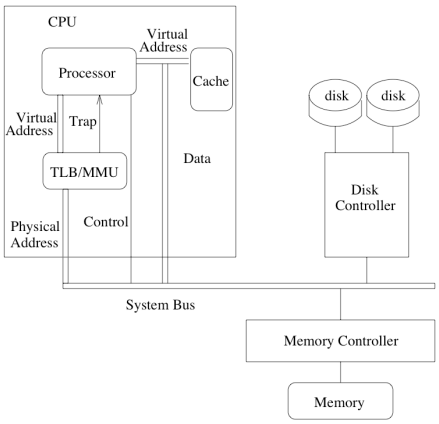
Memory Management Refresher

- Review of memory management techniques

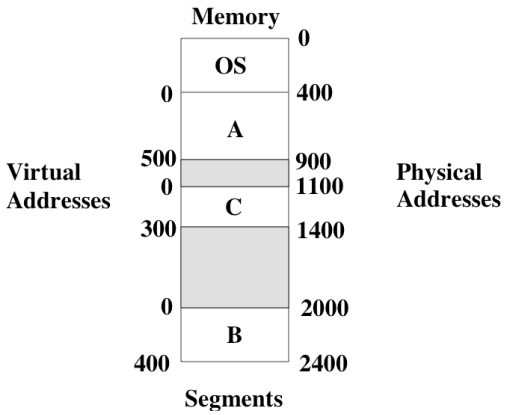
- Contiguous memory allocation
- Paging
- Segmentation
- Demand-paged Virtual Memory

Background: Computer Architecture



- Program executable starts out on disk
- The OS loads the program into memory
- CPU fetches instructions and data from memory while executing the program

Memory Management: Terminology



- **Segment:** A chunk of memory assigned to a process.
- **Physical Address:** a real address in memory
- **Virtual Address:** an address relative to the start of a process's address space.

Where do addresses come from?

How do programs generate instruction and data addresses?

- **Compile time:** The compiler generates the exact physical location in memory starting from some fixed starting position k. The OS does nothing.

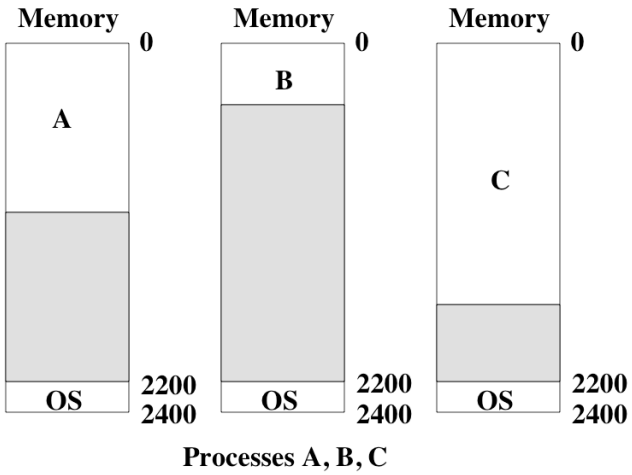
- **Load time:** Compiler generates an address, but at load time the OS determines the process' starting position. Once the process loads, it does not move in memory.

- **Execution time:** Compiler generates an address, and OS can place it any where it wants in memory.

Uniprogramming

- OS gets a fixed part of memory (highest memory in DOS).
- One process executes at a time.
- Process is always loaded starting at address 0.
- Process executes in a contiguous section of memory.
- Compiler can generate physical addresses.
- Maximum address = Memory Size - OS Size
- OS is protected from process by checking addresses used by process.

Uniprogramming



⇒ Simple, but does not allow for overlap of I/O and computation.

Multiple Programs Share Memory

Transparency:

- We want multiple processes to coexist in memory.
- No process should be aware that memory is shared.
- Processes should not care what physical portion of memory they are assigned to.

