Chapter 3: Interprocess Communication

Sukomal Pal, CSE, IIT(BHU)

Chapter 3. Interprocess Communication and Process Synchronization

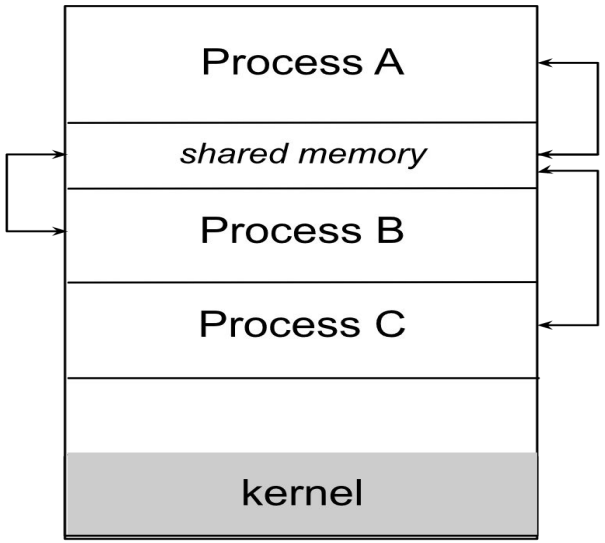
- Inter-process Communication: Critical Section, Race Conditions, Mutual Exclusion, Hardware Solution, Strict Alternation, Peterson's Solution, The Producer Consumer Problem, Semaphores, Event Counters, Monitors, Message Passing,
- Classical IPC Problems: Readers' & Writers' Problem, Dining Philosophers' Problem etc.

INTERPROCESS COMMUNICATION

- Either Processes share some information among them or do not do it at all.
- When they share information among them during execution, they are called **cooperating processes.**
- Otherwise **independent processes**.
- IPC Models:
  1. Shared Memory Model
  2. Message Passing Model

Shared Memory Model

- Processes are allowed to use a memory region in the user space for communication
- One process creates the shared memory.
- Other processes attach and read/write data.



(a)

```
/*****WRITER-PROCESS (shm-writer.c) **/

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(){
// ftok to generate unique key
key_t key = ftok("shmfile",65);
// shmget returns an identifier in shmid
int shmid =
shmget(key,1024,0666|IPC_CREAT);

// shmat to attach to shared memory
char *str = (char*) shmat(shmid,(void*)0,0);
// read input from console
scanf("%s", str);

printf("Data written in memory: %s\n",str);

//detach from shared memory
shmdt(str);

return 0;
}
```

(b)

```
/**READER (shm-reader.c)**/

#include <stdio.h>
#include<string.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(){
// ftok to generate unique key
key_t key = ftok("shmfile",65);

// shmget returns an identifier in shmid
int shmid =
shmget(key,1024,0666|IPC_CREAT);

// shmat to attach to shared memory
char *str = (char*)
shmat(shmid,(void*)0,0);

printf("Data read from shared memory:
%s\n",str);

//detach from shared memory
shmdt(str);

// destroy the shared memory
shmctl(shmid,IPC_RMID,NULL);

return 0;
}
```

(c)

Fig 3.1: (a) Shared Memory Model (b) - (c) IPC implementation of a SM model

Message Passing Model

- The information to be shared among a set of cooperating processes is sent by a process and is received by one or more processes.
- There are several IPC mechanisms that implement message passing:
  - Signal System
  - Message Queuing (MQ)
  - Pipes
  - Sockets

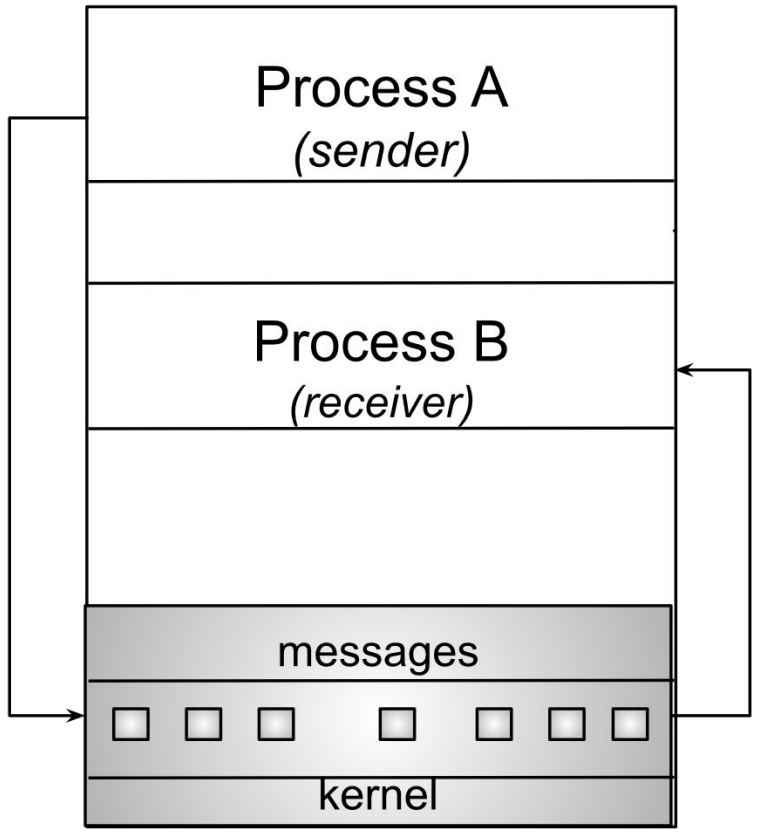


Fig 3.2: Message passing Model

Message Passing Model : Signal System

- Every process has a signal descriptor in its kernel space to get notified on different signals.
- Either the kernel process or any other cooperating process through a syscall sets a particular signal bit ON to notify the recipient process.

