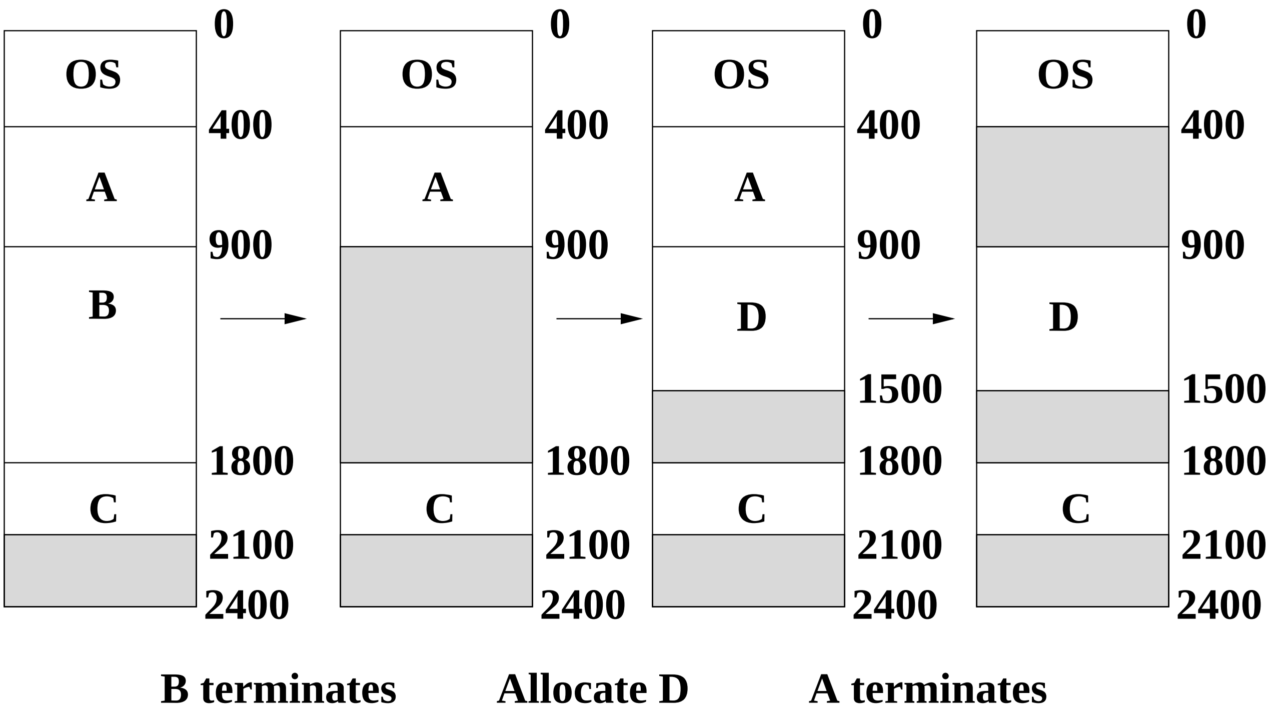
What is Worst-fit? ...Topics in Dynamic Memory Allocation

Assume dynamic relocation!

As processes enter the system, grow, and terminate, the OS must keep track of which memory is available and utilized.



Holes: pieces of free memory (shaded above in figure)

Given a memory request from a starting process: the OS must decide which hole to use for the process.

Memory Allocation Policies

First-Fit: allocate the first one in the list in which the process fits. The search can start with the first hole, or where the previous first-fit search ended.

Best-Fit: Allocate the smallest hole that is big enough to hold the process. The OS must search the entire list or store the list sorted by size.

Worst-Fit: Allocate the largest hole to the process. Again the OS must search the entire list or keep the list sorted.

