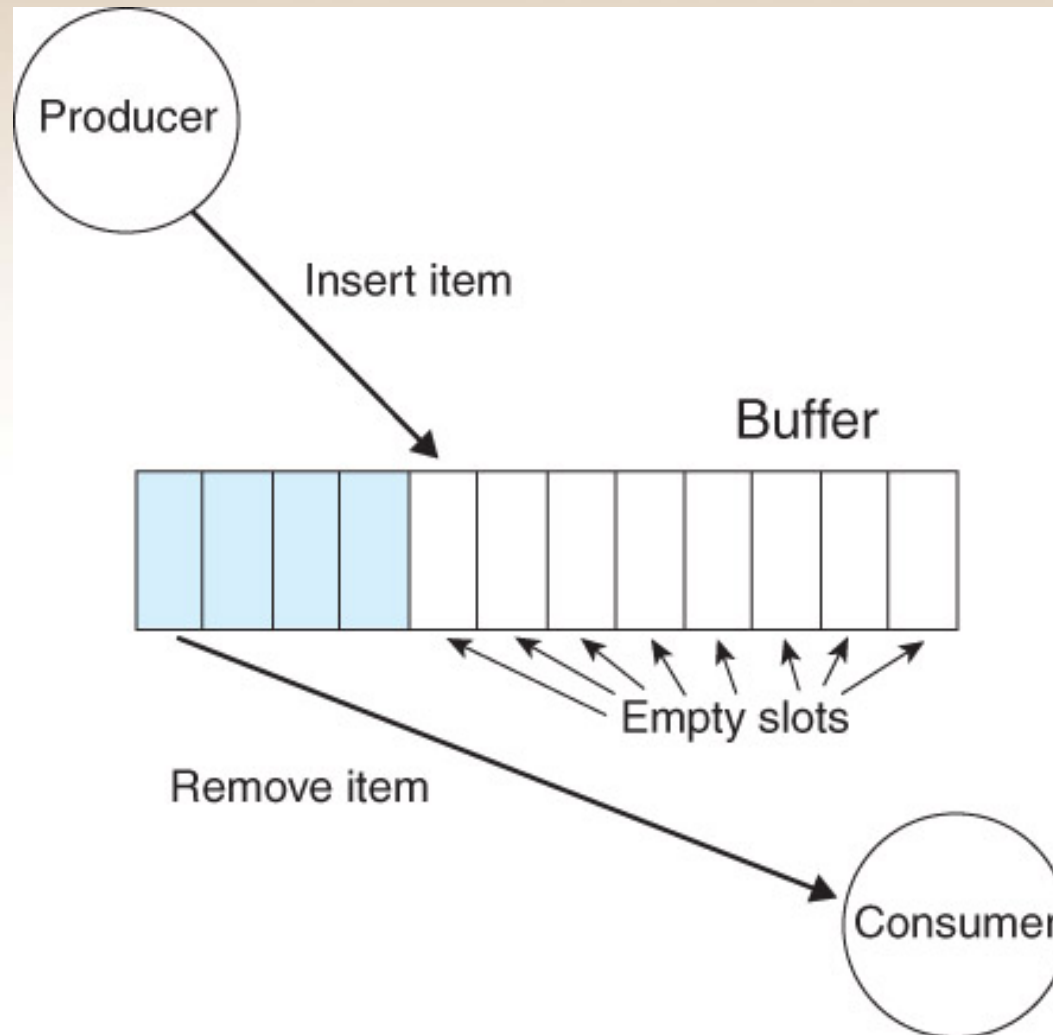
# Producer Consumer Problem

---

- There are two processes that execute continuously and a shared buffer
- The Producer process generates data items
- The Consumer process takes these data items and consumes them
- Buffer - a container of  $N$  slots, filled by the producer process and emptied by the consumer process

# Producer-Consumer Problem

---



# Producer-Consumer Problem(2)

---

- Competition between the two processes to access the shared buffer
- Cooperation of the two processes in order to exchange data through the buffer

# Synchronization

---

The producer and consumer processes must be synchronized:

- Both processes attempt mutual exclusive access to the data buffer
- The producer must wait to insert a new data item if buffer is full
- The consumer process must wait to remove a data item if buffer is empty