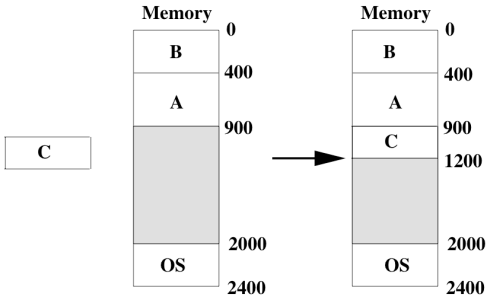
Safety:

- Processes must not be able to corrupt each other.
- Processes must not be able to corrupt the OS.

Efficiency:

- Performance of CPU and memory should not be degraded badly due to sharing.

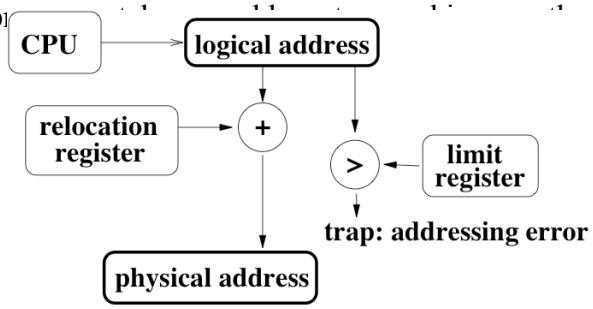
Relocation



- Put the OS in the highest memory.
- Assume at compile/link time that the process starts at 0 with a maximum address = memory size - OS size.
- Load a process by allocating a contiguous segment of memory in which the process fits.
- The first (smallest) physical address of the process is the *base* address and the largest physical address the process can access is the *limit* address.

Relocation

- **Static Relocation:**
  - at load time, the OS adjusts the addresses in a process to reflect its position in memory.
  - Once a process is assigned a place in memory and starts executing it, the OS cannot move it. (Why?)
- **Dynamic Relocation:**
  - hardware adds relocation register (base) to virtual address to get a physical address;
  - hardware compares address with limit register (address must be less than limit).
  - If test fails, the p… physical address.



Dynamic Relocation

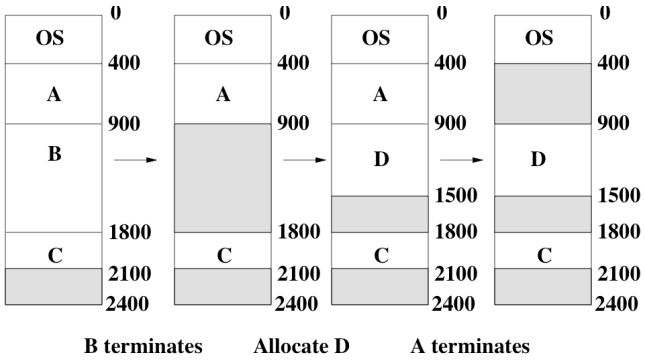
- **Advantages:**
  - OS can easily move a process during execution.
  - OS can allow a process to grow over time.
  - Simple, fast hardware: two special registers, an add, and a compare.
- **Disadvantages:**
  - Slows down hardware due to the add on every memory reference.
  - Can't share memory (such as program text) between processes.
  - Process is still limited to physical memory size.
  - Degree of multiprogramming is very limited since all memory of all active processes must fit in memory.
  - Complicates *memory management*.

Relocation: Properties

- **Transparency:** processes are largely unaware of sharing.
- **Safety:** each memory reference is checked.
- **Efficiency:** memory checks and virtual to physical address translation are fast as they are done in hardware, BUT if a process grows, it may have to be moved which is very slow.

Memory Management: Memory Allocation

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 new process, the OS must decide which hole to use for the process

