Memory Management

- Reading Assignment:
  - Stallings, 7.1-7.5, 8.1, 8.2, 8.4
- Topics covered
  - Historical Perspective
  - Principle of locality
  - Paging (and segmentation) hardware
  - OS support for virtual memory
  - Example: Linux

Historical Perspective

- Let us (briefly) review major steps toward virtual memory and paging
  - Swapping
  - Segmentation
- Although these simple memory management techniques are not used (alone) in modern operating systems, they are important to understanding how memory systems have evolved and provide needed information for a proper discussion of virtual memory (Chapter 8)

3-1 Memory

Main Memory and Secondary Storage

- Physical Organization
  - Secondary storage (traditionally hard disks and nowadays solid-state drives) is the long term store for programs and data, while main memory holds programs and data currently in use
- Moving data between these two levels of memory has always been a major concern of OS memory management software
- Note: Early computing systems left this responsibility to the application programmer!
  - Highly inefficient
  - Tedious, easy to make mistakes

Swapping (a major advance!)

- Processes were automatically swapped in and out of main memory
- Swapping enabled the OS to have a large pool of ready-to-execute processes
- However, a program had to be loaded entirely into main memory prior to execution
- A process often had to be relocated in main memory due to swapping, repositioning due to fragmentation
- Memory references in code (for both instructions and data) translated to “new” addresses
- To do so, relative addressing used by compiler

(External) Fragmentation

- As processes are swapped in and out of memory, eventually small “holes” of free memory emerge
- Periodic compaction needed to create larger spaces

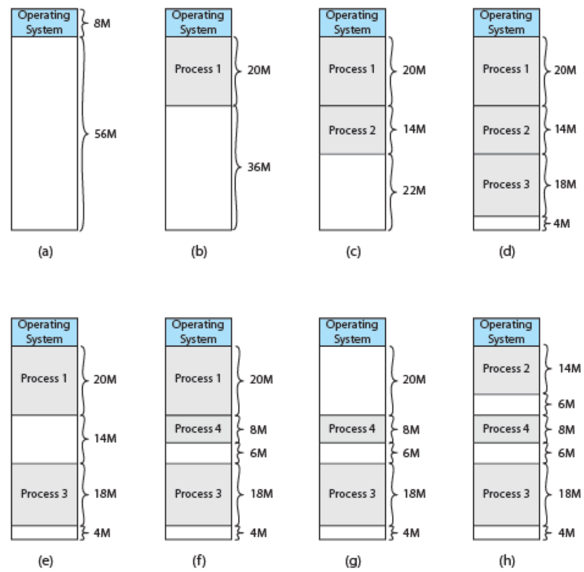


Figure 7.4 The Effect of Dynamic Partitioning

One problem: Finding space for new process…

- Given a process requiring memory size M…
- Best Fit: Find free slot as close as possible to M in size
- Worst Fit: Find free slot as large as possible (as different from M as possible)
- First Fit: Go through memory and find the first slot of size >= M.

- Pros/Cons of each strategy?

Uses in Modern Systems

- With demand paging, placement is no longer a major issue for general-purpose operating systems.
- However, placement is relevant for systems that do not use paging. Such as?!?
- Space-related algorithms (first fit, best fit, etc.) also have application in another aspect of memory management for general-purpose systems.
- Any ideas?

Replacement Problem

- In old swapped systems, when all processes in main memory are blocked, or when a swapped out process is now ready to be brought in, the OS must choose which process to replace
  - The selected process had to be swapped out and replaced by the new process
- The concept of replacement also applies to pages
  - Which page to replace with newly loaded page?
- Later, we will discuss page replacement algorithms for demand-paged virtual memory

