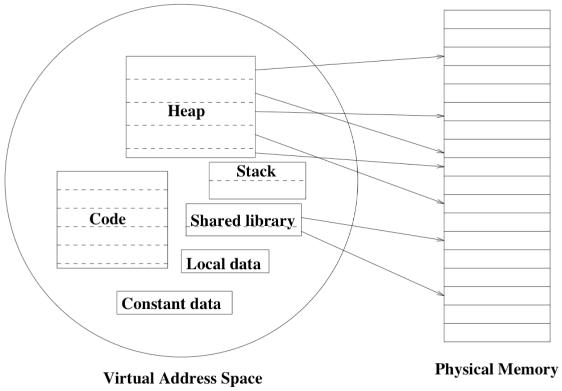
Last Class: Paging & Segmentation

- Paging: divide memory into fixed-sized pages, map to frames (OS view of memory)
- Segmentation: divide process into logical 'segments' (compiler view of memory
- Combine paging and segmentation by paging individual segments



Today: Demand Paged Virtual Memory

- Up to now, the virtual address space of a process fit in memory, and **we assumed it was all in memory.**

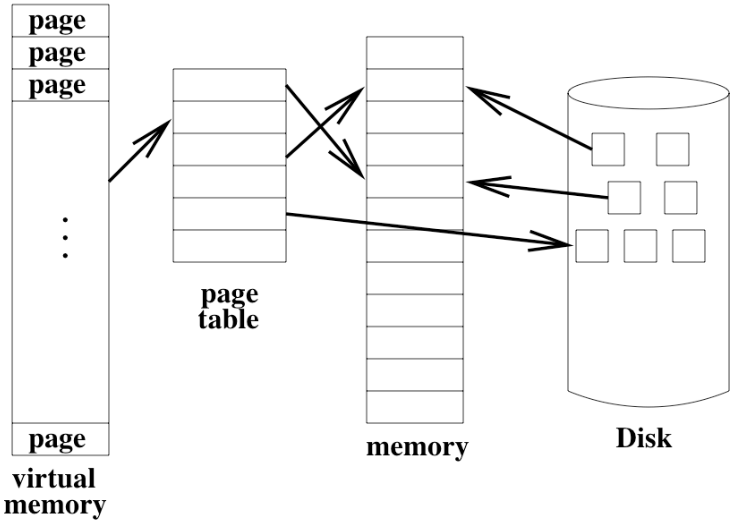
- OS illusions
  1. treat disk (or other backing store) as a much larger, but much slower main memory
  2. analogous to the way in which main memory is a much larger, but much slower, cache or set of registers

- The illusion of an infinite virtual memory enables
  1. a process to be larger than physical memory, and
  2. a process to execute even if all of the process is not in memory
  3. Allow more processes than fit in memory to run concurrently.

Demand Paged Virtual Memory

- Demand Paging uses a memory as a cache for the disk
- The page table (memory map) indicates if the page is on disk or memory using a valid bit
- Once a page is brought from disk into memory, the OS updates the page table and the valid bit
- For efficiency reasons, memory accesses must reference pages that are in memory the vast majority of the time
  - Else the effective memory access time will approach that of the disk
- **Key Idea:** Locality---the *working set* size of a process must fit in memory, and must stay there. (90/10 rule.)

Demand Paged Virtual Memory



When to load a page?

- **At process start time:** the virtual address space must be no larger than the physical memory.
- **Overlays:** application programmer indicates when to load and remove pages.
  - Allows virtual address space to be larger than physical address space
  - Difficult to do and is error-prone
- **Request paging:** process tells an OS before it needs a page, and then when it is through with a page.

When to load a page?

- **Demand paging:** OS loads a page the first time it is referenced.
  - May remove a page from memory to make room for the new page
  - Process must give up the CPU while the page is being loaded
  - *Page-fault:* interrupt that occurs when an instruction references a page that is not in memory.
- **Pre-paging:** OS guesses in advance which pages the process will need and pre-loads them into memory
  - Allows more overlap of CPU and I/O if the OS guesses correctly.
  - If the OS is wrong => page fault
  - Errors may result in removing useful pages.
  - Difficult to get right due to branches in code.

Implementation of Demand Paging

- A copy of the entire program must be stored on disk. (Why?)
- Valid bit in page table indicates if page is in memory.