# Semaphores Used

---

- A binary semaphore for the mutual exclusive access to the data buffer
- A counting semaphore to count the number of full slots in the buffer
- A counting semaphore to count the number of empty slots in the buffer

# Data Declarations for Solution

---

// Shared data

int N = 100; // size of buffer

char *buffer* [N]; // *buffer implementation*

char *nextp*, *nextc*;

Semaphorec *full*, *empty*; // *counting*

*semaphores*

Semaphoreb *mutex*; // *binary semaphore*

# Initializing Semaphore Objects

---

```
full = new Semaphorec(0);    // counting semaphore obj  
empty = new Semaphorec( N); // counting sem obj  
mutex = new Semaphoreb(1); // binary semaphore obj
```

# Body of Producer

---

Producer process

```
while(true) {
```

```
    . . .
```

```
    produce a data item
```

```
    . . .
```

```
    empty.wait(); // any empty slots? Decrease empty slots
```

```
    mutex.wait(); // attempt exclusive access to buffer
```

```
    . . .
```

```
    // instructions to insert data item into buffer
```

```
    . . .
```

```
    mutex.signal(); // release exclusive access to buffer
```

```
    full.signal(); // increment full slots
```

```
    . . .
```

```
}
```

# Body of Consumer

---

Consumer process

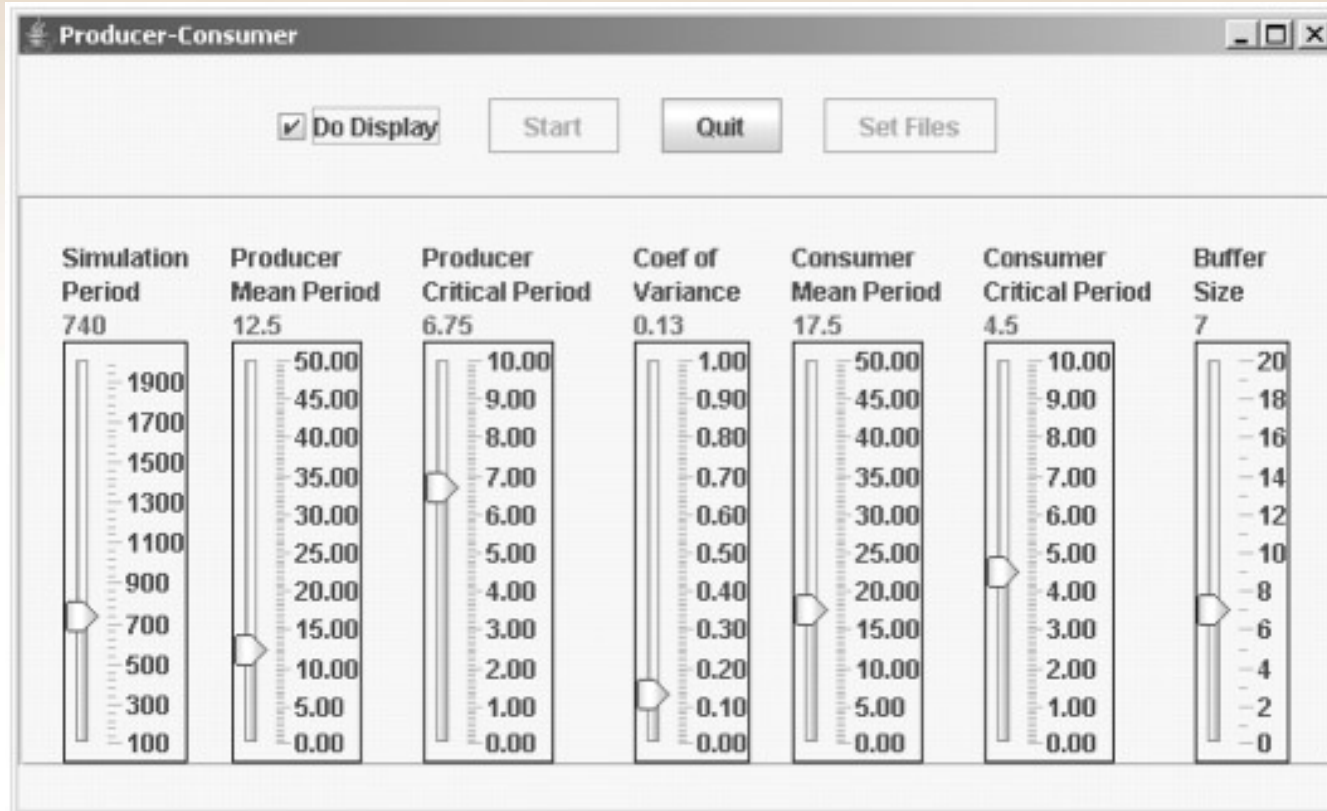
```
while(true) {  
    ...  
    full.wait();    // any full slots? Decrease full slots  
    mutex.wait(); // attempt exclusive access to buffer  
    ...  
    // remove a data item from buffer and put in nextc  
    ...  
    mutex.signal(); // release exclusive access to buffer  
    empty.signal(); // increment empty slots  
    // consume the data item in nextc  
    ... }  
}
```

