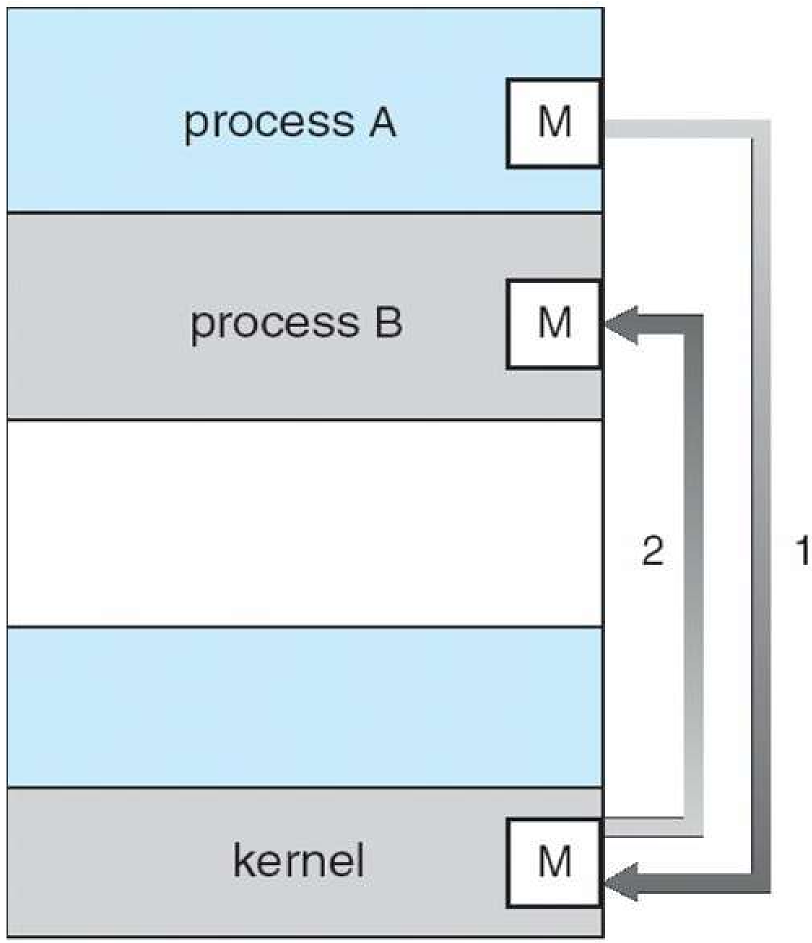
Process Synchronization

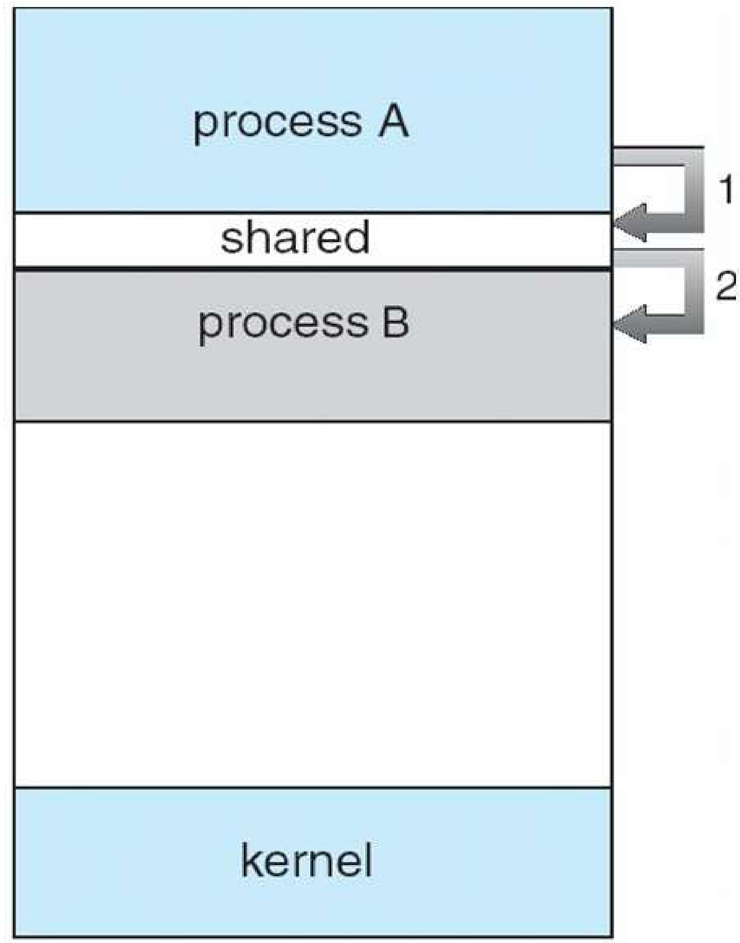
Interprocess Communication

- 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
  - Resources
  - Shared Code
  - Variables
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - **Message passing**

Communications Models



(a)



(b)

Concurrent Processes

Two processes are said to be concurrent if they over lap in their execution.

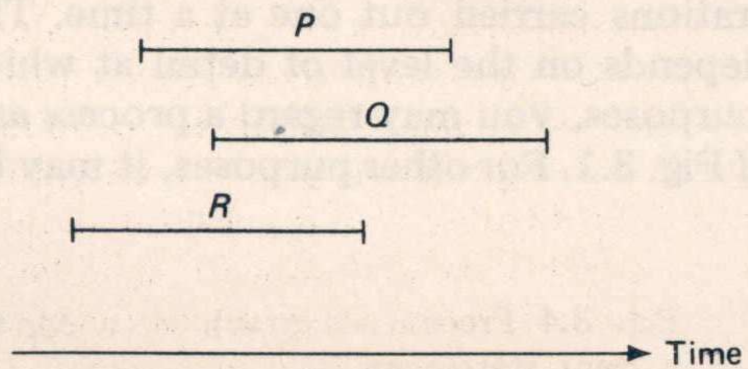


Fig. 3.2 Three concurrent processes P, Q, and R.

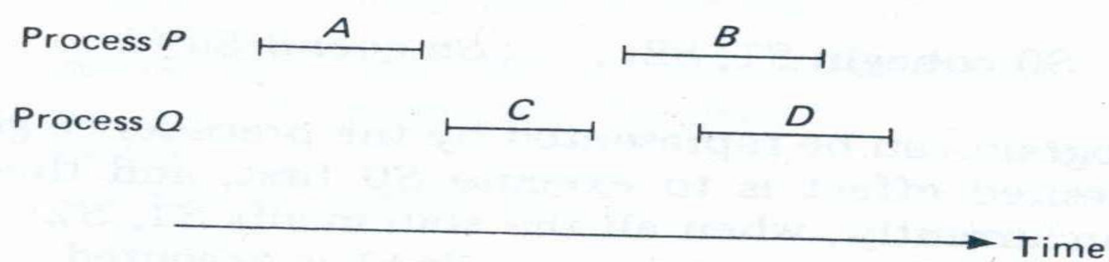


Fig. 3.3 Two concurrent processes P and Q consisting of operations A, B and C, D which are partly interleaved, partly overlapped in time.

Introduction of Process Synchronization

- **counter++ could be implemented as:**
  register1 = counter
  register1 = register1 + 1
  counter = register1

- **counter-- could be implemented as**:
  register2 = counter
  register2 = register2 – 1
  counter = register2

  Consider this execution interleaving with "count = 5" initially:

- S0: producer execute register1 = counter {register1 = 5}
- S1: producer execute register1 = register1 + 1 {register1 = 6}
- S2: consumer execute register2 = counter {register2 = 5}
- S3: consumer execute register2 = register2 – 1 {register2 = 4}
- S4: producer execute counter = register1 {counter = 6 }
- S5: consumer execute counter = register2 {counter = 4}

Race Condition

Critical Section Problem

- Consider system of n processes {p0, p1, … pn-1}
- Each process has critical section segment of code
- Process may be changing common variables, updating table, writing file, etc
- When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
- Critical section is a code segment that can be accessed by only one process at a time.
- Critical section contains shared variables which need to be synchronized to maintain consistency of data variables.