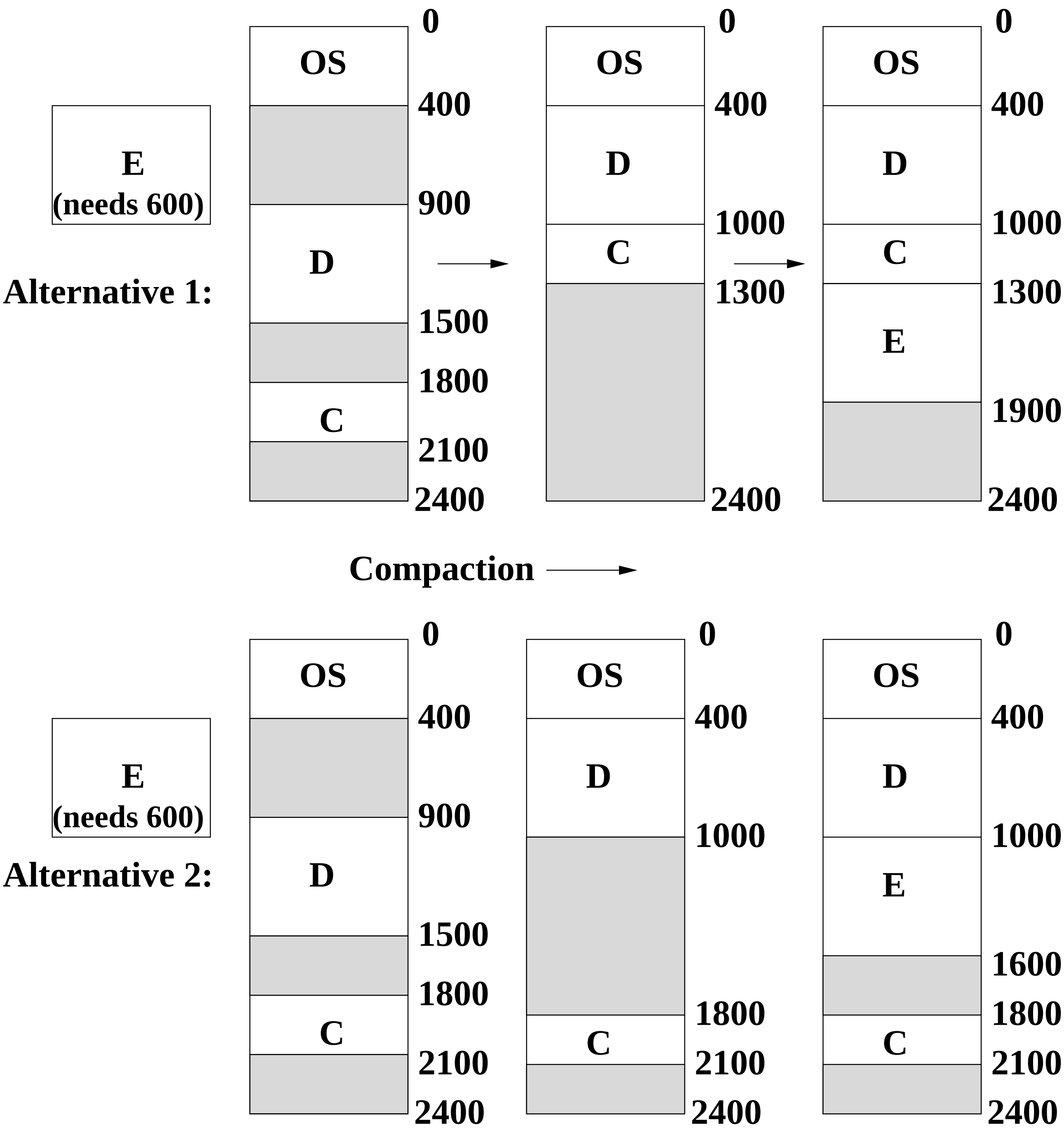
$\Longrightarrow$ What might you expect the distribution of the size of the holes to be over time?

$\Longrightarrow$ Simulations show first-fit and best-fit usually yield better storage utilization than worst-fit; first-fit is generally faster than best-fit.

External Fragmentation - holes between processes.