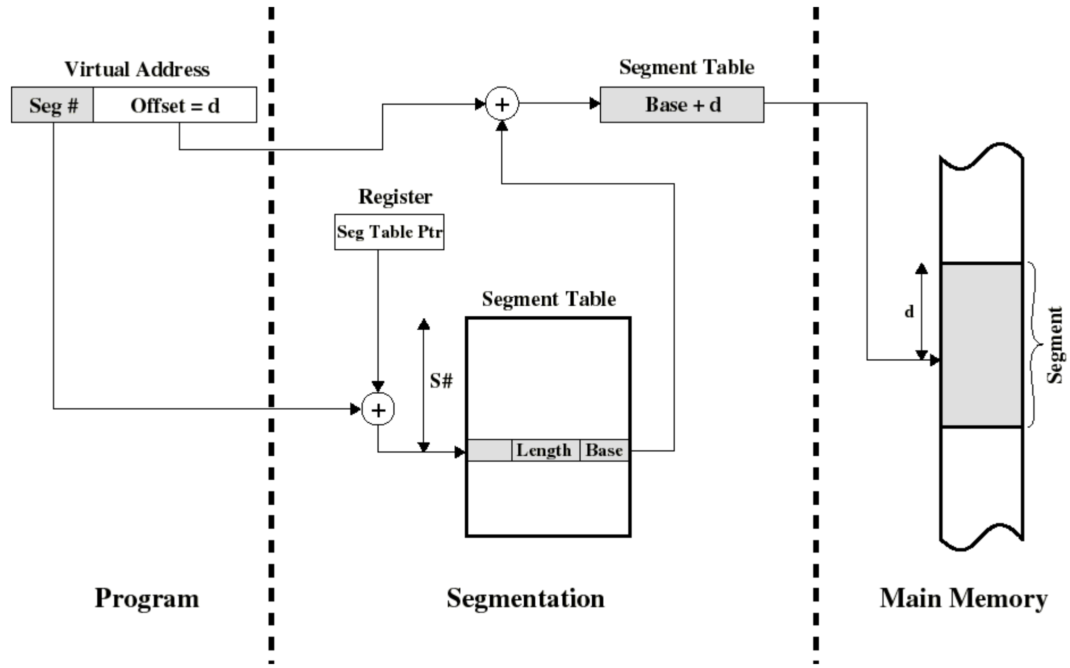
Concept of Segments

- Users typically structure programs as a set of segments with different characteristics
  - Instruction segments (code) are execute/read-only
  - Data segments are either read-only or read/write
  - Stack segment is read/write
  - Some segments are private others are shared
- To effectively deal with user programs:
  - OS and hardware should support protection and sharing
  - These should be done on a per segment basis

Logical address used in segmentation

- In the 1960s and 1970s, some processors were built with hardware support for segmentation
- When a process enters the running state, a CPU register gets loaded with the starting address of the process's segment table.
- When presented with a logical address: (segnum, offset) = (n,m), the CPU indexes (with n) the segment table to obtain the starting physical address k and length l of that segment
- The physical address is obtained by **adding** m to k
  - Segments can start anywhere in memory, in contrast to paging
  - The hardware also compares the offset m with the length l of that segment to determine if the address is valid
  - problem?

Address Translation with Segmentation



Segmentation vs. Paging

- Problems with pure segmentation?

  External fragmentation

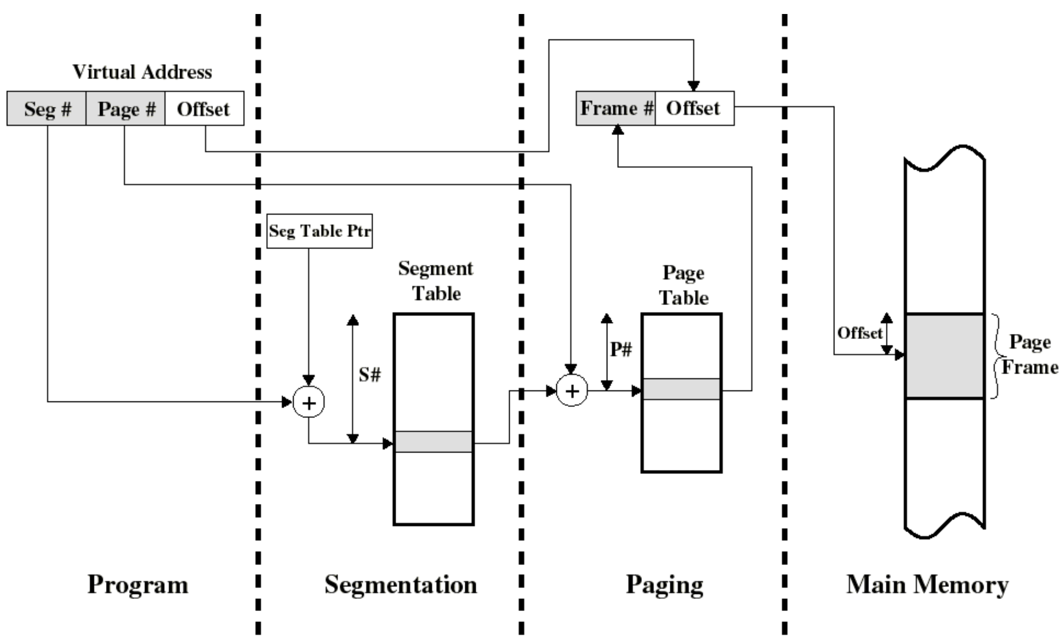
- The fixed size of pages makes it easier to construct the hardware for address translation

