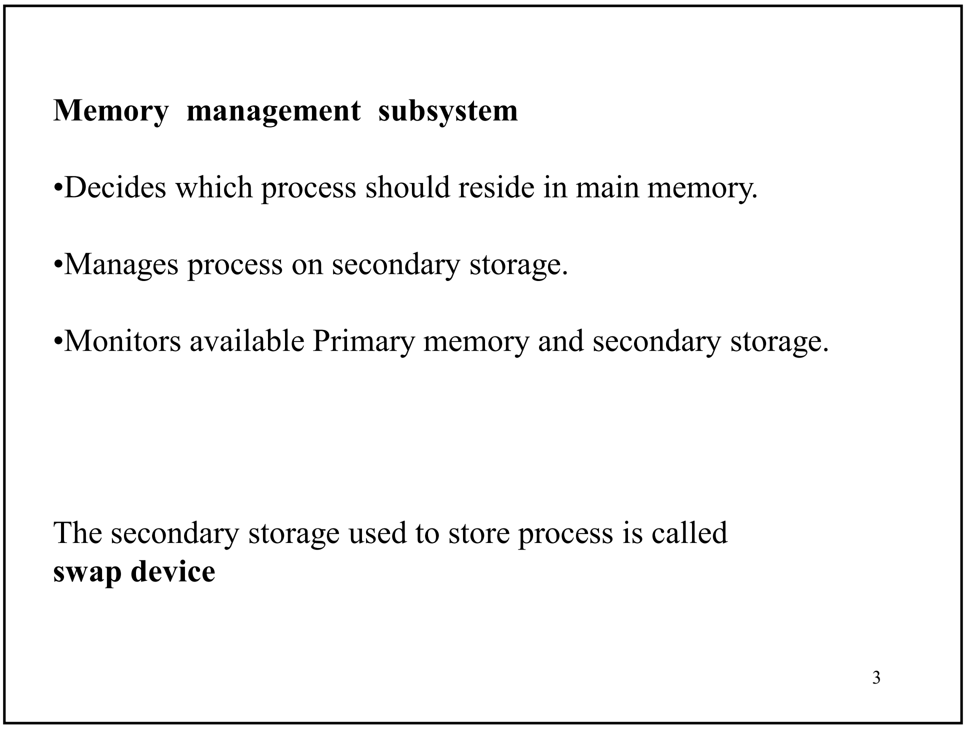
Memory Management Policies

- To Execute a Process at least part of the process must be in main Memory.
- Primary(Main) memory is a precious Resource.
- All active process can not be placed in main memory at once.

How to execute many process simultaneously?

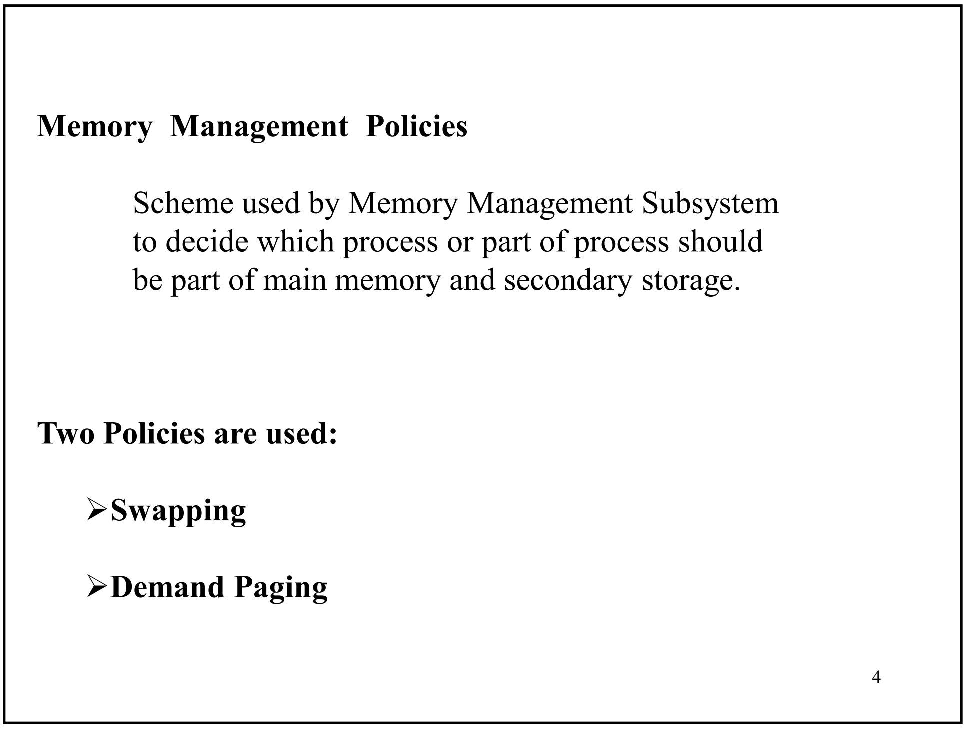
- not to store all process in main memory.
- Part of process may present on secondary storage.
- decision is made by the Memory management subsystem.



Memory management subsystem

- Decides which process should reside in main memory.
- Manages process on secondary storage.
- Monitors available Primary memory and secondary storage.

The secondary storage used to store process is called
swap device

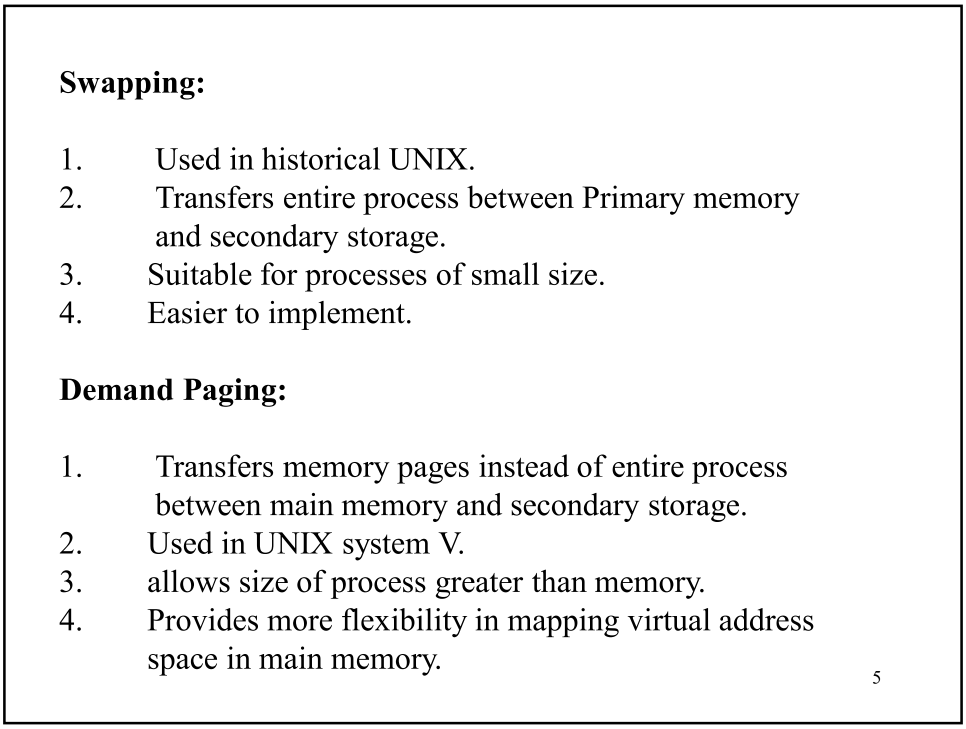


Memory Management Policies

Scheme used by Memory Management Subsystem to decide which process or part of process should be part of main memory and secondary storage.