Memory Allocation Policies

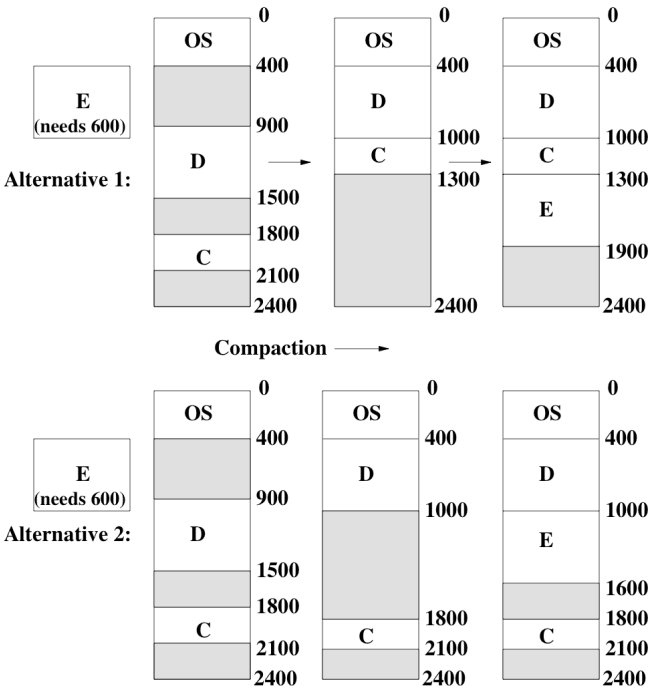
- **First-Fit:** allocate the first one in the list in which the process fits. The search can start with the first hole, or where the previous first-fit search ended.
- **Best-Fit:** Allocate the smallest hole that is big enough to hold the process. The OS must search the entire list or store the list sorted by size hole list.
- **Worst-Fit:** Allocate the largest hole to the process. Again the OS must search the entire list or keep the list sorted.
- Simulations show first-fit and best-fit usually yield better storage utilization than worst-fit; first-fit is generally faster than best-fit.

Fragmentation

- **External Fragmentation**
  - Frequent loading and unloading programs causes free space to be broken into little pieces
  - External fragmentation exists when there is enough memory to fit a process in memory, but the space is not contiguous
  - *50-percent rule:* Simulations show that for every 2*N* allocated blocks, *N* blocks are lost due to fragmentation (i.e., 1/3 of memory space is wasted)
  - We want an allocation policy that minimizes wasted space.
- **Internal Fragmentation:**
  - Consider a process of size 8846 bytes and a block of size 8848 bytes

  ⇒ it is more efficient to allocate the process the entire 8848 block than it is to keep track of 2 free bytes
  - Internal fragmentation exists when memory internal to a partition that is wasted

Compaction



- How much memory is moved?
- How big a block is created?
- Any other choices?

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: Motivation & Features

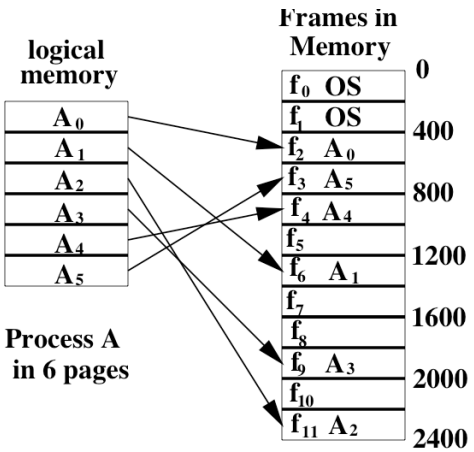
90/10 rule: Processes spend 90% of their time accessing 10% of their space in memory.

=> Keep only 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.
- Paging does not eliminate internal fragmentation (1/2 page per process)

Paging: Example

Mapping pages in logical mem to frames in physical memory

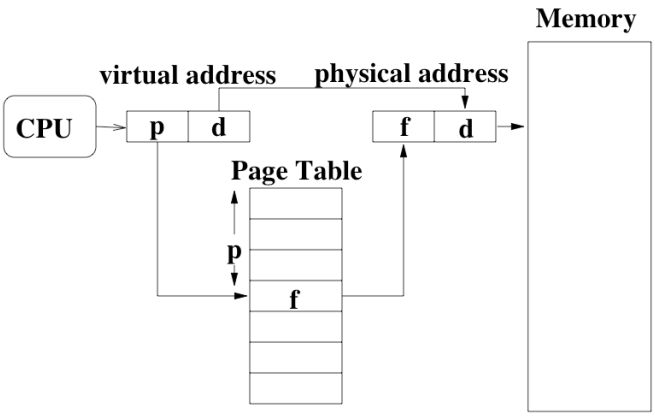


Paging Hardware

- **Problem:** How do we find addresses when pages are not allocated contiguously in memory?
- **Virtual Address:**
  - Processes use a virtual (logical) address to name memory locations.
  - Process generates contiguous, virtual addresses from 0 to size of the process.
  - The OS lays the process down on pages and the paging hardware translates virtual addresses to actual physical addresses in memory.
  - In paging, the virtual address identifies the page and the page offset.
  - *page table* keeps track of the page frame in memory in which the page is located.