- With paging, no external fragmentation and very little internal fragmentation

Combined Segmentation and Paging

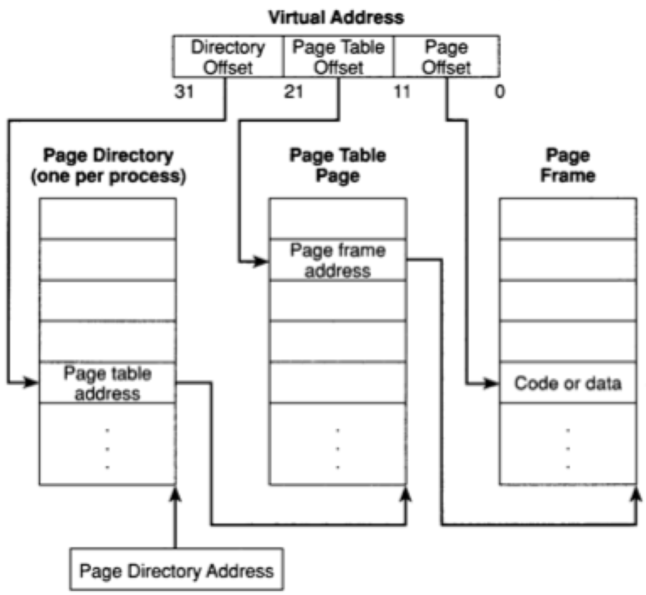
- To combine their advantages, some earlier (1970s, 1980s) processors and operating systems paged the segments.
- Each process had:
  - one segment table
  - several page tables: one page table per segment
- The virtual address consisted of:
  - a segment number: used to index the segment table; entry gives the starting address of the page table for that segment
  - a page number: used to index that page table to obtain the corresponding frame number
  - an offset: used to locate the word within the frame frame

Address Translation with Segmentation/Paging



Look familiar?

Recall Two-Level Page Tables…



Characteristics of Both Paging and Segmentation

- Virtual memory references are dynamically translated into physical addresses at run time
  - a process may be swapped in and out of main memory such that it occupies different regions
- A process may be broken up into pieces (pages, segments, or both) that do not need to be located contiguously in main memory
- Hence: all pieces of a process do not need to be loaded in main memory during execution
  - computation may continue as long the next instruction to be fetched (or the next data to be accessed) is in a piece that resides in main memory
  - takes advantage of locality of reference

Process Execution with Demand Paging

- Today, all general-purpose systems use demand paging
- The OS brings into main memory only a few pages of the program (including its starting point)
- Each page table entry has a present bit that is set only if the corresponding piece is in main memory
- The resident set is the portion of the process that is in main memory
- An interrupt (page fault) is generated when the memory reference is on a piece not present in main memory

Process Execution (cont.)

- On a page fault, the OS places the process in a Blocked state
- OS issues a disk I/O Read request to bring into main memory the page referenced (and possibly others nearby)
- Another process is dispatched to run while the disk I/O takes place
- An interrupt is issued when the disk I/O completes
  - this causes the OS to place the affected process in the Ready state

Advantages of Partial Loading

- More processes can be "active" in main memory
  - only load part of each process into memory
  - with more processes in main memory, it is more likely that at least one process will be in the Ready state at any given time
  - Why important?
- A process can now execute even if it is larger than the main memory size
  - it is even possible to use more bits for logical addresses than the bits needed for addressing the physical memory

