Problems with Bounded-Buffer with Counter

* **- Concurrent access to shared data may result in data inconsistency.**

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

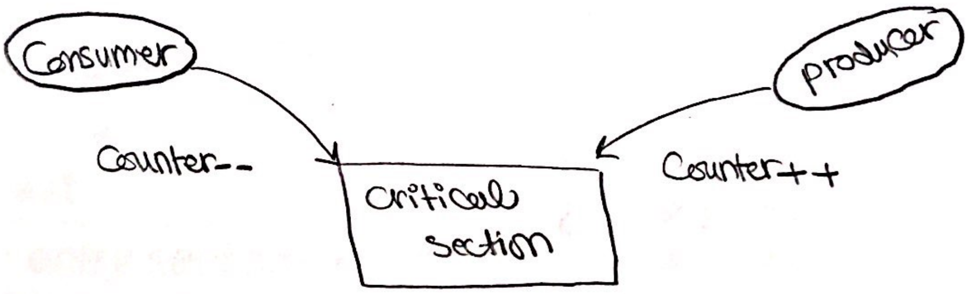
- The statements:

```
o  counter = counter +1;
o  counter = counter -1;
```

must be executed ***atomically***.

to access a shared data Concurrently the shared data must be accessed atomically

Atomically: If one process is modifying counter the other process must wait, that is, as if this is executed sequentially.



The Critical Section Problem

The Problem with Concurrent Execution

- *Concurrent processes (or threads) often need access to shared data and shared resources.* (i.e: Counter in producer_consumer)
- *If there is no controlled access to shared data, it is possible to obtain an inconsistent view of this data.*
- *Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.*

Race Condition: A situation in where several processes access and manipulate data concurrently and the outcome of execution depends on the particular order in which the access takes place.

- *n* processes all competing to use some shared data

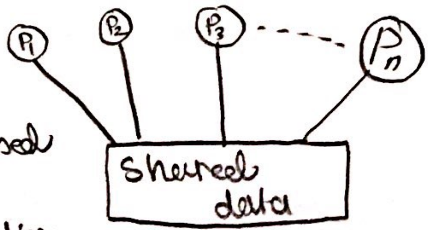
- Each process has a code segment, called *critical section*, in which the shared data is accessed.

- Problem - ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Structure of process P_i

```
repeat
    entry section       <-- شرط دخول ال critical section
        critical section
    exit section        <-- ما العمل عند الخروج من ال critical section
        remainder section
until false;
```

Shared data is Accessed Atomically
↳ one process at a time.



Solution Requirements:

1. **Mutual Exclusion.** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections. "one process at a time"

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.
"If there are no processes in the critical section and process wants to use the critical section it can get it".

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.
"there's a bound for each process on the amount of time it needs to get the critical section"

- Assume that each process executes at a nonzero speed. ← حتى لا تحجز ال critical section طول الوقت، تبقى بقية ال processes waiting
- No assumption concerning relative speed of the n processes.

Solution to Critical Section Problem

Types of Solutions

- ***Software solutions*** Programming
  - Algorithms whose correctness does not rely on any assumptions other than positive processing speed (that may mean no failure).
  - Busy waiting.

- ***Hardware solutions***
  - Rely on some special machine instructions.
    system calls

- ***Operating system solutions*** Ready functions to support the programmer
  - Extending hardware solutions to provide some functions and data structure support to the programmer.

SOFTWARE SOLUTION

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

repeat

entry section

critical section

exit section

remainder section

until *false*;

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

Algorithm 1

- Shared variables: -

```
int turn;  //turn can have a value of either 0 or 1
           //if turn = i, P(i) can enter it's critical
section
```

Process P_i → works based on turns so concurrency isn't used.

```
do
{
  while (turn != i) /*do nothing*/ ;   ← busy waiting

    critical section

  turn = j;

  remainder section
}
while (true)
```

Process P_j

```
do
{
  while(turn != j)
    do nothing ;
  critical section ;
  turn = i ;
  reminder Section
}
```

- **Mutual exclusion** ok

- **Bounded waiting** ok - each only waits at most 1 go.

- **Progress not good** - each has to wait 1 go. P_0 gone into its (long) remainder, P_1 executes critical and finishes its (short) remainder long before P_0, but still has to wait for P_0 to finish and do critical before it can again.

Strict alternation not necessarily good - Buffer is actually pointless, since never used! Only ever use 1 space of it.