Two Policies are used:

- ➢**Swapping**
- ➢**Demand Paging**

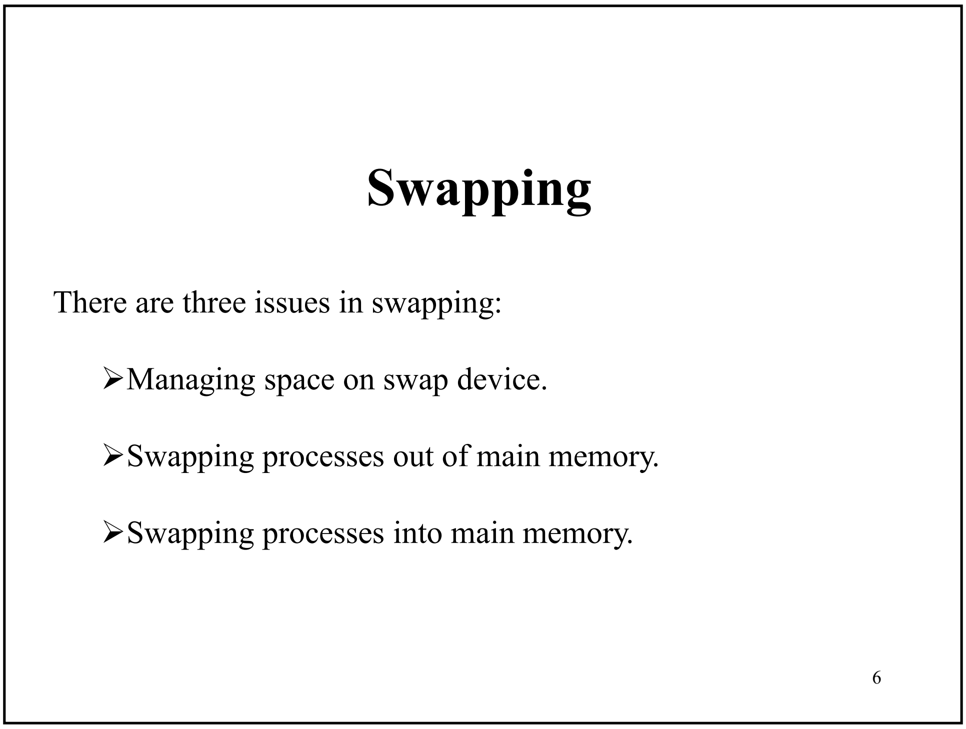


Swapping:

1. Used in historical UNIX.
2. Transfers entire process between Primary memory and secondary storage.
3. Suitable for processes of small size.
4. Easier to implement.

Demand Paging:

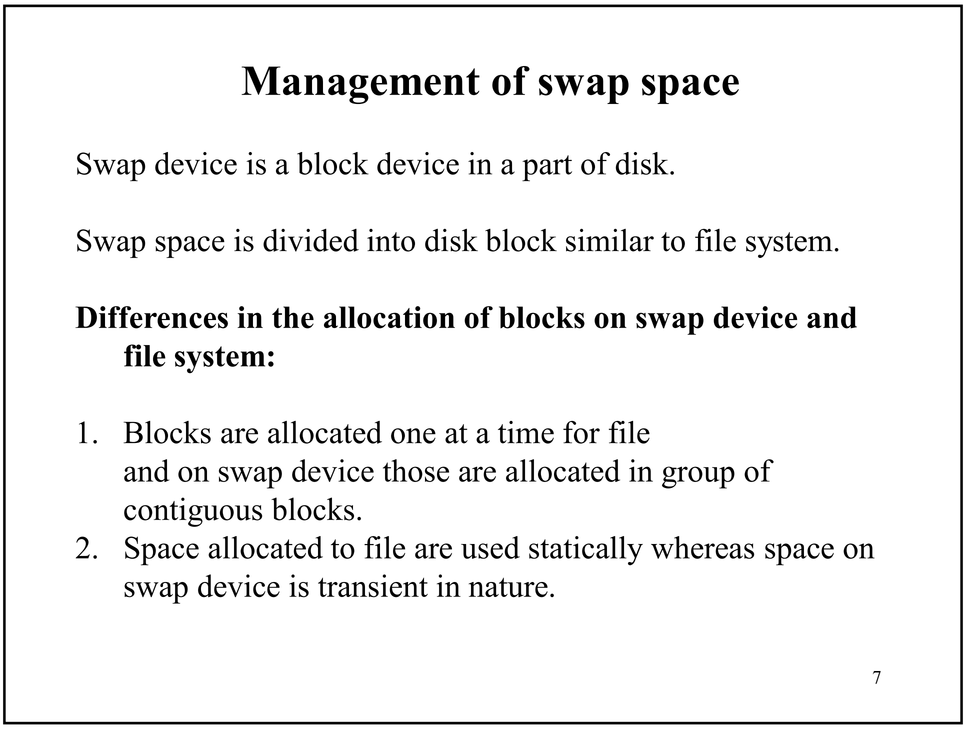
1. Transfers memory pages instead of entire process between main memory and secondary storage.
2. Used in UNIX system V.
3. allows size of process greater than memory.
4. Provides more flexibility in mapping virtual address space in main memory.



Swapping

There are three issues in swapping:

- ➢Managing space on swap device.
- ➢Swapping processes out of main memory.
- ➢Swapping processes into main memory.



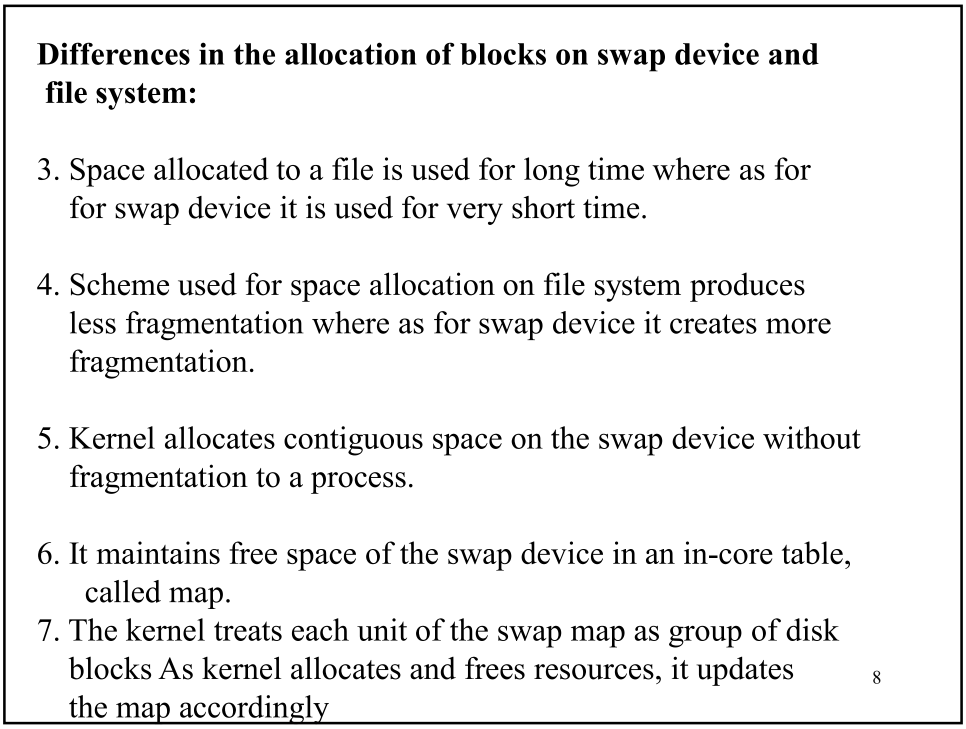
Management of swap space

Swap device is a block device in a part of disk.

