















Dynamic Linking

- A process loads libraries at load time
 Symbol references are resolved at load time
- OS loader finds the dynamic libraries and brings them
 into the process' memory address space

Dynamic Loading

- A process can load a module at runtime on request
 Similar to dynamic linking
- Program is written to load a specific library
- Resolve symbols to get pointers to data & functions
- The library can be unloaded when not needed

Shared libraries

- Dynamic linking + sharing
- Libraries that are loaded by programs when they start - All programs that start later use the shared library
- Program loader searches for needed shared libraries
- · Object code is linked with a stub
- Stub checks whether the needed library is in memory
- If not, the stub loads it
- Stub is then replaced with the address of the library
- · Operating system:
- Checks if the shared library is already in another process' memory
 Shares memory region among processes
- Need position independent code or pre-mapped code (reserved regions of memory that processes share)







Multiple Fixed Partitions

- Divide memory into predefined partitions (segments)
- Partitions don't have to be the same size
- For example: a few big partitions and many small ones
- · New process gets queued for a partition that can hold it
- Unused memory in a partition is wasted – Internal fragmentation
- Unused partitions: external fragmentation
- · Contiguous allocation:
- Process takes up a contiguous region of memory





Allocation algorithms

- First fit: find the first hole that fits
- · Best fit: find the hole that best fits the process
- Worst fit: find the largest available hole

 Why? Maybe the remaining space will be big enough for another process. In practice, this algorithm does not work well.

Variable partition multiprogramming

- What if a process needs more memory?
- Always allocate some extra memory just in case
- Find a hole big enough to relocate the process

Combining holes (fragments) – Memory compaction

Usually not done because of CPU time to move a lot of memory

Segmentation hardware

- Divide a process into segments and place each segment into a partition of memory
- Code segment, data segment, stack segment, etc.
- Discontiguous memory allocation



Paging Paging · Memory management scheme · Translation: - Physical space can be non-contiguous - Divide physical memory into fixed-size blocks: page frames - No fragmentation problems - A logical address is divided into blocks of the same size: pages - No need for compaction - All memory accesses are translated: page → page frame - A page table maps pages to frames · Paging is implemented by the Memory Management Unit (MMU) in the processor • Example: - 32-bit address, 4 KB page size: · Top 20 bits identify the page number · Bottom 12 bits identify offset within the page/frame Displacement (offset), d Page number, p





Hardware Implementation

- Where do you keep the page table? In memory
- Each process gets its own virtual address space
- Each process has its own page table
- Change the page table by changing a page table base register
- CR3 register on Intel IA-32 and x86-64 architectures
- Memory translation is now slow!
- To read a byte of memory, we need to read the page table first
- Each memory access is now 2x slower!

Hardware Implementation: TLB

- Cache frequently-accessed pages
 Translation lookaside buffer (TLB)
 Associative memory: key (page #) and value (frame #)
- TLB is on-chip & fast ... but small (64-1,024 entries) – Locality in the program ensures lots of repeated lookups
- TLB miss = page # not cached in the TLB – Need to do page table lookup in memory
- Hit ratio = % of lookups that come from the TLB

Address Space Identifiers: Tagged TLB

- There is only one TLB per system
- When we context switch, we switch address spaces New page table
- BUT ... TLB entries belong to the old address space

· Either:

- Flush (invalidate) the entire TLB
- Have a Tagged TLB with an Address Space Identifier (ASID)

Protection

- An MMU can enforce memory protection
- Page table stores status & protection bits per frame
 Valid/invalid: is there a frame mapped to this page?
- Read-only
- No execute
- Kernel only access
- Dirty: the page has been modified since the flag was cleared
- Accessed: the page has been accessed since the flag was cleared

Multilevel (Hierarchical) page tables

- Most processes use only a small part of their address space
- Keeping an entire page table is wasteful

Example

32-bit system with 4KB pages: 20-bit page table $\Rightarrow 2^{20} = 1,048,576$ entries in the page table



Inverted page tables

- # of pages on a system may be huge
- # of page frames will be more manageable (limited by physical memory)
- Inverted page table
 ith entry: contains info on what is in page frame i
- Table access is no longer a simple index but a search – Use hashing and take advantage of associative memory

Next Lecture

- Sharing memory across address spaces
- · Copy on write
- Demand paging
- Load needed pages on demand
- Page faults
- Page replacement: FIFO, LRU, second chance
- ThrashingWorking set: time window