Algorithm 2

- *Shared variables*

```
boolean flag[2];
flag[0] = flag[1] = false;
// if flag[i] == true, P(i) ready to enter its critical section
```

Process P i

```
do
 { flag[i]= true;
     while (flag[j]) /*do nothing*/ ;

   critical section

  flag[i]=false;

  remainder section
}
while (true)
```

Process Pj
do
{ Flag[j] =true;
 while (Flag[i])
 do nothing;
}

- Doesn't work at all. Both flags set to true at start. "After you." "No, after you." "I insist." etc.

- Infinite loop

Algorithm 3

Combined shared variables of algorithms 1 and 2.

```
int turn;        //turn can have a value of either 0 or 1
boolean flag[2];         flag[0] = flag[1] = false;
// if flag[i] == true, P(i) ready to enter its critical section
```

process P_i

```
do
  { flag[i]= true;
    turn = j;
    while (flag[j] && turn==j) /*do nothing*/ ;

    critical section

    flag[i]=false;

    remainder section
  }
while (true)
```

Process P_0 → Concurrent ← **Process P_1**

Process P_0

```
do
{ flag[0]= true;
  turn = 1;
  while (flag[1] && turn==1)
      /*do nothing*/ ;

  critical section

  flag[0]=false;

  remainder section

} while (true)
```

Process P_1

```
do
{ flag[1]= true;
  turn = 0;
  while flag[0] && turn==0)
      /*do nothing*/ ;

  critical section

  flag[1]=false;

  remainder section

} while (true)
```

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

- "flag" maintains a truth about the world - that I am at start/end of critical. "turn" is not *actually* whose turn it is. It is just a variable for solving conflict if two processes are ready to go into critical. They all give up their turns so that one will win and go ahead.

- e.g. flags both true, turn=1, turn=0 lasts, P_0 runs into critical, P_1 waits. Eventually P_0 finishes critical, flag =false, P_1 now runs critical, even though turn is still 0. Doesn't matter what turn is, each can run critical so long as other flag is false. Can run at different speeds.

- If other flag is true, then other one is either *in* critical (in which case it will exit, you wait until then) or at start of critical (in which case, you both resolve conflict with turn).

Bakery Algorithm

→ generalization of the solution for (n) processes.

Introduction

This algorithm solves the critical section problem for ***n*** processes in software. The basic idea is that of a bakery; customers take numbers, and whoever has the lowest number gets service next. Here, of course, "service" means entry to the critical section.

Critical section for *n* processes

- Generalization for n processes.
- Each process has an id. Ids are ordered.
- 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...
- Notation <= lexicographical order (ticket #, process id #)
  - (a,b) < (c,d) if a < c or if a = c and b < d
  - max($a_0, \ldots, a_{n-1}$) is a number, k, such that $k >= a_i$ for i = 0, . . . , n - 1
- Shared data

```
1  boolean choosing[n]; //initialise all to false
2  int number[n];   //initialise all to 0

3  do
4  { choosing[i] = true;
5   number[i] = max(number[0], number[1], ...,number[n-1]) + 1;
6   choosing[i] = false;
7   for(int j = 0; j < n; j++)
8     { while (choosing[j]== true)
9          /*do nothing*/
10        while ((number[j]!=0) && (number[j],j)< (number[i],i))
11           /*do nothing*/
12   }

13        critical section

14 number[i] = 0;

15        remainder section

   } while (true)
```

Comments

lines 1-2: Here, *choosing*[*i*] is true if P_i is choosing a number. The number that P_i will use to enter the critical section is in *number*[*i*]; it is 0 if P_i is not trying to enter its critical section.

lines 4-6: These three lines first indicate that the process is choosing a number (line 4), then try to assign a unique number to the process P_i (line 5); however, that does not always happen. Afterwards, P_i indicates it is done (line 6).

lines 7-12: Now we select which process goes into the critical section. Pi waits until it has the lowest number of all the processes waiting to enter the critical section. If two processes have the same number, the one with the smaller name - the value of the subscript - goes in; the notation "(a,b) < (c,d)" means true if a < c or if both a = c and b < d (lines 9-10). Note that if a process is not trying to enter the critical section, its number is 0. Also, if a process is choosing a number when P_i tries to look at it, P_i waits until it has done so before looking (line 8).

line 14: Now P_i is no longer interested in entering its critical section, so it sets *number*[*i*] to 0.