Swap space is divided into disk block similar to file system.

Differences in the allocation of blocks on swap device and file system:

1. Blocks are allocated one at a time for file
   and on swap device those are allocated in group of contiguous blocks.
2. Space allocated to file are used statically whereas space on swap device is transient in nature.



Differences in the allocation of blocks on swap device and file system:

3. Space allocated to a file is used for long time where as for for swap device it is used for very short time.

4. Scheme used for space allocation on file system produces less fragmentation where as for swap device it creates more fragmentation.

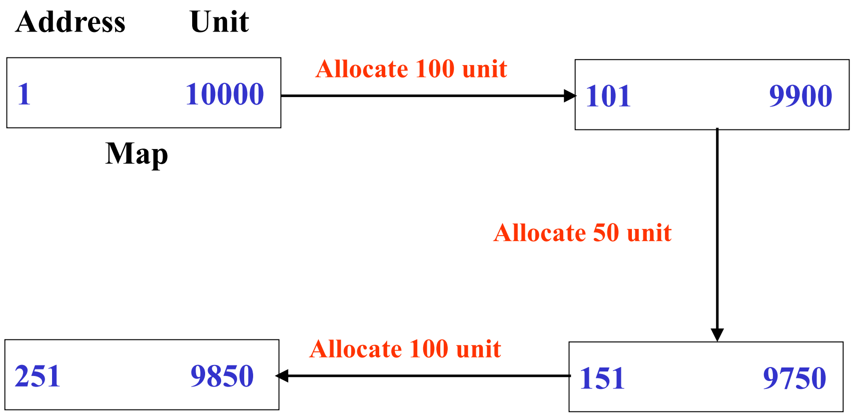
5. Kernel allocates contiguous space on the swap device without fragmentation to a process.

6. It maintains free space of the swap device in an in-core table, called map.
7. The kernel treats each unit of the swap map as group of disk blocks As kernel allocates and frees resources, it updates the map accordingly

Swapping

- The swap device is a block device in a configurable section of a disk
- Kernel allocates contiguous space on the swap device without fragmentation
- It maintains free space of the swap device in an in-core table, called map
- The kernel treats each unit of the swap map as group of disk blocks
- As kernel allocates and frees resources, it updates the map accordingly

Allocating Swap Space



Address Unit

1 10000

Map

Allocate 100 unit

101 9900

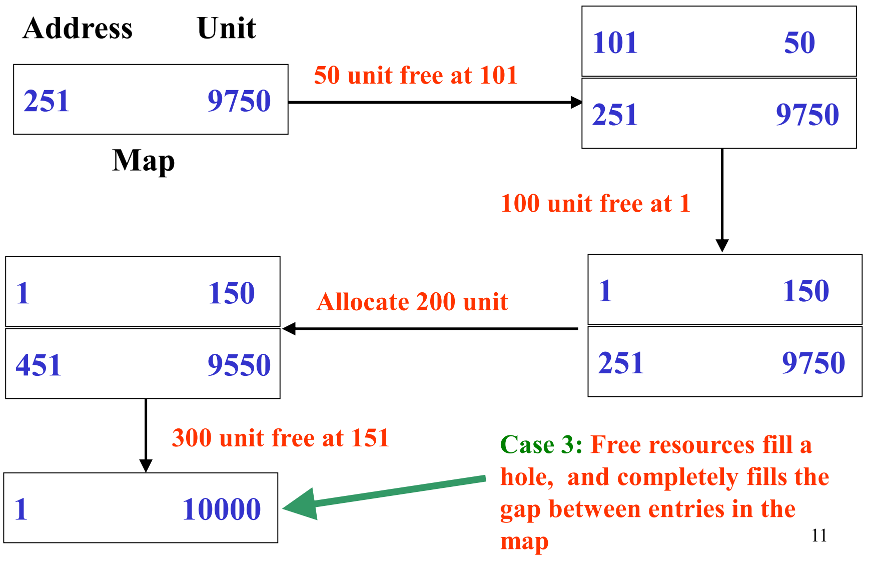
Allocate 50 unit

151 9750

Allocate 100 unit

251 9850

Freeing Swap Space



Address Unit

251 9750

Map

50 unit free at 101

101 50

251 9750

100 unit free at 1

1 150

251 9750

Allocate 200 unit

1 150

451 9550

300 unit free at 151

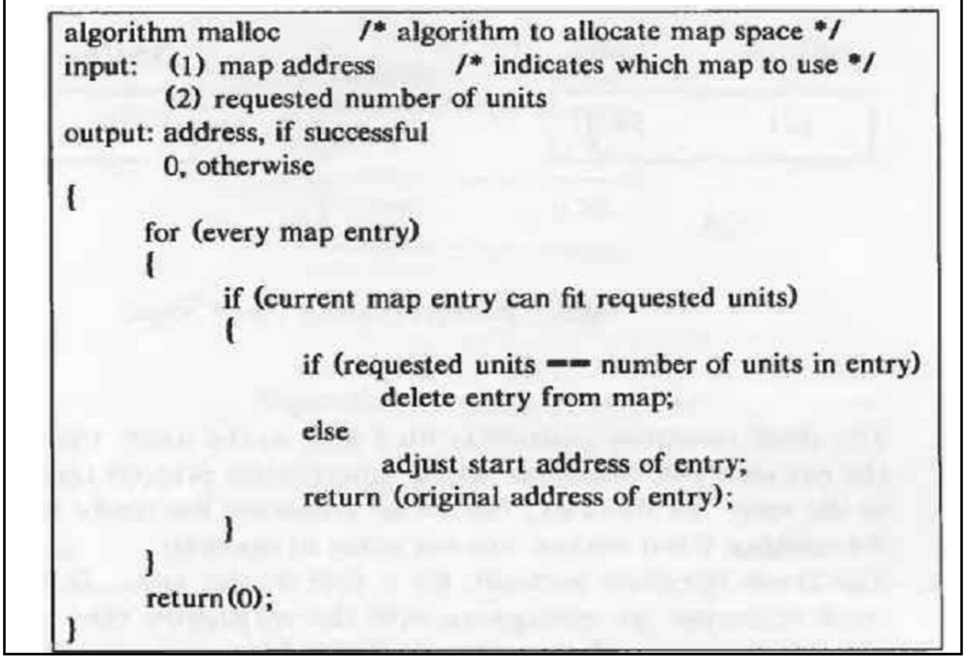
1 10000

Case 3: Free resources fill a hole, and completely fills the gap between entries in the map