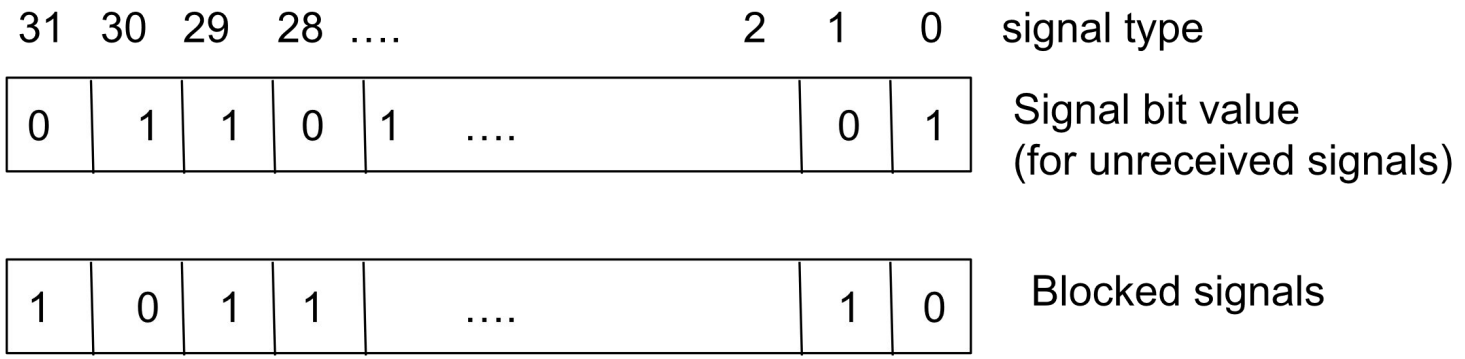
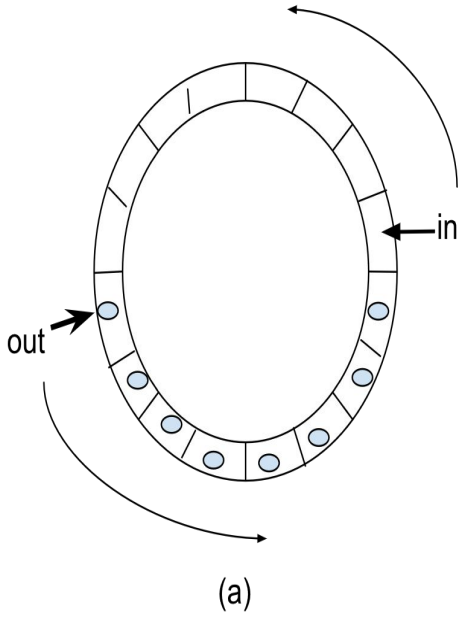


Fig 3.3: Signal descriptor and Blocked signals

Message Passing Model : Message Queuing

- A message buffer acts like a queue and works in strictly FIFO (first-in-first-out) fashion.
- An OS manages message queues but does not see the message contents.
  i. creating message queues (e.g., msgget() function is used in UNIX)
  ii. opening an existing queue (msgget() function with suitable flags)
  iii. sending a message to an open queue (msgsnd() function)
  iv. receiving an available message from an open queue (msgrcv() function)
  v. destroying the queue (msgctl() function).



(a)

Initialisation:

```
#define BUFFER_SIZE N
typedef struct {
. . .
} msg;
msg buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
Int count =0;
```

```
/****PRODUCER *****/
produce() {

 msg next_produced;

while (true) {

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

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

(b)

```
/****CONSUMER *****/

consume(){

 msg next_consumed;

