Process coordination

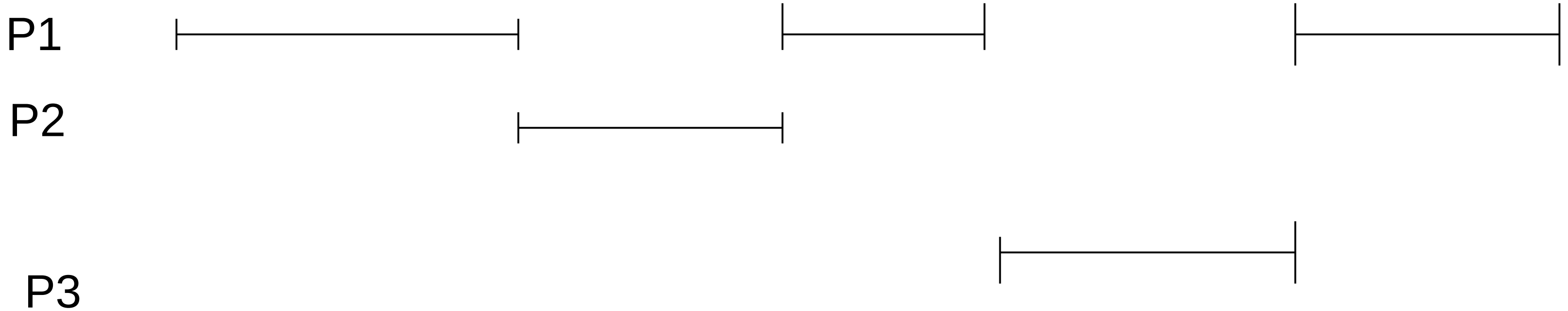
How do we maximize CPU utilization / improve efficiency?

- Multiprogramming
- Multiprocessing

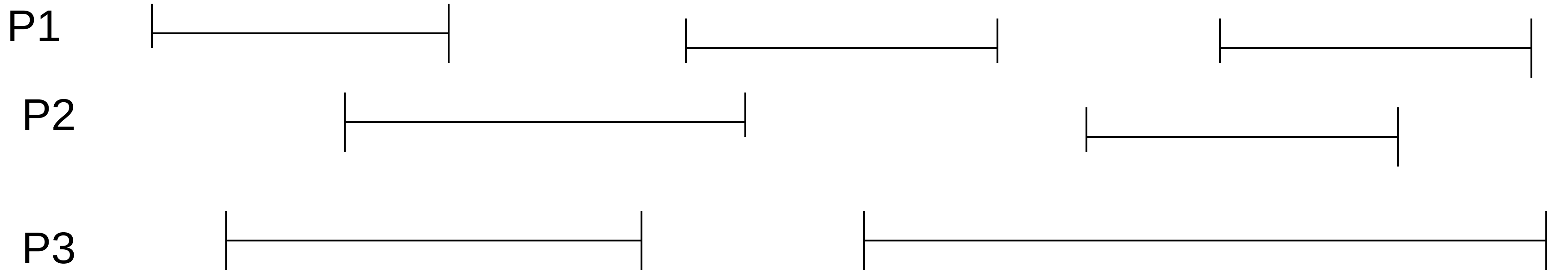
Concurrent operation

- In uni-processor system Processes are interleaved in time
- In multiprocessor system processes are overlapped

Interleaving



Overlapping



Concurrent operation

- Interleaving and overlapping improves processing efficiency

- Interleaved /overlapped process may produce unpredictable results if not controlled properly

Why Problem arises with Interleaving & overlapping

- Finite resources
- Relative speed of execution of processes can not be predicted
- Sharing of resources(non shareable) among processes
  - Sharing of memory is required for Inter process communication

Example

```
Procedure echo;
Var out,in:Character;
Begin
   input (in, keyboard);
   out:=in;
   output(out,Display)
End
```

- Process p1

1. input (in, keyboard);
2. ------------------
3. ------------------
4. ------------------
5. out:=in;
6. output(out,Display)

- Process P2

1. --------
2. input (in, keyboard);
3. out:=in;
4. output(out,Display)

Operating system concern

- The result of a process must be independent of the speed at which the execution is carried out

We need to understand: How processes interact?

- The problem is faced as system has several resources which are to be shared among the processes
- In system we have processes which are
  - unaware of existence of other processes and such processes compete for resources
  - Processes indirectly aware of each other such processes exhibit cooperation
  - Processes directly aware of each other, show cooperation

- When Processes compete for resources,the problem of deadlock , mutual exclusion and starvation may occur
- When processes are directly aware of other processes, the problem of deadlock and starvation still might exist

Concurrency control problem

- Mutual exclusion
- Starvation
- Deadlock

Cooperating & competing processes can cause problem when executed concurrently

How Processes cooperation is achieved ?

