scheme as in the clock algorithm, but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the 1st page encountered in the lowest non-empty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced.

✔ The major difference b/w this alg. & simpler clock alg. is that here we give preference to those pages that have been modified to reduce the no. of I/O required.

5) Counting Based page Replacement: We can keep a counter of the no. of references that have been made to each page & develop the following 2 schemes.

i) Least Frequently used (LFU) page-Replacement Algorithm: Requires that the page with smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is heavily used during the initial phase of a process but then never used again. Since it was used heavily, it has a large count & remains in mem. even though it is no longer needed. 1 soln is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average use count.

ii) Most Frequently Used (MFU) Page-replacement Algorithm: It is based on the argument that the page with smallest count has probably just brought in & has yet to be used.

Hence replace the page with the largest count.

6) Page-Buffering Algorithms: In addition to specific page replacement algs. other improvements can be done.

i) System commonly keeps a pool of free frames. When a page fault occurs, a victim frame is chosen as before. however, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

ii) We can also maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected & is written to the disk & its modify bit is reset. This scheme increases the probability that a page will be clean when it is selected for replacement & will not need to be written out.

Thrashing: If the process does not have enough frames, it needs to support pages in active use, it will quickly give a page fault. At this point it must replace some page. However since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it faults again & again & again replacing the pages that it must bring back in immediately.

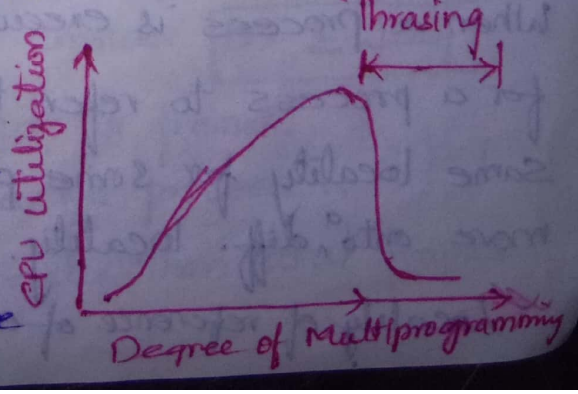
This high paging activity is called Thrashing. A

process is thrashing if it is spending more time in paging than executing.

causes of Thrashing: The OS monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process into the system.

A global page replacement alg is generally used, it replaces pages without regard to the process to which they belong. Now suppose that a process enters into new phase in its execution & needs more frames. It starts faulting & taking away frames from other processes. These processes needs those pages & so they also fault, taking frames from other processes. These faulting processes must use the paging device, to swap pages in & out. As they queue up for paging device, the ready queue empties. As processes waits for paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization & ↑s the degree of multiprogramming as a result. The new processes tries to get started by taking frames from running processes, causing more page faults & longer queue for paging device. As a result, CPU utilization drops even further & CPU scheduler tries to ↑ the degree of multiprogramming even more. The page fault rate ↑s tremendously. No working is done because the processes are spending all their time paging.



from fig: As the degree of multiprogramming ↑s, CPU utilization also ↑s, although more slowly, until a max. is reached. If the degree of multiprogramming is ↑ed even further, thrashing sets in & CPU utilization drops sharply. At this point to ↑ CPU utilization & stop thrashing, we must decrease the degree of multiprogramming.

- We can limit the effect of thrashing by using a local replacement alg. (or priority replacement alg.). With local replacement, if one process starts thrashing, it cannot steal frames from another process & cause the latter to thrash as well.
- To prevent thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it needs?

Soln: ① Working Set Model: This approach defines the locality model.

- A locality model states that as process executes, it moves from locality to locality. A locality is a set of pages that are actively used together.

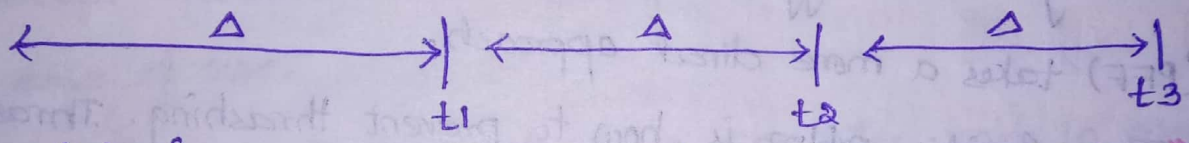
Locality of reference: Prg & data references within a process tend to cluster, which forms diff. localities. When a process is executed, there is more probability for a process to refer the instructions/data from the same locality for some period of time. Then it will move onto a diff. locality.

- Locality of reference of a process refers to its most recent/active pages.

The working set model is based on the assumption of locality. This model uses a parameter Δ (delta) to define the working set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is called as the working set. If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set. Thus, the working set is an approximation of the program's locality.

Eg: Page reference table

...2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 4 4 1 3 2 3 4 4 4 ...



ws(t1) = {1, 2, 5, 6, 7} ws(t2) = {3, 4} ws(t3) = {1, 3, 4}

From the fig. the working set at time t1 = {1, 2, 5, 6, 7} at t2 = {3, 4}, t3 = {1, 3, 4}. Thus at diff time intervals the required frame are diff. Hence based on the working set, the processes are allocated with frames.

The accuracy of working set depends on the selection of Δ. If Δ is too small, it will not encompass the entire locality. If Δ is too large, it may overlap with other localities.

If we compute the working set size, WSS_i, for each process in the sys, we can consider $D = \Sigma WSS_i$

where D = is the total demand for frames.

WSS_i is working set size for process i.

If the total demand is greater than total frames ($D > m$), then thrashing occurs.

✓ Once Δ has been selected, the OS monitors working set of each process & allocates those many frames. If there are extra frames another process can be initiated. If sum of the working set size increasing, exceeding the total no. of available frames, OS selects a process to suspend. Thus it prevents thrashing while keeping the degree of multiprogramming as high as possible.

② Page-fault Frequency: The working-set model is successful & knowledge of the working set can be useful for prepaging, but it seems a clumsy way to control thrasing. A strategy that uses the page-fault frequency (PFF) takes a more direct approach.