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

    next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
    count - -;
  }
}
```

(c)

Fig 3.4: Message Queue Implementation

Message Passing Model :Pipes

- Pipes are an asynchronous and uni-directional message passing mechanism between two related processes.
- Each process has two file descriptors for a pipe: one for read and another for write.
- **Write End:** Process writes data.
- **Read End:** Another process reads data.
- Construction-wise, named pipes are the same as unnamed pipes except that they have names.
- In UNIX, they are called FIFO and created using **mkfifo** syscall

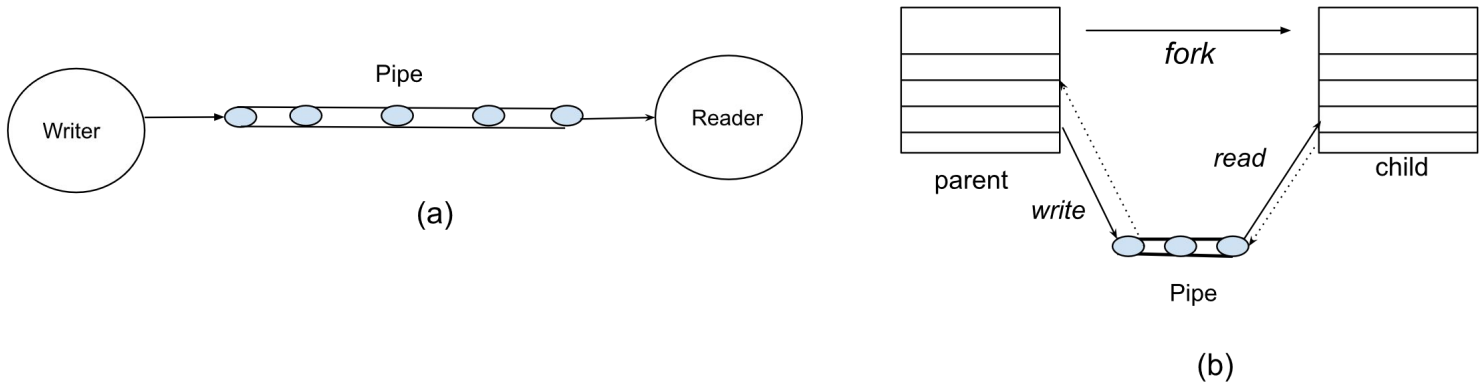
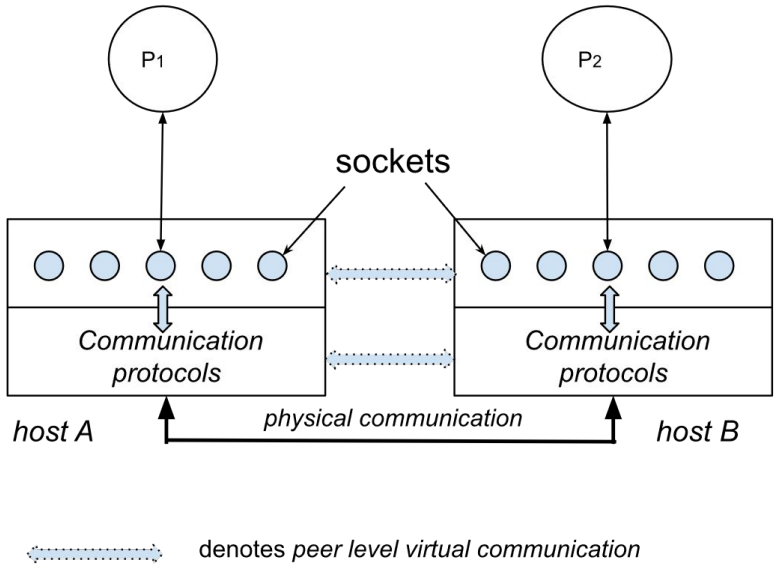


Fig 3.5: (a) Pipe scheme (b) UNIX implementation of pipe

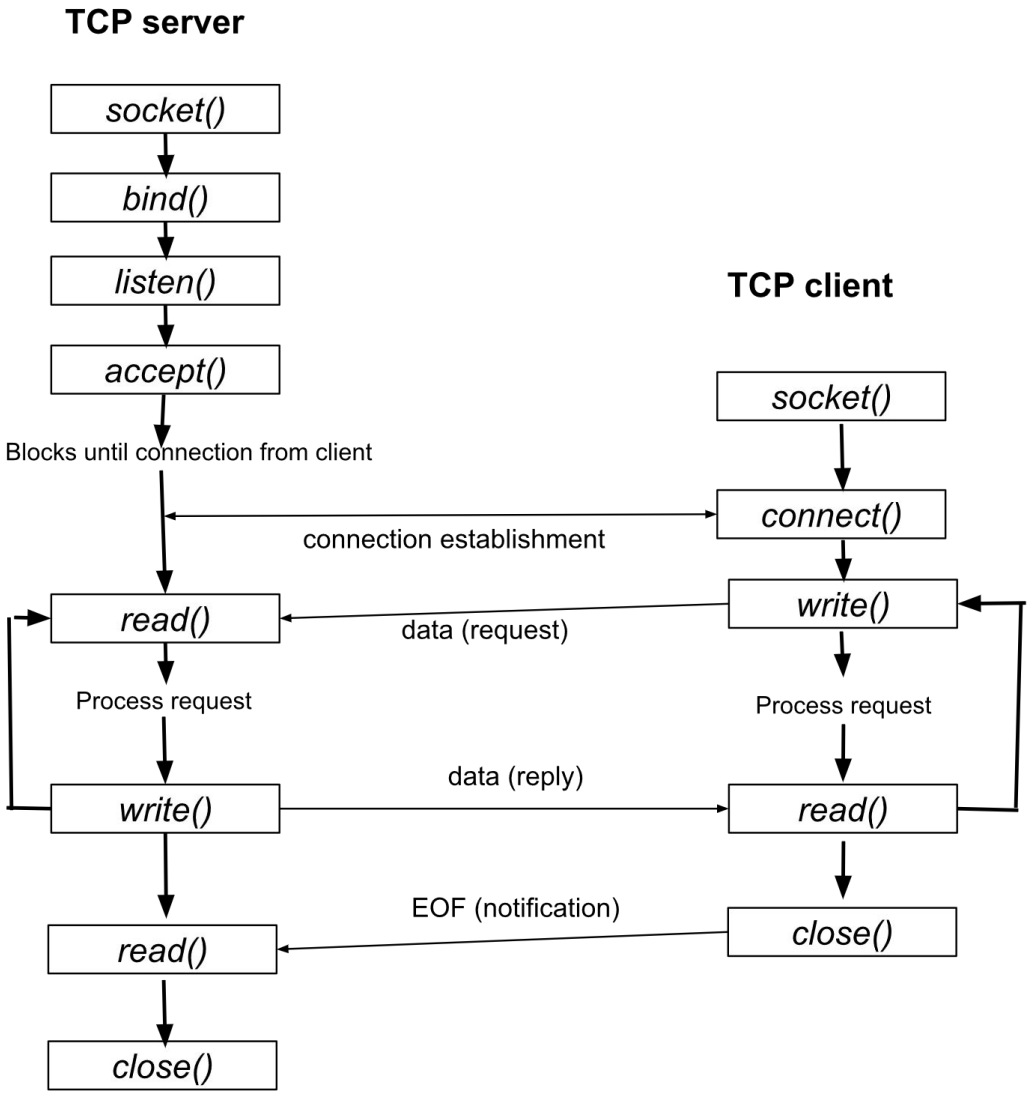
Message Passing Model : Sockets

- Sockets are endpoints of a bi-directional communication channel through which two processes communicate.
- system Calls for Sockets:
  - socket(): Create a socket.
  - bind(): Attach socket to a port.
  - listen(): Wait for connections.
  - connect(): Establish a connection.
  - accept(): Accept an incoming connection.
  - send()/recv(): Transfer data.
  - close(): Close socket.

Message Passing Model : Sockets



(a) Socket conceptual model



(b) Block diagram for TCP client-server socket implementation

Fig 3.6: Sockets

Message Passing Model : Sockets

```
/******SERVER********/
#include<sys/types.h>
#include<sys/socket.h>
#include<string.h>
#include<sys/un.h>
#include<stdlib.h>
#include<stdio.h>
#define ADDRESS "testsocket" /*socket-addr */

int main(){
char c, *server_str = "msg from server\n";
int fromlen;
FILE *fp;
int sd, cd, len;
struct sockaddr_un servaddr, fsaun;

sd = socket(AF_UNIX, SOCK_STREAM,0);      /*create socket */
servaddr.sun_family = AF_UNIX;
strcpy(servaddr.sun_path, ADDRESS);

unlink(ADDRESS);        /* remove earlier file with same name*/
len = sizeof(servaddr.sun_family) + strlen(servaddr.sun_path);
bind(sd, (struct sockaddr *)&servaddr, len);        /*bind socket */
listen(sd, 5);                                      /*listen to client */
cd = accept (sd, &fsaun, &fromlen);           /*accept client request*/
fp = fdopen(cd, "r");                    /*open a fd to read from client */
send(cd, server_str, strlen(server_str), 0);     /*send msg to client*/
while (( c = fgetc(fp)) != EOF) {
putchar(c);                              /*print msg from client*/
if (c == '\n')
break;
}
close(sd); /*close server-side socket */
exit(0);
}
```

```
/*************CLIENT ********/

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdlib.h>
#include <stdio.h>
#define ADDRESS  "testsocket"  /* socket-addr to connect */
char *strs = "A message from client\n"; /*to be sent to server*/

int main(){
char c;
FILE *fp;
int s, len;
struct sockaddr_un saun;

s = socket(AF_UNIX, SOCK_STREAM, 0); /*create socket */
saun.sun_family = AF_UNIX;
strcpy(saun.sun_path, ADDRESS);
len = sizeof(saun.sun_family) + strlen(saun.sun_path);

if (connect(s, &saun, len) < 0)    /*connect to socket */
printf ("error: connect\n");