- Shared Memory
  - Mutual exclusion
- Message passing

Solution to critical section problem

- Successful use of concurrency requires
  - Ability to define critical section
  - Enforce mutual exclusion
- Any Solution to critical section problem must satisfy
  - Mutual exclusion
  - Progress : when no process in critical section, any process that makes a request for critical section is allowed to enter critical section without any delay
  - Processes requesting critical section should not be delayed indefinitely (no deadlock, starvation)
  - No assumption should be made about relative execution speed of processes or number of processes
  - A process remains inside critical section for a finite amount of time

Approach to handle Mutual Exclusion

- Software Approach (User is responsible for enforcing Mutual exclusion)
- Hardware Support
  - Disabling of Interrupt
  - Special Instructions
- OS support
  - Semaphore
  - Monitor

Software Approach (Solution 1)

Var turn :0..1;

Process 0

```
------
------
While turn <>0 do { nothing};

< Critical Section code >;
Turn := 1;
```

Process 1

```
------
------
While turn <>1 do { nothing};

< Critical Section code >;
Turn := 0;
```

- This solution guarantees mutual exclusion

- Drawback 1: processes must strictly alternate
  - Pace of execution of one process is determined by pace of execution of other processes

- Drawback 2: if one processes fails other process is permanently blocked

This problem arises due to fact that it stores name of the process that may enter critical section rather than the process state

Second Approach

Var flag:Array[0..1] of Boolean;

initially flag is initialized to false

While flag[1] do {nop};

Flag[0]:= true;

< critical section>;

Flag[0]:= false;

- --
- --

While flag[0] do {nop};

Flag[1]:= true;

< critical section>;

Flag[1]:= false;

- --
- --

P1

While flag[1] do {nop};

Flag[0]:= true;

< critical section>;

Flag[0]:= false;

- --
- --

P2

While flag[0] do {nop};

Flag[1]:= true;

< critical section>;

Flag[1]:= false;

- --
- --

Second approach

- If one process fails outside its critical section including the flag setting code then the other process is not blocked
- It does not satisfy the Mutual exclusion
- It is not independent of relative speed of process execution
- Mutual exclusion is not satisfied as processes can change their state after it is checked by other process

Third approach

Flag[0]:= true;

While flag[1] do {nop};

< critical section>;

Flag[0]:= false;

- --
- --

Flag[1]:= true;

While flag[0] do {nop};

< critical section>;

Flag[1]:= false;

- --
- --

- This approach Satisfy mutual exclusion
- This approach may lead to dead lock

What is wrong with this implementation ?

- A process sets its state without knowing the state of other. Dead lock occurs because each process can insist on its right to enter critical section
- There is no opportunity to back off from this situation (discourtesies processes)

Fourth approach

Flag[0]:= true;
While flag[1] do
Begin
Flag [0]:=false;
<delay for short time>
Flag[0]:=true
End;
< critical section>;
Flag[0]:= false;

- --
- --

Flag[1]:= true;
While flag[0] do
Begin
Flag [1]:=false;
<delay for short time>
Flag[1]:=true
End;
< critical section>;
Flag[1]:= false;

- --
- --

Fifth Approach

```
Var flag:Array[0..1] of Boolean;
Turn: 0..1;
Procedure p0
Begin
    Repeat
        flag[0]:= true;
        while flag[1] do if turn = 1then
                        begin
                        flag[0]:=false;
                        while turn=1 do {nothing};
                        flag[0]:=true
                        end;
< critical section >
Turn:=1;
Flag[0]:=false;
Forever
End;
```

Mutual Exclusion (Hardware Approach)

- Process interleaving is mainly due to interrupts / system calls in the system.
- Because of interrupts or system call, a running processes gets suspended and another process starts running which results into interleaved code execution.
- Interleaving of processes is main cause due to which mutual exclusion is required

How do we prevent Interleaving ?

- Interleaving can be prevented by disabling the interrupt in the uniprocessor system

Mutual exclusion by disabling interrupt

Repeat

< disable interrupt >;

< Critical Section >;

<enable Interrupt >;

< remainder section >

Forever.

Problem with Hardware Approach

- Interrupt Disabling can degrade the system performance as it will loose ability to handle critical events which occur in the system
- This approach is not suited for multiprocessor system

Special machine instruction

- We find entry into critical section requires eligibility check and this check consists of several operation eg.

Flag[0]= true;

While flag[1] do {nothing}

- We need instruction which can execute these operations in atomic manner.

Test & Set Instruction

```
Function testset (var i:integer):boolean;
Begin
  if i=0 then
      begin
            i:=1;
            testset:=true
      end
Else testset:=false
End.
```

Mutual exclusion using testset

```
Const n;
Var lock; (Initialized to 0 in the beginning)
Procedure p(i:integer);
Begin

        Repeat { nothing } untill testset(lock);
        < critical section >
        lock:=0;
        <remainder section >

end;
```

Properties of machine instruction approach.