- There may be enough memory to fit a process in memory, but because it is fragmented, the process must wait.
- Simulations with first-fit show up to 1/3 of memory may be lost to fragmentation, i.e., the pieces of free memory are too small to fit a process.
- We want an allocation policy that minimizes wasted space.

Compaction



E (needs 600)
Alternative 1:
OS
D
C
E
0
400
900
1000
1300
1500
1800
1900
2100
2400
Compaction
E (needs 600)
Alternative 2:
OS
D
C
E
0
400
900
1000
1500
1600
1800
2100
2400

Swapping

- Roll out a process to disk, releasing all the memory it holds.
- When process becomes active again, the OS must reload it in memory.
  - With static relocation, the process must be put in the same position.
  - With dynamic relocation, the OS finds a new position in memory for the process and updates the relocation and limit registers.
- If swapping is part of the system, compaction is easy to add.
- How could or should swapping interact with CPU scheduling?

Paging

Processes typically do not use their entire space in memory all the time. Paging

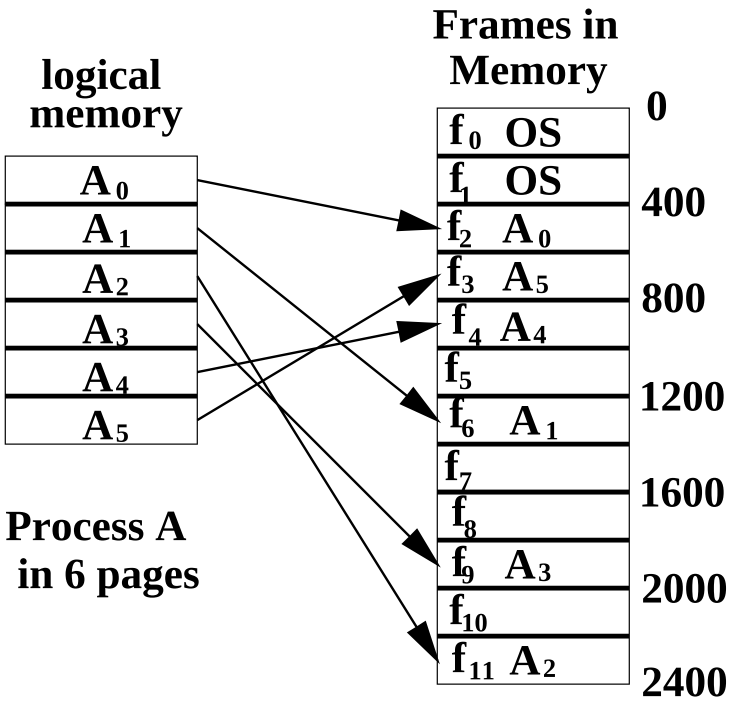
1. divides and assigns processes to fixed sized *pages*,
2. then selectively allocates pages to *frames* in memory, and
3. manages (moves, removes, reallocates) pages in memory.

Motivation & Features of Paging

90/10 rule: Processes spend 90% of their time accessing 10% of their space in memory.

$\Longrightarrow$ Implication: we only want to keep those parts of a process in memory that are actually being used.

- Pages greatly simplify the hole fitting problem.
- The logical memory of the process is contiguous, but pages need not be allocated contiguously in memory.



- By dividing memory into fixed size pages, we can eliminate external fragmentation. Of course, we introduce some internal fragmentation