fp = fdopen(s, "r");  /* opening a fd to read from socket */

while ((c = fgetc(fp)) != EOF){  /*reading from socket */
putchar(c);          /*writing on console */
if(c=='\n')
break;
}
send(s, strs, strlen(strs), 0); /*sending msg to server */
close(s);
exit(0);
}
```

Fig 3.7: Client Server Implementation

SYNCHRONIZATION

- When simultaneous read and write attempts are made or simultaneous writes are attempted on a shared data item by more than one process - their execution order has serious implications.
- These issues may arise because of two reasons:
  1. difference in processing speeds among the cooperating processes
  2. lack of coordination or synchronization among the cooperating processes.
- Synchronization ensures that multiple processes/threads access shared data correctly.
- **Race Conditions:**

  A race condition occurs when multiple processes access and modify shared data simultaneously, leading to unpredictable outcomes.

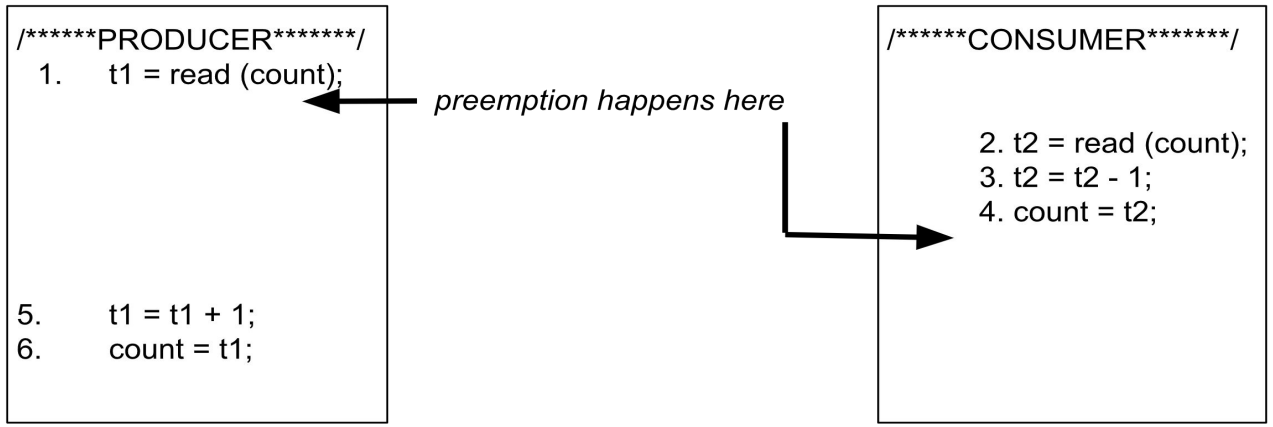


Fig 3.8: Example of a race condition

SYNCHRONIZATION

- **Critical Sections**: A critical section (CS) is a part of a program where shared data is accessed.
- In the example (**Fig 3.8**), both count++ and count-- statements are critical sections in their respective processes.
- The first instruction with which the access begins marks the start of a CS and the instruction where manipulation is complete marks the end of a CS.
- For a given shared variable, there can be several CS for it within a process.
- For the same shared variable, there can be different CS in different processes.

Mutual Exclusion

- Ensures that only one process accesses the critical section at a time.
- Prevents interleaved execution of critical sections.

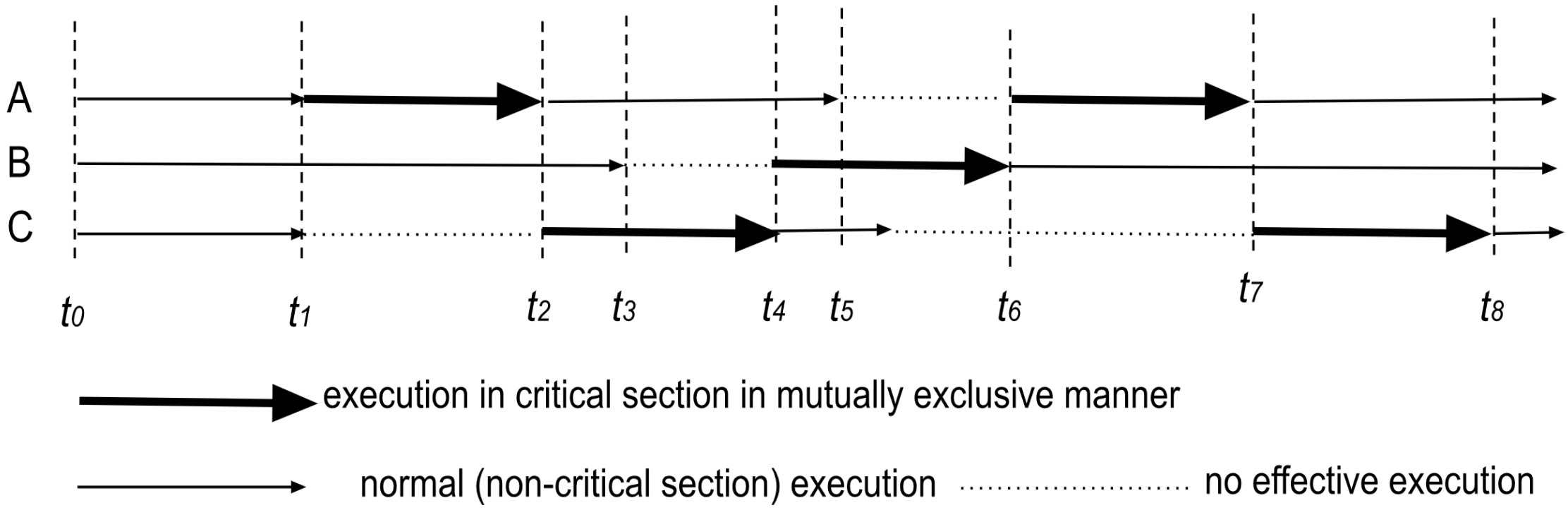


Fig 3.9: An example of mutual exclusion among 3 cooperating processes

CRITICAL SECTION PROBLEM

- Any good solution to the critical section problem must have three following properties:
  - **Mutual Exclusion:** Only one process can enter CS at a time.
  - **Progress:** If no process is in CS, others should be allowed to enter without indefinite waiting.
  - **Bounded Waiting:** Each process gets a fair chance to enter CS.

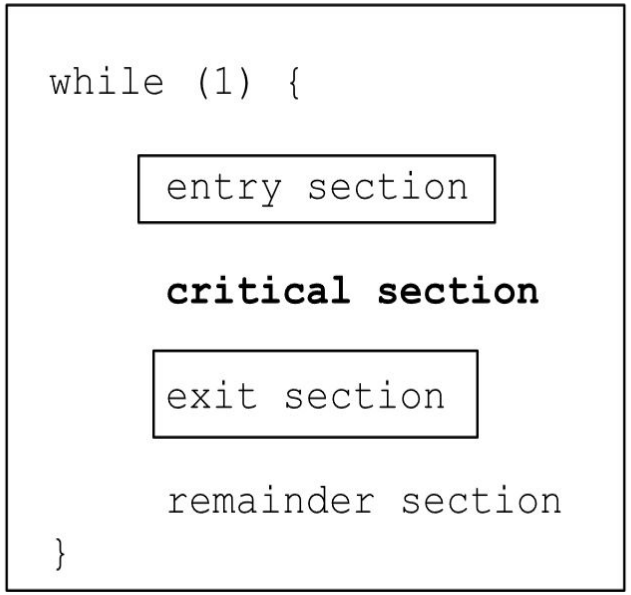


Fig 3.10: critical section structure

SYNCHRONIZATION SOLUTIONS

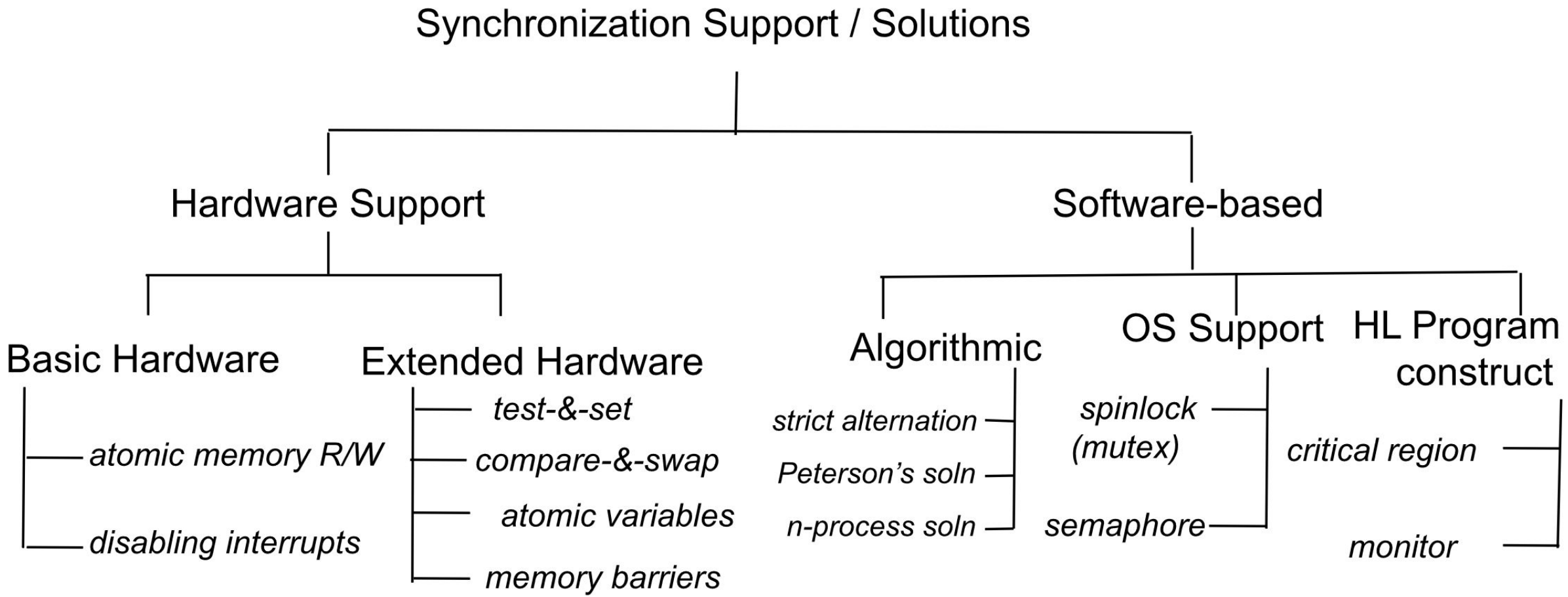


Fig 3.11: Different type of synchronization

Basic Hardware Solutions

- **Atomic memory operations**
  - CPU ensures that a read/write operation completes without interruption.
  - It provides only mutual exclusion

- **Disabling Interrupts**
  - OS disables interrupts before entering CS and re-enables them after exiting.
  - This scheme ensures mutual exclusion, but not progress and bounded wait.
  - Not practical for multiprocessor systems.