- It is applicable to any number of processes
- It can be used to support multiple critical section. Each critical section can be defined by its own variable
- Busy waiting is employed
- Starvation is possible
- Dead lock is possible

Semaphore (OS Approach)

- We can view semaphore as integer variable on which three operations are defined
  - Can be initialized to a non negative value
  - Decrement operation (*wait*)if the value becomes negative then process executing wait is blocked
  - Increment operation (*Signal*) if the value is not positive then a process blocked by wait operation is unblocked
- Other than these three operations, there is no way to inspect or manipulate semaphore

Mutual exclusion

```
Var s:semaphore; initialized to 1
 Begin
 wait (s);
       < critical Section>;
 signal (s)
End.
```

Var s,r: semaphore; s is initialized to 1& r is initialized to zero

Process P0

- -
- ---
- ------

Begin

wait (s);

< critical Section>;

signal (r)

End.

Process P1

- -
- ---
- ------

Begin

wait (r);

< critical Section>;

signal (s)

End.

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;

Wait operation

```
Type semaphore =record
    count: integer;
    queue: List of processes
End;
Var s: semaphore;
Wait(s)
Begin
        s.count:=s.count-1;
        if s.count <0 then
                        begin
                                place the process in s.queue;
                                block this process
                        end;
end;
```

Signal operation

Signal(s):

```
s.count:=s.count+1
if s.count<= 0 then
            Begin
                  remove a process from s.queue;
                  place this process on ready list
            end;
```

Note:

- S.count>= 0, s.count is number of processes that can execute wait(s) without blocking
- S.count<=0, the magnitude of s.count is number of processes blocked waiting in s.queue

Binary semaphore

```
Type binarysemaphore =record
    value: (0,1);
    queue: List of processes
End;
S: binarysemaphore;
Waitb(s):
    If s.value=1 then s.value=0
                 else begin
                      place this process in s.queue;
                      block this process
                      end;
```

Binary semaphore

```
signalb(s):
   If s.queue is empty then s.value=1
                  else begin
                        remove a  process from s.queue;
                        place this process in ready queue
                        end;
```

Dead lock

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait(S);	*wait*(Q);
wait(Q);	*wait*(S);
signal(S);	*signal*(Q);
signal(Q)	*signal*(S);

Producer consumer problem

- One or more producer are producing some items (data) and a single consumer is consuming these items one by one.
- Consumer can not consume until producer has produced
- While producer is producing, consumer can not consume and vice versa
- We assume producer can produce as many items it wants (infinite buffer)

```
Var n:semaphore (:=0)
    s:semaphore (:=1)
Producer:
Begin
  repeat
      produce;
      wait(s)
      append;
      Signal(s);
      signal (n);
  forever
End;
```

```
Consumer:
  Begin
      repeat
          wait(n);
          wait(s);
          take;
          signal(s);
          consume;
      forever
  End;
```

- What happens if signal(s) and signal(n) in producer process is interchanged ?
  - This will have no effect as consumer must wait for both semaphore before proceeding
- What if wait(n) and wait(s) are interchanged ?
  - If consumer enters the critical section when buffer is empty (n.count=0) then no producer can append to buffer and system is in deadlock.

- What happens if signal(s) and signal(n) in producer process is interchanged ?

- What if wait(n) and wait(s) are interchanged ?

- Semaphore provide a powerful tool for enforcing Mutual exclusion and process coordination but it may be difficult to produce correct program by using semaphore
- wait and signal operation are scattered throughout a program, it is difficult to see the overall effect of these operations on semaphores they affect.

Bounded buffer

Var f,e,s :semaphore (:=0);

(In the begning s=1,f=0,e=n)

Producer

```
Begin
   repeat
        produce;
        wait (e)
        wait(s)
        append;
        Signal(s);
        signal (f);
   forever
End;
```

Consumer

```
Begin
        repeat
                wait(f);
                wait(s);
                take;
                signal(s);
                signal (e);
                consume;
        forever
   End;
```

Bounded-buffer

- Shared data

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

Bounded-buffer

- Producer process

```
item nextProduced;

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

Bounded-buffer

- Consumer process

```
item nextConsumed;

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

Bounded Buffer

- The statements

  counter++;
  counter--;

  must be performed *atomically*.

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

Bounded Buffer

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

  register1 = counter

  register1 = register1 + 1
  counter = register1

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

  register2 = counter
  register2 = register2 – 1
  counter = register2

Bounded Buffer

- 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.