Paging Hardware

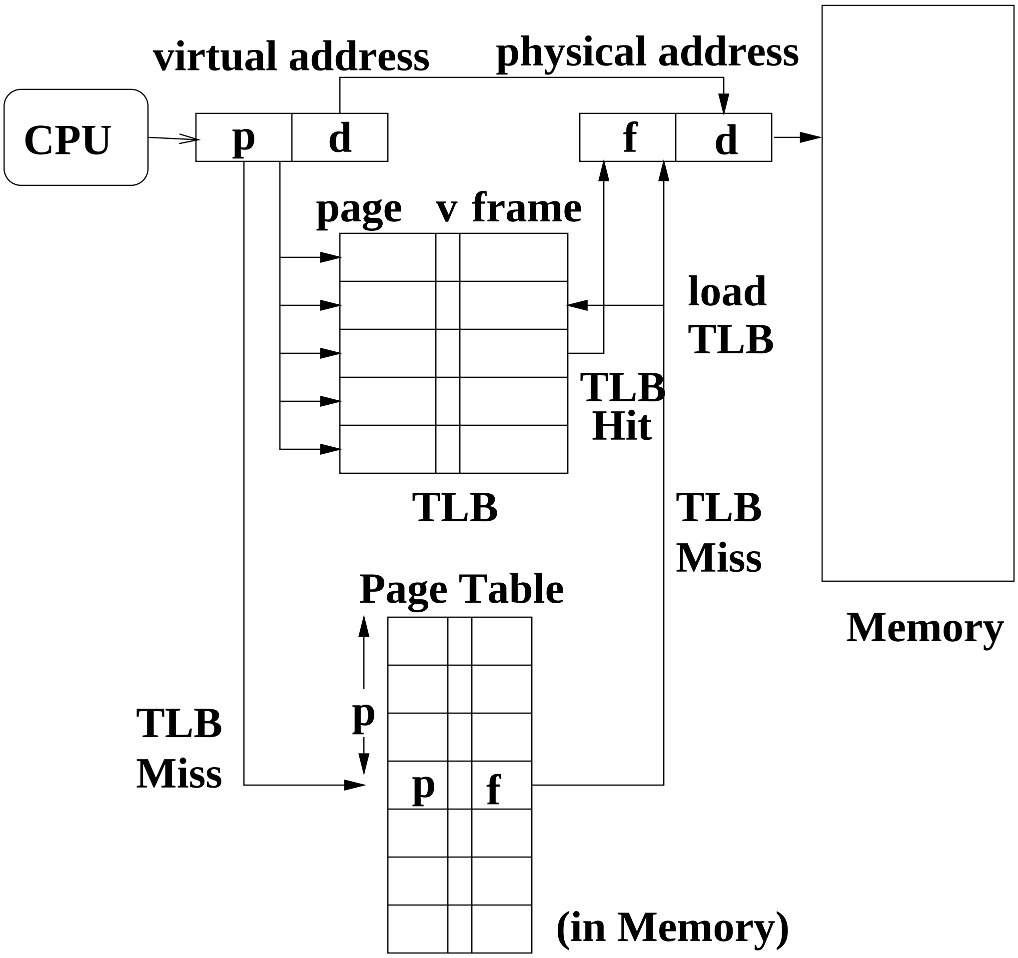
- The page size (frame size), like memory sizes, is typically a power of 2 between 512 bytes and 8192 bytes per page.
- Powers of 2 make the translation of virtual addresses into physical addresses easier. For example, given

  1. virtual address space of size 2^m (bytes or words)
  2. a page of size 2^n, then
  3. the high order $m - n$ bits of a virtual address select the page,
  4. the low order n bits select the offset in the page.

p	d
m-n	**n**

p: page number
d: page offset

The Translation Look-aside Buffer (TLB)



v: valid bit that says the entry is up-to-date

Computing Relative Costs

1. What is the effective memory access cost if the page table is in memory?

2. What is the effective memory access cost with a TLB?

Initializing Memory when Starting a Process

1. Process needing k pages arrives.
2. If k page frames are free, the OS allocates all k pages to the free frames. Otherwise, we need to free a frame that is no longer needed.
3. The OS puts the first page in a frame, and puts the frame number in the first entry in the page table; it then puts the second page in the frame and its frame number in the second entry of the page table and so on.
4. OS marks all TLB entries as invalid (flushes the TLB).
5. OS starts process.
6. As process executes, OS loads TLB entries as each page is accessed, replacing an existing entry if the TLB is full.

Saving/Restoring Memory on a Context Switch

The Process Control Block (PCB) must be extended to contain:

- The page table
- Possibly a copy of the TLB

1. Copy the page table base register value to the PCB.
2. Copy the TLB to the PCB (optionally).
3. Flush the TLB.
4. Restore the page table base register.
5. Restore the TLB if it was saved.

