Midterm

Average: 75
Overall Grades

Average: 72
Paging Advantages

- Easy to allocate memory
  - Memory comes from a free list of fixed size chunks
  - Allocating a page is just removing it from the list
  - External fragmentation not a problem

- Easy to swap out chunks of a program
  - All chunks are the same size
  - Pages are a convenient multiple of the disk block size
  - How do we know if a page is in memory or not?
Page Table Entries (PTEs)

- Page table entries control mapping
  - The Modify bit says whether or not the page has been written
    » It is set when a write to the page occurs
  - The Reference bit says whether the page has been accessed
    » It is set when a read or write to the page occurs
  - The Valid bit says whether or not the PTE can be used
    » It is checked each time the virtual address is used
  - The Protection bits say what operations are allowed on page
    » Read, write, execute
  - The page frame number (PFN) determines physical page
Paging Limitations

- Can still have internal fragmentation
  - Process may not use memory in multiples of a page
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - Solution – use a hardware cache of lookups (more later)
- Memory required to hold page table can be significant
  - Need one PTE per page
  - 32 bit address space w/ 4KB pages = $2^{20}$ PTEs
  - 4 bytes/PTE = 4MB/page table
  - 25 processes = 100MB just for page tables!
  - Solution – page the page tables (more later)
Segmentation

- Segmentation is a technique that partitions memory into logically related data units
  - Module, procedure, stack, data, file, etc.
  - Virtual addresses become <segment #, offset>
  - Units of memory from user’s perspective

- Natural extension of variable-sized partitions
  - Variable-sized partitions = 1 segment/process
  - Segmentation = many segments/process

- Hardware support
  - Multiple base/limit pairs, one per segment (segment table)
  - Segments named by #, used to index into table
Segment Lookups

Virtual Address

Segment #
Offset

Segment Table

limit
base

Physical Memory

Protection Fault

Yes?

<
No?

+
Extensions
- Can have one segment table per process
  - Segment #s are then process-relative (why do this?)
- Can easily share memory
  - Put same translation into base/limit pair
  - Can share with different protections (same base/limit, diff prot)

Problems
- Cross-segment addresses
  - Segments need to have same #s for pointers to them to be shared among processes
- Large segment tables
  - Keep in main memory, use hardware cache for speed
Can combine segmentation and paging
- The x86 supports segments and paging

Use segments to manage logically related units
- Module, procedure, stack, file, data, etc.
- Segments vary in size, but usually large (multiple pages)

Use pages to partition segments into fixed size chunks
- Makes segments easier to manage within physical memory
  » Segments become “pageable” – rather than moving segments into and out of memory, just move page portions of segment
- Need to allocate page table entries only for those pieces of the segments that have themselves been allocated

Tends to be complex…
Virtual memory summary

- **Virtual memory**
  - Processes use virtual addresses
  - OS + hardware translates virtual address into physical addresses

- **Various techniques**
  - Fixed partitions – easy to use, but internal fragmentation
  - Variable partitions – more efficient, but external fragmentation
  - Paging – use small, fixed size chunks, efficient for OS
  - Segmentation – manage in chunks from user’s perspective
  - Combine paging and segmentation to get benefits of both
Managing Page Tables

- We computed the size of the page table for a 32-bit address space with 4K pages to be 4MB
  - This is far too much overhead for each process
- How can we reduce this overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire address space)
- How do we only map what is being used?
  - Can dynamically extend page table...
  - Does not work if address space is sparse (internal fragmentation)
- Use another level of indirection: two-level page tables
Originally, virtual addresses (VAs) had two parts
- Page number (which mapped to frame) and an offset

Now VAs have three parts:
- Master page number, secondary page number, and offset

Master page table maps VAs to secondary page table
- We’d like a manageable master page size

Secondary table maps page number to physical page
- Determines which physical frame the address resides in

Offset indicates which byte in physical page
- Final system page/frame size is still the same, so offset length stays the same
Two-Level Page Tables