# Simulation Models for the Bounded Buffer Problem

---

- The basic simulation model of the bounded-buffer problem (producer-consumer problem) includes five classes: *Semaphore*, *Buffer*, *Producer*, *Consumer*, and *Consprod*.
- The model with graphics and animation includes additional classes for displaying the GUI and the animation.
- The models implemented in Java with the PsimJ simulation package, are stored in the archive files: `consprod.jar` and `consprodanim.jar`.
- The C++ version is in file: `consprod.cpp`

# GUI for the Simulation Model



# Results of a Simulation Run

---

Project: Producer-Consumer Model

Run at: Thu Sep 15 00:00:11 EDT 2006 by jgarrido on Windows XP, localhost

Input Parameters

Simulation Period: 740

Producer Mean Period: 12.5

Prod Mean Buffer Access Period: 6.75

Coef of Variance: 0.13

Consumer Mean Period: 17.5

Cons Mean Buffer Access Period: 4.5

Buffer Size: 7

-----  
Results of simulation: Producer-Consumer Model

Total Items Produced: 23

Mean Prod Buffer Access Time: 0006.735

Total Prod Buffer Access Time: 0154.916

Mean Producer Wait Time: 0000.760

Total Producer Wait Time: 0017.480

Total Items Consumed: 23

Mean Cons Buffer Access Time: 0004.575

Total Cons Buffer Access Time: 0105.218

Mean Consumer Wait Time: 0007.896

Total Consumer Wait Time: 0181.597

# Readers and Writers Problem

---

- Processes attempt to access a shared data object then terminate
- There are two types of processes:
  - Readers
  - Writers

# Readers and Writers Problem

---

- One or more readers can access the shared data object at the same time
- Only one writer can access the shared data object at a time

# Synchronization

---

Two levels of mutual exclusion:

- Individual mutual exclusive access to a shared resource for writers.
- Group exclusive access to a shared resource for readers.

# Access to Shared Data

---

- Writers need “individual” exclusive access to the shared data object
- Readers need “group” exclusive access to the shared data object
  - Once the first reader has gained exclusive access to the shared data object, all subsequent readers in a group are able to immediately access the shared data object

# First and Last Readers

---

- The first reader competes with writers to gain group exclusive access to the shared data object
- The last reader releases group exclusive access to the shared data object
  - Therefore, the last reader gives a chance to waiting writers

# Identifying First and Last Readers

---

- A counter variable, readcount, is needed to keep a count of the number of reader processes that are accessing the shared data
- When a reader requests access to the shared data, the counter is incremented
- When a reader completes access to the shared data, the counter is decremented

# The Counter Variable

---

- The counter variable, readcount, is used by readers only
- Every reader process has to increment this counter variable on requesting access to the shared data
- Every reader has to decrement this counter variable when access to the shared data is completed
- Therefore, access to this counter variable has to be done in a mutual exclusive manner by every reader

# Identifying First and Last Readers

---

- For the first reader, *readcount* is 1
- For the last reader, *readcount* is 0

# Readers-Writers Solution

---

Two binary semaphores are used

- Semaphore *mutex* is used by readers to ensure mutual exclusive access to the variable *readcount* for updating it.
- Semaphore *wrt* controls mutual exclusive access to the shared data object
- Semaphore *wrt* is used by writers; is also used by the first and last readers.

