Lecture Overview

We’ll cover more paging mechanisms:

• **Optimizations**
  ♦ Managing page tables (space)
  ♦ Efficient translations (TLBs) (time)
  ♦ Demand paged virtual memory (space), next lecture

• Project 1 due tonight, no extension!
• Midterm grades to be released soon
• Homework 3 out
All problems in computer science can be solved by another level of indirection.

Butler Lampson
All problems in computer science can be solved by another level of indirection

but that usually will create another problem

– David Wheeler
## Summary: Evolution of Memory Management (before paging)

<table>
<thead>
<tr>
<th>Scheme</th>
<th>How</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple uniprogramming</td>
<td>1 segment loaded to starting address 0</td>
<td>Simple</td>
<td>1 process</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 segment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No protection</td>
</tr>
<tr>
<td>Simple multiprogramming</td>
<td>1 segment relocated at loading time</td>
<td>Simple, Multiple processes</td>
<td>1 segment/process</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>No protection</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>External frag.</td>
</tr>
<tr>
<td>Base &amp; Bound</td>
<td>Dynamic mem relocation at runtime</td>
<td>Simple hardware, Multiple processes</td>
<td>1 segment/process,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>External frag.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple segments</td>
<td>Dynamic mem relocation at runtime</td>
<td>Sharing, Protection, multi segs/process</td>
<td>More hardware,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>External frag.</td>
</tr>
</tbody>
</table>
The Big Picture

- `main.c` and `math.c` are compiled into `main.o` and `math.o` respectively.
- `main.o` and `math.o` are linked to create `a.out`.
- `a.out` is loaded into memory.
- Memory management is set up and managed, with virtual memory mapping.
- Instruction execution is handled with virtual memory.
- Arch execution includes translation and accessing physical memory.
- Load `a.out` to memory and manage memory for the process.
Paging solves the external fragmentation problem by using fixed sized units in both physical and virtual memory.
Paging

Translating addresses
- Virtual address has two parts: virtual page number and offset
- Virtual page number (VPN) is an index into a page table
- Page table determines page frame number (PFN)
- Physical address is PFN::offset ("::" means concatenate)

Page tables
- Map virtual page number (VPN) to page frame number (PFN)
  - VPN is the index into the table that determines PFN
- One page table entry (PTE) per page in virtual address space
  - Or, one PTE per VPN
Virtual address

VPN  offset

Page table

PFN  ...  ...
...
PFN  ...  ...

PFN  offset

Physical address

page table size

error

• Context switch
  ♦ similar to the segmentation scheme

• Pros:
  ♦ easy to allocate memory
  ♦ easy to swap
  ♦ easy to share
Paging Example

- Pages are 4K
  - 4K → offset is 12 bits → VPN is 20 bits ($2^{20}$ VPNs), assuming 32-bit system
- Virtual address is 0x7468
  - Virtual page is 0x7, offset is 0x468 (lowest 12 bits of address)
- Page table entry 0x7 contains 0x2
  - Page frame number is 0x2
  - Seventh virtual page is at address 0x2000 (physical page 2)
- Physical address = 0x2000 :: 0x468 = 0x2468
Deep thinking

- Why does the page table we talked about so far have to be contiguous in the physical memory?
  - Why did a segment have to be contiguous in memory?

- For a 4GB virtual address space, we just need 1M PTE (~4MB), what is the big deal?

- My PC has 2GB, why do we need PTEs for the entire 4GB address space?
Summary so far

• Virtual memory
  ♦ Processes use virtual addresses
  ♦ Hardware translates virtual address into physical addresses with OS support

• Evolution of techniques
  ♦ Single, fixed physical segment per process (no virt mem)
  ♦ Single segment per process, static relocation (no virt mem)
  ♦ Base-and-bound – dynamic relocating whole process
  ♦ Segmentation – multiple (variable-size) segments with dynamic relocation
  ♦ Paging – small, fixed size pages
Page Tables

- Page tables completely define the mapping between virtual pages and physical pages for an address space.
- Each process has an address space, so each process has a page table.
- Page tables are data structures maintained by the OS (and accessed by hardware).
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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>R</td>
<td>V</td>
<td>Prot</td>
<td>Page Frame Number</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>
How many PTEs do we need? (assume page size is 4096 bytes)

• Worst case for 32-bit address machine?

• What about 64-bit address machine?

• Page size?
  ♦ Small page -> big table
  ♦ Large page -> small table but large internal fragmentation
Paging implementation – how does it really work?

- Where to store page table?

- How to use MMU?
  - Even small page tables are too large to load into MMU
  - Page tables kept in mem and MMU only has their base addresses

- What happens at context switches?
Paging Advantages

- Easy to allocate (physical) memory
  - Memory comes from a free list of fix-sized 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
  - Use valid bit to detect references to swapped pages
  - Pages are a convenient multiple of the disk block size
  - More on swapping next time
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 (next lec)
- Memory required to hold page table can be significant
Deep thinking

• Why does the page table we talked about so far have to be contiguous in the physical memory?
  ♦ Why did a segment have to be contiguous in memory?

