

## Cooperating Processes

- Introduction
- Shared-Memory
- Inter-Process Communication (IPC)
- Client-Server Paradigm
- Socket-based Communication

Based on original slides by  
Silberschatz, Galvin and Gagne

1

---

---

---

---

---

---

---

---

## Cooperating Processes (Threads)

- *Independent process*
  - cannot affect or be affected by the execution of another process.
- *Cooperating process*
  - can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

2

---

---

---

---

---

---

---

---

## Cooperation

- Cooperating processes need mechanisms for
  - Communication
  - Synchronization
- Communication between cooperating process can occur by means of
  - Shared memory
  - Message passing

3

---

---

---

---

---

---

---

---

## Producer-Consumer Problem

- Paradigm for cooperating processes
- *producer* process produces information ...
- that is consumed by a *consumer* process.

4

---

---

---

---

---

---

---

---

## Cooperating Processes

- Definition
- **Shared-Memory**
- Inter-Process Communication (IPC)
- Client-Server Paradigm
- Socket-based Communication

5

---

---

---

---

---

---

---

---

## Producer-Consumer Problem

- Paradigm for cooperating processes
- *producer* process produces information ...
- that is consumed by a *consumer* process.
  
- Shared-memory solution
  - ✦ Shared buffer
    - 📖 *unbounded-buffer* places no practical limit on the size of the buffer.
    - 📖 *bounded-buffer* assumes that there is a fixed buffer size.

6

---

---

---

---

---

---

---

---

## Bounded-Buffer Solution

### ■ Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

7

---

---

---

---

---

---

---

---

## Bounded-Buffer Solution

### ■ Producer process

```
item nextProduced;

while (1) {
    while (counter == BUFFER_SIZE)
        /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

8

---

---

---

---

---

---

---

---

## Bounded-Buffer Solution

### ■ Consumer process

```
item nextConsumed;

while (1) {
    while (counter == 0)
        /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

9

---

---

---

---

---

---

---

---

## Bounded-Buffer Solution

- The statements

```
counter++;  
counter--;
```

must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

10

---

---

---

---

---

---

---

---

## Bounded Buffer

- The statement "**counter++**" may be implemented in machine language as:

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

- The statement "**counter--**" may be implemented as:

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

11

---

---

---

---

---

---

---

---

## Bounded-Buffer Solution

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.

12

---

---

---

---

---

---

---

---

## Bounded-Buffer Solution

- Assume **counter** is initially 5. One interleaving of statements is:

producer: **register1 = counter** (*register1 = 5*)  
producer: **register1 = register1 + 1** (*register1 = 6*)

consumer: **register2 = counter** (*register2 = 5*)  
consumer: **register2 = register2 - 1** (*register2 = 4*)

producer: **counter = register1** (*counter = 6*)  
consumer: **counter = register2** (*counter = 4*)

- The value of **count** may be either 4 or 6, where the correct result should be 5.

13

---

---

---

---

---

---

---

---

---

---

## Race Condition

- **Race condition**

- The situation where several processes access and manipulate shared data concurrently.
- The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

14

---

---

---

---

---

---

---

---

---

---

## The Critical-Section Problem

- $n$  processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

15

---

---

---

---

---

---

---

---

---

---

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

### 3. Bounded Waiting.

- A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the  $n$  processes.

16

---

---

---

---

---

---

---

---

## General Process Structure

- General structure of process  $P_i$  (other process  $P_j$ )

```
do {  
    entry section  
    critical section  
    exit section  
    reminder section  
} while (1);
```

- Processes may share some common variables to synchronize their actions.

17

---

---

---

---

---

---

---

---

## Solutions

- Software approaches

- Hardware solutions

- Interrupt disabling
- Special machine instructions

- Operating System Support

- Semaphores

- Programming language Support

- Monitor
- ...

18

---

---

---

---

---

---

---

---

## A Software Solution

```
Boolean lock=FALSE;
Process Pi {
  do {
    while (lock); // do nothing
    lock=TRUE;
    critical section
    lock=FALSE;
    remainder section
  } while (TRUE);
}
```

Does it work?