✓ The specific problm is how to prevent thrashing. Thrasing has high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames. Conversely, if the page-fault we can establish upper & lower bounds on the desired page-fault rate. If the actual page-fault rate exceeds the upper limit, we allocate the process another frame; if the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure & control the page fault rate to prevent thrashing.

File System Management

* Since main mem. is usually too small to accommodate all the data & prgs permanently, the computer sys must provide secondary storage to back up Main mem..

✓ Modern computer systems use disks as the primary on-line storage medium for inf (both prg & data). the file sys. provides the mechanism for on-line storage of & access to data & prgs residing on the disks.

✓ A file is a collection of related inf defined by its creator. The files are mapped by os onto physical devices. Files are normally organized into directories for ease of use.

✓ The devices that attach to a computer vary in many aspects. • Some devices transfer a character or a block of characters at a time

• Some transfer data synchronously, others asynchronously

• Some can be accessed only sequentially, others randomly.

• Some are dedicated, some are shared.

• They can be read-only or read-write.

• They vary gically in speed.

File Concept: Computers can store inf. on various storage media, such as magnetic disks, magnetic tapes & optical disks. So that the computer sys will be convenient to use, the os provides a uniform logic view of inf. storage.

The os abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped by os onto physical devices. These storage devices are usually nonvolatile, so the content are persistent through power failures & sys reboots

* A file is a named collection of related inf that is recorded on secondary storage.
* Files represent prgs & data.
* Data files may be numeric, alphabetic, alphanumeric or binary.
* The inf. in a file is defined by its creator.
* Many diff. types of inf may be stored in a file - Source prg, obj prg, executable prgs, numeric data, text, payroll records, graphic images, sound recordings, & so on.
* A file has a certain defined structure, which depends on its type.
* A text file is a sequence of chars. organized into lines (& possibly pages).
* A source file is a sequence of ~~char~~ subroutines & functions, each of which is further orgainzed as declarations followed by executable statements.
* An object file is a sequence of bytes organized into blocks understandable by the system's linker.
* An executable file is a series of code sections that the loader can bring into mem. & execute.

File Attributes: A file is named, for humans convenience, & refer them by its name. A name is a string of chars, such as example.c (usually). So systems differ-entiate blw upper case & lower case whereas other systems do not. When a file is named, it becomes independent of the process, the user, & even the sys that created it.

✓A file's attributes vary from one OS sys to another but typically consist of these:

1) Name: The symbolic file name is the only inf kept in human readable form.

2) Identifier: This unique tag, usually a no., identifies the file within the file sys; it is the non-human-read-able name for the files.

3) Type: The inf is needed for sys's that support diff. types of files.

4) Location: This inf. is a pointer to a device & to the location of the files on that device.

5) Size: The current size of the file (in bytes, words, or blocks) & possibly the max. allowed size are included in this attributes.

6) Protection: Access-control inf. determines who can do reading, writing, executing, & so on.

7) Time, date & User identification: This inf may be kept

for creation, last modification, & last use. These data can be useful for protection, security, & usage monitoring. The inf. abt all files is kept in the directory struct, which also resides on secondary storage. The directory consists of file's name & its unique identifier.

* File Operations: A file is an abstract data type. To define file, we need to consider that operations that can be performed on files. The OS can provide Sys calls to create, write, read, reposition, delete & truncate files.

1) Creating a file: Two steps are necessary to create a file. ① Space in the file Sys must be found
② An entry for the new file must be made in the directory.

2) Writing a file: To write a file, we make a Sys call specifying both the name of the file & the inf to be written to the file. On giving name of the file, the Sys searches the directory to find the file's location. Then, the Sys must keep a write ptr to the location in the file where the next write is to take place. And update the write ptr as write occurs.

3) Reading a file: To read a file, we use Sys call that specifies the name of the file & where the nxt block of the file shld be put (in mem). Again, the directory is searched for the associated entry, & the Sys needs to keep a read ptr to the loc. in the file where the nxt read is

to take place. Once the read has taken place, the read ptr is updated.

Becoz a process is usually either reading from (or) writing to a file, the current operation location can be kept as a per-process current-file-position ptr. Both the read & write operations use this same ptr, saving space & reducing sys complexity.

4) Repositioning with-in a file: The directory is searched for the appropriate entry & the current-file-position ptr is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file seek

5) Deleting a file:

6. Synchronization

* A cooperating process is 1 that can affect or be affected by other processes executing in the system. cooperating processes can either directly share a logical address space (i.e., both code & data) or be allowed to share data may result in data inconsistancy. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that the data consistancy is maintained.

Critical section problem: Consider a system consisting of n processes $\{P_0, P_1, P_2, \ldots, P_{n-1}\}$. Each process has a segment of code, called as critical section, in which the process may be changing common variables, updating a table, writing to a file, & so on...

✓ The important feature of the system is that, when 1 process is executing its critical section, no other process is to be allowed to execute in their critical section. i.e., No 2 processes are executing in their critical section at the same time.

✓ The critical section prblm is to design a protocal that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of the code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

The general structure of a typical process P_i is shown below:

```
do
{
    Entry Section
        critical section
    Exit Section
        Remainder Section
}
```

* A soln to critical section problem must satisfy the following three requirements.

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 & some processes wish to enter their critical sections, then only those processes which are not executing in their remainder sections can participate in deciding which will enter its critical section next, & this selection cannot be postposned indefinitely.

3) Bounded Waiting: There exists a bound, or limit, on the no. of times that other processes are allowed to enter their critical section after a process has made a request to enter its critical section & before that request is granted.

Race condition (Concurrency): Occurs when several processes try to access & manipulate data concurrently.

✓ The order of execution of instructions influences the result produced.

* Two general approaches are used to handle critical sections in OS: (1) Preemptive Kernels, (2) Non-Preemptive Kernels.

✓ A preemptive kernel allows a process to be preempted while it is running in kernel mode. Non-preemptive kernel does not allow a process running in kernel mode to be preempted.

✓ Non-preemptive kernel is essentially free from race condition on kernel data structures, as only 1 process is executing/active in the kernel at a time. We cannot say the same about preemptive kernels.

✓ Preemptive kernel is more suitable for real-time programming (adv. over Non-preemptive kernel).