Bounded Buffer

- 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*)

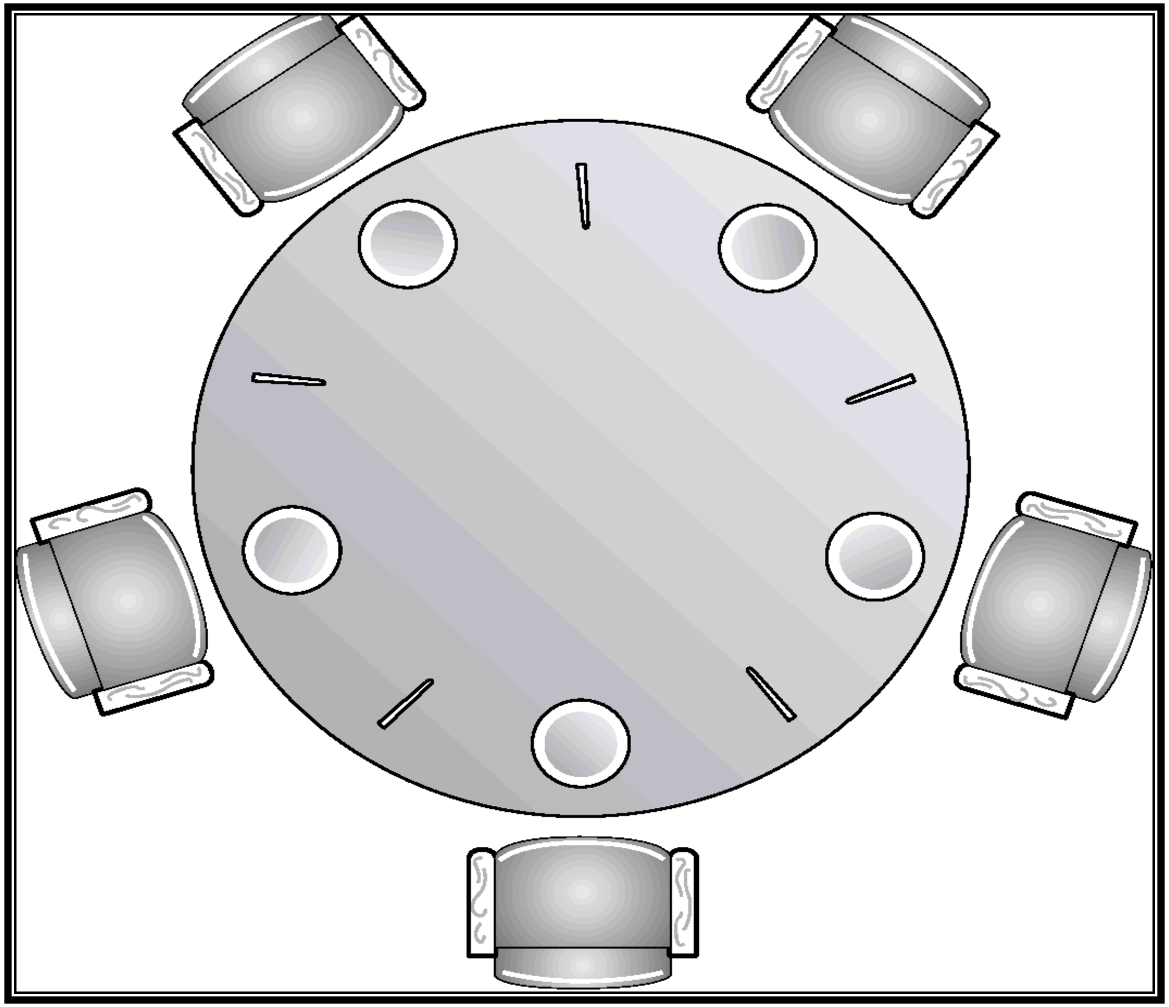
- The value of **count** may be either 4 or 6, where the correct result should be 5.

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**.

Dining-Philosophers Problem



Shared data **chopstick[5]: semaphore; initially it is initialized to 1**

- To start eating, a philosopher needs two chopsticks

- A philosopher may pick up only one chopstick at a time

- After eating, the philosopher releases both the chopsticks

```
do {
        wait(chopstick[i])
        wait(chopstick[(i+1) % 5])
          …
         eat
          …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
          …
         think
          …
        } while (1);
```

The solution is not deadlock free!!

➔ All philosophers pick up left chopsticks!!

➔ Allow at most 4 philosophers to be sitting on the table

➔ Allow a philosopher to pick chopsticks only if both are available

➔ An *odd* philosopher picks up first the left and then the right chopstick while a *even* philosopher does the reverse

Monitors

- Monitor is a software module
- Chief characteristics
  - ➔Local data variables are accessible only by the monitor
  - ➔Process enters monitor by invoking one of its procedures
  - ➔Only one process may be executing in the monitor at a time

Monitor

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

type *monitor-name* = **monitor**

variable declarations

procedure entry *P1* :(…);

begin … **end**;

procedure entry *P2*(…);

begin … **end**;

⋮

procedure entry *Pn* (…);

begin…**end**;

begin

initialization code

end

Monitor

- To allow a process to wait within the monitor, a *condition* variable must be declared, as

  var *x, y: condition*

- Condition variable can only be used with the operations c*wait* and c*signal*.