Memory sharing across processes

Paging introduces the possibility of sharing since the memory used by a process no longer needs to be contiguous.

- Shared code must be reentrant, that means the processes that are using it cannot change it (e.g., no data in reentrant code).
- Sharing of pages is similar to the way threads share text and memory with each other.
- A shared page may exist in different parts of the virtual address space of each process, but the virtual addresses map to the same physical address.
- The user program (e.g., emacs) marks text segment of a program as reentrant with a system call.
- The OS keeps track of available reentrant code in memory and reuses them if a new process requests the same program.
- Can greatly reduce overall memory requirements for commonly used applications.

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. Memory is as big and cheap as disk
  2. Memory is as fast as DRAM (cache adds the illusion that memory is as fast as SRAM)
- Put another way
  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. more processes than fit in memory to run concurrently.

Demand Paged Virtual Memory

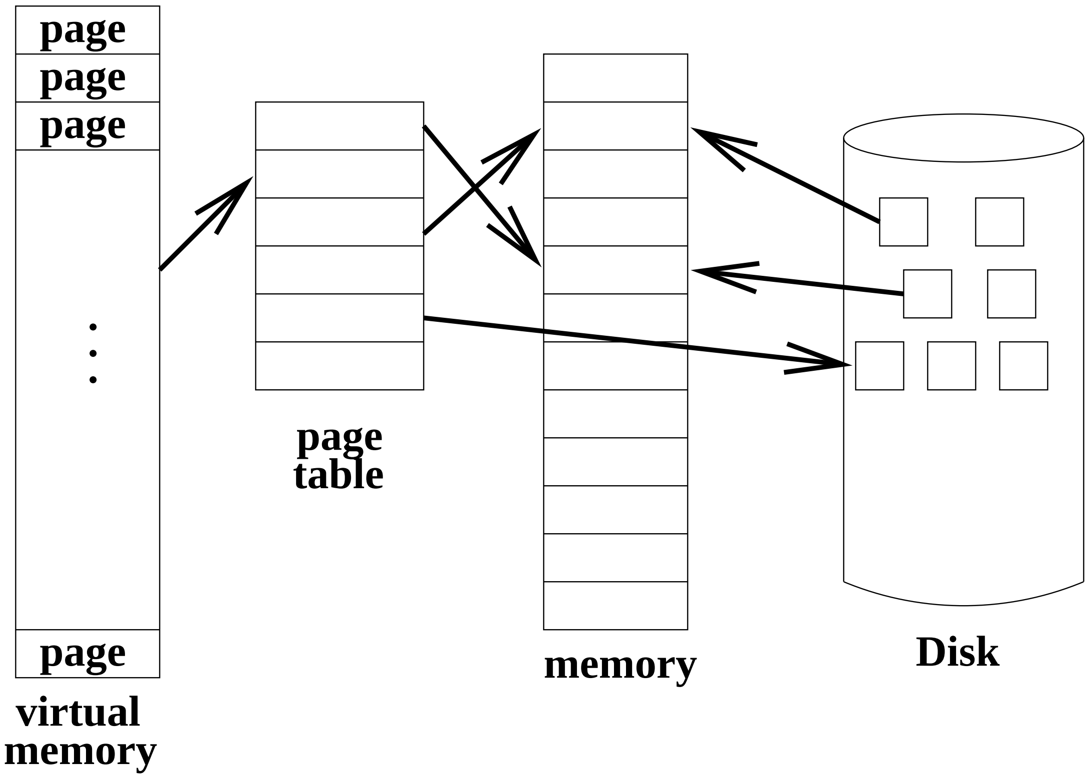
Idea – Demand Paging uses memory as a cache for disk.

The page table (memory map) indicates if the page is on disk, and thus is not in a physical page frame in memory right now using a valid bit. To access this page, the OS needs to put the page in a physical page frame. Once the page is in a frame, the OS updates the page table to reflect the placement.