Drawbacks of Software Solutions

- **Complicated to program**
- **Busy waiting (wasted CPU cycles**
- **It would be more efficient to *block* processes that are waiting (just as if they had requested I/O).**

HARDWARE SOLUTION

Hardware Solution **Disable Interrupts**

On a uni-processor, you can get mutual exclusion by locking out interrupts. Observations:

- You can only afford to do this for a little while, so you don't lose any interrupts (of course in general you don't want to protect expensive things with spin locks).
- Nothing else works if you're sharing memory with a device you sure can't use a spin lock! (DEADLOCK).
- Correct solution for a uni-processor machine, but this doesn't work on multiprocessors, the solution is not correct.
- During critical section multiprogramming is not utilized - performance penalty.

```
Repeat
  disable interrupts
  critical section
  enable interrupts
  remainder section
Forever
```

Hardware Solution **Test and Set** → must be executed Atomically

Use better (more powerful) atomic operations:

- **Test and modify the content of a word atomically.**

```
boolean Test_and_Set(Boolean & target)
 {boolean test = target;
   target = true;
  return test;
 }
```

→ call by reference to return the value of target.

- Shared data: `boolean lock = false;`

Process P_i

```
do
{ while (Test-and-Set(lock))
      /*do nothing*/ ;
      critical section
   lock = false;
      remainder section
}while (true)
```

