Synchronization as a Concept

The \textit{coordination} of the activities of the processes

\begin{itemize}
  \item Processes compete for resources
  \item Processes interfere with each other
  \item Processes cooperate
\end{itemize}
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

Local variable $y$

$x += 4;\quad y = x;$

Critical section of P1

Process P1

Global variable $x$

$x ++;\quad y = x;$

Local variable $y$

Critical section of P2

Process P2
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 processes:
  - Producer:
    - produces a character,
    - places the character in buffer.
  - Consumer:
    - removes a character from the buffer,
    - consumes the character.
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.
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 ();
    public signal ();
} // end of class Semaphore
Creating a Semaphore Object

// declaration of object ref
Semaphore mutex;
// create object and initialize
mutex = new Semaphore (1);
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
Synchronization with 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
Critical Section Protocol

A semaphore object referenced by *mutex*, is used here with the two operations: *wait* and *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 woken 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
}
Event 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 $P1$, $\text{write}(x)$ must be executed before $P2$ executes $\text{read}(x)$.

• This synchronization problem can be solved with semaphores.

• The following is a solution that uses Semaphore $\text{exord}$, initialized to zero:
Code of Processes P1 and P2

// process P1
... write(x);
exord.signal();
...

// process P2
... exord.wait();
read(x);
...
Synchronization Case Studies

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

• See simulations on the CD
Bounded-Buffer 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

```java
full  = new Semaphorec(0);  // counting semaphore obj
empty = new Semaphorec(N);  // counting sem obj
mutex = new Semaphoreb(1);  // binary semaphore obj
```
Implementation 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
    ...
}
Implementation 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
...
}
Synchronization with Semaphores in Java

• The Java programming language has facilities for defining, creating, and manipulating Java threads.
• Java also provides a mechanism for basic exclusive access to shared resources.
• The wait/notify mechanism is used to synchronize Java threads.
• See Section 6.6.2.
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, \textit{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 \textit{incremented}
• When a reader completes access to the shared data, the counter is \textit{decremented}
The Counter Variable

• The counter variable, \textit{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 \textit{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;   // binary semaphores

mutex = new Semaphoreb (1);
wrt = new Semaphoreb (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 (){
    . . .
    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
POSIX Threads

• Developed for using with the C programming language
• The library includes functions for creating, synchronizing, terminating threads
• Synchronization is used with *mutex variables* and semaphore variables
• See Section 6.7
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 \textit{signal} operation must exit the monitor immediately. The \textit{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.
IPC Mechanism

• A set of mechanisms provided by the OS that enable processes to exchange data
• Data is formatted as messages
• Transfer of messages requires synchronization
  – Asynchronous communication (acomm)
  – Synchronous communication (scomm)
Asynchronous Communication

- Similar to consumer-producer problem
- Indirect communication between sender and receiver processes
- No direct interaction between processes
- A mailbox stores the message
- Sender does not need to wait (?)
- Receiver waits until message becomes available

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com
Asynchronous Communication

Sender process

Receiver process

Send message

Receive message

Operating System

Mailbox

Messages

© 2012 Jones & Bartlett Learning, LLC
www.jblearning.com
Synchronous Communication

• Direct interaction between processes
• Processes have to wait until the communication occurs
• The processes have to participate at the same time during the communication interval
• A channel is established between the processes
Synchronous Communication
Multi-Part Transaction

Move money from account #1 to account #2
Lock accounts
Decrement Account #1 Balance
Increment Account #2 Balance
Unlock accounts

System Failure
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