# Putting it all together

- **Relocation** using Base and Limit registers
  - simple, but inflexible
- **Segmentation:**
  - compiler's view presented to OS
  - segment tables tend to be small
  - memory allocation is expensive and complicated (first fit, worst fit, best fit).
  - compaction is needed to resolve external fragmentation.



# Putting it all together

- **Paging:**
  - simplifies memory allocation since any page can be allocated to any frame
  - page tables can be very large (especially when virtual address space is large and pages are small)
- **Segmentation & Paging**
  - only need to allocate as many page table entries as we need (large virtual address spaces are not a problem).
  - easy memory allocation, any frame can be used
  - sharing at either the page or segment level
  - increased internal fragmentation over paging
  - two lookups per memory reference

# Today: Demand Paged Virtual Memory

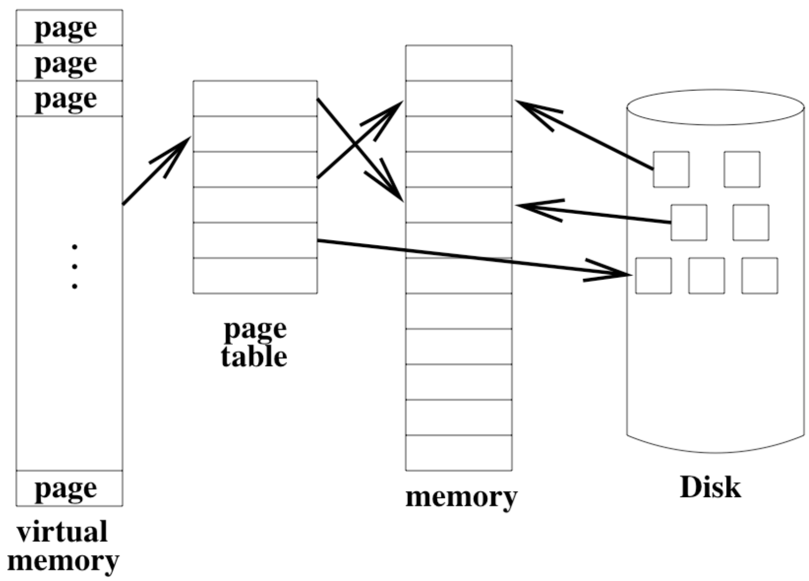
- Up to now, the virtual address space of a process fit in memory, and **we assumed it was all in memory.**
- OS illusions:
  1. treat disk (or other backing store) as a much larger, but much slower main memory
  2. analogous to the way in which main memory is a much larger, but much slower, cache or 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)
- If the page is not in memory, trap to the OS on first the reference
- 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 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.
  - *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 = $(1-p) \times ma + p \times$ page fault time
  - If memory access time is 200 ns and a page fault takes 25 ms
  - 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?

# 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.



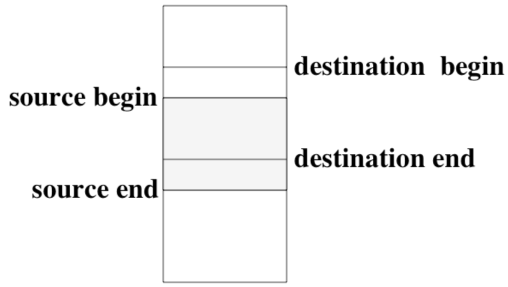
# Transparent Page Faults

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)
  – mov a, (r10)+ : moves a into the address contained in register 10 and increments register 10.
- 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) Look into the future and throw out the page that will be accessed farthest in the future (provably optimal [Belady'66]). Problem?
- **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 page Frames

4 virtual Pages: A B C D

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

**FIFO:** First-In-First-Out

| | A | B | C | A | B | D | A | D | B | C | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | |

Number of page faults?



# 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 | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | |

Number of page faults?

# 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 | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | |

Number of page faults?



# Example: LRU

• When will LRU perform badly?

| | A | B | C | A | B | D | A | D | B | C | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| frame 1 | | | | | | | | | | | |
| frame 2 | | | | | | | | | | | |
| frame 3 | | | | | | | | | | | |

# 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