* Two process solution / Strict Alternation Solution:

consider 2 processes, a boolean variable turn which can be either have 0 or 1 value.

P0

```
while (1)
{
    while (turn != 0);
        critical section
    turn = 1;
        Remainder section
}
```

P1

```
while (1)
{
    while (turn != 1);
        critical section
    turn = 0;
        Remainder section
}
```

* Let's consider initial value of turn = 0.

✓ Note that after while statement there is a semicolon

⇒ There is no body for the while loop.

⇒ If the condition in the while statement is true, then again it will keep on checking the condition infinitely until the condition becomes false. When the condition is

false then the control will come out of while loop & then process goes to its critical section.

Now, turn=0 & consider P_0 process.

✓ since (turn != 0) is false, then P_0 enters its critical section & P_0 can complete executing its instructions in the critical section & make the value of turn to 1.

✓ while P_0 is executing critical section P_1 can also preempt the process P_0, & If P_1 try to execute its critical section, (turn != 1) is true hence again it will check condition until it becomes false. Hence here although P_1 can preempt P_0 while its executing in its critical section but P_1 cannot enter into its critical section.

✓ P_0 after executing cs successfully turn=1, Now P_0, if it wants to enter its critical section again,

(turn != 0) ⇒ True ⇒ Not possible.

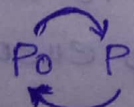
✓ P_1 : (turn != 1) ⇒ False ⇒ P_1 can enter its critical section

Again in the way,

if turn = 0 ⇒ P_0 executes
if turn = 1 ⇒ P_1 executes
} This will ensure mutual exclusion.

② progress = ?

In this alg, we are not at all considering whether some process is interested in executing its cs & hence strict alternation is followed here i.e., intially P_0 is executed then P_1, then P_0, then P_1 & so on



If P_0 is interested & P_1 is not interested in going to its cs. It is not possible for process P_0 to execute it

cs more than once. Although P1 is not interested, alg makes it compulsory for P1 to go to its cs.

⇓

progress is not ensured

⇒ Not a correct soln (Suffering from Strict Alternation).

* Solution II: In the previous alg we did not consider the interest of the process, whether its wants to execute its cs or not.

For that lets consider flag, which stores whether the process is interested or not.

P0

```
while (1)
{
    flag[0] = T;
    while (flag[1]);
        CS
    flag[0] = F;
}
```

P1

```
while(1)
{
    flag[1] = T;
    while (flag[0]);
        CS
    flag[1] = F;
}
```

If a process wants to go into critical section it will set flag value to true.

eg: flag[0] = T ⇒ P0 wants to enter
flag[1] = F ⇒ P1 not interested
} flag (Initially) P0 P1 | F | F |

Now, P0 code: flag[0] = T ⇒ P0 is interested
& flag[1] = F ⇒ Condition (while) becomes false
hence P0 will enter cs.

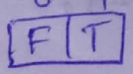
Checking Mutual Exclusion: While P0 executing in cs, P1 may preempt P0.

P1 code: flag[1] = T; (∴ flag 0 1 | T | T |)

while(flag[0]);
⇓ true
Infinite & P1 will not enter.

After P0 completing its cs, flag[0] = F

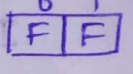
Now we can execute P1 code

while(flag[0]); (∵ flag 

)
⇓ false
P1 can enter its cs.

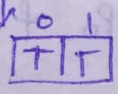
∴ Mutual Exclusion Satisfied. //

② Progress = ?

Lets consider P1 is not interested ⇒ P0 shld execute any no. of times. (∵ flag 

)
⇓
true (possible)

consider the following case:

P0 code: After executing flag[0]=T & before checking condition, if the cs occurs
⇓
P1 code: After executing flag[1]=T & before checking condition, if context switch occurs.
⇓
flag 

⇒ Both processes wants to go to its critical section but none of them are able to go. ⇒ Deadlock.
⇓
progress is not satisfied.

* Peterson's ~~algorSolution~~: In the first case, we were not considering the interest of the process & strict alternation prblm occured & in the 2nd case we have taken flag array to consider the interest of process but we are ended up with deadlock.

In this Soln, we will consider both turn variable & flag array variable.

P0

```
while(1)
{
    flag[0] = T;
    turn = 1;
    while (turn==1 && flag[1]==T);
        CS
    flag[0] = F;
}
```

P1

```
while (1)
{
    flag[1] = T;
    turn = 0;
    while (turn==0 && flag[0]==T);
        CS
    flag[1] = F;
}
```

Initially turn = 0/1

flag [F | F] (0, 1)

a) **Mutual Exclusion:** flag[0] = T ⇒ P0 interested

& turn = 1

while (turn==1 && flag[1] == T);

T && F

F ⇒ P0 will enter CS.

If P1 wants to preempt P0 & wants to enter CS

flag[1] = T & turn = 0

while (turn==0 && flag[0] == T);

T && T

T ⇒ infinite loop

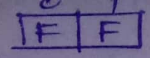
P1 can't enter CS.

After executing P0 in CS, flag[0] = F

P1 can enter CS

⇒ ∴ ME is satisfied.

b) Progress: turn = 0/1

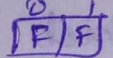
flag 

can Po go for multiple iterations = ? ⇒ Possible

strict alternation ⇒ ✗ not there

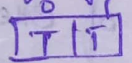
Does deadlock exists = ?

turn = 0/1

flag 

If Po is executing & context switch occurs after turn = 1 ⇒ flag[0] = T & turn = 1

& P1 is executing & context switch occurs after turn = 0 ⇒ flag[1] = T & turn = 0.

flag 

, turn = 0

Since (turn == 1) is false (Po code)

⇓

Po will enter cs.

At any point of time turn will have either 0 or 1, which will help 1 process to enter into its cs.

⇒ ∴ Progress satisfied. //

c) Bounded wait: Whenever some process goes out of cs, the other process if it is waiting to go into critical section ⇒ Possible.

Max. a process will wait for 1 turn & hence bounded wait is satisfied.

Drawback: Holds for 2 processes only.

* Semaphores: It gives a 'n' process soln.