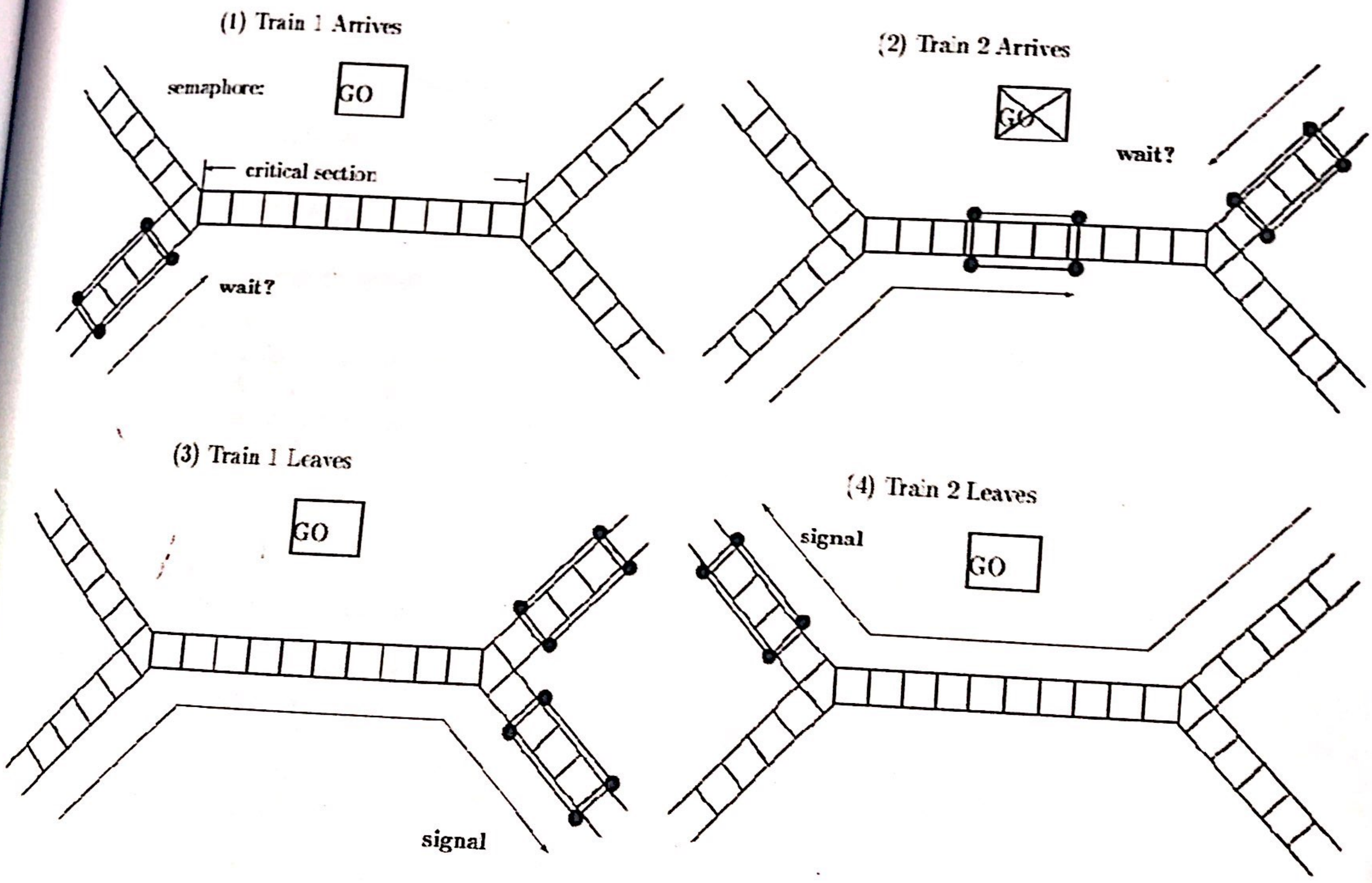
OPERATING SYSTEM SOLUTION

Semaphores

→ checks if the critical section is Empty or not "close"

Semaphore: wait and signal

↳ "opens the critical section"



Semaphore S - integer variable

↳ int S=1; - can only be accessed via two indivisible (**atomic**) operations

```
wait(s)      : while (S<=0) { /*do nothing*/ };
               S = S-1;

signal(S)    : S = S + 1;
```

mutual exclusion

```
mutex : semaphore =1;
Repeat
  wait( mutex );
      critical section
  signal( mutex );
      remainder section
Forever
```

Note :- The wait and signal instruction must be executed atomicaly.

Problem (semaphore): busy waiting.

Semaphore Implementation

- Define a semaphore as a record/structure

```
struct semaphore
 { int value;
   List *L;     //a list of processes
 }
```

↳ Pending

- Assume two simple operations:
  - ***block*** suspends the process that invokes it.
  - ***wakeup(P)*** resumes the execution of a blocked process *P*.
- Semaphore operations now defined as

```
wait(S)
{ S.value = S.value -1;
  if (S.value <0)
    { add this process to S.L;
      block;
    }
}

signal(S)
{ S.value = S.value + 1;
  if (S.value <= 0)
    { remove a process P from S.L;
      wakeup(P);
    }
}
```

Classical Problems of Synchronization

- Bounded Buffer Problem
- Readers and Writers Problem
- Dining Philosophers Problem

Bounded Buffer Problem

- Shared data

```
char item;                 // could be any data type
char buffer[n];
semaphore full = 0;        // counting semaphore
semaphore empty = n;       // counting semaphore
semaphore mutex = 1;       // binary semaphore
char nextp, nextc;
```

↓mutual execlusion.

- Producer process

```
do
{ produce an item in nextp
  wait (empty); → checks if Buffers is full
  wait (mutex); → counter
  add nextp to buffer
  signal (mutex);
  signal (full);
}
while (true)
```

- Consumer process

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

Readers-Writers Problem

- Shared data

```
semaphore mutex = 1;
semaphore wrt = 1;
int readcount = 0;
```

- Writer process

```
wait(wrt);
  writing is performed
signal (wrt);
```

- Reader process

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

Dining Philosopher Problem

- Shared data

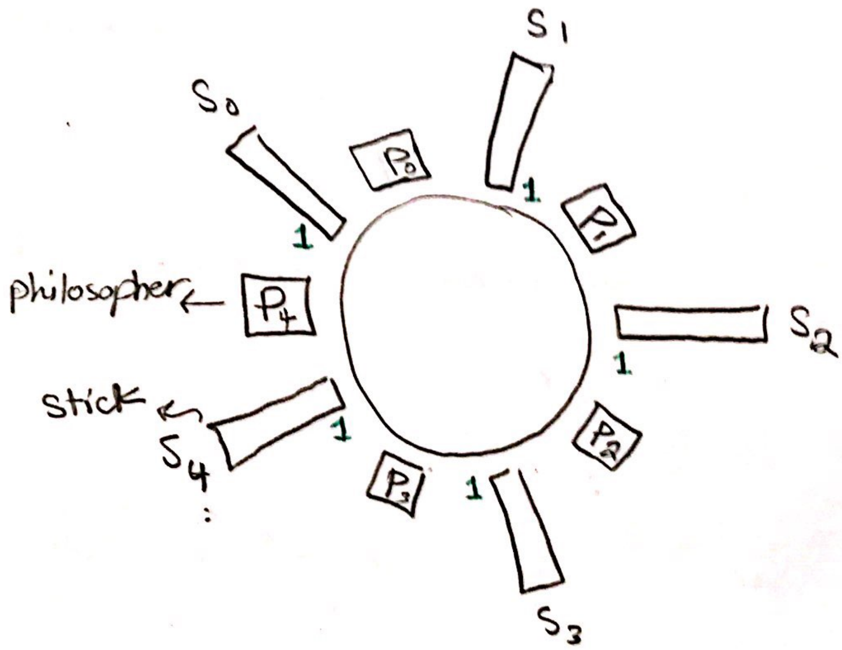
```
semaphore chopstick[5];
chopstick[] = 1;
```