```
while (1) {
        disable INTR
        /*entry section*/

        critical section

        enable INTR
        /*exit section*/

        remainder section
}
```

Fig 3.12: Disabling interrupts within process

Extended Hardware Solutions

1. test-&-set lock (TSL):

```
boolean testAndset(boolean *val){
 boolean temp = *val;
 *val = true;
 return temp;
}
```

2. compare-&-swap (CAS):

```
int compareAndswap(int *target, int exp, int newval){
 int temp = *target;

 if (*target == exp)  *val = newval;

 return temp;
}
```

/****** use of test-&-set ****/

```
while (1) {
     while (testAndset(&lock))
     ; /*entry section*/

     critical section

     lock = false; /*exit section */

     remainder section
}
```

(a)

/****** use of compare-&-swap ****/

```
while (1) {
     while (compareAndswap(&lock, 0, 1) != 0)
     ; /*entry section*/

     critical section

     lock = 0; /*exit section */

     remainder section
}
```

(b)

Fig 3.13: Implementation of mutual exclusion using extended hardware support (special instructions)

- There is no guarantee on which process among several contenders will get the chance to go into the CS.
- Bounded wait is not met by the above implementation.

Extended Hardware Solutions

3. **Atomic variables:**
   - Atomic variables ensure that increment/decrement operations execute without interruption.
   - It may not always mitigate the race condition.
4. **Memory barriers:**
   - Some processors reorder instructions to improve performance, which can cause unexpected behavior.
   - Memory barriers ensure that memory operations occur in the correct order.

```
/******Thread 1 ****/
while (!flag)
;
display x;
```

(a)

```
/******Thread 2 ****/

X = 10;
flag = true;
```

(b)

Fig 3.14: Data sharing by two threads and instruction reordering

```
/******Thread 1 ****/
while (!flag)
 memory_barrier();
display x;
```

(a)

```
/******Thread 2 ****/
x = 10;
memory_barrier();
flag = true;
```

(b)

Fig 3.15: Use of memory barrier to stop instruction reordering

Algorithmic Solutions

1. Strict Alternation

- A simple or naive solution to 2-process CSP is to alternately allow the processes to enter CSP.
- Even though all 3 necessary conditions are satisfied, it assumes that both processes would like to enter CS all the time.

```
shared volatile boolean turn = 0;
const boolean self = i;
constant boolean other = 1 - i;

while (1) {

 while (turn != self)
  ; /*entry section */

 <critical section>

turn = other; /*exit section */

}
```

(a) Process P_i

```
turn = 0;
self = 0;
other = 1;

while (1) {

 while (turn != self)
 ;/*entry section*/

 <critical section>

  turn = other;
 /*exit section */
}
```

(b) Process P_0

```
turn = 0;
self = 1;
other = 0;

while (1) {

  while (turn != self)
   ;/*entry section*/

 <critical section>

   turn = other;
 /*exit section */
}
```

(c) Process P_1

Fig 3.16: Strict Alternation for 2-process CSP

Algorithmic Solutions

2. The Peterson Solution
   - Take into consideration the intent of the process to enter CS.
   - Remaining parts of Solution are almost like earlier naive solution to 2-process CSP.