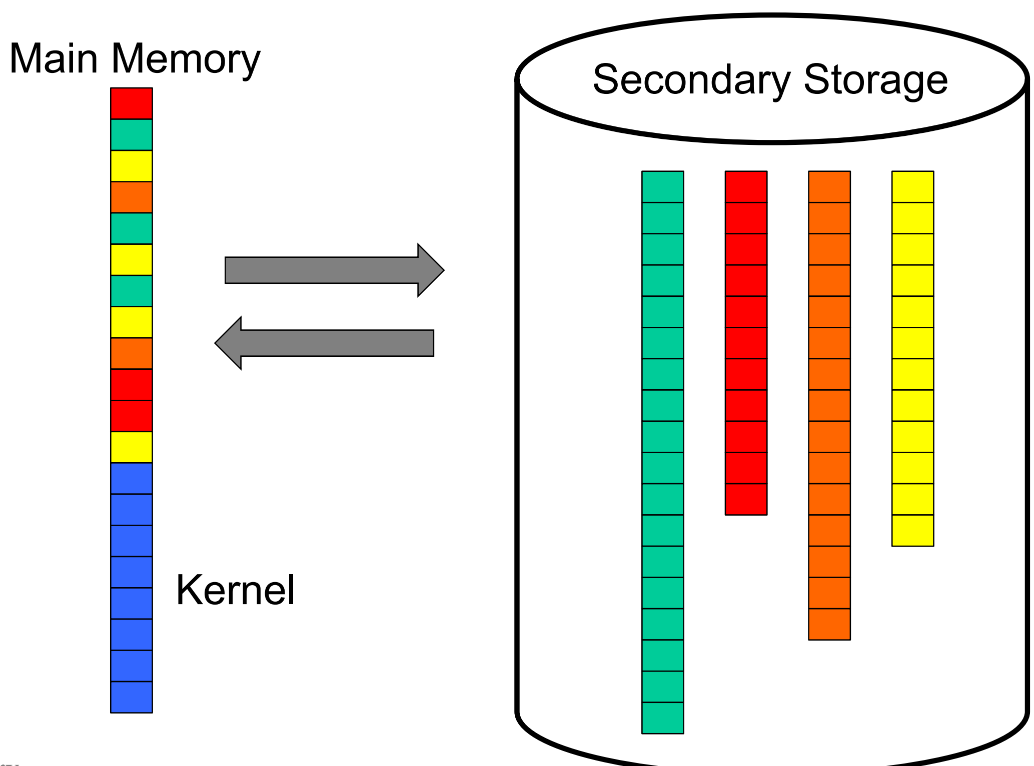
Principle of Locality

- The key to the success of this approach to memory management is that instruction and data references in a program sequence tend to cluster
- Hence, only a portion of the process need be in memory for efficient execution. This portion is called the working set (Denning)
- Moreover, based on recent execution it is possible to predict with good accuracy which instructions/data will be accessed in the near future

Demand Paging

- Pages (of both text and data) are loaded into main memory as needed
- A page is placed in a page frame of main memory
- What if all the page frames are being used and a new page needs to be brought in?
  - A page that has not been accessed recently is selected for replacement
  - This page needs to be copied out to disk (unless we already have a copy there) before the new page is loaded
- The selection of such pages is called the page replacement algorithm

Illustration



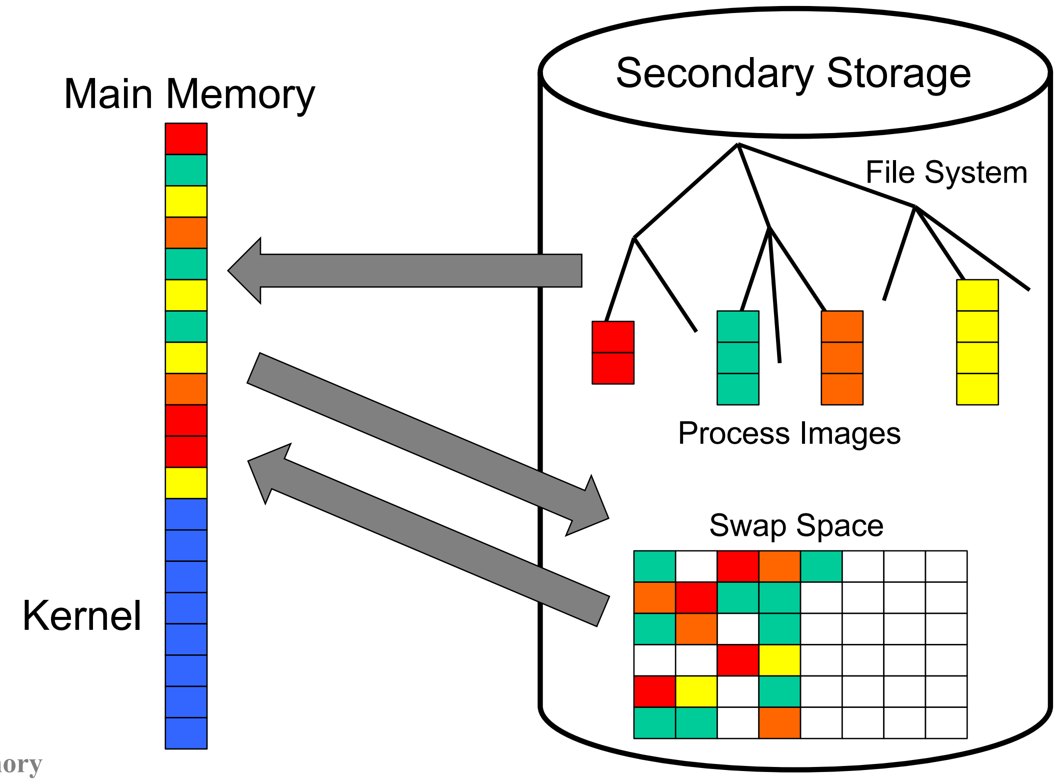
Swap Space

- Pages selected for replacement need to be stored on secondary storage (hard disk or solid-state drive).
- Unlike the previous figure, process images are NOT stored in contiguous areas of disk
- For performance reasons, the file system is often bypassed and parts of virtual memory (typically data areas) are stored in a special area of the disk called the swap space, or swap area
  - pages of the process in swap space are accessed directly by block numbers, as opposed to a hierarchical file system structure