# Declaring Data and Objects

---

```
integer readcount = 0;           // used by readers only
```

```
Semaphoreb mutex, wrt;
```

```
mutex = new Semaphoreb (1);  
wrt = new Semaphore (1);
```

# Writer Process

---

```
Writer() {  
    . . .  
    wrt.wait (); // get access to the shared object  
  
    // write to shared data object in a mutual  
    // exclusive manner  
  
    wrt.signal ();  
    . . .  
} // end of writer
```

# Reader Process

---

```
Reader (){
  . . .
  mutex.wait ();
  increment readcount;
  if readcount equals 1 then // first reader?
    wrt.wait ();           // gain group access to shared data
  mutex.signal ();
  // Critical Section read shared data object
  mutex.wait();
  decrement readcount;
  if readcount equals zero then // last reader?
    wrt.signal (); // release group access to shared data
  mutex.signal ();
  . . .
} // end Reader()
```

# Observations on R-W Problem

---

- If a writer is in the critical section and  $n$  readers are waiting, then one reader is queued on *wrt* and  $n-1$  readers are queued on *mutex*.
- When a writer executes *wrt.signal()*, the OS resumes the execution of either:
  - waiting readers, or
  - a single waiting writer.

# Readers & Writers Problem

---

- Prob. 1 – Readers have implicit priority -- No reader will be kept waiting unless a writer has already obtained permission to access the shared data
- Prob. 2 - Once a writer is ready to access the shared data, the writer performs its operation as soon as possible (i.e., no new readers may start reading).
- This second strategy gives priority to the writer processes

# Simulation Models of the Readers Writers Problem

---

- The simulation models of the readers-writers problem also follow very closely the theoretical discussion. The Java implementation of the model includes eight classes: *Buffer*, *Condition*, *Reader*, *ReaderArrivals*, *ReaderWriter*, *SimDisplay*, *SimInputs*, *Writer*, and *WriterArrivals*.
- These Java files are stored in the archive file `readwrite.jar` and the model with the GUI is stored in the archive file `rwrite2.jar`.
- The C++ version of this model is stored in file `reawriter.cpp`. The model implements the first strategy for solving the readers-writers problem.

# Partial Results of a Simulation Run

---

*Project: Concurrent Readers/Writers Problem*

*Run at: Thu Mar 02 15:20:37 EST 2006*

-----  
*Input Parameters*

*Simulation Period: 740*

*Arrivals Stop: 400*

*Reader Inter Arrival Time: 5.5*

*Writer Inter Arrival Time: 7.5*

*Mean Write Time: 17.5*

*Mean Read Time: 12.75*

*Writer Priority: 10*

*Reader Priority: 10*

*Results of simulation: Concurrent Readers/Writers Problem*

-----  
*Results of simulation:*

*Number of Readers that Arrived: 87*

*Number of Readers that completed: 87*

*Average reader wait period: 0164.1250*

*Number of Writers that Arrived: 45*

*Number of Writers that completed: 28*

*Average writer wait period: 0234.5263*

# Synchronization Using Monitors

---

- Monitors are higher-level mechanisms for the synchronization of interacting processes, compared with semaphores.
- Monitors are abstract data types. They take full advantage of the encapsulation principle provided by object-oriented programming languages.
- Only a single process can be executing an operation of the monitor at any given time.

# Monitors

---

- A monitor implements a mechanism that facilitates the use of mutual exclusion
- In addition to the entry queue for waiting processes that need to enter the monitor, there are one or more condition queues
- Each condition queue corresponds to a condition variable.

# Synchronization with Monitors

---

- In a similar manner to the use of semaphores, monitors are used to solve various synchronization problems.
- The main advantage of using monitors in solving synchronization problems is the higher-level construction.
- Brinch Hansen's approach for the semantics of monitor requires that a process that executes a *signal* operation must exit the monitor immediately. The *signal* statement must be the last one defined in the monitor function. Anthony Hoare proposed a slightly different semantics for monitors.