To effectively use large virtual address spaces, we need an efficient implementation that finds the page in memory the vast majority of the time, otherwise the effective memory access time will approach that of the disk.

Key – 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?

When the process starts: This is what we have talked about so far. 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, but it is painful to do and is error-prone.

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. A *page-fault* is the name for the interrupt that occurs when an instruction references a page that is not in memory. Process must give up the CPU while the page is being loaded.

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, the page is unnecessarily faulted in. Errors also may result in removing useful pages.

Implementation of Demand Paging

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

   1: in memory

   0: not in memory (either on disk or bogus address)
3. If the page is not in memory, the hardware traps to the OS on the reference to the invalid page.
4. The OS checks that the address is valid. Assuming it is, it...
   (a) selects a page to replace (page replacement algorithm)
   (b) invalidates the old page in the page table
   (c) starts loading new page into memory from disk
   (d) switches to another process while I/O is being done
   (e) gets interrupt that page is loaded in memory
   (f) updates the page table entry
   (g) continues faulting process (why not continue current process?)

All of this is functionally transparent to the user (it may not be performance transparent!). The OS keeps the processor busy by running another job.

Swap Space

What happens when a page is removed from memory to make room for a new page?

- If the page contained code, we could simply remove it since it can be re-loaded 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.
- A portion of the disk is reserved as *swap space* where pages are moved when they are forced to give up their physical frame.

$\Longrightarrow$ At any given time, a page of virtual memory might exist in one or more of:

- The file system
- Physical memory
- Swap space

$\Longrightarrow$ 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 (or more!) of memory with each instruction.
- Fortunately, processes typically exhibit *locality of reference* which yields reasonable performance for demand paging.
- Locality:

  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 = $(1 - p) \times$ ma + $p\times$ page fault time

  Assume memory access time is 200 nanoseconds while it takes 25 milliseconds to service a page fault. Then...

  Effective access time = $(1 - p) \times$ 200 + $p\times$ 25,000,000

  If we want the effective access time to be only 10% slower than memory access time, what value must p have?

Software-Loaded TLB with Virtual Memory

We have to keep the page table up-to-date too.

- In some implementations, the hardware loads the TLB on a TLB miss.
- If the TLB hit rate is very high, we can afford to have the software load the TLB.
- An implementation,
  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.

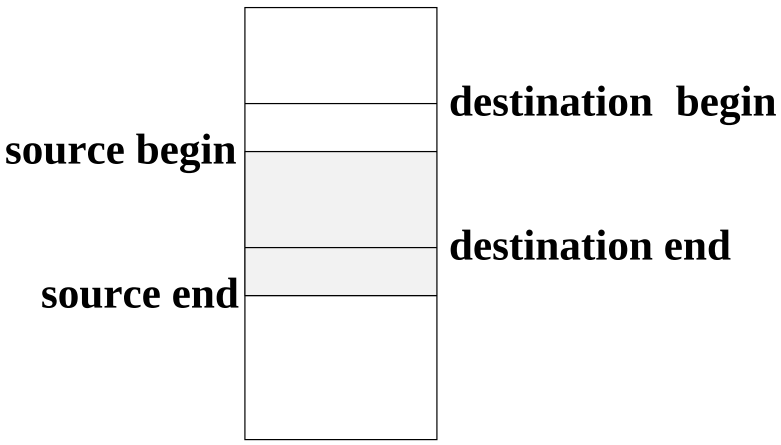
Transparent Page Faults

How does the OS transparently restart a faulting instruction?

- Hardware has to help out by saving (1) the faulting instruction, and (2) the CPU state.
- What about instructions with side-effects? (CISC)

  mov a, (r10)+

  Moves a into the address contained in register 10 and increments register 10.
- Solution: unwind side effects
- 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

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.

Next time: page replacement algorithms—if we must bring in a page from disk, we must first kick out a page currently in physical memory. How do we choose the victim? (Experimental evaluation of various strategies is the goal of our project 2.)