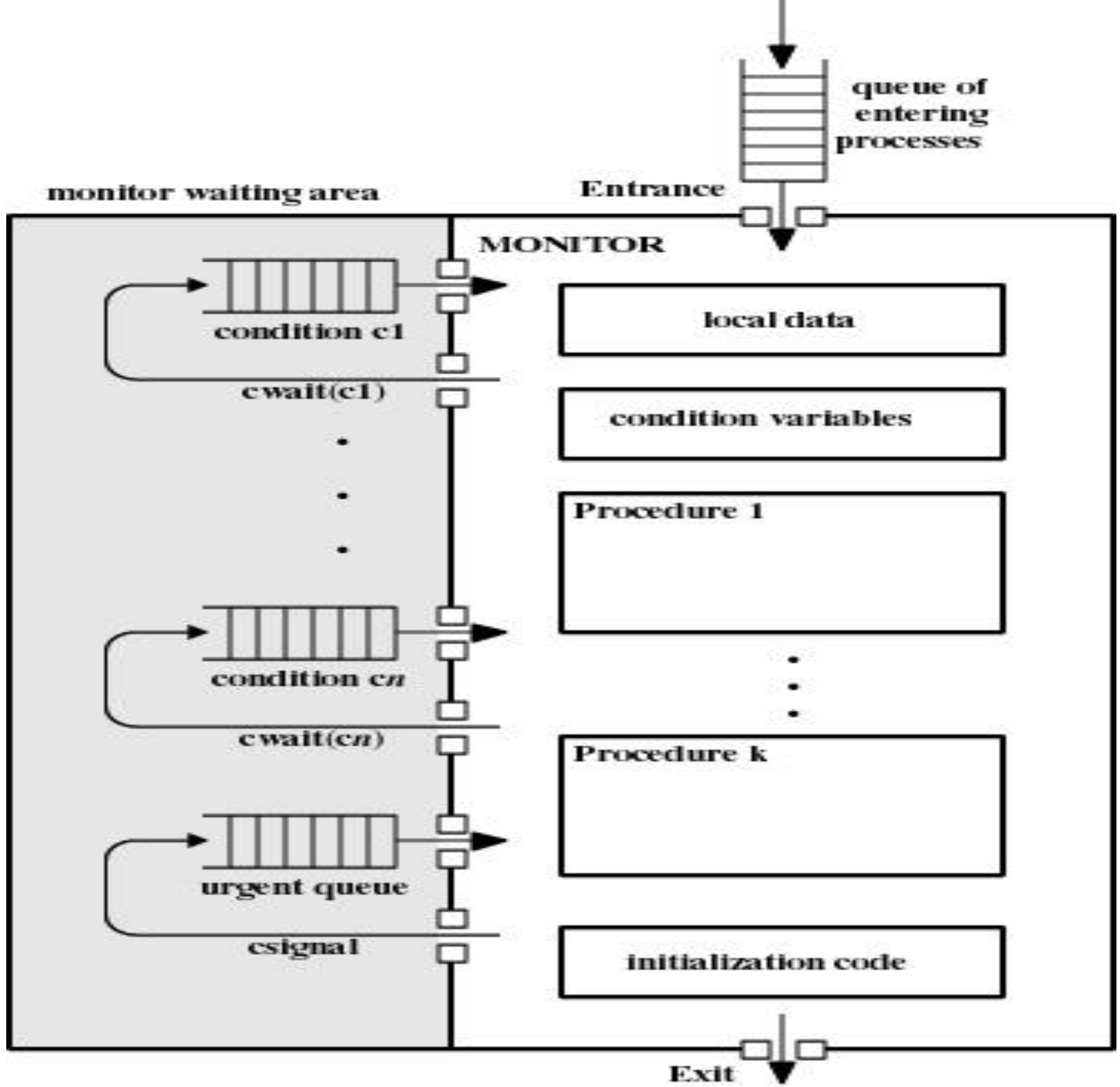
  ➔The operation

  C*wait (x);*

  means that the process invoking this operation is suspended until another process invokes

  C*signal (x)*;

  ➔The C*signal* (x) operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.


queue of entering processes
Entrance
monitor waiting area
MONITOR
condition c1
cwait(c1)
local data
condition variables
Procedure 1
condition cn
cwait(cn)
Procedure k
urgent queue
csignal
initialization code
Exit

Producer/consumer

```
Monitor  bounded-bufer
Buffer array [0..N] of char ;
in, out ,count :integer;
Full,empty :condition;

Procedure append(x:char);
Begin
If count = N then cwait(full) ;
Buffer[in]:=x;
in:= (in+1) mod N;
Count :=count +1;
csignal (empty)
End;
```

```
Procedure take(x:char);
Begin
If count =0 then
    cwait(empty);
x:=buffer[out];
out:=(out+1) mod N;
Count=count-1;
Csignal(full);
End


Begin
In:=0;out:=0;count:=0;
End;
```

```
Procedure producer:
Var X char;
begin
repeat
produce(x)
append(x)
forever
end;
```

```
Procedure consumer;
Var x;
begin
repeat
take(x);
consume(x)
forever
end;
```