# Simulation Model of the Producer-Consumer with Monitors

---

- The simulation models of the producer-consumer problem with monitors also follow very closely the theoretical discussion. The Java implementation of the model includes the classes: *Buffer*, *Condition*, *Consprod*, *Consumer*, *PCmonitor*, *Producer*, and *Semaphore*.
- These Java files are stored in the archive file `consprodm.jar`.
- The C++ version of this model is stored in file `consprodm.cpp`. The model implements the first strategy for solving the readers-writers problem.

# Dining Philosophers

---

- Illustrates processes that are competing for exclusive access to more than one resources, such as tape drives or I/O devices.
- A monastery's dining table
  - Circular table
  - Five plates
  - Five forks (critical)
  - Plate of noodles at center of table (endless supply)
  - Philosophers can only reach forks adjacent to their plate (left and right forks)

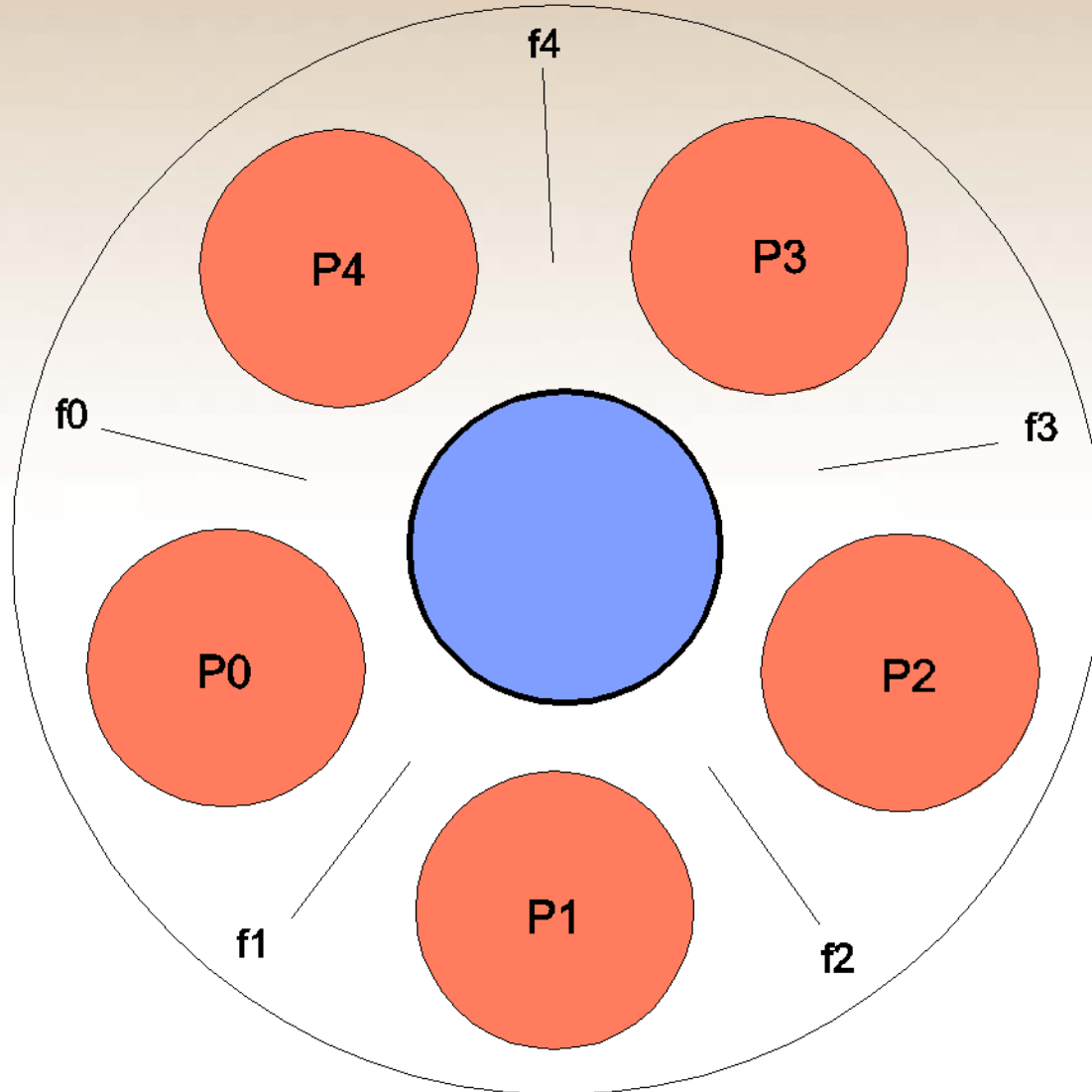
# Life of a Philosopher

---

1. Spends some time thinking
2. Gets hungry
3. Sits at the table,
4. Attempts to get the left fork
5. Attempts to get the right fork
6. Starts eating, this activity takes a finite time
7. Releases the left and right forks
8. Returns to the initial state (thinking)