"A semaphore S is an integer variable that, apart from initialization, is accessed only through 2 standard atomic operations: wait() & signal().

Initialization : int S=1;

wait(): is an atomic operation which test the condition & decrement the value of S by 1.

signal(): is an atomic operation which increments the value of S by 1.

Semaphores will solve : (Applications)
1) CS problem
2) Order of execution among process
3) Resource mngt.

int S=1;
Consider n processes P1, P2, P3, ... Pn

```
do
{
   wait(S);
   [CS]
   signal(S);
   [RS]
} while(True)
```

```
wait(S)
{
   while(S<=0);
   S=S-1;
}
```

```
signal(S)
{
   S=S+1;
}
```

Lets consider that initially process P1 is executing, it will execute wait() fun. & since the cond. while(S<=0) is false it will come out of loop & decrements the value of S (i.e., S=0 now). & P1 enters in its CS. Now if P2 wants to enter into CS, it has to execute wait() operation but the condition while(S<=0) becomes true & hence P2 will have to wait until the condition becomes

false. i.e., it is unable to enter into cs. similarly until P1 comes out of cs & increments the s value by 1, NO other process can enter into its cs.

⇓

① Mutual Exclusion is guaranteed

② Progress = ?

Here, only those processes that are interested in entering cs only will execute/check the wait() operation & they may enter. We are not restricting if the processes are not interested.

⇓

Progress is guaranteed

③ Bounded Wait ?

Whenever a process comes out of cs, any other process/some process can decrement the value of s and can enter the cs. So hence we can't determine a logical bound time for any process because the entry is basically determined by the OS, randomly (i.e., whichever the process is having control over the CPU will enter the cs).

⇓

Bounded Wait is not guranteed.

ME & Progress are the 2 mandatory criterias which are satisfied by semaphores. But it will not satisfy bounded wait, which is not essential. Hence we can call this as a soln to cs.

There are 2 types of Semaphores:

1) Binary Semaphores: can range only between 0 & 1
2) Counting Semaphores: can range over unrestricted domain

* On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual Exclusion.

```
do
{
    wait (mutex);
        [CS]
    signal (mutex);
        [RS]
} while (True);
```

* On binary semaphores mutex is initialized to 1.

* Classic Problems of Synchronization:

1) Producer-Consumer Problem / Bounded-Buffer Problem:

We assume that the pool consists of n buffers, each capable of holding 1 item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool & is initialized to the value 1. The empty & full semaphores count the no. of empty & full buffers. The semaphore empty is initialized to n & full is initialized to 0.

```
mutex = 1;
empty = n;
full = 0;
```

Producer Process

```
do
{
    ...
    // produce an item
    ...
    wait (empty);
    wait (mutex);
    ...
    // add item to buffer
```

Consumer Process

```
do
{
    wait (full);
    wait (mutex);
    ...
    // Remove an item from buff
    ...
    signal (mutex);
```

```
...
signal(mutex);
signal(full);
}while(true);
```

```
signal(empty);
...
//consume the item
...
}while(true);
```

Producer: Produces the items & adds them to the buffer. Producer will decrement empty value & checks mutex condition, & produces the item & increments the value of full & then releases mutex (decrements by 1).

✓ Producer has to check the overflow condition: He can produce the items only if atleast 1 buffer is empty. If empty = 0 ⇒ condition becomes true & waits until empty is incremented by the consumer.

Consumer: Consumes the items & deletes them from the buffer. consumer will decrement the value of full & checks mutex condition & enters cs to consume the item. After consuming, consumer will increment mutex as well as empty value.

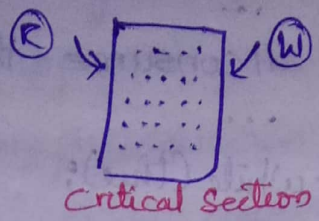
Consumer has to check the underflow condition. He can consume items only if atleast 1 buffer/item is available If full = 0 ⇒ condition becomes true & hence consumer waits until producer increments the value of full to atleast 1.

2) **Reader - Writer Problems:** Lets consider a database or file & 2 types of processes → Reader process & writer process.

Reader process: Reads the file or database

Writer process: can do both - Reading & Writing.

Here the file or db is the critical section.



✓ For Readers its actually not a cs because more than one reader can read at the same time & hence more than 1 reader can enter into cs. But it is cs for readers, only when writer wants to enter into cs. Even if 1 reader is present in the cs, no writer shld be allowed to enter the cs.

✓ For writers its a cs in both the ways. When a writer is present in the cs, we shld not allow either a reader or another writer to enter into cs.

✓ Lets solve it using semaphores: Consider 2 Semphores i.e., wrt & mutex.

For Writer

```
while(1)
{
  wait(wrt);
  // write operation
  signal(wrt);
}
```

Initialization:

```
wrt = 1
mutex = 1
readcount = 0
```

For Reader

```
while(1)
{
  wait(mutex);
    readcount++;
    if(readcount==1)
    wait(wrt)
  signal(mutex);
  // Read operation
  wait(mutex);
    readcount--;
    if(readcount==0)
    signal(wrt);
  signal(mutex);
```

Entry

Exit

* Writer: We have used semaphore wrt = 1, whenever writer wants to perform write operation, wait(wrt) is performed & value becomes 0. until writer increments the value

of wrt/complete the write operation, no other Reader/Writer process is allowed to enter cs.

✓Very easy.

For Reader: If reader is present in the cs, & any other process wants to enter into cs, we shld check whether its a reader process or a writer process. If it is a reader process, we can allow it to enter cs but if it is a writer process, we shld not allow.

✓We shld also note that whenever the 1st reader enters into cs, it is the responsibility of the first reader to stop writer processes to enter into cs. In the similar fashion, the last reader process, which is exiting from the cs shld allow the writer to enter into cs.

readcount = 0 (initially)

Mutex is used to get synchronization blw reader processes i.e., only 1 reader process shld increament the readcount or decrement the readcount at any point of time.

3) The Dining-Philosphers Problem: Consider 5 philosphers, who spend their lives thinking & eating. The philosophers share a circular table surrounded by 5 chairs, each belonging to 1 philosopher. In the centre of the table is a bowl of rice & the table is laid with 5 single chopsticks. When a philosophur thinks, he does not interact with his collegues. From time to time, a philosopher gets hungry & tries to pick up the 2 chopsticks that are closest to him (the chopsticks that are blw his left & right neighbours.).