19

---

---

---

---

---

---

---

---

## Algorithm 1

### ■ Shared variables:

- **int turn;**  
initially **turn = 0**
- **turn = i** ⇒  $P_i$  can enter its critical section

### ■ Process $P_i$

```
do {
  while (turn != i);
  critical section
  turn = j;
  remainder section
} while (1);
```

### ■ Satisfies mutual exclusion, but not progress

20

---

---

---

---

---

---

---

---

## Algorithm 2

### ■ Shared variables

- **boolean flag[2];**  
initially **flag [0] = flag [1] = false.**
- **flag [i] = true** ⇒  $P_i$  ready to enter its critical section

### ■ Process $P_i$

```
do {
  flag[i] := true;
  while (flag[j]);
  critical section
  flag [i] = false;
  remainder section
} while (1);
```

### ■ Satisfies mutual exclusion, but not progress requirement.

21

---

---

---

---

---

---

---

---

### Algorithm 3

- Combined shared variables of algorithms 1 and 2.

- Process  $P_i$

```
do {  
    flag [i] := true;  
    turn = i;  
    while (flag [j] and turn = j) ;  
        critical section  
    flag [i] = false;  
    remainder section  
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.

22

---

---

---

---

---

---

---

---

### Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes  $P_i$  and  $P_j$  receive the same number, if  $i < j$ , then  $P_i$  is served first; else  $P_j$  is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

23

---

---

---

---

---

---

---

---

### Bakery Algorithm

- Notation  $\leq$  lexicographical order (ticket #, process id #)
  - $(a,b) < (c,d)$  if  $a < c$  or if  $a = c$  and  $b < d$
  - $\max(a_0, \dots, a_{n-1})$  is a number,  $k$ , such that  $k \geq a_i$  for  $i = 0, \dots, n-1$
- Shared data
  - boolean choosing[n];**
  - int number[n];**Data structures are initialized to **false** and **0** respectively

24

---

---

---

---

---

---

---

---

## Bakery Algorithm

```
do {
  choosing[i] = true;
  number[i] = max(number[0], number[1], ..., number [n - 1])+1;
  choosing[i] = false;
  for (j = 0; j < n; j++) {
    while (choosing[j]) ;
    while ((number[j] != 0) && (number[j,j] < number[i,i]) ;
  }
  critical section
  number[i] = 0;
  remainder section
} while (1);
```

25

---

---

---

---

---

---

---

---

## Solution based on Locks

```
Process Pi {
  do {
    acquire lock
    critical section
    release lock
    remainder section
  } while (TRUE);
}
```

26

---

---

---

---

---

---

---

---

## Interrupt disabling

### ■ General process structure

```
do {
  <disable interrupt>
  critical section
  <enable interrupt>
  remainder section
} while (1);
```

- Disabling Interrupts for long times may create problems
- Does not work on multiprocessor systems

27

---

---

---

---

---

---

---

---

## Synchronization Hardware

- Test and modify the content of a word atomically

```
boolean TestAndSet(boolean &target) {  
    boolean rv = target;  
    target = true;  
  
    return rv;  
}
```

28

---

---

---

---

---

---

---

---

## Mutual Exclusion with Test-and-Set

- Shared data:  
    boolean lock = false;

```
Process  $P_i$   
do {  
    while (TestAndSet(lock)) ;  
    critical section  
    lock = false;  
    remainder section  
}
```

29

---

---

---

---

---

---

---

---

## Synchronization Hardware

- Atomically swap two variables.

```
void Swap(boolean &a, boolean &b) {  
    boolean temp = a;  
    a = b;  
    b = temp;  
}
```

30

---

---

---

---

---

---

---

---

## Mutual Exclusion with Swap

- Shared data (initialized to false):  
`boolean lock;`  
`boolean waiting[n];`
- Process  $P_i$   
`do {`  
    `key = true;`  
    `while (key == true)`  
        `Swap(lock, key);`  
        critical section  
    `lock = false;`  
    remainder section  
`}`

31

---

---

---

---

---

---

---

---

## Semaphores

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

`wait(S)`  
`signal(S)`

32

---

---

---

---

---

---

---

---

## 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);`

33

---

---

---

---

---

---

---

---

## Semaphore Implementation

- Define a semaphore as a record

```
typedef struct {  
    int value;  
    struct process *L;  
} semaphore;
```

- Assume two simple operations:

- `block` suspends the process that invokes it.
- `wakeup(P)` resumes the execution of a blocked process `P`.

34

---

---

---

---

---

---

---

---

## Implementation

- Semaphore operations now defined as

```
wait(S):  
    S.value--;  
    if (S.value < 0) {  
        add this process to S.L;  
        block;  
    }
```