```
shared volatile boolean turn = 0;
Shared volatile boolean flag[2] = 0;

while (1) {
   flag[i] = 1;
   turn = 1 - i;

   while (flag[1-i] && turn == 1-i)
     ; /*entry section */

  <critical section>

  flag[i] = 0; /*exit section */
}
```

(a) Process Pi

```
turn = 0;
flag[0]=0; flag[1]=0;

while (1) {
 flag[0] = 1;
 turn = 1;
 while (flag[1] &&
turn == 1)
 ;/*entry section*/

<critical section>

  flag[0] = 0;
 /*exit section */
}
```

(b) Process P_0

```
turn = 0;
Flag[0] = 0; flag[1]=0;

while (1) {
 flag[1] = 1;
 turn = 0;
 while (flag[0] && turn
== 0)
 ;/*entry section*/

  <critical section>

  flag[1] = 0;
 /*exit section */
}
```

(c) Process P_1

Fig 3.17: Peterson's solution to 2-process CSP

Algorithmic Solutions

3. *n* - process Solution
   - The above solutions do not work for n-process CSP ($n > 2$).
   - The updates on the shared variables flag[j] and lock are very important
   - They are to be done in the given order in all the processes.

P_i	P_0	P_{n-1}

while (1) {
flag[i] = true;
 	
while (flag[i] &&
testAndset (&lock));
/*busy-wait*/
flag[i] = false;
 	
<critical section>
 	
j = (i + 1) % n;
while ((j != i) && !flag[j])
j = (j + 1) % n;
 	
if (j == i)
lock = 0;
else
flag[j] = false;
 	
/* *remainder section* */
}
	while (1) {
	flag[0] = true;
	
	while (flag[0] &&
	testAndset(&lock)); /*busy-wait*/
	flag[0] = false;
	
	<critical section>
	
	j = (0 + 1) % n;
	while ((j != 0) && !flag[j])
	j = (j + 1) % n;
	
	if (j == 0)
	lock = 0;
	else
	flag[j] = false;
	
	/* *remainder section* */
	}
		while (1) {
		flag[n-1] = true;
		
		while (flag[n-1] &&
		testAndset(&lock)); /*busy-wait*/
		flag[n-1] = false;
		
		<critical section>
		
		j = (n) % n;
		while ((j != n-1) && !flag[j])
		j = (j + 1) % n;
		
		if (j == n-1)
		lock = 0;
		else
		flag[j] = false;
		
		/* *remainder section* */
		}

Fig 3.18: *n*-process algorithmic solution to CSP

Operating System Support

1. Mutex:
   - Mutex is the acronym of mutual exclusion.
   - A mutex lock is a simple synchronization tool designed to ensure mutual exclusion.
   - A mutex lock allows two simple operations: acquire() and release() which are considered atomic.

```
typedef struct {
boolean avail;
} mut_lock;
```

(a) Definition of a mutex lock

```
acquire(){
   while (!avail);
    /* busy wait */

avail = false;
}
```

(b) Acquiring a mutex lock

```
release(){
avail = true;
}
```

(c) Releasing a mutex lock

Fig 3.19: Definition of a simple mutex lock and its implementational prototype

Operating System Support

2. Semaphores:
   - A semaphore *S* can be considered as an integer variable.
   - Two atomic operations `wait()` and `signal()`

```
typedef struct {
int val;
} semaphore;

semaphore S;
```

(a) Definition of a semaphore

```
wait(S){
 while (S.val <=0);
    /* busy wait */

 S.val-- ;
}
```

(b) waiting with busy probing

```
signal(S){
S.val++;
}
```

(c) semaphore release

Fig 3.20: Simple definition of semaphore and its basic operations

Semaphores

- Types of semaphores:
  - **counting semaphore:** Allows multiple but limited number of processes to simultaneously access a shared resource.
  - **binary semaphore:** like a mutex lock. Its value thus can be 0 or 1.
- Implementation of a semaphore:

```
typedef struct {
int val;
struct process *list;
} semaphore;

semaphore *S;
```

(a) Definition of a semaphore

```
sem_wait(S){
  S→val--;
  if (S→val < 0) {
    add this process to S→list;
    sleep();}
}
```

(b) wait without busy loop

```
sem_signal(S) {
   S→val++;
 if (S→val <= 0) {
  remove process P from S→list;
 wakeup(P);}
}
```

(c) signal

Fig 3.21: A semaphore and its implementational prototype

- Use of semaphores:

```
 sem_wait(s); /*entry section */

<critical section>

sem_signal(s); /*exit section */
```

(a) Semaphore for mutual exclusion

```
/* process P1 */

  S1;
sem_signal(sync);
```

```
/* process P2 */

  sem_wait(sync);
  S2;
```

(b) Semaphore for serializing sentences

Fig 3.22: Illustrative uses of semaphores

Programming Language Constructs

- Mutex lock, Spinlock or Semaphores do not provide ready-made solutions to CSPs.
- They need to be intelligently used along with their associated functions.

```
sem_wait(s);  /*entry section */

<critical section>

sem_wait(s);  /*exit section */
```

(a) Wrong use of semaphore for mutual exclusion