Swap Space

- In older systems, both text and data (global data, stack) of active processes were stored in swap
- Nowadays, the OS uses the file system copy of text pages for "backing store" of text pages
- NOTE: Swap space is used only when needed.
  - data pages start in memory and might get swapped out
  - NOT the other way around
  - That is, we don't make a copy of the data of the process on swap and then page it in.
  - Data/stack pages get created in main memory, might get swapped out later.

A slightly more realistic illustration…



Questions:

- Why are images in the filesystem small?
- Why unidirectional transfer of pages from images?
- Why is total image size of each process smaller than before?

Possibility of Thrashing

- To accommodate as many processes as possible, only a few pages of each process are maintained in main memory
- But main memory may be full: when the OS brings one page in, it might need to swap another page out
- The OS should not swap out a page of a process just before that page is needed
- How can it avoid doing so?
- If this scenario occurs too often the result is thrashing:
  – The processor spends most of its time swapping pieces in and out of memory rather than executing user instructions
- Question: Can thrashing occur with only one user process?

Page Tables and Virtual Memory

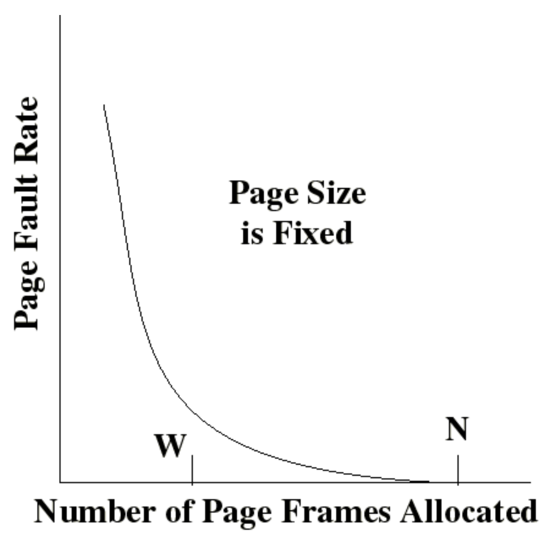
- Most computer systems support a very large virtual address space
  – 32 to 48 bits are used for virtual addresses
  – If (only) 32 bits are used with 4KB pages, a page table can have up to 2^20 entries, 1024 pages, 4MB
  – With 48-bit addresses, 2^36 entries, 128M pages, 512GB!
- The entire page table may take up too much main memory. Hence, increasingly page tables are often also stored in virtual memory and subjected to paging
  – Obviously, when a process is running, at least part of its page table must be in main memory (including the page table entry of the currently executing page)

Operating System Role in Paging

- OS memory management duties depend on whether the hardware supports paging, segmentation, or both
- Pure segmentation systems are obsolete. Segments (vm areas) are nearly always paged.
- Hence, OS memory management focuses on how to perform paging most effectively.
- To achieve good performance, a major goal is to minimize page fault rate.

Page Fault Rate

- Typically, page fault rate is indirectly related to the number of page frames allocated to a process
- Page faults drop to a reasonable value when W frames are allocated, where W is the working set size
- Drops to 0 when the number (N) of frames is such that the process is entirely in memory



Fetch Policy

- Determines when a page should be brought into main memory. Two common policies:
- On-demand only brings a page into main memory (if it is not already resident in memory) when a reference is made to a location in the page
  - many page faults occur when a process starts but rate should decrease as more pages are loaded
- Pre-paging brings in more pages than needed
  - locality of reference suggests that it is more efficient to bring in pages that reside contiguously in VM, especially if near each other on secondary storage
  - however, this efficiency not definitely established: the extra pages brought in are “often” not referenced

Placement policy

- Determines where in physical memory a newly loaded page of the process is placed
- Back in pure segmentation systems:
  - first-fit, next fit... are possible choices (a big issue)
- For paging:
  - the chosen frame location is **irrelevant** since all page frames are equivalent
  - typically, the OS maintains a “free list” identifying page frames that are available to be filled (overwritten?) by newly loaded pages