A philosopher may pick up only 1 chopstick at a time. Obviously, she can't pickup a chopstick that is already in hand of a neighbour.

When a hungry philosopher have both chopsticks at the same time, he eats without releasing her chopsticks. When she is finished eating, she puts down both of his chopsticks & starts thinking again.

One simple Soln to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore; she releases his chopsticks by executing the signal() operation on the appropriate semaphore. Thus the shared data are:

```
Semaphore chopstick[5];
```

where all the elements of chopsticks are initialized to 1.

The structure of philosopher 'i' is shown below:

Soln 1:

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

The above Soln gurantees that no 2 philosophers are eating simultaneously but it is rejected because it could create a deadlock. Suppose that all 5 philosophers are hungry, each grabs his left chopstick. All the 5 elements of chopsticks will now be equal to zero. And deadlock occurs.

Soln 2: Allow atmost 4 philosophers to pick up left chopstick first & other philosopher picks up right chopstick first.

Lefty

```
wait(chopstick[i]);
wait(chopstick[(i+1)%5]);
//eat
signal(chopstick[i]);
signal(chopstick[(i+1)%5]);
//think
}while(True);
```

Righty

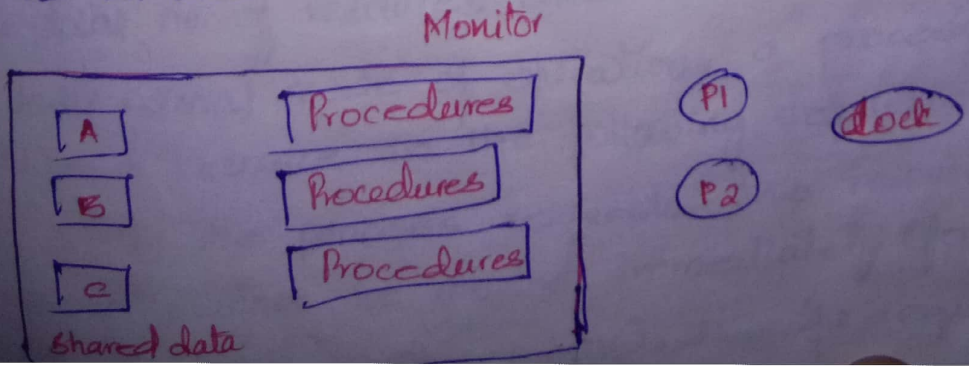
```
do
{
wait(chopstick[(i+1)%5]);
wait(chopstick[i]);
//eat
signal(chopstick[(i+1)%5]);
signal(chopstick[i]);
//think
}while(True);
```

This will solve the deadlocks problem, as atleast 1 philosopher will get 2 chopsticks.

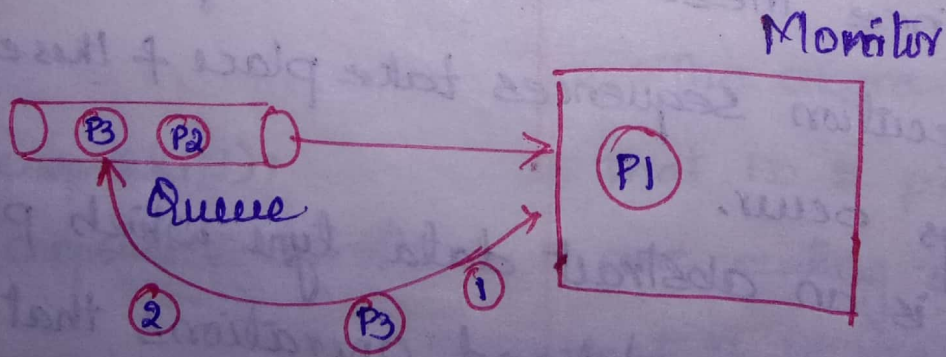
*Monitors: Although semaphores provides a convinient & effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors can happen only if some particular execution sequences take place & these sequence do not always occur.

A monitor is an abstract data type which presents a set of programmer defined operations that are provided mutual exclusion within the monitor.

It encapsulates private data with public methods to operate on that data.



- Maintors contains shared data & also procedures which needs to access data. Processes will not directly access data. Processes have to call procedures & procedures in turn allow access to data. And only one process can access data at any point of time.
- Monitors are associated with locks & process which have the lock can only access data.
- Monitor is a module that encapsulates:
  1) Shared data structures
  2) Procedures that operate on the shared data
  3) Synchronization blw concurrent procedure invocation.
- If process P1 wants to access shared data it will contact the procedure & gets a lock & enters CS now other processes will not be allowed to enter the monitor.



7. Deadlocks

In a multiprogramming environment, several processes may compete for a finite no. of resources. A process requests a resource; if all the resources are not available at that time, the process enters a waiting state. Sometimes a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called as a deadlock.

System Model: A system consists of finite no. of resources to be distributed among a no. of competing processes. The resources are partitioned into several types, each consisting of some no. of identical instances. Memory space, CPU cycles, files, I/O devices (printers, DVD driver) are examples of resource types. If a system has 2 CPU's, then the resource type CPU has 2 instances. If a process requests an instance of a resource type, the allocation of any instances of the type will satisfy the request.

A process must request a resource before using it and must release the resource after using it. The no. of resources required by a process shld not exceed the total no. of resources available in the system.

Under normal mode of operation, a process may utilize a resource in the following sequence:

1) Request: The process requests the resource. If the resource cannot be granted immediately (for eg. it is being used by another process), then the requesting

process must wait until it can acquire the resour

2) Use: The process can operate on the resource.

3) Release: The process releases the resource.

✓ A programmer who is developing multithreaded applications must pay particular attention to this prb. Multithreaded progs are good candidates for deadlock because multiple threads can compete for shared resources.

Deadlock Characterization: Features that characterize deadlocks.

Necessary conditions: A deadlock situation can arise if the following 4 conditions hold simultaneously in the syst.