```
/* process P1 */

  S1;
sem_wait(sync);
```

```
/* process P2 */


sem_signal(sync);
  S2;
```

(b) Wrong use of semaphore for serializing sentences

Fig 3.23: Examples of wrong uses of semaphores

Programming Language Constructs

1. **Critical Regions**
   - A critical region is a region consisting of one or more critical sections.
   - General construct of a critical region: "region cr do S;"
   - cr is the critical region variable, S is the critical section.
   - Unconditional critical regions are not always sufficient.
   - **Conditional Critical Region:** "region cr do S_1; … **await (E)** .... S_n;"

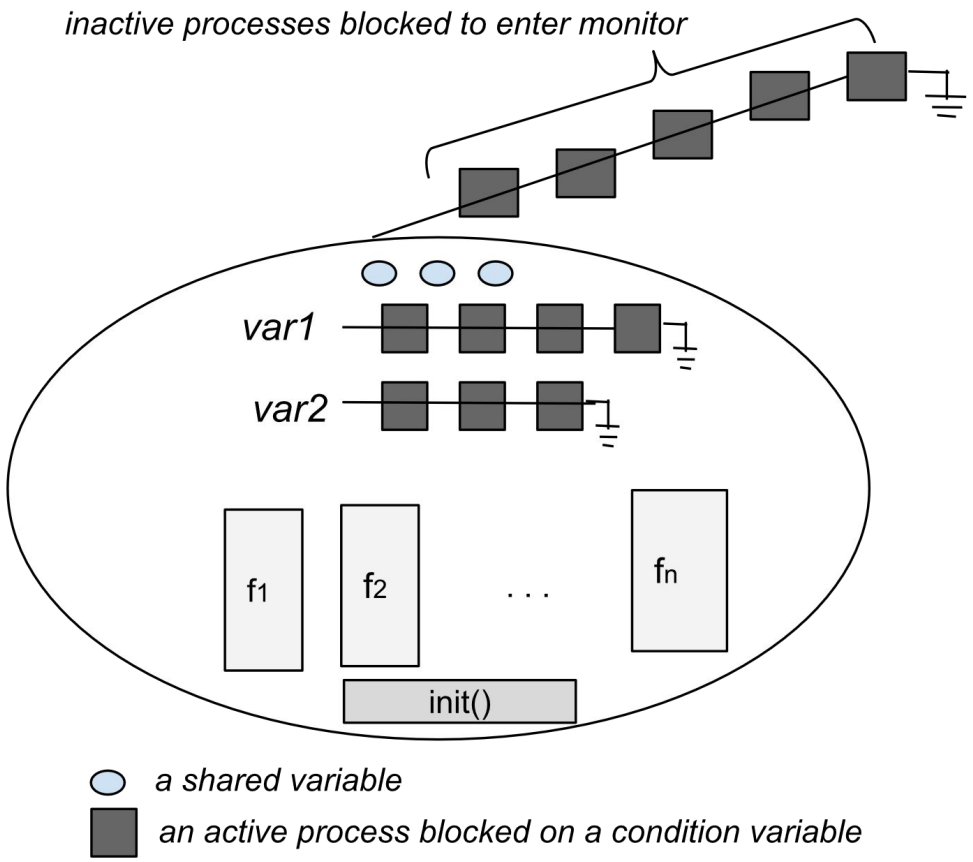
Programming Language Constructs

2. Monitors

- Encapsulate both data and methods, resembling objects in C++ or Java.

```
monitor <monitor-name> {

/* shared variable declarations */

  condition variable <var1, var2>

  function <f1> ( . . . ) {
   . . .
  }
  function <f2> ( . . . ) {
   . . .
  }
  .
  .
  .
   function <fn> ( . . . ) {
   . . .
   }
   function init( . . . ) {
   . . . /*initialization */
   }
}
```

a) structure of a monitor



b) conceptual diagram of a monitor

Fig 3.24: A monitor and condition variables and blocked processes

Programming Language Constructs

- **Implementation of a monitor using semaphores**

```
wait(mutex);
  …
<monitor-function>;
  …
if (next_count >0)
   signal(next);
else
  signal (mutex);
```

Fig 3.25: An implementation of monitor using semaphores

```
/*** x.wait() ***/

x_count++;

if (next_count > 0)
  signal (next);
else
  signal(mutex);

wait(sem_x);
x_count--;
```

```
/*** x.signal() ***/

if (x_count > 0){
    next_count++ ;

    signal (sem_x);

    wait (next);
    next_count-- ;
}
```

Fig 3.26: Illustrative implementation of condition variable actions of a monitor

Solutions without enforcing mutual exclusion

1. EventCounts
   - EventCounts track event occurrences using an integer counter.
   - advance(E): Increments event counter.
   - await(E, v): Blocks a process until event E reaches value v.
   - read(E): Reads the event counter value.
2. Sequencers
   - Sequencers generate unique sequence numbers for ordered execution.

```
sem_wait(S){
  int t = ticket (S.T);
  await (S.E, t - S.I);
}
```

```
sem_signal(S){
  advance(S.E);
}
```

Fig 3.27: Implementation of semaphore using an eventcount and a sequencer

CLASSICAL IPC PROBLEMS

1. The Producer-Consumer Problem
2. The Readers-Writers Problem
3. The Dining Philosophers Problem

The Producer-Consumer Problem

- A producer generates items and places them in a buffer while a consumer retrieves them.

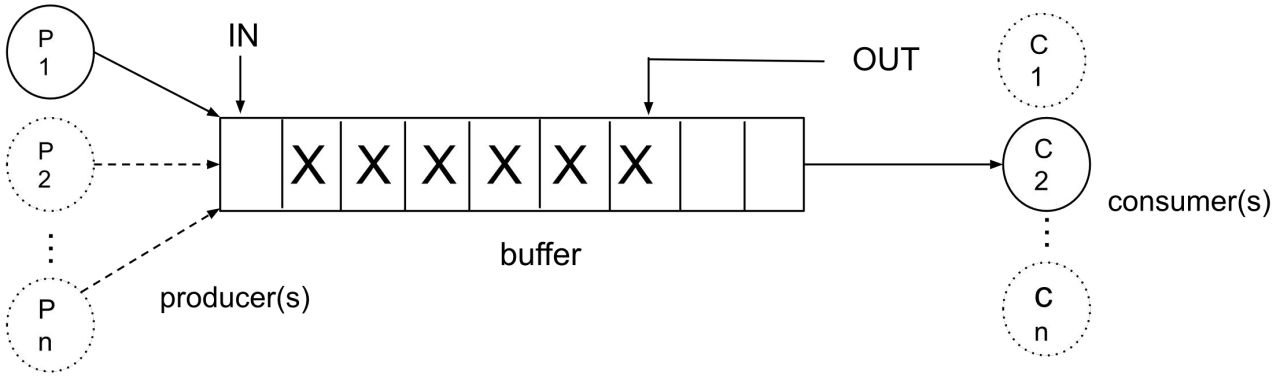


Fig 3.28: Producer-Consumer Model

- The producer and consumer can simultaneously access the buffer asynchronously.
- The producer must stop when the buffer is full and loops around in a busy-wait state.
- Similarly, the consumer must be in busy-wait when the buffer is empty.

The Producer-Consumer Problem

```
/** Common data structures and initialisations **/

constant n  = #slots in the buffer;/*BUFFERSIZE*/

semaphore empty = n;   /*counts empty slots */
semaphore full = 0;        /*counts occupancy */
semaphore mutex = 1;  /*mut-ex access to slots*/
```