Replacement Policy

- Deals with the selection of a page in main memory to be replaced when a new page is brought in
  - NOT the actual writing back to disk of a page
- Earlier edition of your text: “This occurs whenever main memory is full (no free frame available).”
- In a real OS: Replacement occurs often, as the OS wants to have space ready for pages as they are accessed.
- Indeed, a **real** OS goes to great lengths to ensure that the **free list** does not become empty…

Replacement Policy

- Not all pages in main memory can be selected for replacement
- Some frames are locked (cannot be paged out):
  - much of the kernel is held in locked frames as well as key control structures and I/O buffers
- Two main policies for selecting page for replacement:
  - limited to those pages of the process that has suffered the page fault (local replacement)
  - the set of all pages in unlocked frames (global replacement)
  - Which is more common?
    global replacement

Basic algorithms for the replacement policy

- The Optimal policy selects for replacement the page for which the time to the next reference is longest.
- Produces the fewest number of page faults
- Problem? requires seeing the future
- Serves as a standard to compare with the other algorithms we shall study:
  - Least recently used (LRU)
  - First-in, first-out (FIFO)
  - Clock

The Least Recently Used (LRU) Policy

- Replaces the page that has not been referenced for the longest time
  - By the principle of locality, this should be the page least likely to be referenced in the near future
  - performs nearly as well as the optimal policy
- Simple example: A process of 5 pages, resident set size of 3

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2 3	2 3	2 3 1	2 3 5 F	2 3 5	4 3 5 F	4 3 5	4 3 5	2 3 5 F	2 3 5	2 3 5
LRU	2	2 3	2 3	2 3 1	2 5 1 F	2 5 1	2 5 4 F	2 5 4	3 5 4 F	3 5 2 F	3 5 2	3 5 2

Note on counting page faults

- When the main memory is empty, each new page we bring in is a result of a page fault
- For the purpose of comparing the different algorithms where the number of frames is F
  - we are not counting the initial F page faults
  - these are the same for all replacement algorithms
- But, in contrast to what is shown in the figures, these initial references are really producing page faults

Implementation of the LRU Policy

- Each page could be tagged (e.g., in the page table entry) with the time at each memory reference.
- The LRU page is the one with the smallest time value (needs to be searched at each page fault)
- This would require expensive hardware and a great deal of overhead.
- Consequently, very few computer systems provide sufficient hardware support for true LRU replacement policy
- Other algorithms (heuristics) are used instead

The First-In, First Out (FIFO) Policy

- Treats page frames allocated to a process as a circular buffer
- When the buffer (resident set) is full, the oldest page is replaced. Hence: first-in, first-out
  - This is not necessarily the same as the LRU page
  - Problem? page brought in early might be heavily used
- Simple to implement
  - requires only a pointer that circles through the page frames allocated to the process

Comparison of FIFO with LRU

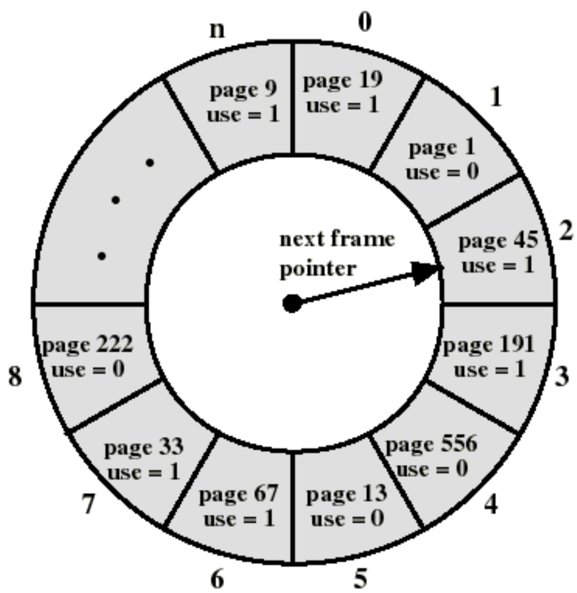
Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
LRU	2	2 3	2 3	2 3 1	2 5 1 F	2 5 1	2 5 4 F	2 5 4	3 5 4 F	3 5 2 F	3 5 2	3 5 2
FIFO	2	2 3	2 3	2 3 1	5 3 1 F	5 2 1 F	5 2 4 F	5 2 4	3 2 4 F	3 2 4	3 5 4 F	3 5 2 F

- LRU recognizes that pages 2 and 5 are referenced more frequently than others but FIFO does not
- FIFO performs relatively poorly