```
signal(S):  
    S.value++;  
    if (S.value <= 0) {  
        remove a process P from S.L;  
        wakeup(P);  
    }
```

35

---

---

---

---

---

---

---

---

## Semaphore as a General Synchronization Tool

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

```
      Pi          Pj  
      ⋮          ⋮  
      A          wait(flag)  
      signal(flag) B
```

36

---

---

---

---

---

---

---

---

## Deadlock and Starvation

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

- Let S and Q be two semaphores initialized to 1

```
      P0                P1
wait(S);             wait(Q);
wait(Q);             wait(S);
⋮                   ⋮
signal(S);           signal(Q);
signal(Q)            signal(S);
```

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

37

---

---

---

---

---

---

---

---

---

---

## Two Types of Semaphores

- *Counting* semaphore – integer value can range over an unrestricted domain.
- *Binary* semaphore – integer value can range only between 0 and 1; can be simpler to implement.
- Can implement a counting semaphore S as a binary semaphore.

38

---

---

---

---

---

---

---

---

---

---

## Implementing S as a Binary Semaphore

- Data structures:

```
binary-semaphore S1, S2;
int C;
```

- Initialization:

```
S1 = 1
S2 = 0
C = initial value of semaphore S
```

39

---

---

---

---

---

---

---

---

---

---

## Implementing S

### ■ wait operation

```
wait(S1);  
C--;  
if (C < 0) {  
    signal(S1);  
    wait(S2);  
}  
signal(S1);
```

### ■ signal operation

```
wait(S1);  
C ++;  
if (C <= 0)  
    signal(S2);  
else  
    signal(S1);
```

40

---

---

---

---

---

---

---

---

## Classical Problem of Synchronization

### ■ Bounded-Buffer Problem

41

---

---

---

---

---

---

---

---

## Bounded-Buffer Problem

### ■ Shared data

**semaphore full, empty, mutex;**

Initially:

**full = 0, empty = n, mutex = 1**

42

---

---

---

---

---

---

---

---

## Bounded-Buffer Problem

### ■ Producer Process

```
do {  
  ...  
  <produce an item in nextp>  
  ...  
  wait(empty);  
  wait(mutex);  
  ...  
  <add nextp to buffer>  
  ...  
  signal(mutex);  
  signal(full);  
} while (1);
```

### ■ Consumer Process

```
do {  
  wait(full);  
  wait(mutex);  
  ...  
  <remove item from buffer to nextc>  
  ...  
  signal(mutex);  
  signal(empty);  
  ...  
  <consume the item in nextc>  
  ...  
} while (1);
```

43

---

---

---

---

---

---

---

---

---

---

## Cooperating Processes

- Definition
- Shared-Memory
- **Inter-Process Communication (IPC)**
- Client-Server Paradigm
- Socket-based Communication

44

---

---

---

---

---

---

---

---

---

---

## Inter-Process Communication (IPC)

- Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- The communication link is provided by the OS

45

---

---

---

---

---

---

---

---

---

---

## Implementation Questions

### ■ Physical implementation

- Shared memory (single- or multi-processor systems)
- Hardware bus (multi-processor systems)
- Network (distributed systems)

### ■ Logical properties

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

46

---

---

---

---

---

---

---

---

## Other aspects

### ■ Addressing

### ■ Synchronization

- between sender and receiver

47

---

---

---

---

---

---

---

---

## Direct Addressing

### ■ Processes must name each other explicitly.

### ■ Symmetric scheme

- **send** (*D, message*) – send a message to process *D*
- **receive**(*S, message*) – receive a message from process *S*

### ■ Asymmetric scheme

- **send** (*D, message*) – send a message to process *D*
- **receive**(*proc, message*) - receive a message from any process *proc*

### ■ Logical properties

- A communication link exists between exactly two process
- Links are established automatically
- Links are usually FIFO

48

---

---

---

---

---

---

---

---

## Indirect Addressing

- Messages are sent and received through **mailboxes**
  - shared data structures where messages are queued temporarily
  - Sometimes referred to as **ports**
- Processes can communicate only if they share a mailbox.
- Relationships
  - One-to-one (private communication)
  - Many-to-one (client-server communication)
  - Many-to-many (multicast communication)
- Properties of communication link
  - Link established only if processes share a common mailbox
  - Link may be unidirectional or bi-directional.

49

---

---

---

---

---

---

---

---

## Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send**(*mb, message*) – send a message to mailbox *A*
  - receive**(*mb, message*) – receive a message from mailbox *mb*

50

---

---

---

---

---

---

---

---

## Indirect Communication

- Mailbox sharing
  - $P_1$ ,  $P_2$ , and  $P_3$  share mailbox *A*.
  - $P_1$  sends;  $P_2$  and  $P_3$  receive.
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

51

---

---

---

---

---

---

---

---

## Synchronization

- **send and receive** primitives may be
  - **Blocking (synchronous)**
  - **Non-blocking (asynchronous)**
- **Blocking send, blocking receive**
  - Rendez-vous between sender and receiver
- **Non-blocking send, blocking receive**
  - Most useful combination (used by servers)
  - Variations: receive with timeout, select, proactive test
- **Non-blocking send, Non-blocking receive**
  - Neither party is required to wait

52

---

---

---

---

---

---

---

---

## Buffering

- Queue of messages attached to the link; implemented in one of three ways.
  1. Zero capacity – 0 messages  
Sender must wait for receiver (rendezvous).
  2. Bounded capacity – finite length of  $n$  messages  
Sender must wait if link full.
  3. Unbounded capacity – infinite length  
Sender never waits.

53

---

---

---

---

---

---

---

---

## Producer-Consumer: IPC-based solution (1)

Mailbox mb;