```
/****PRODUCER *****/

while (true) {
        …
/* produce next item */
        …
    sem_wait (empty);
    sem_wait (mutex);
        …
/* put next item in buffer */
        …
    sem_signal (mutex);
    sem_signal (full);
}
```

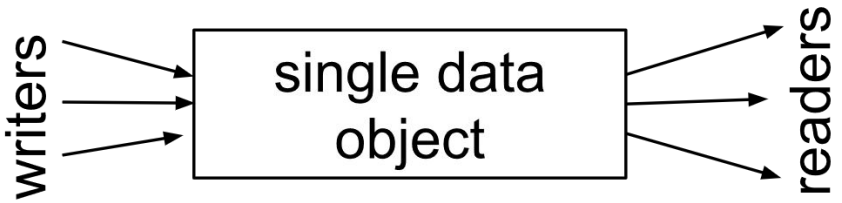
```
/****CONSUMER *****/

while (true) {
    sem_wait (full);
    sem_wait (mutex);
        …
/*pick first item from buffer*/
        …
    sem_signal (mutex);
    sem_signal (empty);
        …
/*  consume picked-up item */
        …
}
```

Fig 3.29: Producer-Consumer Problem Solution

- Progress and bounded wait are met for single producer-single consumer cases, but not for multiple producers.

The Readers'-Writers' Problem

- A shared resource (e.g., file, database record) can be read by multiple readers but cannot be read & written simultaneously.
- Either the readers can be given priority or the writers.




```
/** Common data structures and initialisations **/

semaphore rw_mutex = 1;  /* b/w read & write */
int r_count =0;              /* #simultaneous reads */
semaphore mutex = 1;      /* for update r_count */
```

```
/****WRITER ****/

while (true) {

    sem_wait (rw_mutex);

     ...
 <write operation>
       …

  sem_signal (rw_mutex);

}
```

```
/****READER *****/
while (true) {
     sem_wait (mutex);
     r_count++;

      if (r_count ==1)
        sem_wait (rw_mutex);
      sem_signal (mutex);
        …
       <read operation>
        …
      sem_wait (mutex);
       r_count - -;

      if (r_count ==0)
       sem_signal (rw_mutex);

      sem_signal (mutex);
 }
```

Fig 3.30: Solution to the readers-writers problem (readers' priority) using semaphores

The Dining Philosophers' Problem

- Five philosophers are sitting around a round-table to dine. They primarily think but attempt to eat when they feel hungry. Each philosopher has a private plate but there are only five forks. Assume, each philosopher needs two forks (both left and right) to eat and thus not all of them can eat at the same time.
- can we devise an algorithm that all the philosophers can complete dining without any issues or difficulties?

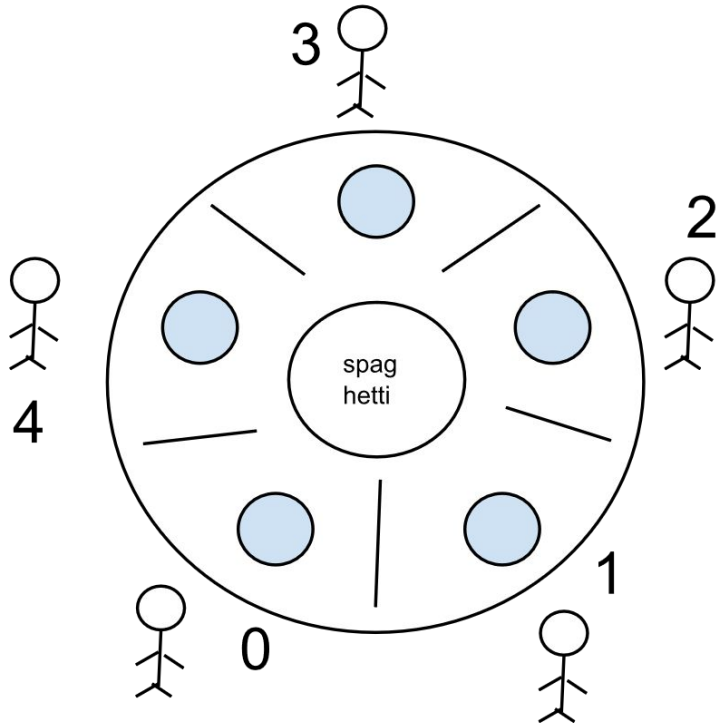


Fig 3.31: Dining Philosophers

The Dining Philosophers' Problem

```
/** PHILOSOPHER 0 **/
int left = 0;
int right = 1;

while (true) {
  wait(fork[left]);
  wait(fork[right]);
     . . .
    /* eats for some time */
     . . .
   signal(fork[left]);
   signal(fork[right]);
        . . .
       /* think again*/
      . . .
}
```

. . .

```
/** PHILOSOPHER i  **/
int left = i;
int right = (i+1)%n;

while (true) {
  wait(fork[left]);
  wait(fork[(right]);
     . . .
    /* eats for some time */
     . . .
   signal(fork[left]);
   signal(fork[right]);
        . . .
       /* think again*/
      . . .
}
```

. . .

```
/* PHILOSOPHER (n-1) */
int left = n-1;
int right = 0;

while (true) {
   wait(fork[left]);
   wait(fork[right]);
     . . .
    /* eats for some time */
     . . .
   signal(fork[left]);
   signal(fork[right]);
        . . .
       /* think again*/
      . . .
}
```

Fig 3.32: Naive solution attempt to the dining philosophers problem using semaphores

The Dining Philosophers' Problem

```
monitor DP {
constant n  = #philosophers          /* should be >=2 */
boolean fork[n] = {1,1,...,1}        /*initialised as fork available */
condition cfork[n];                  /* condition for each fork */

void get_forks(int i){

  int left = i;
  int right = (i +1)%n;

  if (!fork[left]) cfork[left].wait();        /*wait for left-fork */
  fork[left] = 0;                             /*left-fork acquired */

  if (!fork[right]) cfork[right].wait();      /*wait for right-fork */
  fork[right] = 0;                            /*right-fork acquired */
}

void put_forks(int i){
  int left = i;
  int right = (i +1)%n;

   fork[left] = 1;                            /*left-fork freed */
   fork[right] = 1;                           /*right-fork freed */

   fork[left].signal();
  fork[right].signal();
}
```

```
/* from a philosopher (i), outside monitor */

while (true) {
        …
        <think>
        …
        DP. get_forks(i);
        …
        <eat>
        …
        DP.put_forks(i);
        …
        <think>
        …
}
```

```
/* the main driver program outside monitor */

void main() {

parbegin (philosopher [0], philosopher[1], …,
          philosopher[n-1]);

/* parbegin indicates parallel or concurrent
execution of its arguments */
}
```

Fig 3.33: Solution to the dining philosophers problem using a monitor & condition variables

Reference

[1] “OPERATING SYSTEMS”, Author: Dr. Sukomal Pal Associate Professor Department of Computer Science & Engineering IIT (BHU), Varanasi, UP