Critical Section Problem


```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

Critical Section Problem

Any solution to the critical section problem must satisfy three requirements:

- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress** : If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection cannot be postponed indefinitely.
- **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.

Two process solution for critical section

First Solution: Boolean turn=0;

	P_0	P_1

do {
 entry section
 critical section
 exit section
 remainder section
 } while (TRUE);
	while(1)
	 {
	 While(turn!=0);
	 Critical Section
	 Turn=1;
	 Remainder Section
	 }
		while(1)
		 {
		 While(turn!=1);
		 Critical Section
		 Turn=0;
		 Remainder Section
		 }

- **Solution first is following Mutual Exclusion and fails to follow Progress as both process spin in some order that is both process can be executed one by one. Any Process P0 or P1 cannot be executed two times sequentially.**

Two process solution for critical section

Second Solution: Boolean Flag[2]=

F	F
0	1

	P_0	P_1

do {
entry section
critical section
exit section
remainder section
} while (TRUE);
	while(1)
	{
	Flag[0]= T;
	while(flag[1]);
	Critical Section
	flag[0]=F;
	Remainder Section
	}
		while(1)
		{
		Flag[1]= T;
		while(flag[0]);
		Critical Section
		flag[1]=F;
		Remainder Section
		}

Peterson’s Solution

- Peterson’s Solution is a classical software based solution to the critical section problem.
- In Peterson’s solution, we have two shared variables:
  - boolean flag[i]: Initialized to FALSE, initially no one is interested in entering the critical section
  - int turn : The process whose turn is to enter the critical section.

Peterson's Solutions

i = 1-j

```
do {

        flag[i] = TRUE ;
        turn = j ;
        while (flag[j]  &&  turn == j) ;

            critial section

        flag[i] = FALSE ;

            remainder section

    }  while (TRUE) ;
```

Two process solution for critical section

~~Second~~ Solution: Boolean Flag[2]=

F	F
0	1

	P_0	P_1

do {
entry section
critical section
exit section
remainder section
} while (TRUE);
	while(1)
	{
	Flag[0]= T;
	Turn=1;
	while(turn==1 &&flag[1]==T);
	Critical Section
	flag[0]=F;
	Remainder Section
	}
		while(1)
		{
		Flag[1]= T;
		Turn=0
		while(turn==0 &&flag[0]==T);
		Critical Section
		flag[1]=F;
		Remainder Section
		}

Semaphore

- Semaphore was proposed by Dijkstra in 1965 to manage concurrent processes by using a simple integer value, known as a semaphore.
- Semaphore is simply an integer variable which is non-negative and shared between threads.
- This variable is used for critical section in the multiprocessing environment.
- It uses two basic function wait(S) and Signal(S).

wait (*S*):
while *S*≤ 0 do *no-op*;
***S*--;**

signal (*S*):
***S*++;**

Critical Section of *n* Processes

- Shared data:

  semaphore mutex; //initially *mutex* = 1

- Process *Pi:*

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (1);
```

wait (*S*):

while $S \leq 0$ **do** ***no-op;***

S--;

signal (*S*):

S++;

Two Types of Semaphores

- **Binary Semaphore** – This is also known as mutex lock. It can have only two values – 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.
- **Counting Semaphore** – Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances
- Can implement a counting semaphore *S* as a binary semaphore.

Example

- There are TWO Processes P1 with Statement S1 and P2 with S2
- **CONDITION :** S2 be executed only after S1. (S1 □S2)

Semaphore synch□0;

P1 {

S1;
Signal(synch);

}

P2 {

Wait(synch);
S2;

}

P1

P

wait(S) {
while(S ≤
S--1
}

Signal(S)
S+
}

Classical Problems of Synchronization

- Producer Consumer with Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Producer Consumer with Bounded-Buffer

- We have a buffer of fixed size.
- A producer can produce an item and can place in the buffer.
- A consumer can pick items and can consume them.
- We need to ensure that when a producer is placing an item in the buffer, then at the same time consumer should not consume any item.
- In this problem, buffer is the critical section.
- To solve this problem, two counting semaphores – **Full and Empty**.
- **"Full"** keeps track of number of items in the buffer at any given time and **"Empty"** keeps track of number of unoccupied slots.