Master page number → Secondary → Offset

Virtual Address

Page table

Master Page Table

Page frame

Secondary Page Table

Page frame → Offset

Physical Address

Physical Memory
An Example w/4-byte PTEs

4K pages = 12-bit offset; 1 master page = 4K/4 = 1K entries = 10bits
Secondary page table size = 32 - 12 - 10 = 10 bits = 1K entries * 4 = 4K
● Physical memory
  - Easy to address, no translation required
  - But, allocated page tables consume memory for lifetime of VAs

● Virtual memory (OS virtual address space)
  - Cold (unused) page table pages can be paged out to disk
  - But, addressing page tables requires translation
  - How do we stop recursion?
  - Do not page the outer page table (called wiring or pinning)

● If we’re going to page the page tables, might as well page the entire OS address space, too
  - Need to wire special code and data (fault, interrupt handlers)
Efficient Translations

- Our original page table scheme already doubled the cost of doing memory lookups
  - One lookup into the page table, another to fetch the data
- Now two-level page tables triple the cost!
  - Two lookups into the page tables, a third to fetch the data
  - And this assumes the page table is in memory
- How can we use paging but also have lookups cost about the same as fetching from memory?
  - Cache translations in hardware
  - Translation Lookaside Buffer (TLB)
  - TLB managed by Memory Management Unit (MMU)
**TLBs**

- **Translation Lookaside Buffers**
  - Translate *virtual page #s into PTEs* (not physical addresses)
  - Can be done in a single machine cycle
- **TLBs implemented in hardware**
  - Fully associative cache (all entries looked up in parallel)
  - Cache tags are virtual page numbers
  - Cache values are PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address
- **TLBs exploit locality**
  - Processes only use a handful of pages at a time
    - 16-48 entries/pages (64-192K)
    - Only need those pages to be “mapped”
  - Hit rates are therefore very important
Most address translations are handled using the TLB

- >99% of translations, but there are misses (TLB miss)...

Who places translations into the TLB (loads the TLB)?

- Software loaded TLB (OS)
  - TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
  - Must be fast (but still 20-200 cycles)
  - CPU ISA has instructions for manipulating TLB
  - Tables can be in any format convenient for OS (flexible)

- Hardware (Memory Management Unit)
  - Must know where page tables are in main memory
  - OS maintains tables, HW accesses them directly
  - Tables have to be in HW-defined format (inflexible)
Managing TLBs

- OS ensures that TLB and page tables are consistent
  - When it changes the protection bits of a PTE, it needs to invalidate the PTE if it is in the TLB
- Reload TLB on a process context switch
  - Invalidate all entries
  - Why? What is one way to fix it?
- When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  - Choosing PTE to evict is called the TLB replacement policy
  - Implemented in hardware, often simple (e.g., Last-Not-Used)
Paged Virtual Memory

- We’ve mentioned before that pages can be moved between memory and disk
  - This process is called demand paging

- OS uses main memory as a page cache of all the data allocated by processes in the system
  - Initially, pages are allocated from memory
  - When memory fills up, allocating a page in memory requires some other page to be evicted from memory
    - Why physical memory pages are called “frames”
  - Evicted pages go to disk (where? the swap file/partition)
  - The movement of pages between memory and disk is done by the OS, and is transparent to the application
Page Faults

- What happens when a process accesses a page that has been evicted?
  1. When it evicts a page, the OS sets the PTE as invalid and stores the location of the page in the swap file in the PTE
  2. When a process accesses the page, the invalid PTE will cause a trap (page fault)
  3. The trap will run the OS page fault handler
  4. Handler uses the invalid PTE to locate page in swap file
  5. Reads page into a physical frame, updates PTE to point to it
  6. Restarts process

- But where does it put it? Has to evict something else
  - OS usually keeps a pool of free pages around so that allocations do not always cause evictions
Next time…

- Demand paging, a.k.a., more Chapter 9…