# Dining Table

---



# Dining-Philosophers Problem

---

Every philosopher needs two forks and these must be accessed in a mutual exclusive manner

```
// Shared data
```

```
Semaphore fork[] = new Semaphore[5];
```

# Philosophers Code

---

```
Philosopher run() {  
    while (true) {  
        // get left fork  
        fork[i].wait();  
        // get right fork  
        fork[(i + 1) % 5].wait();  
        // eat for a finite time interval  
        // return left fork  
        fork[i].signal();  
        // return right fork  
        fork[(i + 1) % 5].signal();  
        // think for finite time interval  
    }  
} // end run()
```

# Why Synchronize?

---

- When a philosopher is hungry
  - obtains 2 forks (1 at a time)
  - Proceeds to eat
- When a philosopher has satisfied hunger returns both forks and goes back to think
- Problem: There are 5 competing philosopher processes
- Using semaphores as before, is not sufficient for solving the problem

# Dining Philosophers 1st Attempt

---

- Suppose all philosophers want to eat at the same time
- Each one will pick up the left fork first then block trying to pickup the right fork
- All processes will now block indefinitely - deadlock

# Dining Philosophers 2nd Attempt

---

- After taking the left fork, if the right fork is not available, philosopher puts back the left fork and waits for a while.
- Problem: Suppose all philosophers want to eat at the same time:
  - Each will pick up the left fork first and then try to pick up the right fork.
  - The right fork is not available, so all philosophers put back left forks and wait.
  - After some time, philosophers pick up the left fork and try again - the cycle repeats.

# Dining Philosophers 3rd Attempt

---

- Use a *mutex* semaphore to make eating mutually exclusive.
- A philosopher is guaranteed to get both forks in this case.
- Problem: Only one philosopher can be eating at any given time.

# 4th Attempt to a Solution

---

- A philosopher's neighbors are defined by macros LEFT & RIGHT.
- A philosopher may move only into eating state if neither neighbor is eating.
- Use an array, *state*, to keep track of whether a philosopher is eating, thinking or hungry (trying to acquire forks).
- Use an array of semaphores, one per philosopher, so hungry philosophers can block if the needed forks are busy.

# Atomic Transactions

---

- What is an atomic transaction?
  - More than one data item must be changed to complete the transaction

# Multi-Part Transaction

---

Move money from account #1 to account #2

Lock accounts

Decrement Account #1 Balance

← System Failure

Increment Account #2 Balance

Unlock accounts

# What is a System Failure?

---

- Power failure or other problem that stops the system from doing its normal processing

# System Failure Recovery

---

- Remember that if any part of an Atomic Transaction is completed, then the entire operation must be completed
- Failure Recovery must either:
  - Run Atomic Transactions to completion after system restart
  - Undo the incomplete parts of transactions already executed

# Run Atomic Transaction to Completion

---

- On a general purpose system...
  - Impossible to do
    - Memory contents were lost
- Some embedded & realtime systems will do this
  - Requires specialized hardware support

# Undo Incomplete Transactions

---

- Checkpoint – Restart
- Transaction Logging

# Checkpoint - Restart

---

- Periodically, record the state of the system (with all completed transactions)
- At Failure Restart, restore the saved system status from the Checkpoint
- Loses all transactions since last Checkpoint

# System Failure Recovery

---

- Write-Ahead Logging
  - **Prior** to each step of a transaction, the details of that step are written to a separate Transaction Log file.
  - After system restart, the log is used to undo the steps of any incomplete transactions

# System Failure Recovery

---

- A good system will use both techniques
- Transaction Logging will be the primary recovery technique
- Checkpoint Restart will be used in case of catastrophic failure