11

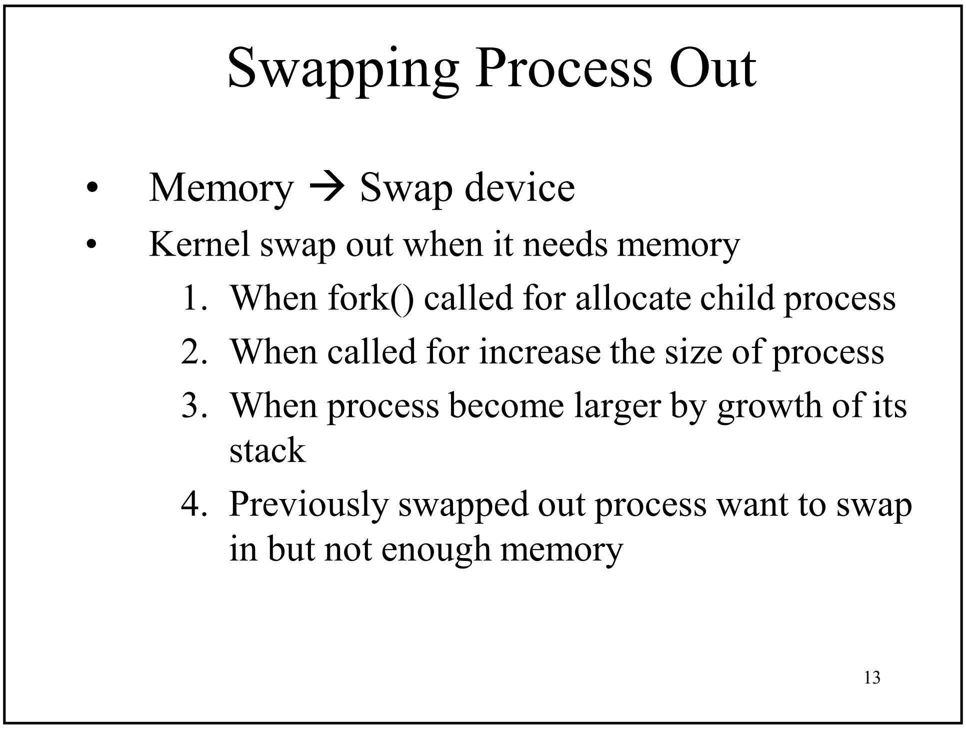
11

Algorithm for allocation of Swap space



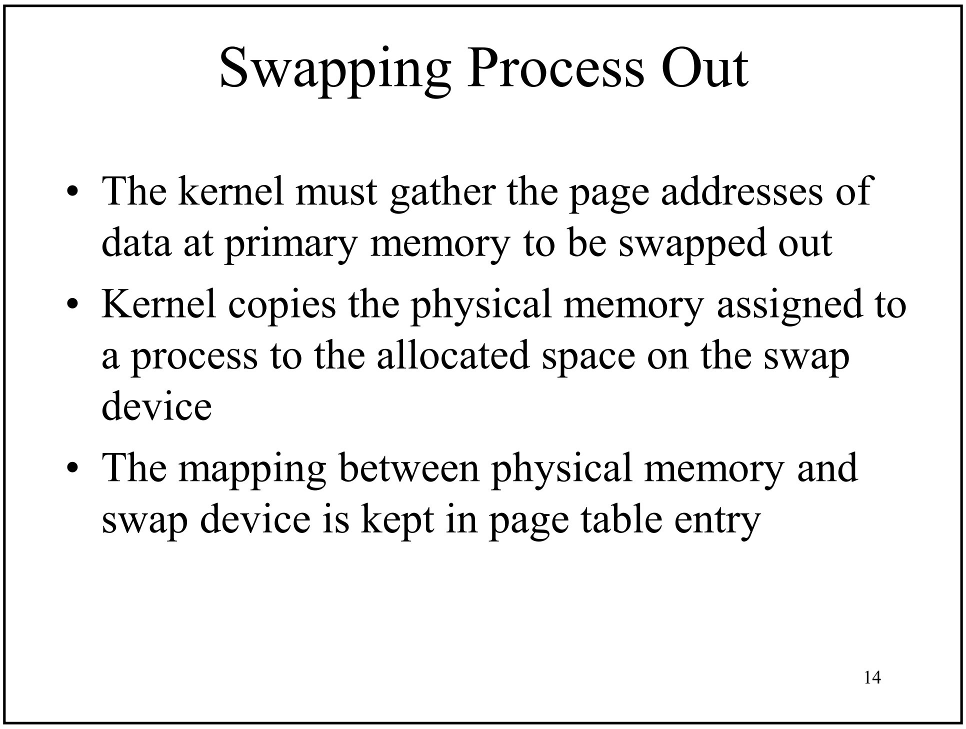
```
algorithm malloc          /* algorithm to allocate map space */
input:  (1) map address         /* indicates which map to use */
        (2) requested number of units
output: address, if successful
        0, otherwise
{
        for (every map entry)
        {
                if (current map entry can fit requested units)
                {
                        if (requested units == number of units in entry)
                                delete entry from map;
                        else
                                adjust start address of entry;
                        return (original address of entry);
                }
        }
        return(0);
}
```

12



Swapping Process Out

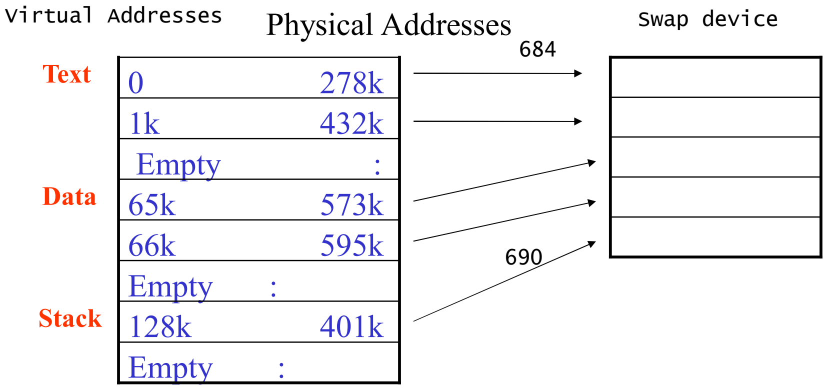
- Memory → Swap device
- Kernel swap out when it needs memory
  1. When fork() called for allocate child process
  2. When called for increase the size of process
  3. When process become larger by growth of its stack
  4. Previously swapped out process want to swap in but not enough memory



Swapping Process Out

- The kernel must gather the page addresses of data at primary memory to be swapped out
- Kernel copies the physical memory assigned to a process to the allocated space on the swap device
- The mapping between physical memory and swap device is kept in page table entry

Swapping Process Out

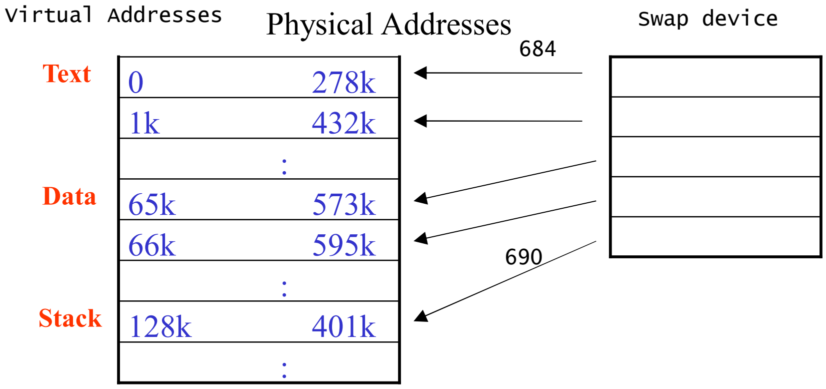


	Virtual Addresses	Physical Addresses
Text	0	278k
	1k	432k
	Empty	:
Data	65k	573k
	66k	595k
	Empty	:
Stack	128k	401k
	Empty	:

