One of the problems with Optimal (OPT) Page Replacement Technique is that it cannot decide which page should be dropped for maximum efficiency particularly when the current pages in page table do not occur in near future. Suppose that we have three pages P1, P2 and P3 in the page table.

Page Table
All three pages P1, P2 and P3 are least likely to occur in the future, so any of these pages can be faulted to make place for P4.

**Global Page Replacement Algorithm** –
When a process incurs a page fault, a Global page replacement algorithm is free to select any page.
In case of page fault, we can take page from any process (P1, P2 or P3). Global Page Replacement algorithm is adaptive and tries to reach to reach the equilibrium with memory utilization. Disadvantage of this algorithm is that users can impact one other.

**Per-Process Page Replacement Algorithm** –

When a process incurs a page fault, a Per-Process page replacement algorithm selects for replacement some page that belongs to that same process.

Per-Process Replacement algorithm is non-optimal and isolates different processes. It mainly aims at guarding the page replacement from encountering the worst case where different processes may contend for some shared resource or data.

**CLOCK Algorithm**

The clock algorithm keeps a circular list of pages in memory, with the hand pointing to the oldest page in the list. When a page fault occurs and no empty frames exist, then the A (accessed) bit is inspected at the hand's location. If A is 0, the new page is put in place of the page the "hand" points to, otherwise the A bit is cleared. Then, the clock hand is incremented and the process is repeated until a page is replaced.

Performance of Clock algorithm is asymptotically similar to LRU algorithm. It gives less overhead as there is no need of writing back a page if A=1. As the hand advances, we clear the A bit so that a page can be dropped.
Suppose that we have following page sequence –

\[
\begin{array}{cccccccccccccccccccc}
1 & 2 & 3 & 4 & 2 & 1 & 5 & 6 & 2 & 1 & 2 & 3 & 7 & 6 & 3 & 2 & 1 & 2 & 3 & 6
\end{array}
\]

*) Initially pages 1,2,3 and 4 are loaded into the queue and A is set to zero.

*) We get a hit on next two pages i.e 1 and 2, so A bit of 1 and 2 is changed to 1.
*) On encountering page 5, clock hand is incremented and since A bit of page 3 = 0, it is replaced with page 5 with A=0.

*) Similarly page 4 is replaced with page 6.
*) Next three pages 2, 1 and 2 will give a hit and access bits will be changed to 1.
*) Page 3 will replace page 5.
*) Page 7 will replace page 6.

*) Now clock hand will point to page 1 and clear the A bit and page 6 will replace page 1.
*) Page 3 and 2 will give a hit and A bit is changed to 1.

*) Page 1 will replace page 7.

*) Pages 2,3 and 6 will give a hit.

Problem with above mentioned Clock algorithm is A bit for incoming pages is assumed to be 0 which actually should be 1 as pages are accessed when they are read in the queue. But if pages come in with A=1, they have to go all the way around to get cleared and again all the way around to get evicted.

To overcome these problems, we use Two-handed clock algorithm. The front hand rotates ahead of the back hand, clearing the referenced and modified bits for each page.
The trailing back hand then inspects the referenced and modified bits some time later. Pages that have not been referenced or modified are swapped out and freed.

Page X is loaded with $A = 1$

Page Y is loaded with $A = 1$ and A bit of X is set to 0.
Page Z is loaded with $A = 1$ and A bit of $Y$ is set to 0.

**Mid Term review**

Topics covered till lecture 5

- pre HW1:
  1. Memory maps
  2. Reading on UNIX and DOS.
  3. Simple machine model

- HW1
  1. OS interface (Question 2 of HW1)
  2. Program loading (Question 1 of HW1)
  3. Command line interface (Question 3 of HW1)
  4. Context Switching (Question 4 of HW1)
  5. Multiple Processes (Question 4 of HW1)

- Synchronization primitives
  1. Mutex

Mutex has two operations – lock and unlock. Its main purpose is to avoid race condition and simultaneous access by making an operation atomic.

```c
lock (obj) {
  do stuff
} unlock (obj)
```
2. Semaphore

Semaphores are mainly used to sequence/synchronize threads by causing them to sleep or awake at certain conditions.

Semaphore (n)
enter : n=n-1 if n>0
       else sleep

leave : n=n+1 if no waiter
       else wake 1

3. Monitor
Monitors are better form of semaphores. Monitor check a specific condition and perform a task accordingly.

wait (condition)
signal (condition)
broadcast (condition)

- Page table
  1. Tree structure
  2. Translation algorithm
  3. TLB

- Page replacement
  1. LRU
  2. FIFO
  3. OPT
  4. Clock with A bit
**Calling Stack**

Using a stack to save the return address has important advantages over alternatives. One is that each task has its own stack, and thus the subroutine can be reentrant, that is, can be active simultaneously for different tasks doing different things. Another benefit is that recursion is automatically supported. When a function calls itself recursively, a return address needs to be stored for each activation of the function so that it can later be used to return from the function activation. This capability is automatic with a stack.

Call stack can be used for –
1. Local data storage
2. Parameter passing
3. Return address

```assembly
caller:
    push arg1
    push arg2
    call f
    add sizeof(args), SP
f:
    sizeof (locals)
    sub #, SP
    --------
    --------
    --------
    add #, SP
    return
```
By switching between stacks, we switch between threads.