  1: in memory 0: not in memory (either on disk or bogus address)
- When referenced, if the page is not in memory, trap to the OS
- The OS checks that the address is valid. If so, it
  1. selects a page to replace (page replacement algorithm)
  2. invalidates the old page in the page table
  3. starts loading new page into memory from disk
  4. context switches to another process while I/O is being done
  5. gets interrupt that page is loaded in memory
  6. updates the page table entry
  7. continues faulting process (why not continue current process?)

Swap Space

- What happens when a page is removed from memory?
  - If the page contained code, we could simply remove it since it can be reloaded from the disk.
  - If the page contained data, we need to save the data so that it can be reloaded if the process it belongs to refers to it again.
  - *Swap space:* A portion of the disk is reserved for storing pages that are evicted from memory
- At any given time, a page of virtual memory might exist in one or more of:
  - The file system
  - Physical memory
  - Swap space
- Page table must be more sophisticated so that it knows where to find a page

Performance of Demand Paging

- Theoretically, a process could access a new page with each instruction.
- Fortunately, processes typically exhibit *locality of reference*
  - **Temporal locality:** if a process accesses an item in memory, it will tend to reference the same item again soon.
  - **Spatial locality:** if a process accesses an item in memory, it will tend to reference an adjacent item soon.

- Let p be the probability of a page fault ($0 \leq p \leq 1$).
- Effective access time = $(1\text{-}p)$ x $ma + p$ x page fault time
  - If memory access time is 200 ns and a page fault takes 20 ms
  - Effective access time = $(1\text{-}p)$ x $200 + p$ x 20,000,000
- If we want the effective access time to be only 10% slower than memory access time, what value must p have?
  - 1.1 x 200 = (1-p) x 200 + p x 20,000,000
  - p = 1 / 100,000

Updating the TLB

- In some implementations, the hardware loads the TLB on a TLB miss.
- If the TLB hit rate is very high, use software to load the TLB
  1. Valid bit in the TLB indicates if page is in memory.
  2. on a TLB hit, use the frame number to access memory
  3. trap on a TLB miss, the OS then
     a) checks if the page is in memory
     b) if page is in memory, OS picks a TLB entry to replace and then fills it in the new entry
     c) if page is not in memory, OS picks a TLB entry to replace and fills it in as follows
        i. invalidates TLB entry
        ii. perform page fault operations as described earlier
        iii. updates TLB entry
        iv. restarts faulting process

All of this is still functionally transparent to the user.

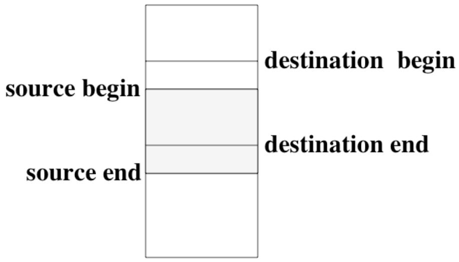
Transparent Page Faults

How does the OS transparently restart a faulting instruction?

- Need hardware support to save
  1. the faulting instruction,
  2. the CPU state.
- What about instructions with side-effects? (CISC)
  - *move -(r10), a* : decrements register 10, then moves r10 to address *a*
- Solution: unwind side effects

Transparent Page Faults

- Block transfer instructions where the source and destination overlap can't be undone.



- Solution: check that all pages between the starting and ending addresses of the source and destination are in memory before starting the block transfer

Page Replacement Algorithms

On a page fault, we need to choose a page to evict

Random: amazingly, this algorithm works pretty well.

- **FIFO:** First-In, First-Out. Throw out the oldest page. Simple to implement, but the OS can easily throw out a page that is being accessed frequently.
- **MIN:** (a.k.a. OPT). Throw out the page that will not be accessed for the longest time (provably optimal [Belady'66]).
- **LRU:** Least Recently Used. Approximation of MIN that works well if the recent past is a good predictor of the future. Throw out the page that has not been used in the longest time.

Example: FIFO

3 physical Frames

4 virtual Pages: A B C D

Reference stream: A B C A B D A D B C A

FIFO: First-In-First-Out

	A	B	C	A	B	D	A	D	B	C	A
frame 1	A*	A	A	A	A	D*	D	D	D	C*	C
frame 2		B*	B	B	B	B	A*	A	A	A	A
frame 3			C*	C	C	C	C	C	B*	B	B

Number of page faults? **7**

Example: MIN

MIN: Look into the future and throw out the page that will be accessed farthest in the future.

