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.

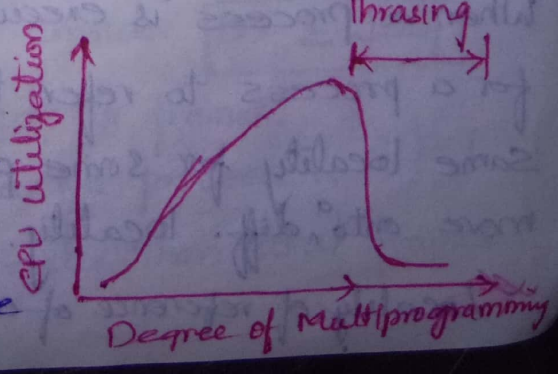
This high paging activity is called Thrashing. A

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: ⓐ 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.

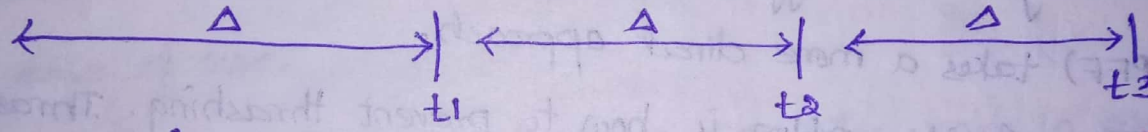
Locality of reference: Prg & data references within a process tend to cluster, which forms diff. localeties. When a process is executed, there is more probability for a process to refer the instructions/data from the same locality for some period of time. Then it will move onto a diff. locality.

- 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 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, 2, 3, 4}

From the fig. the working set at time t1 = {1, 2, 5, 6, 7} at t2 = {3, 4}, t3 = {1, 2, 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 work set of each process & allocates those many frames. & there are extra frames another process can be initiated. If sum of the working set size ds, exceeding the tota no. of available frames, OS selects a process to suspen 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 prblm 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-faul rate is too low, then the process may have too many fames. (Conversely, if the pagefault) 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 in 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 greatly 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.

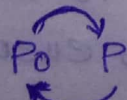
✓ P_1 : (turn != 1) ⇒ False ⇒ P_1 can enter its critical section

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 it 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]);
⇓ false

(∵ flag [F|T] (0,1))

P1 can enter its cs.

∴ Mutual Exclusion Satisfied. //

② Progress = ?

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

(∵ flag [F|F] (0,1))

⇓ 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 [T|T] (0,1)

⇒ Both processes wants to go to its critical section but none of them are able to go. ⇒ Deadlock.

⇓ progress is not satisfied.

* Peterson's algorithm: 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 [F | F]

can Po go for multiple iterations = ? ⇒ Possible

Strict alternation ⇒ ✗ not there

Does deadlock exists = ?

turn = 0/1

flag [F | F]

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 [T | T], 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.