1. Mutual Exclusion: Atleast 1 resource must be held in a nonsharable mode; that is only 1 process at a time can use the resource. If another process requests the resource, the requesting process must be delayed until the resource has been released.

2. Hold and Wait: A process must be holding atleast 1 resource & waiting to acquire additional resources that are currently being held by other processes.

3. No preemption: Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. Circular Wait: A set $\{P_0, P_1, \ldots P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ... P_{n-1} is

waiting for a resource held by P_n & P_n is waiting for a resource held by P_0.

We emphasize that all 4 conditions must hold for a deadlock to occur. These conditions are not completely independent.

Resource-Allocation Graph: Deadlocks can be described more precisely in terms of a directed graph called a system Resource-Allocation graph.

This graph consists of a set of vertices V and set of edges E. The set of vertices V is partitioned into 2 different types of nodes: $P = \{P_1, P_2, P_3, \ldots P_n\}$, the set consisting of all the active processes in the system, & $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

Notations:

① $P_i \rightarrow R_j$: Process P_i has requested an instance of resource type R_j & is currently waiting for the resource (Request Edge).

② $R_j \rightarrow P_i$: An instance of resource type R_j has been allocated to process P_i. (Assignment Edge).

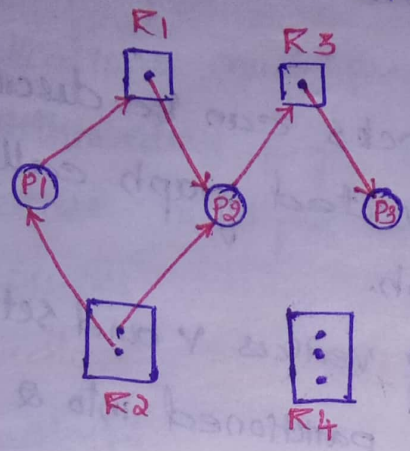
Pictorially: P_i is represented as circle & R_j as rectangle. Since R_j may have more than 1 instance, we represent each instance as a dot within rectangle. Note that the request edge points to only the rectangle R_j, whereas an assignment edge must also designate one of the dots in rectangle.

eg: RAG (Resource Allocation Graph)

P = {P1, P2, P3}

R = {R1, R2, R3, R4}

E = {P1→R1, P2→R3, R1→P2, R2→P2, R2→P1, R3→P3}



Instances:

R1 = 1

R2 = 2

R3 = 1

R4 = 3

Process states:

✓ P1 is holding

Process	Holding	Waiting
P1	R2	R1
P2	R1, R2	R3
P3	R3	

Given a definition of the resource Allocation Graph, it can be shown that, if the graph contains no cycles then, no process in the system is deadlocked. If the graph does contain a cycle, then deadlock may exist.

① Eg: Cycle → Deadlock: Consider the previous RAG, & suppose P3 requests an instance of resource type R2. Since no resource instance is currently available, a request edge P3→R2 is added to the graph. At this point 2 minimal cycles exist in the system.

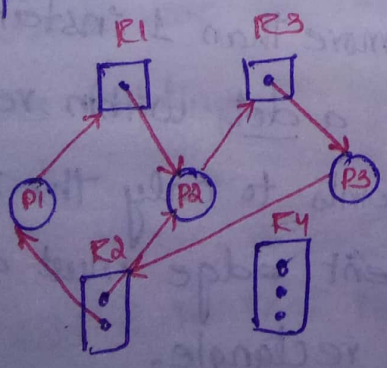
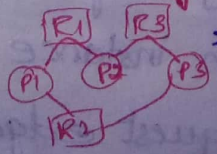
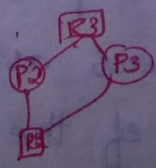


Fig: RAG with a deadlock

Cycles:



P1 → R1 → P2 → R3 → P3 → R2 → P1

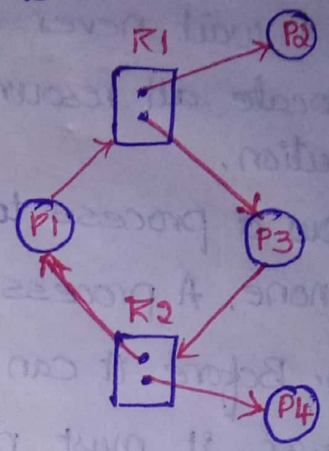


P2 → R3 → P3 → R2 → P2

Processes P1, P2 & P3 are deadlocked

process P2 is waiting for R3, which is held by P3. P3 is waiting for either P1 or P2 to release R2. In addition, P1 is waiting for P3 to release R1.

2) eg: cycle → but no deadlock: There is no deadlock, because P4 process may release its instance of resource type R2. That resource can be allocated to P3, breaking the cycle, Then the system may or may not be all in a dead-locked state.



Cycle: P1 → R1 → P3 → R2 → P1

Methods for Handling Deadlocks:

1) Deadlock Prevention: Provides a set of methods for ensuring that atleast 1 of the necessary conditions cannot hold.

2) Deadlock Avoidance: Requires that OS be given in advance additional info concerning which resource a process will request & use during lifetime.

3) Deadlock Detection & Recovery: If above 2 methods are not used, then deadlock may occur. Then it shld be detected and recovered.

① Deadlock Prevention:

i) Mutual Exclusion: The mutual exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes.

* Sharable resources, do not required ME. access & thus cannot be involved in deadlock.

* In general, we cannot prevent deadlocks by denying

the ME Condition, because some resources are intrinsically non-sharable.

ii) Hold & Wait: The following protocal can be used to ensure hold & wait never occurs.

a) Allocate all resources to a process before it begins execution.

b) Allow a process to request resources only when it has none. A process may request some resources & use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.

Disadvantages:

1) Resource utilization may be low, since resources may be allocated but unused for a long period.

2) Starvation is possible: A process that needs several popular resources may have to wait indefinitely, because atleast 1 of the resources that it needs is always allocated to some other process.

3) No Preemption: The foll. protocol can be used: If a process is holding some resources & requests another resource that cannot be immediately allocated to it. (that is the process must wait), then all resources, the process is currently holding are preempted. (released).