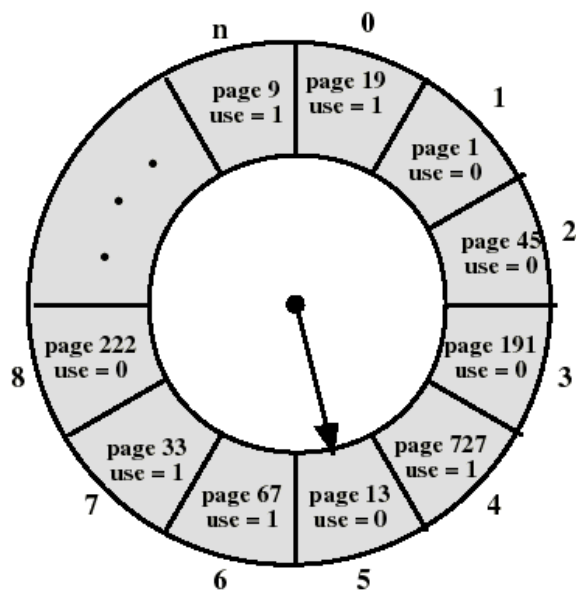
The Clock Policy

- The set of frames that are candidates for replacement is considered as a circular buffer
- When a page is replaced, a pointer is set to point to the next frame in buffer
- A use (reference) bit the frame set to 1 whenever
  - a page is first loaded into the frame
  - the corresponding page is referenced (TLB entry updated)
  - When it is time to replace a page, the first frame encountered with the use bit set to 0 is replaced.
  - During the search for replacement, each use bit set to 1 is changed to 0

The Clock Policy: An example

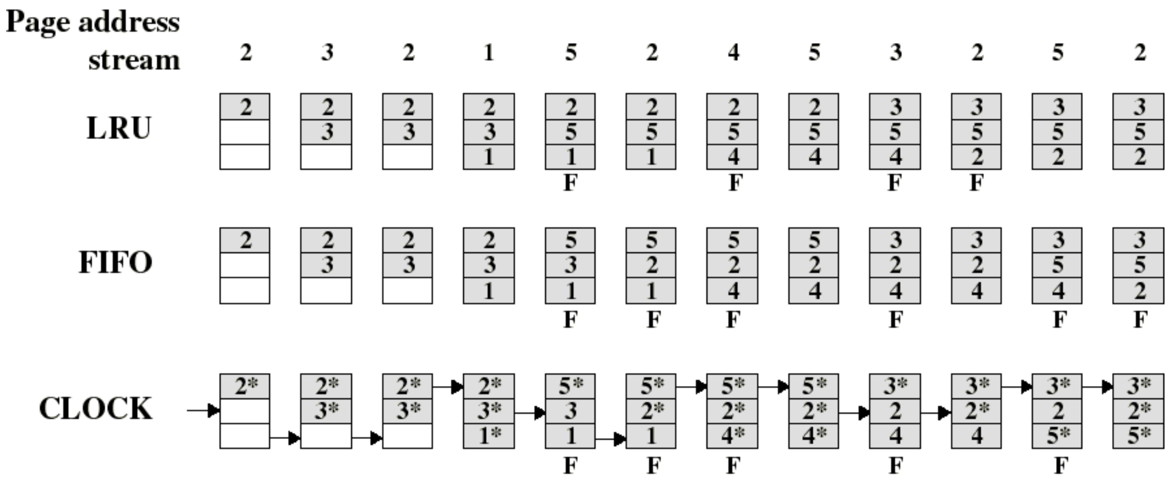


(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Comparison of Clock, FIFO and LRU



- Asterisk indicates that the corresponding use bit is set to 1
- Clock protects frequently referenced pages by setting the use bit to 1 at each reference

Comparison of Clock, FIFO and LRU

- Numerical experiments tend to show that performance of Clock is close to that of LRU
- Experiments have been performed when the number of frames allocated to each process is fixed and when pages local to the page-fault process are considered for replacement
  - When few (6 to 8) frames are allocated per process, there is almost a factor of 2 difference in page faults between LRU and FIFO
  - This factor reduces close to 1 when several (more than 12) frames are allocated. (But then more main memory is needed to support the same level of multiprogramming)

Cleaning Policy

- When should a modified page be written to disk?
- Demand cleaning
  - a page is written out only when its frame needs to be replaced (now!)
  - but now a process that suffers a page fault may have to wait for two page transfers
- Precleaning
  - modified pages eventually need to be written to disk
  - can we write them before we actually need the frames?
- A good compromise can be achieved with page buffering

Page Buffering (real operating systems)

- Pages to be replaced are kept in main memory for a while to guard against poorly performing replacement algorithms such as FIFO (bet hedging)
- Two lists of pointers are maintained: each entry points to a frame selected for replacement
  - a free page list for frames that have not been modified since brought in (no need to swap out)
  - a modified page list for frames that have been modified (need to write them out)

When a page is selected for replacement…

- A (pointer to the) frame to be replaced is added to the tail of either the free list or the modified list
- The present bit is cleared in the corresponding page table entry
- However, the page **remains** in the same page frame
- Pages on the modified list are periodically written to disk
  - in batches, for efficiency (elevator algorithm)
  - then moved to the free list (still in memory!)