↳ available

- Philosopher *i*:

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

1: available



⚠ Problems:

(1) Deadlock.

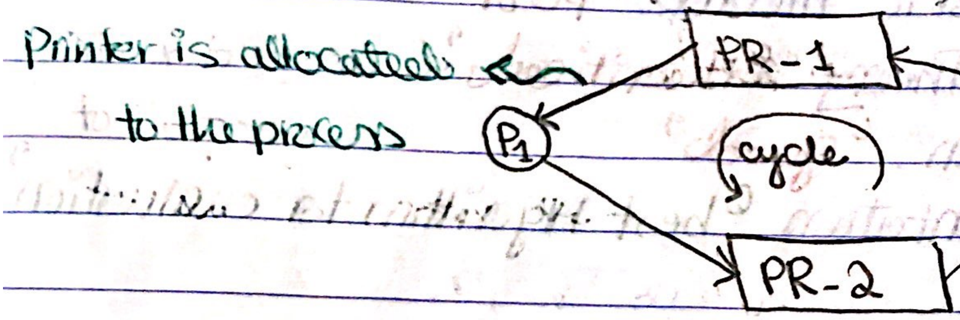
(2) Starvation.

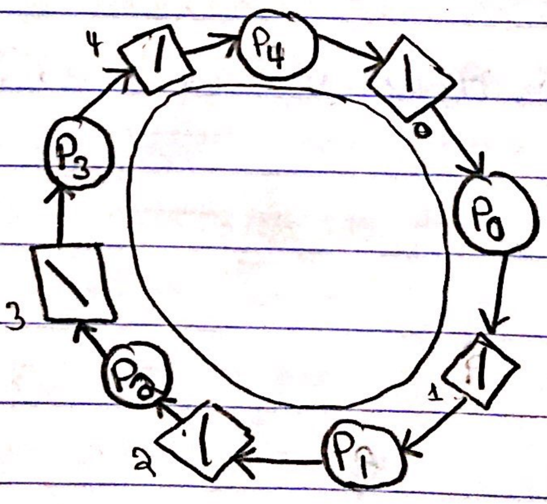
Chapter #7

DeadLocks.

Definition:

Two processes are deadlocked, if every process is holding a resource & waiting for the other process to release its resource.

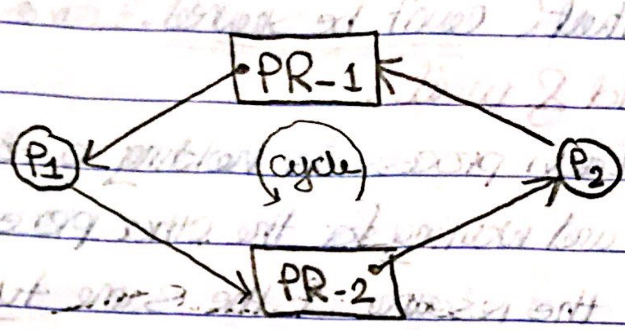




Monday
April 16. 2018

Deadlock

Deadlock: A set of waiting (blocked processes), each process is holding a resource & waiting for other processes to release its resources



System model:

HD tapes printers

- we have the resource types $R_0, R_1 ----, R_{n-1}$
- we have W_j instances of each resource type $\overset{5}{W_0}, \overset{2}{W_1} ---- , \overset{3}{W_n}$
- Each process use the resources in the following order:
  - * Requests the resources.
  - * uses the resources.
  - * Releases the resources.

Deadlock handling:

The OS handles the deadlock in one of two methods;

(1) Allow the system to enter a deadlock and then recovers from it. "UNIX"

(2) The OS prevents the system from entering a deadlock state.

[1] Necessary Conditions:

4 necessary conditions must hold simulatanously in order for a deadlock to occur.