4) Circular wait: One way to ensure that this condition never holds is to impose a total ordering of all resource types & to require that each process requests in an increasing order of enumeration

Let $R = \{R1, R2, R3, \ldots Rm\}$ be the set of resource types. We assign a unique integer no. to each resource type, which allows us to compare 2 resources & to determine whether 1 precedes another in our ordering.

Formally, we define a one-to-one function.

$F: R \rightarrow N$, where N is set of Natural numbers.

eg: $F(R1) = 1$
$F(R2) = 2$
$F(R3) = 3$
$\vdots$
$F(Rm) = m$

Now, each process can request resources only in an increasing order of enumeration.

i.e., A process can request any no. of instances of resource types. Say R_i. After that, the process can request instances of resource type R_j if any only if $F(R_j) > F(R_i)$.

Alternatively, we can require that a process requesting an instance of resource R_j must have released any resources R_i such that $F(R_i) \geq F(R_j)$.

* Deadlock Avoidance: Deadlock prevention may result in low device utilization & reduced sys. throughput. If the os has knowledge of what resources are requested by the processes before execution, then os can decide whether or not those resources are allocated to the processes to avoid possible deadlock in the future.

Each request requires that in making this decision the sys considers the resources currently available, the

resources currently alloted to each process, & the future requests & releases of each process.

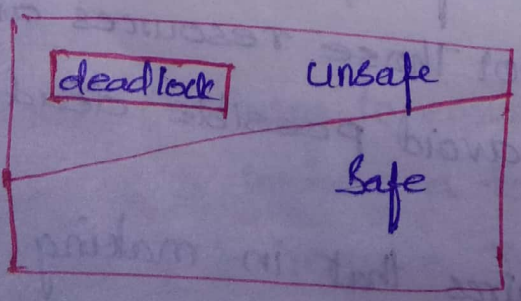
Safe State: A state is safe if the Sys can allocate resources to each process (upto its maximum) in some order & still avoid a deadlock.

More formally, a Sys is in a safe state only if there exists a safe sequence.

A sequence of processes $\langle P_1, P_2, \ldots P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i, the resource requests that P_i can still be satisfied by the currently available resources plus the resources held by all P_j, with $j < i$.

In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, & terminate. When P_i terminates P_{i+1} can obtain its needed resources and so on. If no such sequence exists, then the Sys State is said to be unsafe.

* A deadlocked state is an unsafe state & an unsafe state may lead to a deadlock.



Eg: consider 12 tape driver & 3 processes.

Process	Max Need	Currently holding / allocated
P0	10	5
P1	4	2
P2	9	2

(a) Find whether the sys. is in safe state or not & also find the safe sequence?

Total tape drivers = 12

Already alloted = 5+2+2 = 9

Free tape drives = 12-9 = 3//

We need to use the available tape drives & find the safe sequence.

safe sequence = <P1, P0, P2>//

P1 needs 4 & it already has 2, allocate 2 more drives = 4, P1 can complete execution & it releases 4 drives. Now total drives available = 4+1 = 5//. Now we can allocate these 5 drives to P0 ⇒ if release 10 drives. then P2 can be executed

chance of unsafe state: If P2 is allocated with 1 more tape drive = ?

P1 can be executed but neither P0 nor P2 can complete their execution ⇒ Unsafe state.

Here idea is to keep the sys in a safe state. Initially, the sys is in a safe state. Whenever a process requests a resource that is currently availab -le, the sys must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the sys in a safe state.

Resource Allocation Graph Alg: If we have a RA Sys. with only 1 instance of each resource type, we can use a variant of the resource-allocation graph for deadlock avoidance.

- In addition to the request & assignment edges, a new type of edge called as claim edge is introduced. This is represented by a dashed line.

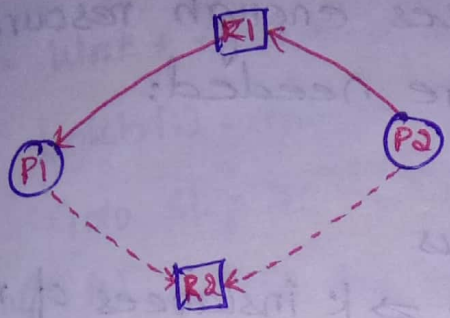
$P_i \dashrightarrow R_j$ ⇒ Process P_i may request resource R_j at sometime in the future.

- When the process P_i requests resource R_j, the claim edge $P_i \dashrightarrow R_j$ is converted into a request edge $P_i \rightarrow R_j$. Similarly, when a resource R_j is released by P_i, the assignment edge $R_j \rightarrow P_i$ is reconverted into a claim edge $P_i \dashrightarrow R_j$.
- The resources must be claimed by the process before starting the execution of process i.e., all claim edges must appear in the RAG.
- Now suppose that process P_i requests resource R_j. The resource can be granted only if converting the requesting edge $P_i \rightarrow R_j$ to assignment edge $R_j \rightarrow P_i$ does not result in the formation of cycle in RAG. We check for the safety by using a cycle-detection alg. Alg. may take an order of n^2 operations (where n is the no. of processes in the Sys).

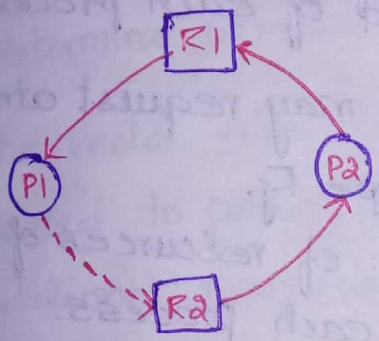
If no cycles exists, then the allocation of the resources will leave the sys in a safe state. If a cycle is found, then the allocation will put the

system in an unsafe state. In that case P_i will have to wait for its requests to be satisfied.

Eg: Consider the foll. situation.



Now suppose that P2 requests R2. Although R2 is currently free, we cannot allocate it to P2, since this allocation/action will create a cycle in the graph.



This indicates that the sys. is in an unsafe state. (If P1 requests R2 & P2 requests R1, then a deadlock will occur).

P1 → R2 → P2 → R1 → P1

Drawback: RAG alg. is not applicable to a resource allocation sys with multiple instances of each resource type. The next, Bankers algorithm is applicable to such sys. but less efficient than RAG.