	A	B	C	A	B	D	A	D	B	C	A
frame 1	**A***	A	A	A	A	A	A	A	A	A	C
frame 2		**B***	B	B	B	B	B	B	B	**C***	B
frame 3			**C***	C	C	**D***	D	D	D	D	D

Number of page faults? **5**

Example: LRU

- **LRU:** Least Recently Used. Throw out the page that has not been used in the longest time.

	A	B	C	A	B	D	A	D	B	C	A
frame 1	**A***	A	A	A	A	A	A	A	A	**C***	C
frame 2		**B***	B	B	B	B	B	B	B	B	B
frame 3			**C***	C	C	**D***	D	D	D	D	**A***

Number of page faults? **6**

Example: LRU

- When will LRU perform badly?

	A	B	C	D	A	B	C	D	A	B	C
frame 1	**A***	A	A	**D***	D	D	**C***	C	C	**B***	B
frame 2		**B***	B	B	**A***	A	A	**D***	D	D	**C***
frame 3			**C***	C	C	**B***	B	B	**A***	A	A

- Number of page faults? **11**

Adding Memory: FIFO

Does adding memory always reduce the number of page faults?

FIFO:

	A	B	C	D	A	B	E	A	B	C	D	E
frame 1												
frame 2												
frame 3												
frame 1												
frame 2												
frame 3												
frame 4												

- **Belady's Anomaly:** Adding page frames may actually cause **more** page faults with certain types of page replacement algorithms (such as FIFO).

Adding Memory: LRU

LRU:

	A	B	C	D	A	B	E	A	B	C	D	E
frame 1	**A***	A	A	**D***	D	D	**E***	E	E	**C***	C	C
frame 2		**B***	B	B	**A***	A	A	A	A	A	**D***	D
frame 3			**C***	C	C	**B***	B	B	B	B	B	B
frame 1	**A***	A	A	A	A	A	A	A	A	A	A	**E***
frame 2		**B***	B	B	B	B	B	B	B	B	B	B
frame 3			**C***	C	C	C	**E***	E	E	E	**D***	D
frame 4				**D***	D	D	D	D	D	**C***	C	C

- With LRU, increasing the number of frames always decreases the number of page faults. Why?

Implementing LRU:

- **Perfect LRU:**
  - Keep a time stamp for each page with the time of the last access. Throw out the LRU page. **Problems?**
    - OS must record time stamp for each memory access, and to throw out a page the OS has to look at all pages. Expensive!
  - Keep a list of pages, where the front of the list is the most recently used page, and the end is the least recently used.
    - On a page access, move the page to the front of the list. Doubly link the list. **Problems?**
      - Still too expensive, since the OS must modify multiple pointers on each memory access

Approximations of LRU

- **Hardware Requirements:** Maintain reference bits with each page.
  - On each access to the page, the hardware sets the reference bit to '1'.
  - Set to 0 at varying times depending on the page replacement algorithm.
- **Additional-Reference-Bits:** Maintain more than 1 bit, say 8 bits.
  - On reference, set high bit to 1
  - At regular intervals or on each memory access, shift the byte right, placing a 0 in the high order bit.
  - On a page fault, the lowest numbered page is kicked out.

=> Approximate, since it does not guarantee a total order on the pages.

=> Faster, since setting a single bit on each memory access.

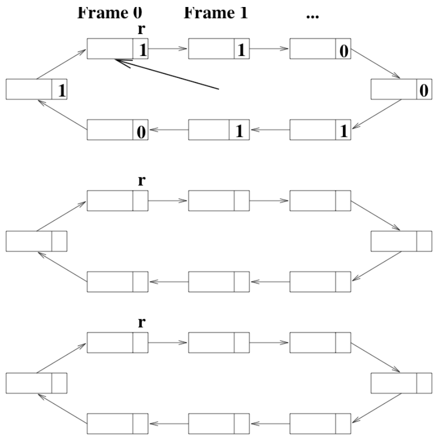
- Page fault still requires a search through all the pages.

Second Chance Algorithm: (a.k.a. Clock)

Use a single reference bit per page.