(1) Mutual Exclusion:

The resource type must be used exclusively that's can't be shared. "for more than one process at a time"

(2) Hold & wait:

Each process is holding a resource type and waiting for the other process to release the resource of the same type.

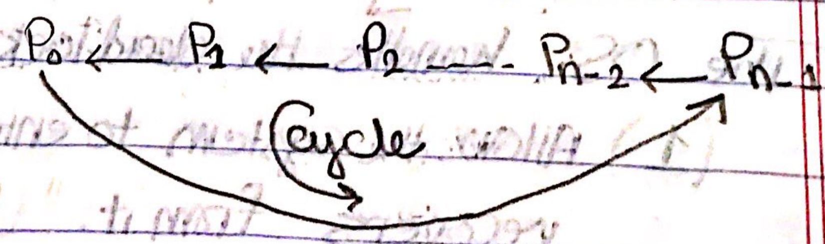
(3) No Pre-emption:

Can't remove any of the resources.

(4) Circular wait: "cycle"

There exists a sequence of processes $\langle P_0, P_1, P_2, \ldots, P_{n-1} \rangle$ such that:

- P_0 is waiting for P_1 to release its resources.
- P_1 is waiting for P_2 to release its resources.
- P_{n-2} is waiting for P_{n-1} to release its resources.



Resource Allocation Graph:

Generally, a graph $G = (V, E)$

V = Set of vertices.

E = Set of edges.

→ In the deadlock case:

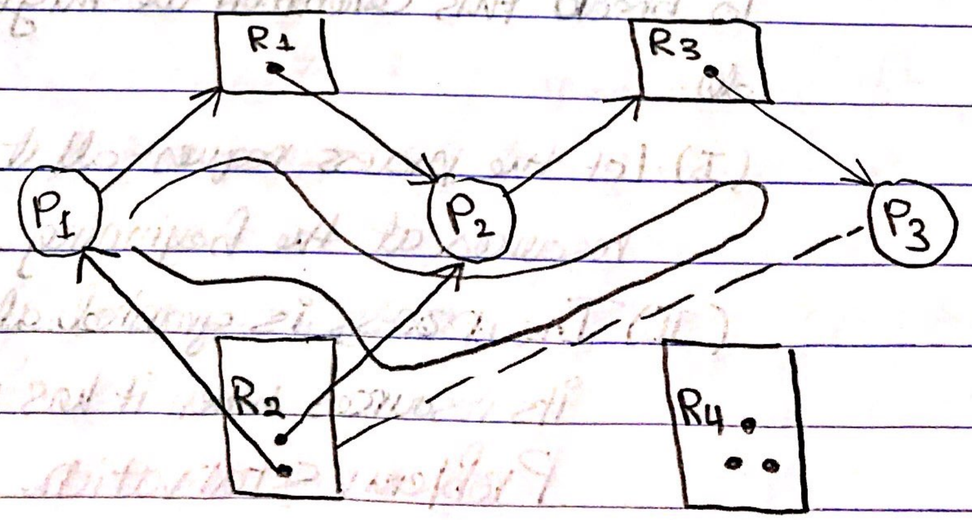
$V = \begin{cases} P: \text{Process.} \\ R: \text{resource type.} \end{cases}$

$E = \begin{cases} (P_i, P_j): \text{Process } P_i \text{ is requesting one instance of resource type } R_j \\ (R_j, P_i): \text{one instance of resource type } R_j \text{ is allocated or given to process } P_i \end{cases}$