Paging Hardware

Translating a virtual address to physical address

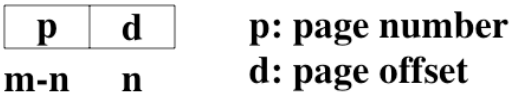


Paging Hardware

- Paging is a form of dynamic relocation, where each virtual address is bound by the paging hardware to a physical address.
- Think of the page table as a set of relocation registers, one for each frame.
- Mapping is invisible to the process; the OS maintains the mapping and the hardware does the translation.
- Protection is provided with the same mechanisms as used in dynamic relocation.

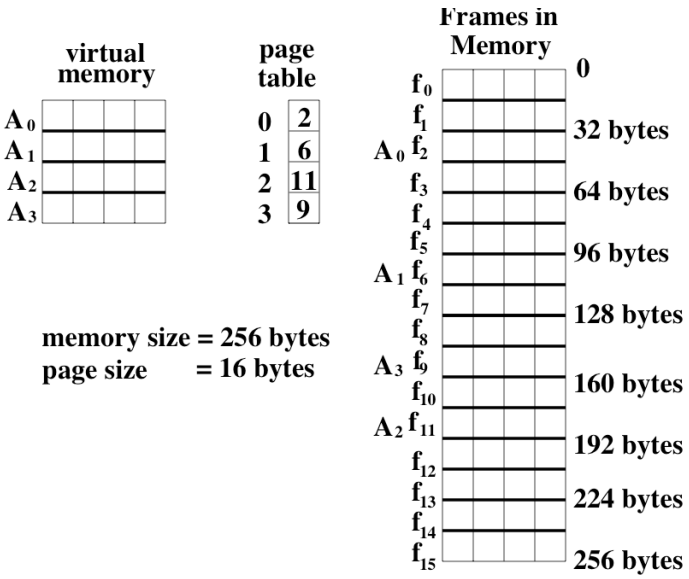
Paging Hardware: Practical Details

- Page size (frame sizes) are 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
- virtual address space of size 2^m bytes and a page of size 2^n, then
- the high order m-n bits of a virtual address select the page,
- the low order n bits select the offset in the page



p: page number
d: page offset

Address Translation Example



Address Translation Example

- How big is the page table?
- How many bits for an address. Assume we can address 1 byte increments?
- What part is p, and d?
- Given virtual address 24, do the virtual to physical translation.

Address Translation Example

- How big is the page table?
  - 16 entries
- How many bits for an address. Assume we can address 1 byte increments?
  - 8 bits, 4 for page and 4 for offset
- What part is p, and d?
- Given virtual address 24, do the virtual to physical translation.
  - p=1, d=8
  - f=6, d=8

Address Translation Example

- How many bits for an address? Assume we can address only 1 word (4 byte) increments?
- What part is p, and d?
- Given virtual address 13, do the virtual to physical translation.
- What needs to happen on a context switch?