1. OS keeps frames in a circular list.
2. On reference, set page reference bit to 1
3. On a page fault, the OS
   a) Checks the reference bit of the next frame.
   b) If the reference bit is ‘0’, replace the page, and set its bit to ‘1’.
   c) If the reference bit is ‘1’, set bit to ‘0’, and advance the pointer to the next frame

Clock Example



=> One way to view the clock algorithm is as a crude partitioning into two categories: young and old pages.

- Why not partition pages into more than two categories?

Second Chance Algorithm

- Less accurate than additional-reference-bits, since the reference bit only indicates if the page was used at all since the last time it was checked by the algorithm.
- Fast, since setting a single bit on each memory access, and no need for a shift.
- Page fault is faster, since we only search the pages until we find one with a '0' reference bit.
- Simple hardware requirements.

Will it always find a page?

What if all bits are '1'?

Enhanced Second Chance

- It is cheaper to replace a page that has not been written
  - OS need not be write the page back to disk

  => OS can give preference to paging out un-modified pages

- Hardware keeps a *modify* bit (in addition to the reference bit)

  ‘1’: page is modified (different from the copy on disk)

  ‘0’: page is the same as the copy on disk

Enhanced Second Chance

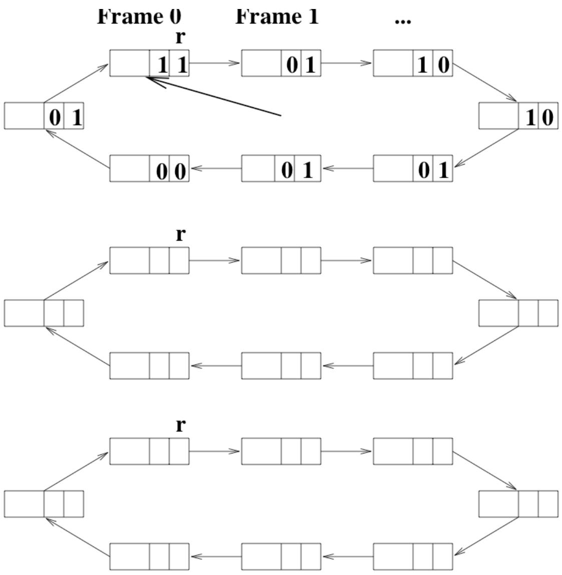
- The reference bit and modify bit form a pair (r,m) where
  1. (0,0) neither recently used nor modified - replace this page!
  2. (0,1) not recently used but modified - not as good to replace, since the OS must write out this page, but it might not be needed anymore.
  3. (1,0) recently used and unmodified - probably will be used again soon, but OS need not write it out before replacing it
  4. (1,1) recently used and modified - probably will be used again soon and the OS must write it out before replacing it

- On a page fault, the OS searches for the first page in the lowest nonempty class.

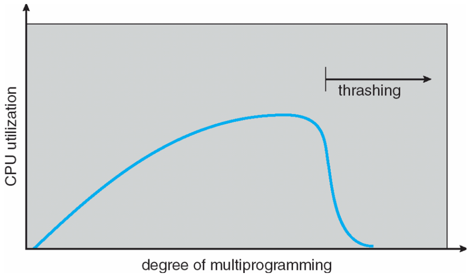
Page Replacement in Enhanced Second Chance

- The OS goes around at most three times searching for the (0,0) class.
  1. Page with (0,0) => replace the page.
  2. Page with (0,1) => initiate an I/O to write out the page, locks the page in memory until the I/O completes, clears the modified bit, and continue the search
  3. For pages with the reference bit set, the reference bit is cleared.
  4. If the hand goes completely around once, there was no (0,0) page.
     - On the second pass, a page that was originally (0,1) or (1,0) might have been changed to (0,0) => replace this page
     - If the page is being written out, waits for the I/O to complete and then remove the page.
     - A (0,1) page is treated as on the first pass.
     - By the third pass, all the pages will be at (0,0).

Clock Example



Multiprogramming and Thrashing



- **Thrashing:** the memory is over-committed and pages are continuously tossed out while they are still in use
  - memory access times approach disk access times since many memory references cause page faults
  - Results in a serious and very noticeable loss of performance.
- What can we do in a multiprogrammed environment to limit thrashing?