Monitor

- A process exits the monitor immediately after executing the csignal
- If process doing csignal is not done then two additional context switch is required
  - One to suspend this process
  - Resume when the monitor becomes available
- Process scheduling should be reliable
  - When csignal is issued, the process from condition queue must get activated immediately

- Allow the executing process to continue
- Replace csignal with cnotify
- Cnotify(x) : it causes x condition to be notified but allows the signaling process to continue
- The signaled condition could get modified

```
Procedure
   append(x:char);
Begin
While  count = N do
   cwait(full) ;
Buffer[in]:=x;
in:= (in+1) mod N;
Count :=count +1;
cnotify (empty)
End;
```

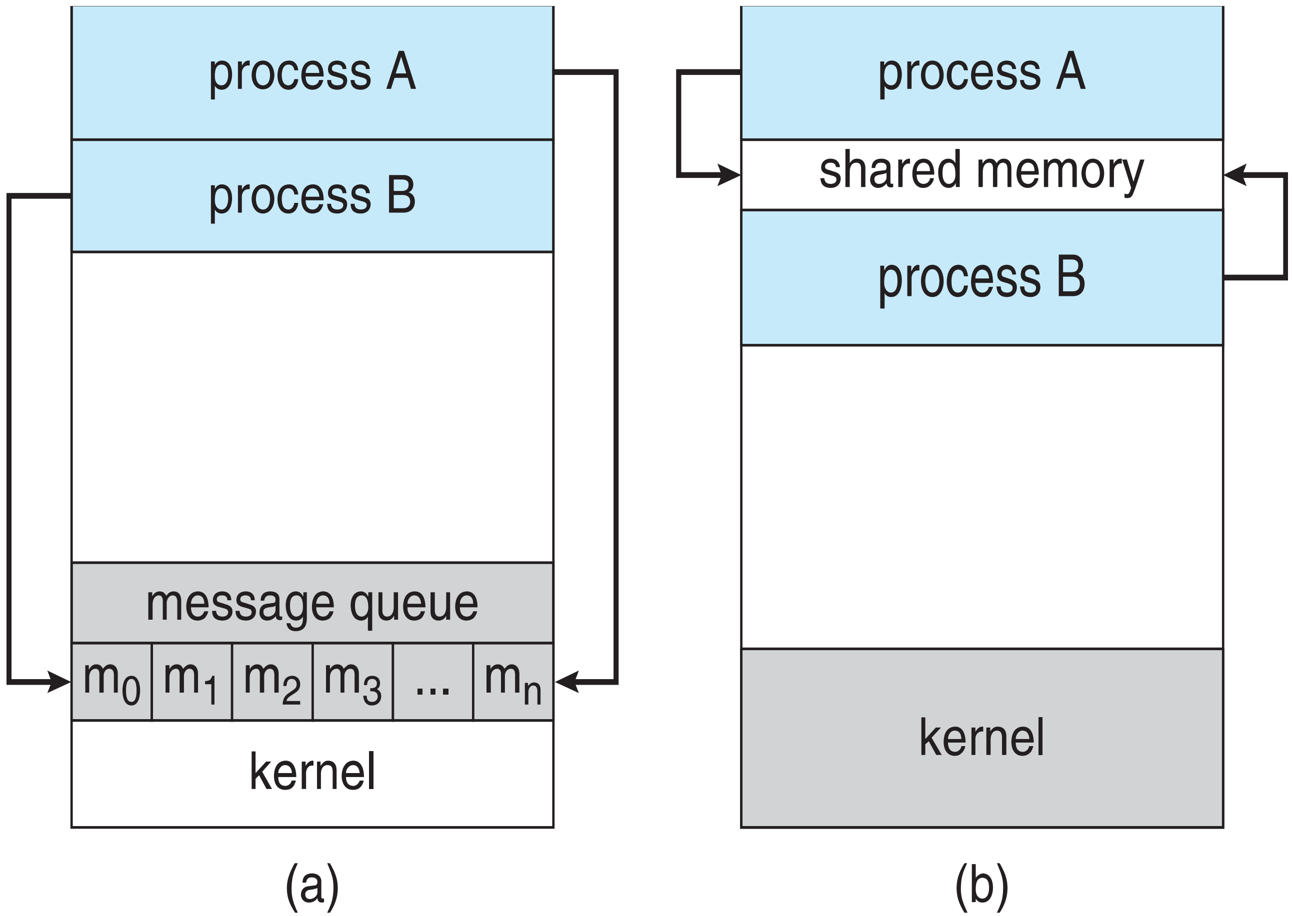
```
Procedure take(x:char);
Begin
While count =0 do
   cwait(empty);
x:=buffer[out];
out:=(out+1) mod N;
Count=count-1;
cnotify(full);
End
```

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

```
(a) Message passing.  (b) shared memory.
```



Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Some synchronization methods already discussed

Message Passing

- The actual function of message passing is normally provided in the form of a pair of primitives:

- send (destination, message)
- receive (source, message)

- Communication requires synchronization
  - Sender must send before receiver can receive
  - Sender and receiver may or may not be blocking (waiting for message)

IPC synchronization

- When send is executed :
  - Sending process can be blocked untill message is received or
  - Is allowed to proceed
- When process issues a receive
  - If message has been previously sent, it receives
  - If there is no waiting message
    - The process is blocked until message is received
    - Process continues to execute abandoning the receive

Blocking send, Blocking receive

- Both sender and receiver are blocked until message is delivered
- Known as a *rendezvous*
- Allows for tight synchronization between processes.