• For a 4GB virtual address space, we just need 1M PTE (~4MB), what is the big deal?

• My PC has 2GB, why do we need PTEs for the entire 4GB address space?
Managing Page Tables

- How can we reduce page table space overhead?
  - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)

- How can we be flexible?
  - “All computer science problems can be solved with an extra level of indirection.”
    - two-level page tables
Page Table Evolution

Linear (Flat) Page Table

Virtual Address Space

Physical Memory

Page 0
Page 1
Page 2

Page N-1
Page Table Evolution

Hierarchical Page Table

Virtual Address Space

Physical Memory

Master

Secondary

Page 0

Page 1

Page 2

Page N-1
Page Table Evolution

Hierarchical Page Table

Virtual Address Space

Physical Memory

Master

Secondary

Page 0

Page 1

Page 2

Page N-1

Unmapped

Not Needed
Two-Level Page Tables

• Two-level page tables
  ♦ Virtual addresses (VAs) have three parts:
    » Directory (master page number), secondary page number, and offset
  ♦ Directory page table maps VAs to secondary page table
  ♦ Secondary page table maps page number to physical page
  ♦ Offset indicates where in physical page address is located
Two-Level Page Tables

- Virtual address
- Directory
  - table addr
  - ...
- Page
- Page table
  -PFN
  - ...
  - ...
  - ...
  - ...
  - ...
  - ...
  - ...
  - ...
-PFN
- offset

Physical address
Two-Level Page Tables

• Example
  ♦ 4KB pages, 4 bytes/PTE, 32-bit address space
  ♦ How many bits in offset?
    » 4KB = 12 bits
  ♦ Want directory page table in one page, how many entries can we have in the directory page table?
    » 4KB/4 bytes = 1KB entries (each entry is a 32-bit address)
  ♦ Hence, 1KB secondary page tables. How many bits?
  ♦ Directory (1KB) = 10, offset = 12, 32 – 10 – 12 = 10 bits left
    » One secondary page table can host 4K/4bytes=1KB PTEs
    » 10 bits (inner) => exactly 1KB PTEs
Multiple-level page tables
Multi-level page tables

• 3 Advantages?
  ♦ L1, L2, L3, L4 tables do not have to be consecutive
  ♦ They do not have to be allocated before use!
  ♦ They can be swapped out to disk!

The power of an extra level of indirection!

• Problems?
Efficient Translations

- Our original page table scheme already increased the cost of doing memory lookups
  - Two lookups into the page table, another to fetch the data
  - One lookup and one data access for original flat page table
- Now 4-level page tables require five DRAM accesses for one memory operation!
  - Four lookups into the page tables, a fifth to fetch the data
- Solution: *reference locality!*
  - In a short period of time, a process is likely accessing only a few pages
  - Store part of the page table that is “hot” in a fast hardware unit
Translation Look-aside Buffer (TLB)

- Translation Look-aside Buffers
  - Translate VPNs into PFNs
- TLBs implemented in hardware
  - TLB hit is very fast $\leq$ 1 CPU cycle
  - Fully associative cache $\Rightarrow$ least conflict misses
  - New entries can be inserted anywhere in the TLB
  - All entries looked up in parallel
    $\Rightarrow$ TLB can’t be made very big, typically 64 – 4096 entries

- Optional (useful) bits
  - ASIDs -- Address-space identifiers (process tags)
Translation Look-aside Buffer (TLB)

Virtual address

VPN   offset

VPN   PFN   ...

VPN   PFN   ...

...  ...

VPN   PFN   ...

TLB

Miss

VPN   PFN   ...

PFN   offset

Real page table

Hit

Physical address
Miss handling: Hardware-controlled TLB

• On a TLB hit, MMU checks the valid bit
  ♦ If valid, perform address translation
  ♦ If invalid (e.g. page not in memory), MMU generates a page fault
    » OS performs fault handling
    » Restart the faulting instruction

• On a TLB miss
  ♦ MMU parses page table and loads PTE into TLB
    » Needs to replace if TLB is full
    » Page table layout is fixed
  ♦ Same as hit …
Miss handling: Software-controlled TLB

• On a TLB hit, MMU checks the valid bit
  ♦ If valid, perform address translation
  ♦ If invalid (e.g. page not in memory), MMU generates a page fault
    » OS performs page fault handling
    » Restart the faulting instruction

• On a TLB miss, HW raises exception, traps to the OS
  ♦ OS parses page table and loads PTE into TLB
    » Needs to replace if TLB is full
    » Page table layout can be flexible
  ♦ Same as in a hit…
Hardware vs. software controlled

• Hardware approach
  ♦ Efficient – TLB misses handled by hardware
  ♦ OS intervention is required only in case of page fault
  ♦ Page structure prescribed by MMU hardware -- rigid

• Software approach
  ● Less efficient -- TLB misses are handled by software
  ● MMU hardware very simple, permitting larger, faster TLB
  ● OS designer has complete flexibility in choice of MM data structure
Deep thinking

- Without TLB, how MMU finds PTE is fixed
- With TLB, it can be flexible, e.g. software-controlled is possible
- What enables this?
- TLB is an extra level of indirection!
More TLB Issues

- When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
  - Which TLB entry should be replaced?
    » Random
    » LRU

- What happens when changing a page table entry (e.g. because of swapping, change read/write permission)?
  - Change the entry in memory
  - flush (eg. invalidate) the TLB entry
    » INGLPG on x86
What happens to TLB in a process context switch?