Replacement Policies for Multiprogramming

- **Proportional allocation:** allocate more page frames to large processes.
  - alloc = s/S * m
- **Global replacement:** put all pages from all processes in one pool so that the physical memory associated with a process can grow
  - **Advantages:** Flexible, adjusts to divergent process needs
  - **Disadvantages:** Thrashing might become even more likely (Why?)

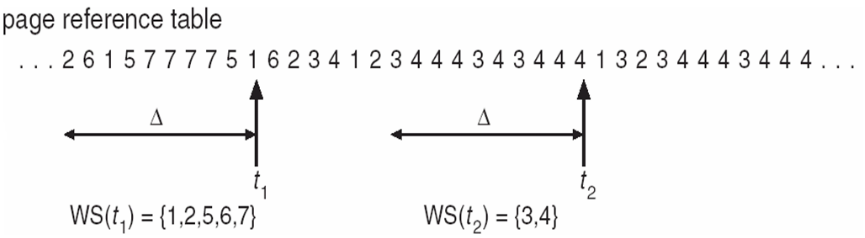
Replacement Policies for Multiprogramming

- **Per-process replacement:** Each process has its own pool of pages.
- Run only groups of processes that fit in memory, and kick out the rest.
- How do we figure out how many pages a process needs, i.e., its working set size?
  - Informally, the working set is the set of pages the process is using right now
  - More formally, it is the set of all pages that a process referenced in the past T seconds
- How does the OS pick T?
  - 1 page fault = 10msec
  - 10msec = 2 million instructions

=> T needs to be a whole lot bigger than 2 million instructions.

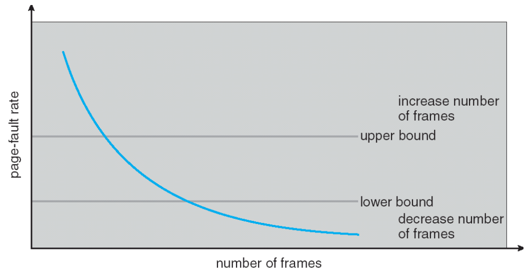
  - What happens if T is too small? too big?

Working Set Determination



Per-process Replacement

- Working sets are expensive to compute => track page fault frequency of each process instead
  - If the page fault frequency > some threshold, give it more page frames.
  - If the page fault frequency < a second threshold, take away some page frames
- **Goal:** the system-wide mean time between page faults should be equal to the time it takes to handle a page fault.
  - May need to suspend a process until overall memory demands decrease.



Page-fault Frequency Scheme

- **Advantages:** Thrashing is less likely as process only competes with itself. More consistent performance independent of system load.
- **Disadvantages:** The OS has to figure out how many pages to give each process and if the working set size grows dynamically adjust its allocation.

Page Sizes

- Reasons for small pages:
  - More effective memory use.
  - Higher degree of multiprogramming possible.
- Reasons for large pages:
  - Smaller page tables
  - Amortizes disk overheads over a larger page
  - Fewer page faults (for processes that exhibit locality of references)
- Page sizes are growing because:
  - Physical memory is cheap. As a result, page tables could get huge with small pages. Also, internal fragmentation is less of a concern with abundant memory.
  - CPU speed is increasing faster than disk speed. As a result, page faults result in a larger slow down than they used to. Reducing the number of page faults is critical to performance.

Summary of Page Replacement Algorithms

- Unix and Linux use variants of Clock, Windows NT uses FIFO.
- Experiments show that all algorithms do poorly if processes have insufficient physical memory (less than half of their virtual address space).
- All algorithms approach optimal as the physical memory allocated to a process approaches the virtual memory size.
- The more processes running concurrently, the less physical memory each process can have.
- A critical issue the OS must decide is how many processes and the frames per process that may share memory simultaneously.

Summary

Benefits of demand paging:

- Virtual address space can be larger than physical address space.
- Processes can run without being fully loaded into memory.
  - Processes start faster because they only need to load a few pages (for code and data) to start running.
  - Processes can share memory more effectively, reducing the costs when a context switch occurs.
- A good page replacement algorithm can reduce the number of page faults and improve performance

Summary

- Allocating memory **within** a process and **across** processes
- Page Replacement Algorithms
- LRU approximations:
  - Second Chance
  - Enhanced Second Chance
- Hardware support for page replacement algorithms
- Replacement policies for multiprogramming