Address Translation Example

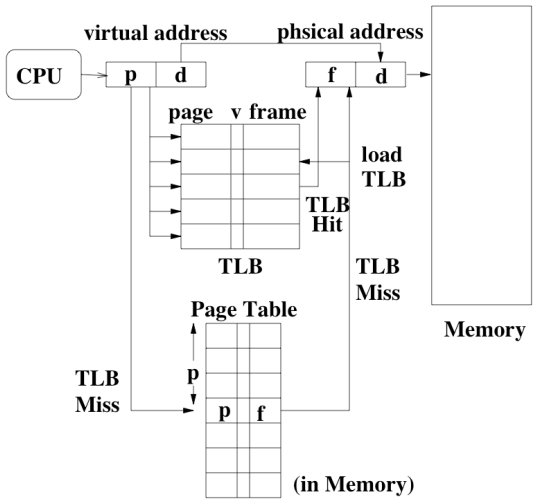
- How many bits for an address? Assume we can address only 1 word (4 byte) increments?
  - 6 bits, 4 for page, 2 for offset
- What part is p, and d?
- Given virtual address 13, do the virtual to physical translation.
  - p=3, d=1 (virtual)
  - F=9, offset=1 (physical)
- What needs to happen on a context switch?
  - Need to save the page table in PCB. Need to restore the page table of new process.

Making Paging Efficient

How should we store the page table?

- **Registers:** Advantages? Disadvantages?
- **Memory:** Advantages? Disadvantages?
- **TLB:** a fast fully associative memory that stores page numbers (key) and the frame (value) in which they are stored.
  - if memory accesses have locality, address translation has locality too.
  - typical TLB sizes range from 8 to 2048 entries.

The Translation Look-aside Buffer (TLB)



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

Costs of Using The TLB

- What is the effective memory access cost if the page table is in memory?
- What is the effective memory access cost with a TLB?

A large TLB improves hit ratio, decreases average memory cost.

Costs of Using The TLB

- What is the effective memory access cost if the page table is in memory?
  - ema = 2 * ma

- What is the effective memory access cost with a TLB?
  - ema = (ma + TLB) * p + (2ma + TLB) * (1-p)

A large TLB improves hit ratio, decreases average memory cost.

Initializing Memory when Starting a Process

1. Process needing *k* pages arrives.
2. If *k* page frames are free, then allocate these frames to pages. Else free frames that are no longer needed.
3. The OS puts each page in a frame and then puts the frame number in the corresponding entry in the page table.
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
- On a context switch:
  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.
- **Multilevel Paging:** If the virtual address space is huge, page tables get too big, and many systems use a multilevel paging scheme

Sharing

Paging allows sharing of memory across processes, since 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.

Summary

- Paging is a big improvement over segmentation:
  - They eliminate the problem of external fragmentation and therefore the need for compaction.
  - They allow sharing of code pages among processes, reducing overall memory requirements.
  - They enable processes to run when they are only partially loaded in main memory.

- However, paging has its costs:
  - Translating from a virtual address to a physical address is more time-consuming.
  - Paging requires hardware support in the form of a TLB to be efficient enough.
  - Paging requires more complex OS to maintain the page table.

Next: Segmentation

Segments take the user's view of the program and gives it to the OS.

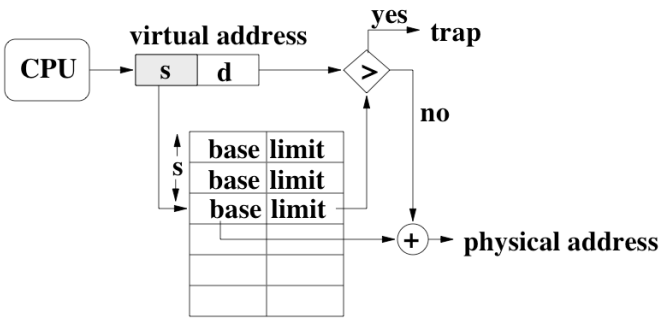
- User views the program in logical *segments*, e.g., code, global variables, stack, heap (dynamic data structures), not a single linear array of bytes.
- The compiler generates references that identify the segment and the offset in the segment, e.g., a code segment with offset = 399
- Thus processes thus use virtual addresses that are segments and segment offsets.

⇒ Segments make it easier for the call stack and heap to grow dynamically. Why?

⇒ Segments make both sharing and protection easier. Why?

Implementing Segmentation

- Segment table: each entry contains a base address in memory, length of segment, and protection information (can this segment be shared, read, modified, etc.).
- Hardware support: multiple base/limit registers.