Mapping process onto the swap device

15

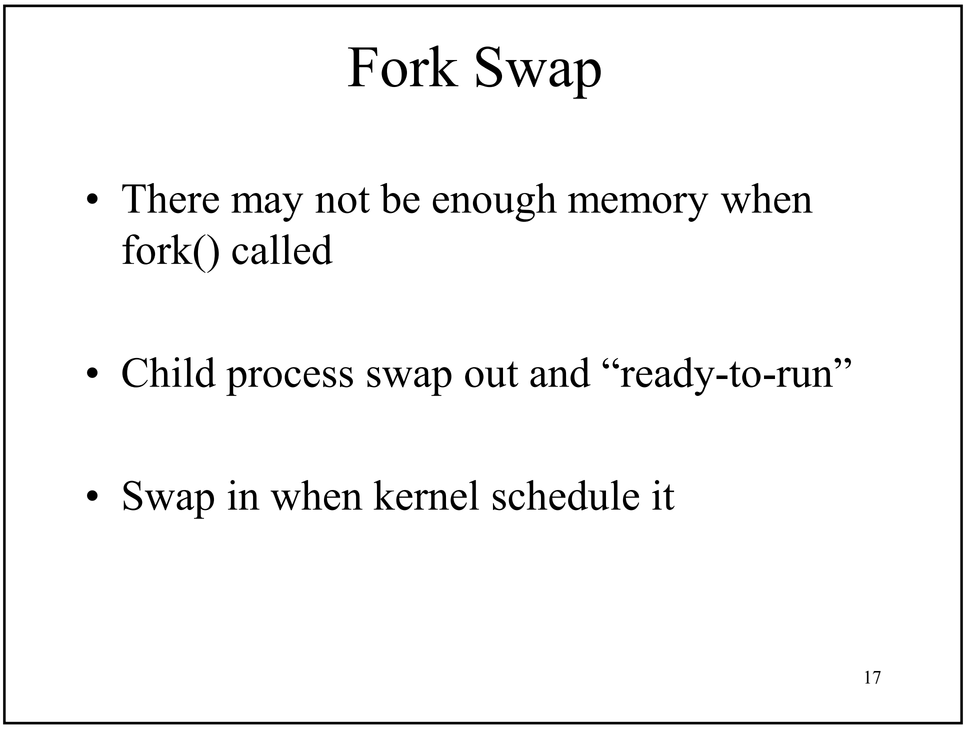
Swapping Process In



	Virtual Addresses	Physical Addresses
Text	0	278k
	1k	432k
		:
Data	65k	573k
	66k	595k
		:
Stack	128k	401k
		:

Swapping a process into memory

16

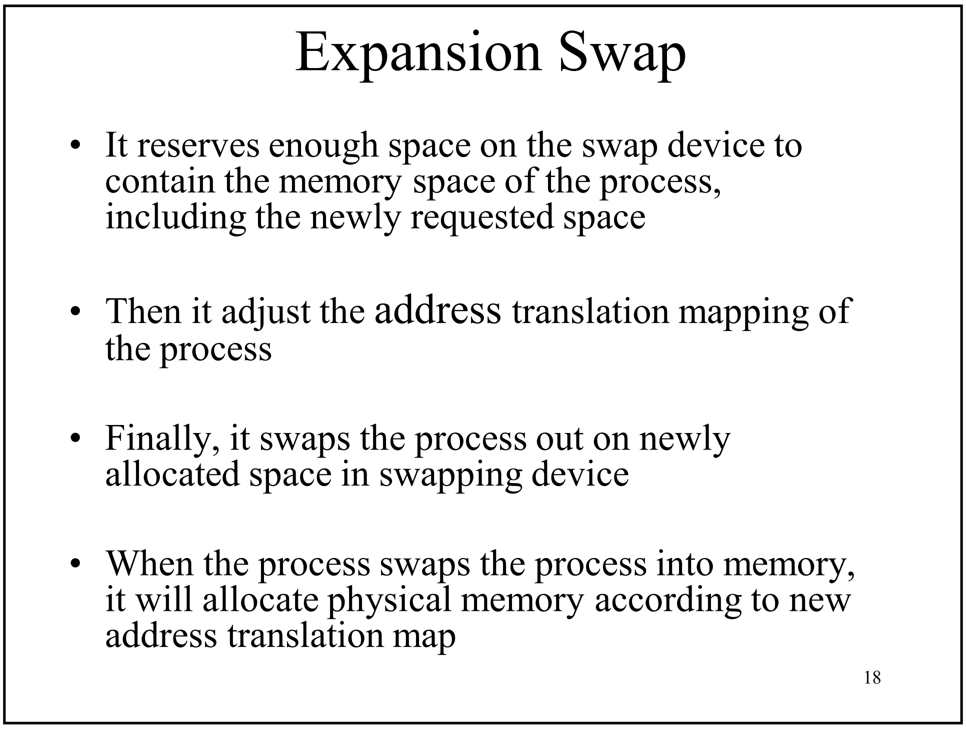


Fork Swap

- There may not be enough memory when fork() called
- Child process swap out and “ready-to-run”
- Swap in when kernel schedule it

17

17

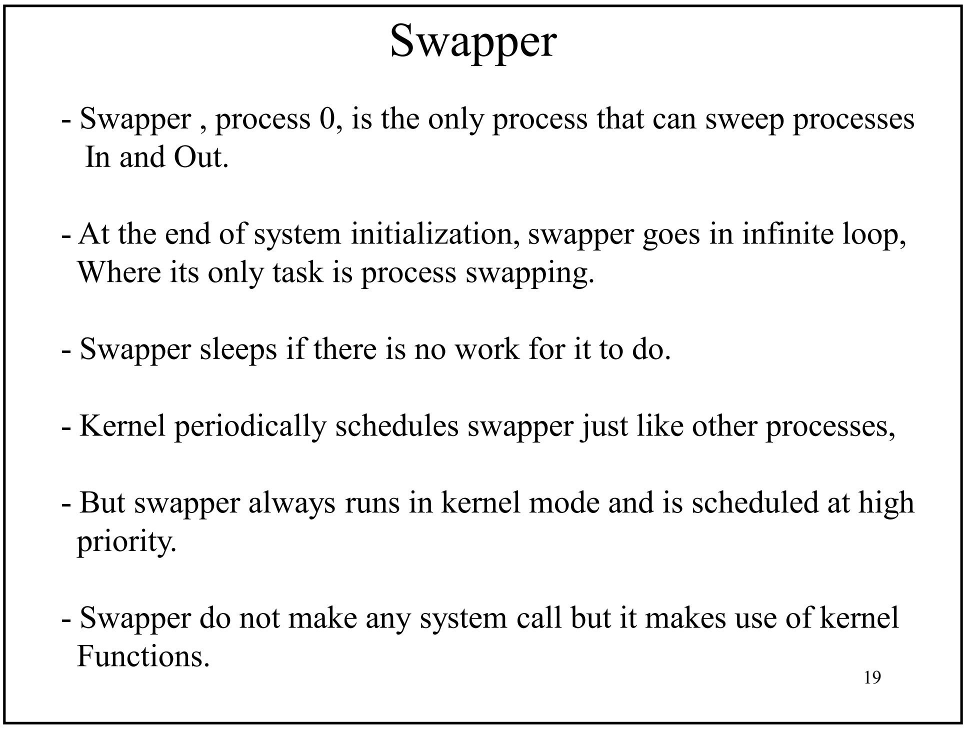


Expansion Swap

- It reserves enough space on the swap device to contain the memory space of the process, including the newly requested space
- Then it adjust the address translation mapping of the process
- Finally, it swaps the process out on newly allocated space in swapping device
- When the process swaps the process into memory, it will allocate physical memory according to new address translation map

18

18



Swapper

- Swapper , process 0, is the only process that can sweep processes In and Out.