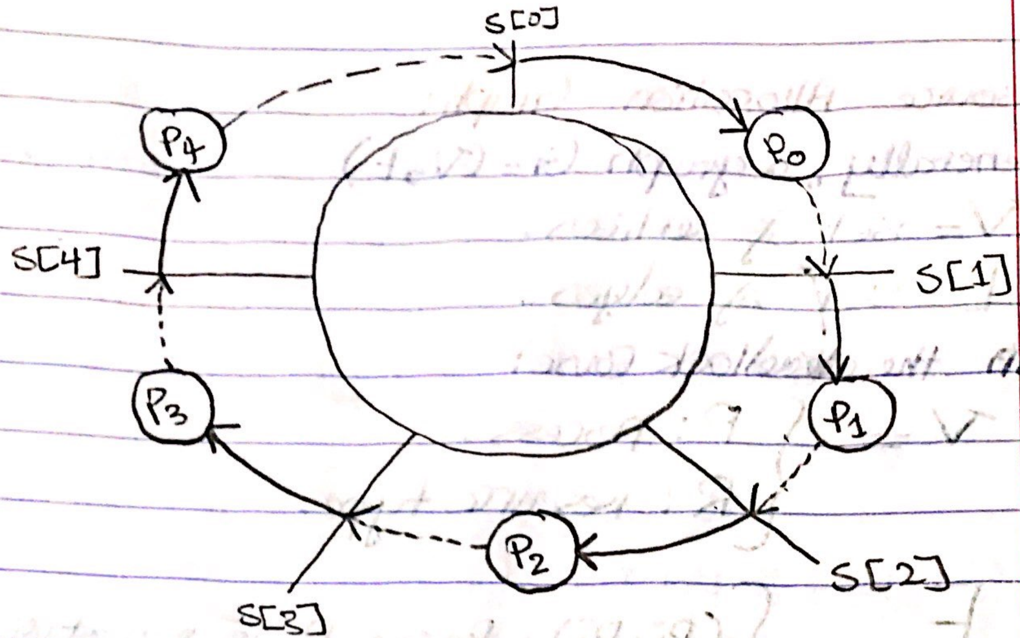
Example: $P = \{P_1, P_2, P_3\}$

$R = \{R_1(1), R_2(2), R_3(1), R_4(3)\}$

$E = \{(P_1, R_1), (P_2, R_3), (R_1, P_2), (R_2, P_2), (R_2, P_1), (R_3, P_3)\}$



→ Assume P_3 demands an instance of R_3.



4) Deadlock prevention:-

To make sure at least one of the four necessary conditions don't hold:

1_ **mutual exclusion.**

By default, some resources are mutually exclusive, and we can't do anything about it, such as printers.

2_ **Hold & wait**

To break this condition we might do:

(I) Let the process request all its resources at the Beginning.

(II) The process is granted all its resources when it has none.

Problem: Starvation

3- Non pre-emption:

If a process requests a resource which is not available, it must release the resources it has.

Problem: low system utilization. "poor performance", in addition to starvation.

4- **Circular wait:**

1. Card reader.
2. Hard disk.
3. Tape.
4. Printer.

Process P_i;

```
Semaphor int s[i] = {1,1,1,1}
Repeat {
    Think ;
    wait(si); ——> wait(S min(i, ((i+1)%5)))
    wait(s((i+1)%5) ); ——> wait(S max(i,((i+1)%5)))
    Eat ;
    Signal (s((i+1)%5) ) ;
    Signal (si);
}until False.
```

4) Deadlock Avoidance:

Definition: A system is in a safe state, if there exists a sequence of processes $<P_0, P_1, P_2, ---, P_{n-1}>$ such that:
P_0 can take all available resources, execute & finish.
P_1 can take all available resources, & resources released by P_0, execute & Finish.
P_2 can take all available resources, & resources released By P_0, P_0, execute & Finish.
⋮
P_{n-1} can take all available resources and resources released by $P_0, P_1, P_2 -- P_{n-2}$, execute & Finish.

Definition: If there's such a sequence, then the system is safe, NO deadlock.

example: A system with 12 tape units and 3 processes, A snapshot at the system looks like:

Process	max needs	allocated	current needs
P_0	10	5	5
P_1	4	2	2
P_2	9	~~2~~ 3	~~7~~ 6

the available at this time: 3 → 12 - allocated(9)

- Is the system safe.

available: ~~3~~ ~~1~~ (5)   <P1, P0, P2>

~~5~~ ~~0~~ (10)

~~10~~ ~~3~~ (12)   Safe (✓)

- Assume, process 2 demanded extra tape & the OS granted the request. is the system safe?

the available at this time: 2

available: ~~2~~ 4   <P1, ??

(X) No safe sequence - deadlock

Process	Allocation A	B	C	max A	B	C	current needs A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	3 ~~0~~	3 ~~0~~	2	4	3	3	1 ~~4~~	0 ~~3~~	1

- Is the System Safe ?! Is there a safe sequence?!

Available

~~3~~ 3 2

~~5~~ ~~3~~ ~~2~~

~~7~~ ~~4~~ ~~3~~

~~7~~ ~~5~~ ~~3~~

~~10~~ ~~5~~ ~~5~~

10	5	7

A	B	C
10	5	7

total

<P1, P3, P0, P2, P4>