Implementing Segmentation

- Compiler needs to generate virtual addresses whose upper order bits are a segment number.
- Segmentation can be combined with a dynamic or static relocation system,
  - Each segment is allocated a contiguous piece of physical memory.
  - External fragmentation can be a problem again
- Similar memory mapping algorithm as paging. We need something like the TLB if programs can have lots of segments

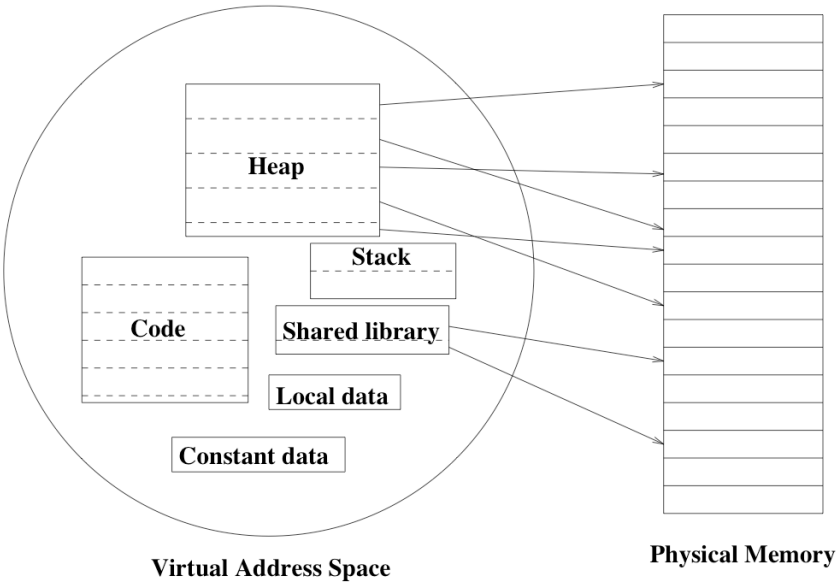
- **Let's combine the ease of sharing we get from segments with efficient memory utilization we get from pages.**

Combining Segments and Paging

- Treat virtual address space as a collection of segments (logical units) of arbitrary sizes.
- Treat physical memory as a sequence of fixed size page frames.
- Segments are typically larger than page frames,

⇒ Map a logical segment onto multiple page frames by paging the segments

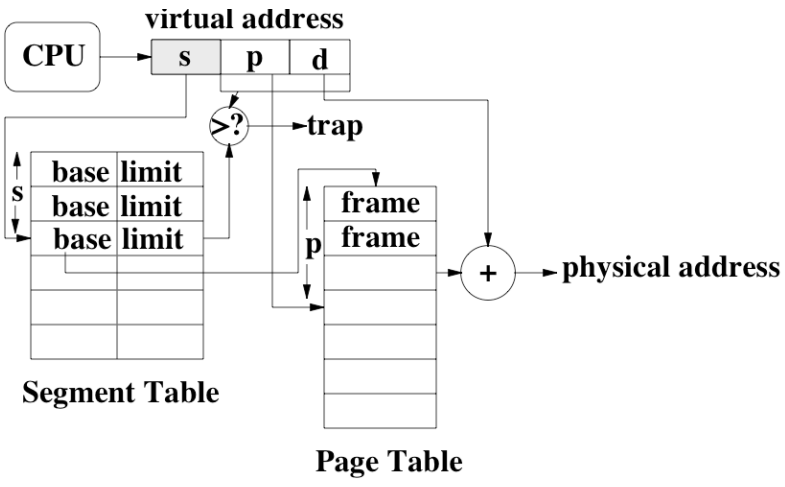
Combining Segments and Paging



Addresses in Segmented Paging

- A virtual address becomes a segment number, a page within that segment, and an offset within the page.
- The segment number indexes into the segment table which yields the base address of the page table for that segment.
- Check the remainder of the address (page number and offset) against the limit of the segment.
- Use the page number to index the page table. The entry is the frame. (The rest of this is just like paging.)
- Add the frame and the offset to get the physical address.

Addresses in Segmented Paging



Addresses in Segmented Paging: Example

- Given a memory size of 256 addressable words,
- a page table indexing 8 pages,
- a page size of 32 words, and
- 8 logical segments

- How many bits is a physical address?
- How many bits is a virtual address?
- How many bits for the seg #, page #, offset?
- How many segment table entries do we need?
- How many page table entries do we need?