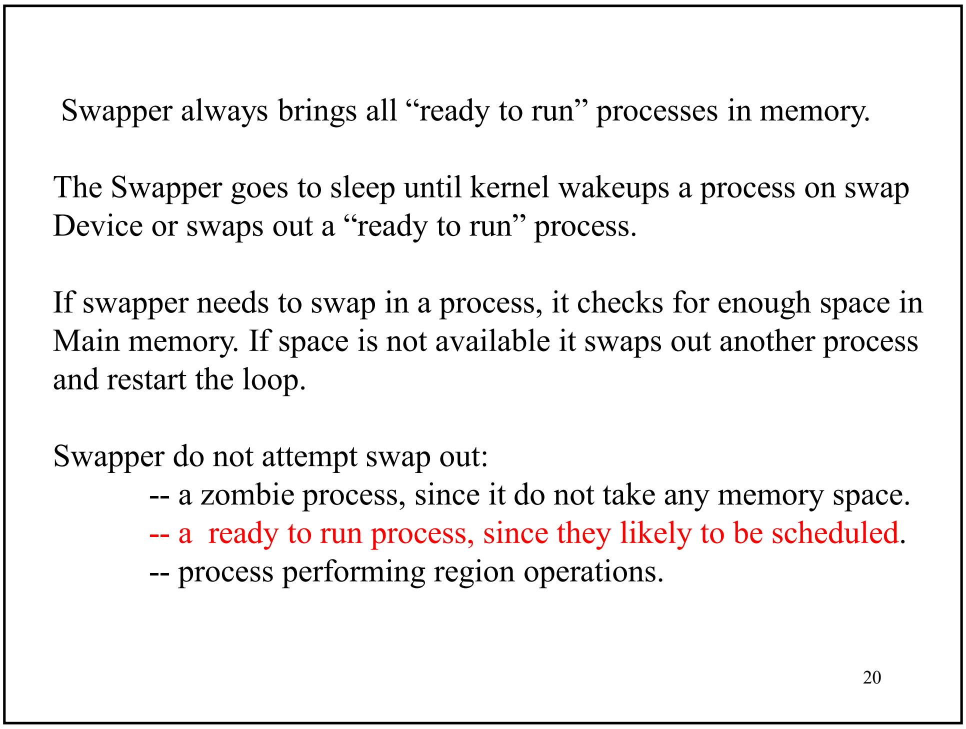
- At the end of system initialization, swapper goes in infinite loop, Where its only task is process swapping.

- Swapper sleeps if there is no work for it to do.

- Kernel periodically schedules swapper just like other processes,

- But swapper always runs in kernel mode and is scheduled at high priority.

- Swapper do not make any system call but it makes use of kernel Functions.



Swapper always brings all “ready to run” processes in memory.

The Swapper goes to sleep until kernel wakeups a process on swap Device or swaps out a “ready to run” process.

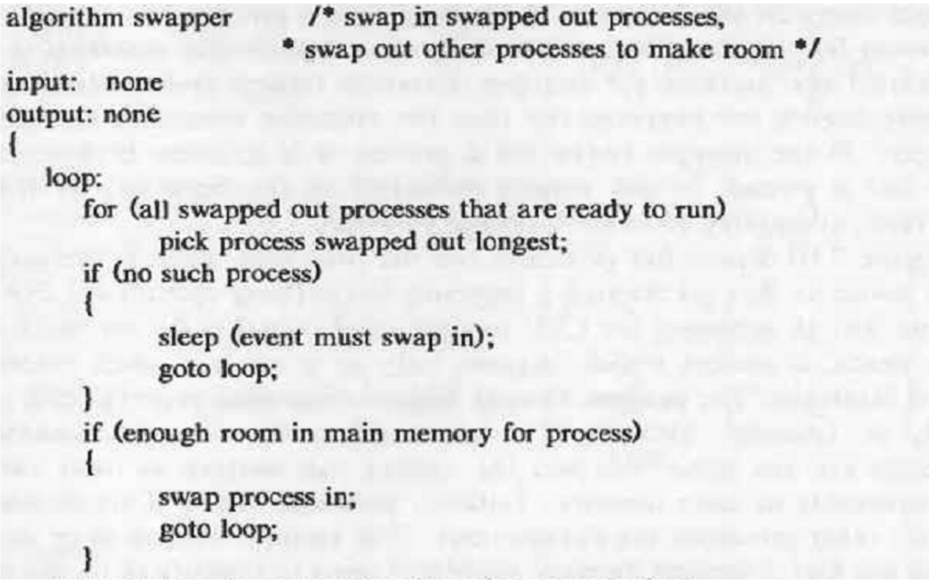
If swapper needs to swap in a process, it checks for enough space in Main memory. If space is not available it swaps out another process and restart the loop.

Swapper do not attempt swap out:

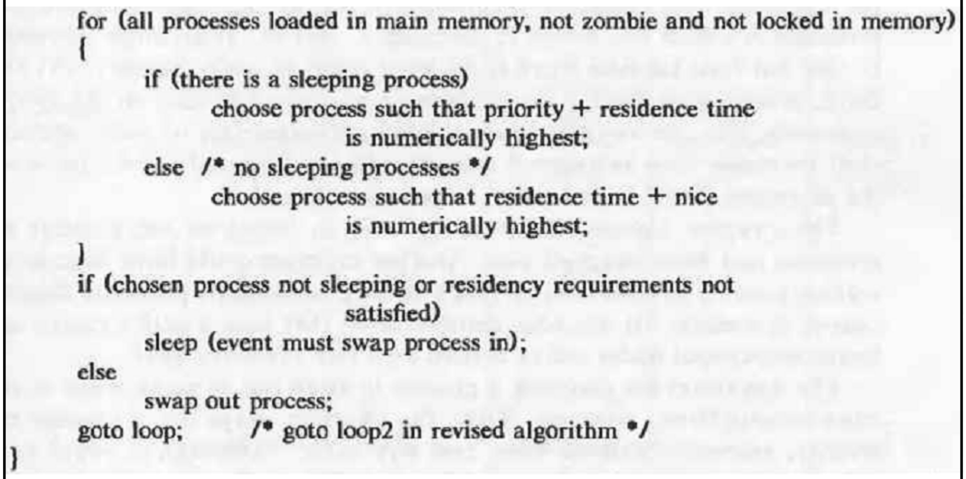
-- a zombie process, since it do not take any memory space.
-- a ready to run process, since they likely to be scheduled.
-- process performing region operations.

Swappers makes a choice of sleeping processes for swap out
Based on their priority and time spend in memory.

Swapper may swap a “ready to run” process based on its nice
Value.



```
algorithm swapper       /* swap in swapped out processes,
                         * swap out other processes to make room */
input:  none
output: none
{
  loop:
     for (all swapped out processes that are ready to run)
           pick process swapped out longest;
     if (no such process)
     {
           sleep (event must swap in);
           goto loop;
     }
     if (enough room in main memory for process)
     {
           swap process in;
           goto loop;
     }
```



```
for (all processes loaded in main memory, not zombie and not locked in memory)
{
        if (there is a sleeping process)
                choose process such that priority + residence time
                        is numerically highest;
        else  /* no sleeping processes */
                choose process such that residence time + nice
                        is numerically highest;
}
if (chosen process not sleeping or residency requirements not
                        satisfied)
        sleep (event must swap process in);
else
        swap out process;
goto loop;      /* goto loop2 in revised algorithm */
}
```

Demand Paging

Possible only on the machines having MM unit supporting “Paging” And CPU supporting “restartable” instructions.