Non-blocking Send

- More natural for many concurrent programming tasks.
- Nonblocking send, blocking receive
  - —Sender continues on
  - —Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - —Neither party is required to wait

Addressing

- Sending process need to be able to specify which process should receive the message
  - Direct addressing
  - Indirect Addressing

Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive could handle in two ways
  - know ahead of time which process a message is expected and explicitly designate the sending process
  - Receive primitive could use source parameter to return a value when the receive operation has been performed

Indirect addressing

- Messages are sent to a shared data structure consisting of queues
- Queues are called *mailboxes*
- One process sends a message to the mailbox and the other process picks up the message from the mailbox
- Indirect addressing decouple the sender & receiver. It gives rise to
  - One to One , One to Many , Many to One and Many to Many relation ship

Indirect Process Communication

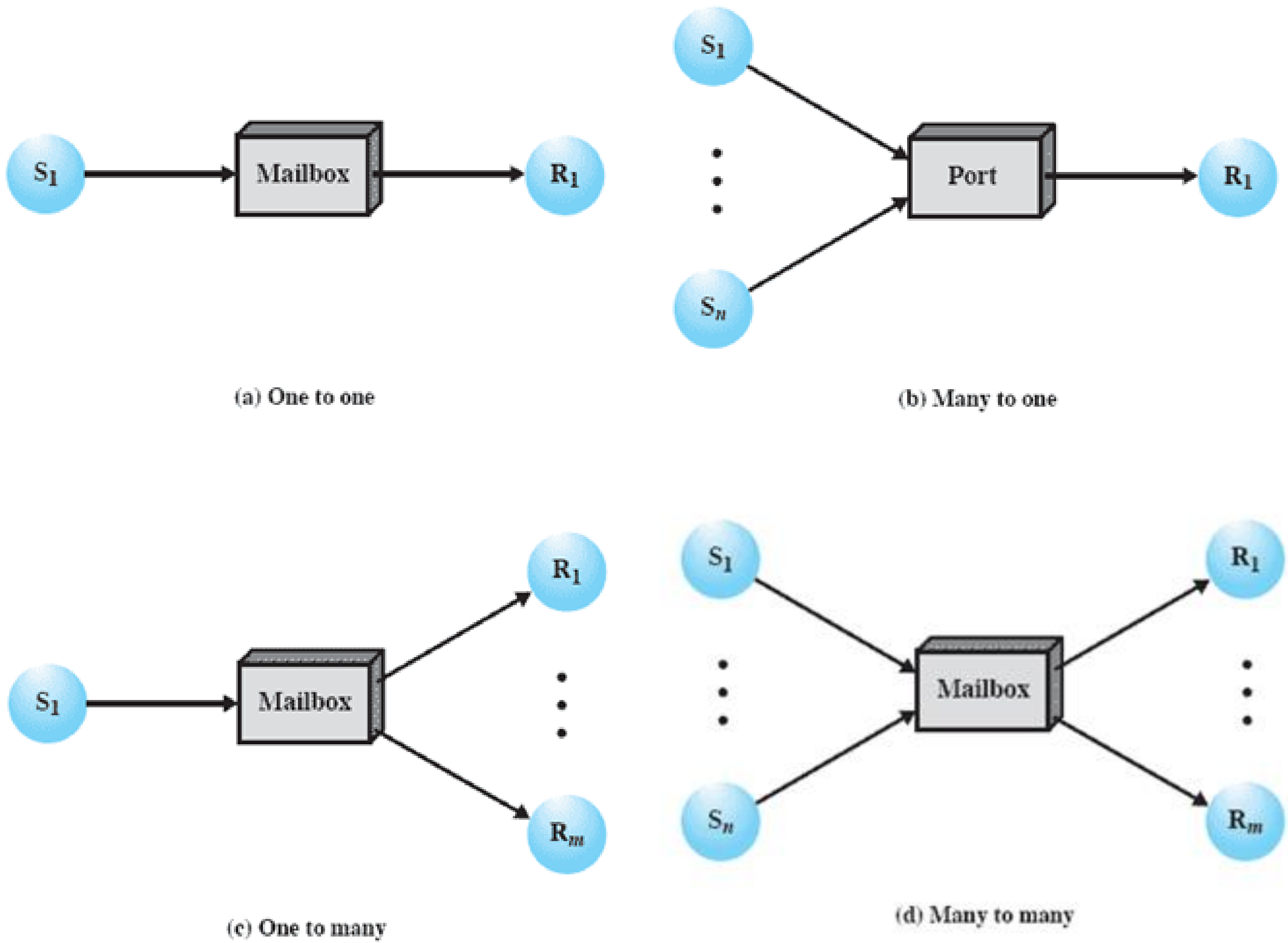


Figure 5.18 Indirect Process Communication

One to one relationship

- One to one relationship allows private communication link to be setup between sender and receiver
- Many to one is useful for client server interaction.
  - One process provides service to number of other process
  - In this case mailbox is known as port
- One to many : used for broadcast

- Association of process to mailbox can be static or dynamic
- Port is created and assigned to process permanently (static association)
- For one to one relationship, it is assigned permanently
- When many senders are sending to mailbox, the association is dynamic
- Connect, disconnect primitive are used for dynamic association

Ownership of mailbox

- A port is typically owned and created by receiving process