```
Process Producer {
  while (1) {
    /* produce message in nextProduced */
    send(mb, nextProduced);
  }
}
```

```
Process Consumer {
  while (1) {
    receive(mb, nextConsumed);
    /* consume message in nextConsumed */
  }
}
```

54

---

---

---

---

---

---

---

---

### Producer-Consumer: IPC-based solution (2)

```
Mailbox mb1, mb2;
Process Producer {
  while (1) {
    /* produce message in nextProduced */
    receive(mb2, consumer_ready);
    send(mb1, nextProduced);
  }
}
Process Consumer {
  while (1) {
    send(mb2, ack);
    receive(mb1, nextConsumed);
    /* consume message in nextConsumed */
  }
}
}
```

55

---

---

---

---

---

---

---

---

### Cooperating Processes

- Definition
- Shared-Memory
- Inter-Process Communication (IPC)
- Client-Server Paradigm
- Socket-based Communication

56

---

---

---

---

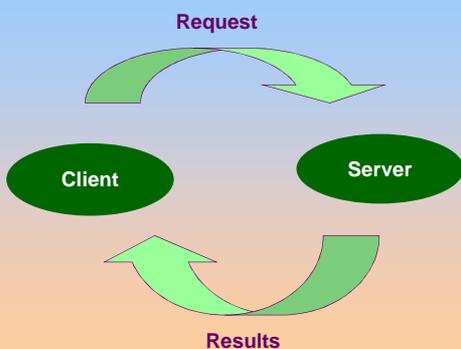
---

---

---

---

### Client-Server Communication



57

---

---

---

---

---

---

---

---

## Sockets

- A socket is defined as an *endpoint for communication*.
- Concatenation of IP address and port
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets.

58

---

---

---

---

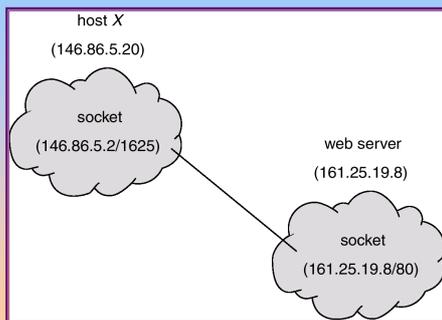
---

---

---

---

## Socket Communication



59

---

---

---

---

---

---

---

---

## Remote Procedure Calls

- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems.
- **Stubs** – client-side proxy for the actual procedure on the server.
- The client-side stub locates the server and *marshalls* the parameters.
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server.

60

---

---

---

---

---

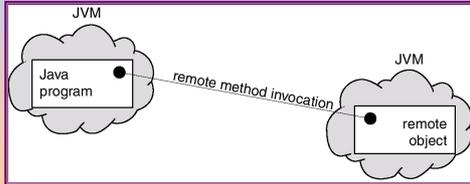
---

---

---

## Remote Method Invocation

- Remote Method Invocation (RMI) is a Java mechanism similar to RPCs.
- RMI allows a Java program on one machine to invoke a method on a remote object.



61

---

---

---

---

---

---

---

---

## Questions?



62

---

---

---

---

---

---

---

---