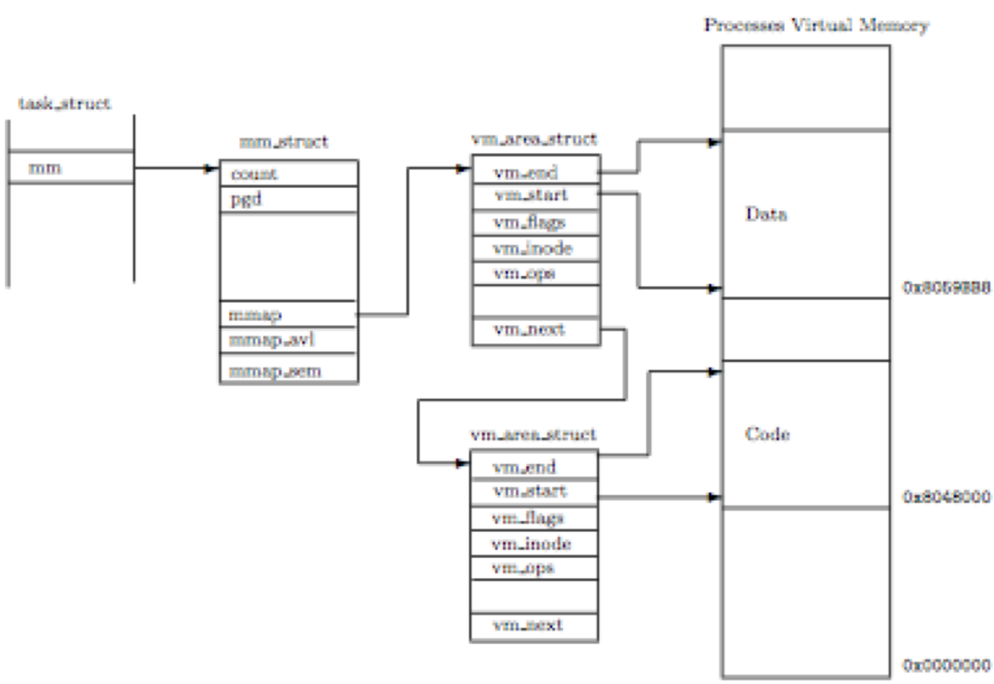
Page Buffering (cont.)

- At each page fault the two lists are first examined to see if the referenced page is still in main memory
  - If it is, we just need to set the present bit in the corresponding page table entry (and remove the entry in the relevant page list)
  - If it is not, then the needed page needs to be brought in from disk.
    - Page is placed in the frame pointed to by the head of the free frame list (overwriting the page that was there)
    - The head of the free frame list is moved to the next entry

Advantages of Page Buffering?

less I/O for page faults
hedge our bets, keep in memory in case referenced
more efficient disk writes

Linux Virtual Memory Data Structures



Linux VM Areas

- Since an area of memory may be associated with an image on disk, the `vm_area_struct` has inode pointer.
- Also, the `vm_ops` field points to the specific functions to be used to map and unmap this area, etc.
- A very important such operation is the `nopage()` operation, which specifies what to do when a page fault occurs (for example, bring in page from an image on disk)
- Different page fault routines may be applied to different areas.

Demand Paging

- When a page fault occurs, Linux will search the vm_area_structs to find which area is involved. Possibilities:
  – if no such area is found, what happens? segfault
  – legal area, but wrong operation. example? writing to RO segment
- Assuming address is legal, Linux checks to see if the page is
  – in swap file (page table entry is marked invalid but address is not empty)
  – in an executable image on the file system, associated with the inode in the corresponding vm_area_struct
- Issues appropriate disk read operation.

Kernel Swap Daemon (kswapd)

- Description
  - job is to keep enough free pages in system for Linux's needs
  - kernel thread – executes and remains in kernel mode
- Operation
  - periodically wakes up and makes sure the number of free pages is not too low
  - tries to free up 4 pages each time it runs
  - adds unmodified pages to free list and if necessary writes modified pages to secondary storage

Summary

- VM serves multiple purposes
  - address space protection
  - address space larger than physical memory
  - solves placement problem
- In general-purpose systems, VM is coupled with demand paging and usually the concept of segments (for permissions, etc.)
- VM requires support from both the processor... and the **operating system!**