- When the process is destroyed, the port is also destroyed
- For general mailbox, OS may offer a create mailbox service.
- Such mailboxes are owned by creating process or OS

General Message Format

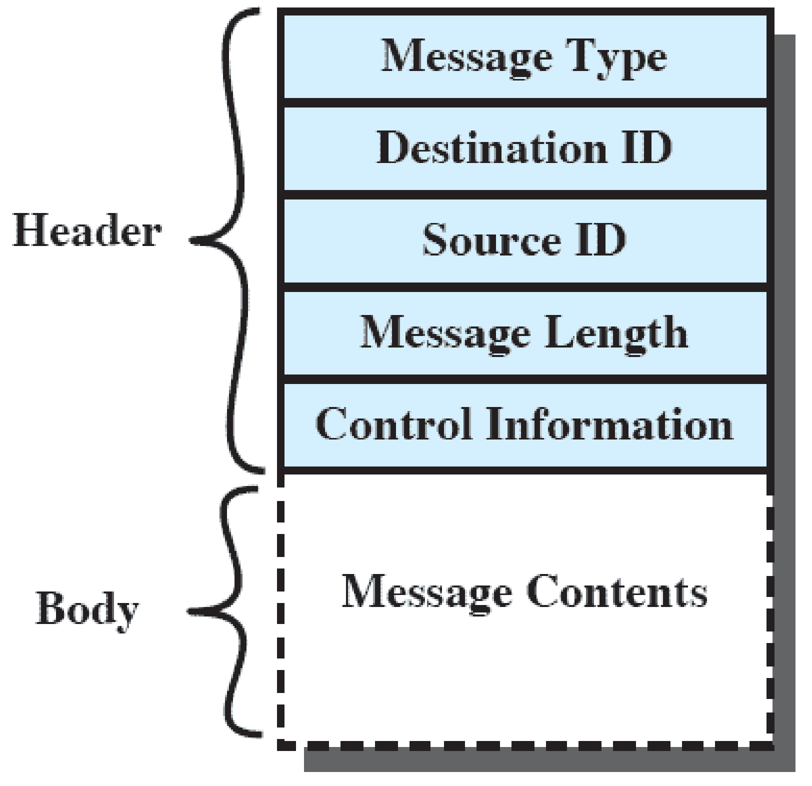


Figure 5.19 General Message Format

Mutual exclusion

- We use blocking receive and non blocking send
- Assume a mailbox named share1
- All process can use mailbox for send & receive operation
- The mailbox is initialized to contain a single message with null content
- A process wishing to enter critical section attempts to receive a message. If the mailbox is empty then the process is blocked .
- Once process has acquired message it enters the critical section, and on completion , places the message back in the mailbox
- The message functions as a token that is passed from process to process. The process having the token enters the CS

receive (share1, msg)

<CS>

Send(share1,msg)

<remainder section>

Producer-Consumer

- We use two mailboxes
  - Consume_1 .
    - It will contains items produced by the producer
    - As long there is one message, the consumer will be able to consume
    - Consume mailbox is serving as buffer . The data in buffer are organized as queue of messages
  - Produce_1
    - Initially mailbox produce_1 has number of null messages equal to size of buffer
    - The number of messages will shrink with each production and increase with each consumption

```
const  capacity =  ….
Null =…..
Var I; integer;
<parent process>
Begin
Create_mailbox (produce);
Create_mailbox (consume);
For i=1 to capicity send (produce, null);
```

```
Procedure consumer;
Var cmsg :message;
Begin
While true do
    begin
    receive(consume_1,cmsg);
     consume(cmsg);
    send(produce_1,null);
    end
end
```

```
Procedure producer;
Var pmsg :message;
Begin
While true do
    begin
    receive(produce_1,pmsg);
    pmsg=produce ;
    send(consume_1,pmsg);
    end
end
```