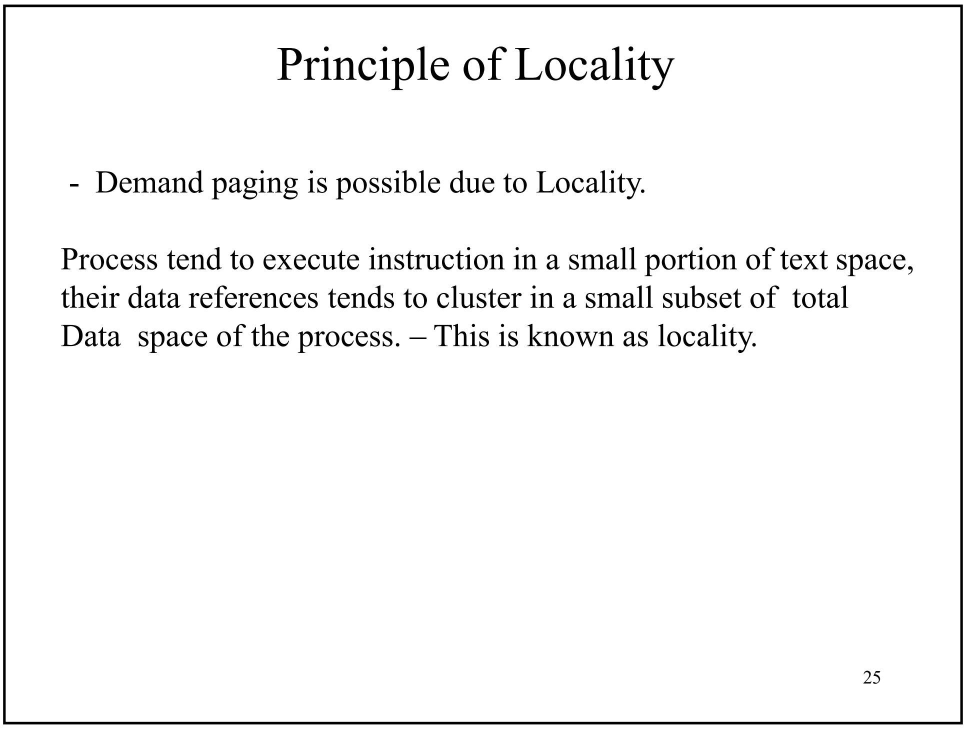
Not all page of process resides in main memory.

No size limitation for process. A machine having 1 M of physical Memory can run process of size more than 1 M.

But kernel still puts limit on maximum virtual size of process.

When a process accesses a page that is not part of its working set, it incurs a page fault.

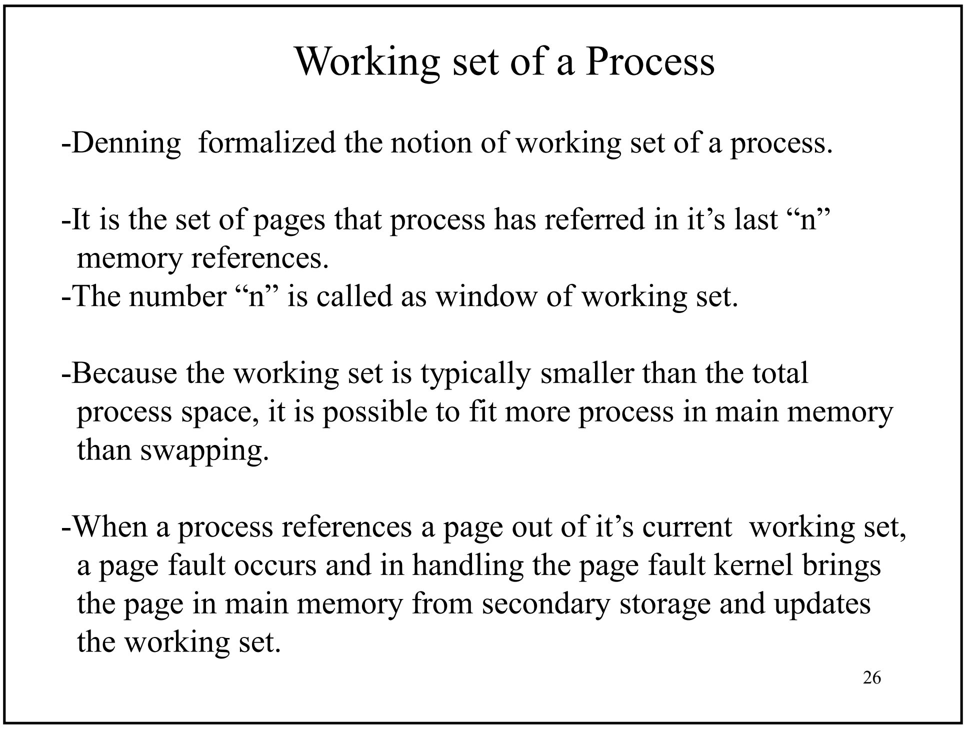
The kernel suspends the execution of the process until it reads the page into memory and makes it accessible to the process



Principle of Locality

- Demand paging is possible due to Locality.

Process tend to execute instruction in a small portion of text space, their data references tends to cluster in a small subset of total Data space of the process. – This is known as locality.



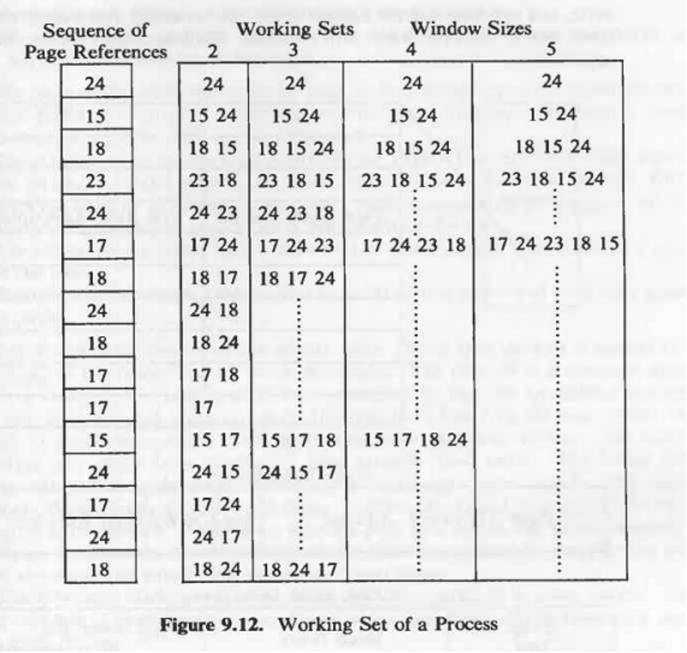
Working set of a Process

-Denning formalized the notion of working set of a process.

-It is the set of pages that process has referred in it's last "n" memory references.
-The number "n" is called as window of working set.

-Because the working set is typically smaller than the total process space, it is possible to fit more process in main memory than swapping.

-When a process references a page out of it's current working set, a page fault occurs and in handling the page fault kernel brings the page in main memory from secondary storage and updates the working set.