Producer Consumer with Bounded-Buffer

- Shared data : **semaphore full, empty, mutex;**

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

Pr.

```
e an item in nextp
.
mpty);
utex);
.
xtp to buffer
.
(mutex);
(full);
```

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

Readers-Writers Problem

- **semaphore mutex, wrt;**
  Initially **mutex = 1**, **wrt = 1**, **readcount = 0**

Writer

Reader

.

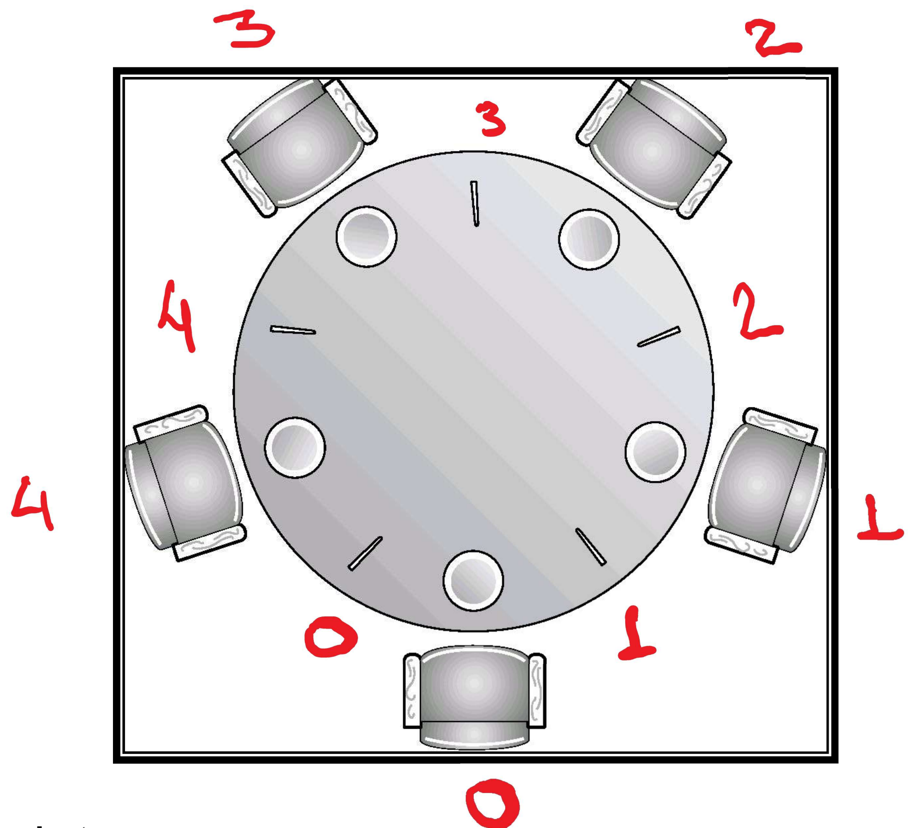
g is performed

.

);

```
wait(mutex);
readcount++;
if (readcount == 1)
        wait(rt);
signal(mutex);
    ...
    reading is performed
        ...
wait(mutex);
readcount--;
if (readcount == 0)
    signal(wrt);
signal(mutex):
```

Dining-Philosophers Problem



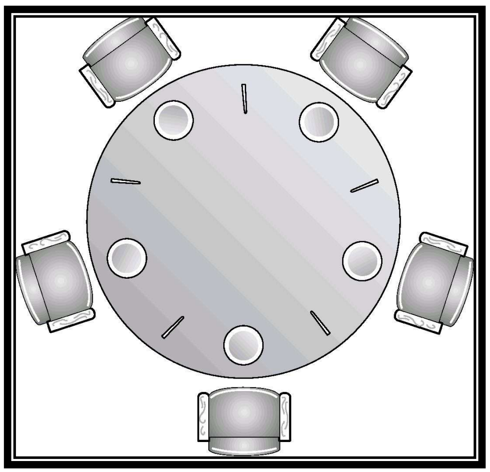
- Shared data

  semaphore chopstick[5];

Initially all values are 1

Dining-Philosophers Problem

- Philosopher *i*:



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


Thank you