Sharing Pages and Segments

- Share individual pages by copying page table entries.
- Share whole segments by sharing segment table entries, which is the same as sharing the page table for that segment.
- Need protection bits to specify and enforce read/write permission.
  - When would segments containing code be shared?
  - When would segments containing data be shared?

Sharing Pages and Segments: Implementation Issues

- Where are the segment table and page tables stored?
  - Store segment tables in a small number of associative registers; page tables are in main memory with a TLB (faster but limits the number of segments a program can have)
  - Both the segment tables and page tables can be in main memory with the segment index and page index combined used in the TLB lookup (slower but no restrictions on the number of segments per program)
- Protection and valid bits can go either on the segment or the page table entries
- **Note:** Just like recursion, we can do multiple levels of paging and segmentation when the tables get too big.

Segmented Paging: Costs and Benefits

- **Benefits:** faster process start times, faster process growth, memory sharing between processes.
- **Costs:** somewhat slower context switches, slower address translation.
- Pure paging system => (virtual address space)/(page size) entries in page table. How many entries in a segmented paging system?
- What is the performance of address translation of segmented paging compared to contiguous allocation with relocation? Compared to pure paging?
- How does fragmentation of segmented paging compare with contiguous allocation? With pure paging?

Inverted Page Tables

- Techniques to scale to very large address spaces
- Multi-level page tables
- Inverted index
  - Page table is a hash table with key-value lookups
    - Key=page number, value = frame number
    - Page table lookups are slow
    - Use TLBs for efficiency

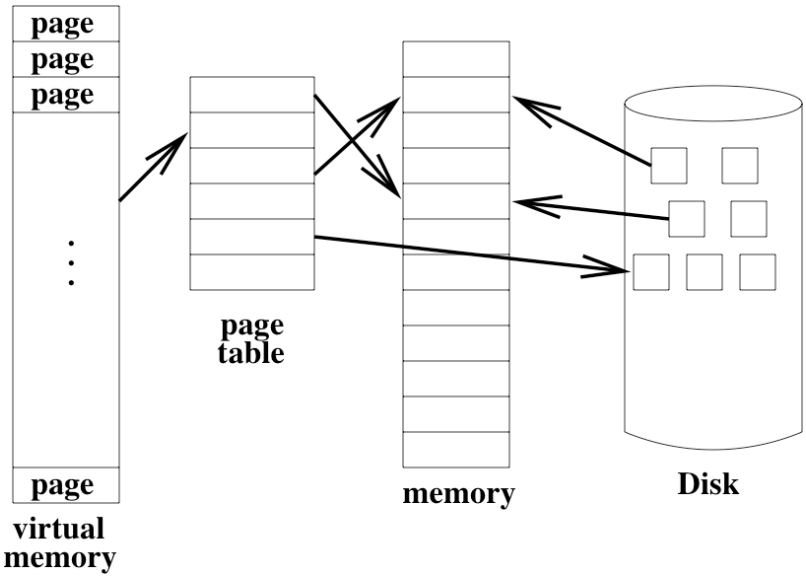
Demand Paged Virtual Memory

- Up to now, the virtual address space of a process fit in memory, and **we assumed it was all in memory.**
- OS illusions:ac
  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) x ma + p x page fault time
  - If memory access time is 200 ns and a page fault takes 25 ms
  - Effective access time = (1-p) x 200 + p x 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.

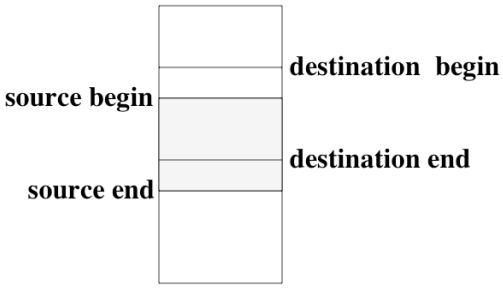
Transparent Page Faults

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

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