Sequence of	Working Sets		Window Sizes	
Page References	2	3	4	5
24	24	24	24	24
15	15 24	15 24	15 24	15 24
18	18 15	18 15 24	18 15 24	18 15 24
23	23 18	23 18 15	23 18 15 24	23 18 15 24
24	24 23	24 23 18	⋮	⋮
17	17 24	17 24 23	17 24 23 18	17 24 23 18 15
18	18 17	18 17 24	⋮	⋮
24	24 18	⋮	⋮	⋮
18	18 24	⋮	⋮	⋮
17	17 18	⋮	⋮	⋮
17	17	⋮	⋮	⋮
15	15 17	15 17 18	15 17 18 24	⋮
24	24 15	24 15 17	⋮	⋮
17	17 24	⋮	⋮	⋮
24	24 17	⋮	⋮	⋮
18	18 24	18 24 17	⋮	⋮

Figure 9.12. Working Set of a Process

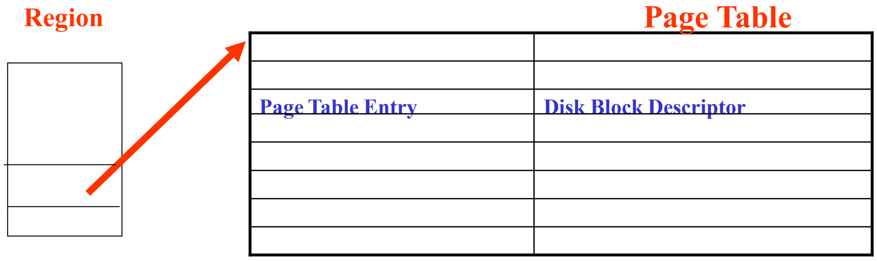
- A larger working set demands more pages in the main memory.

-Pages from working sets are removed according to LRU algorithm as process references new pages not currently in working set.

Data Structure for Demand Paging

- Page table entry
- Disk block descriptors
- Page frame data table
- Swap use table

Data Structure for Demand Paging



Page Table Entry

Page address	age	Cp/wrt	mod	ref	val	prot

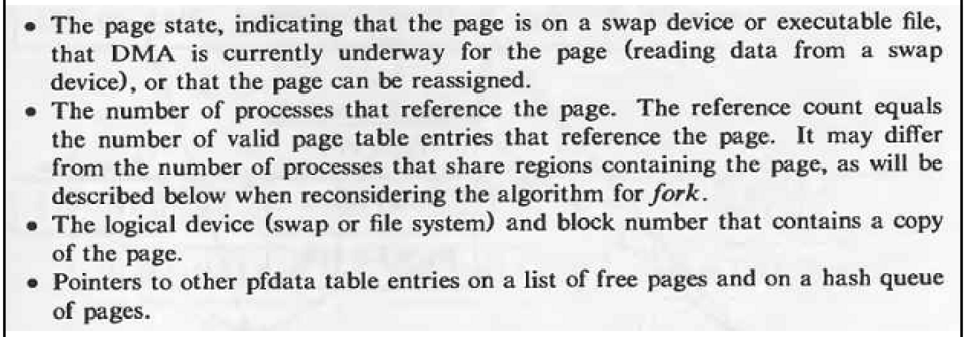
Disk Block Descriptor

Swap device	Block num	Type

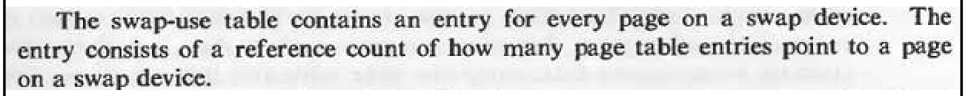
Page Table Entry

- Contains the physical address of page and the following bits:
  - Valid: whether the page content legal
  - Reference: whether the page is referenced recently
  - Modify: whether the page content is modified
  - copy on write: kernel must create a new copy when a process modifies its content (required for fork)
  - Age: Age of the page
  - Protection: Read/ write permission

Page Frame Data Table (pfdata)



- The page state, indicating that the page is on a swap device or executable file, that DMA is currently underway for the page (reading data from a swap device), or that the page can be reassigned.
- The number of processes that reference the page. The reference count equals the number of valid page table entries that reference the page. It may differ from the number of processes that share regions containing the page, as will be described below when reconsidering the algorithm for *fork*.
- The logical device (swap or file system) and block number that contains a copy of the page.
- Pointers to other pfdata table entries on a list of free pages and on a hash queue of pages.



The swap-use table contains an entry for every page on a swap device. The entry consists of a reference count of how many page table entries point to a page on a swap device.

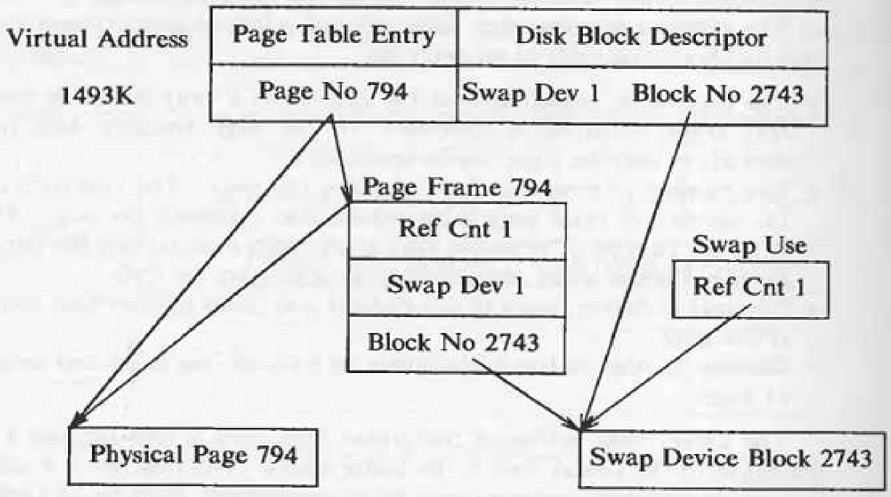


Figure 9.14. Relationship of Data Structures for Demand Paging

Fork in Paging System

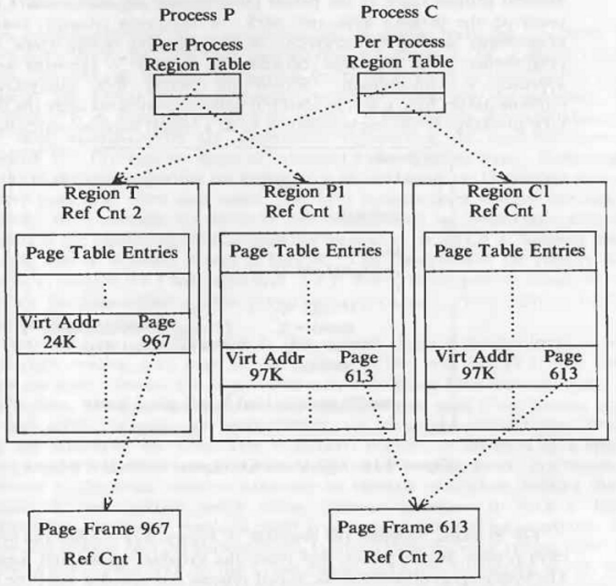
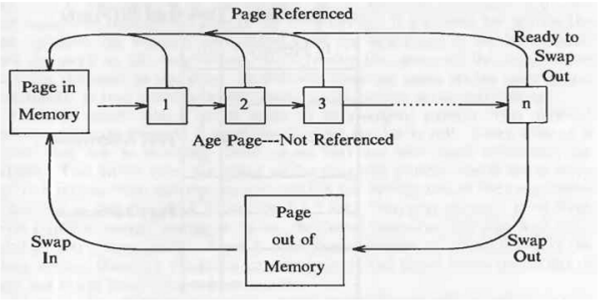


Figure 9.15. A Page in a Process that Forks

Page Stealer Process



35

35