Bankers Algorithm: The name was choosen because the alg. could be used in a banking sys to ensure that the bank never allocated its available cash in such away that it could no longer satisfy the needs of all its customers.

* When a new process enters the sys, it must declare the max no. of instances of each resource type that it may need. Now, the sys will determine

whether the allocation of these resources will leave the sys in a safe state. If it will, the resources are allocated; otherwise the process must wait until some other process releases enough resources.

✓ The foll. datastructures are needed:

n: no. of processes

m: no. of resource types.

1) **Available:** Available[j] = k ⇒ k instances of resource type R_j are available (Total resoures)

2) **Max:** n×m matrix (max. demand of each process

max[i][j] = k ⇒ Process P_i may request atmost k instances of resource type R_j.

3) **Allocation:** n×m matrix defines no. of resources of each type currently allocated to each process.

Allocation[i][j] = k ⇒ Process P_i is currently allocated with k instances of resource type R_j.

4) **Need:** n×m matrix defines the remaining resource need of each process.

Need[i][j] = k ⇒ P_i may need k more instances of resource type R_j to complete its task.

Note: Need[i][j] = Max[i][j] − Allocation[i][j].

Safety Alg: The alg for finding out whether or not a sys is in a safe state.

Let **work** & **finish** be vectors of length m & n respectively.

Initialization: 1) Work = Available

Finish[i] = False for i = 0, 1, 2, ..., n−1

2) Find an index i such that both

a. Finish[i] == false;

b. Need$_i$ <= Work

If no such i exists goto step 4.

3) Work = Work + Allocation$_i$;

finish[i] = true;

Goto Step 2.

4) If finish[i] == true for all i, then the Sys is in a safe state.

This alg. may require an order of $m \times n^2$ operations to determine whether a state is safe.

The vector Allocation$_i$ specifies the resources currently allocated to process P_i; the vector Need$_i$ specifies the additional resources that process P_i may still request to complete its task.

* Resource Request Alg: Alg to determine whether the requests can be safely granted or not.

Let Request$_i$ be the request vector for process P_i. If request$_i$[j] = k, then process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken.

1. If Request$_i$ <= Need$_i$, goto step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Request$_i$ <= Available, goto step 3. Otherwise, P_i must wait, since the resources are not available.
3. Have the System pretend to have allocated the requested

resources to process P_i by modifying the state as follows. (If 2 is true).

Available = Available − Request$_i$;

Allocation = Allocation$_i$ + Request$_i$;

Need = Need$_i$ − Request$_i$;

If the resulting resource allocation state is safe, the transaction is completed, & process P_i is allocated its resources. However if the new state is unsafe, then P_i must wait for Request$_i$, & the old resource-allocation state is restored.

Deadlock Detection: If a sys. does not employ either a deadlock-prevention or a deadlock-avoidance alg, then a deadlock situation may occur. In this environment the sys may provide:

1) An alg. that examines the state of the sys to determine whether a deadlock has occured.

2) An alg. to recover from the deadlock.

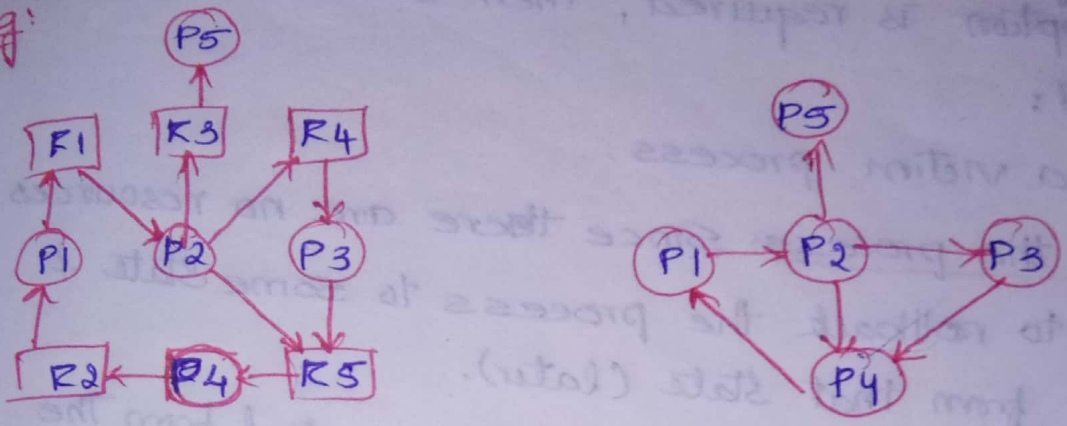
I. Single Instance of each Resource type: If all resources have only a single instance, then we can define a deadlock detection alg that uses a variant of the resource allocation graph, called a **wait-for** graph. We obtain this graph from RAG by removing the resource nodes & collapsing the appropriate edges.

More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait for graph if & only if the corresponding

RAG graph contains 2 edges $P_i \rightarrow R_q$ & $R_q \rightarrow P_j$ for some resource R_q.

✓ If there is a cycle ⇒ Deadlock.

eg:



II. Several Instances of a Resource type.

✓ Bankers algorithm.

Recovery from Deadlock: When a deadlock detection alg determines that a deadlock exists, several alternatives are available. One possibility is to inform the operator that a deadlock has occured & let the operator deal with the deadlock manually. Another possibility is to let the sys. recover from the deadlock automatically.

✓ There are 2 options for breaking a deadlock.

1) Process Termination 2) Resource Preemption.

1) Process Termination: 1) Abort all deadlocked processes: will break the deadlock but at a great expense. The deadlocked processes may have computed for long time.

2) Abort 1 process at a time until the deadlock cycle is eliminated: Incurs a considerable overhead. After each process is aborted, a deadlock detection alg. must be invoked.

2) Resource Preemption: Preempt some resources from process & give these resources to other processes until the deadlock cycle is broken.

If preemption is required, then 3 issues need to be addressed:

i) Selecting a victim process.

ii) Rollback the process: Since there are no resources we need to rollback the process to some state & restart it from that state (later).

iii) Starvation: If resources are preempted from the same process again & again then that process will be starved.