- During a process context switch, cached translations cannot be used by the next process
  - Invalidate all entries during a context switch
    » Lots of TLB misses afterwards
  - Tag each entry with an ASID
    » Add a HW register that contains the process id of the current executing process
    » TLB hits if an entry’s process id matches that register
Cache vs. TLB

• Similarities:
  ♦ Both cache a part of the physical memory

• Differences:
  ♦ Associatively
    » TLB is usually fully associative
    » Cache can be direct mapped
  ♦ Coherence
    » No hardware provided coherence between TLB and main memory
    » Software needs to flush TLB entries for coherence
    » Cache: hardware-provided (via snooping bus) coherence across multiple cores and main memory
More on coherence issues

• No hardware maintains coherence between DRAM and TLBs:
  ♦ OS needs to flush related TLBs whenever changing a page table entry in memory

• On multiprocessors, when you modify a page table entry, you need to do “TLB shoot-down” to flush all related TLB entries at all the cores
Summary so far

- Virtual memory address: a level of indirection to decouple static time (compiler) from run time (OS)
- Paging: avoiding external fragmentation, great flexibility
- Single-level page tables are too big
- Multi-level page tables reduce the space overhead (leveraging indirection) but increases the performance overhead
- TLB improves paging performance (leveraging locality)
- But TLB shootdown is costly (esp. on many cores)
- One last thing about paging…
Kernel Address Space

- Wait… how does the OS virtual address space work?
- We have talked about it as a separate address space
- But it is typically implemented as an extension of the user-level process address space
  - The bottom portion is for the user-level process
  - The top portion is for the operating system/kernel
  - VMS, early Unix: user 2GB, kernel 2GB (32-bit)
  - Linux, Windows: user 3GB, kernel 1GB (32-bit)
Process Address Space

Address space used by process

0x00000000

0xBFFFFFFF

User Address Space (3GB)

Stack

Heap

Static Data (Data Segment)

Code (Text Segment)
Kernel Address Space

Address space used by process

Address space used by kernel

Same in all page tables

Trap to kernel

Stack

Code (Text Segment)

Static Data (Data Segment)

Heap

User Address Space (3GB)

0xBFFFFFFF

0x00000000

OS Code, Data, Heap, Drivers, …

Kernel Address Space (1GB)

0xFFFFFFFF

0xBFFFFFFF

0xC0000000

0x00000000
Kernel Address Space

• When CPU is in user mode, a process can only access the user-level portion
• When CPU is in kernel/privileged mode, the OS can access the entire region
• This arrangement is very convenient for the OS
  ♦ The OS can access any memory in the user-level portion of the current process (e.g., copying system call arguments)
  ♦ But the OS region is protected from the process
• As a result, the OS is mapped into every process
  ♦ The upper portion of every process address space is the OS
  ♦ Context switching effectively just switches the bottom portion
• This works well until Meltdown (mitigation: kernel page-table isolation KPTI) read more: here (Meltdown and KPTI not in exam)
Next time

- Swapping, memory allocation, memory sharing
Memory Hierarchy Revisited

What does this imply about L1 addresses?

Where do we hope requests get satisfied?
Physical (Address) Caches

- Memory hierarchy so far: physical caches
  - Indexed and tagged by PAs
    - Physically Indexed (PI)
    - Physically Tagged (PT)
  - Translate to PA to VA at the outset
  - Cached inter-process communication works
    - Single copy indexed by PA
    - Slow: adds at least one cycle to $t_{hit}$
Virtual Caches (VI/VT)

- **Alternative: virtual caches**
  - Indexed and tagged by VAs (VI and VT)
  - Translate to PAs only to access L2
  - Fast: avoids translation latency in common case
  - Problem: VAs from different processes are distinct physical locations (with different values) (call homonyms)

- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags

- Does inter-process communication work?
  - **Synonyms**: multiple VAs map to same PA
    - Can’t allow same PA in the cache twice
    - Also a problem for DMA I/O
  - Can be handled, but very complicated
Memory Hierarchy Re-Revisited

What does this imply about L1 addresses?

Any speed benefits?
Any drawbacks?
Parallel TLB/Cache Access (VI/PT)

- Compromise: access TLB in parallel
  - *In small caches, index of VA and PA the same*
    - \( VI == PI \)
  - Use the VA to index the cache
  - Tagged by PAs
  - Cache access and address translation in parallel
  - + No context-switching/aliasing problems
  - + Fast: no additional \( t_{hit} \) cycles
  - Common organization in processors today