Paging Hardware
Questions

• What if the page size is very large?
  – Internal fragmentation: if we don’t need all of the space inside a page frame

• What if the page size is very small?
  – Many page table entries, which consume memory
  – Bad TLB use (will be covered later)

• Can we share memory between processes?
  – The same page frames are pointed to by the page tables of processes that share the memory
  – This is how processes share the kernel address space, program code, and library code
Paging and Copy on Write

- UNIX fork with copy on write
  - Copy page table of parent into child process
  - Mark all pages (in both page tables) as read-only
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution
Issues with the array-based page table implementation

- Large memory consumption due to the table
  - 32-bit space, 4KB page size => $2^{32}/2^{12}= 1$M entries, 4 bytes for each entry => 4M memory
  - 64-bit machine currently uses 48 bits address space
    - $2^{48}/2^{12} = 64$G entries => 256 G memory
Multilevel Page Table

Program

Paging Mechanism

Root page table (contains 1024 PTEs)

4-kbyte page table (contains 1024 PTEs)

Main Memory

Virtual Address

10 bits 10 bits 12 bits

Frame # Offset

Root page table ptr

Page Frame
Questions

• What is the range of the virtual address space that is covered by an entry in the L1 page table?
  • $2^{22} = 4M$

• Recall that we consistently need 4MB memory if we use the single-level implementation; how much memory do we need at most with the multi-level page table?
  • 4KB memory for the L1 page table (1024 entries, each 4 bytes)
  • 1024 L2 page tables = 4MB memory
  • $1 + 1024 = 1025$ frames, that is, 4KB + 4MB
  • Even worse? This is just the worst case
Questions

• If the process needs 8MB ($2^{23}$) memory, how many page frames are used as page tables at least? How many at most?
  • Best case: all pages are together in the virtual space. 1 frame for the L1 page table + 2 for L2 page tables ($2^{23} / 2^{12} = 2^{11} = 2048$ entries, which correspond to 2 L2 page tables)
  • Worst case: pages scatter. 1 frame for the L1 page table; 2048 scatter in all the L2 page tables, which are 1024
• Which is the case in reality?
Anatomy of the virtual address space of a process

Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault

0xc0000000 == TASK_SIZE
Random stack offset

Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset

Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so

3GB
program break
brk
start_brk
Random brk offset

Heap

BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;

Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";

Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
Multilevel Translation

• Pros:
  – Save the physical memory used by page tables

• Cons:
  – Two steps of (or more) lookups per memory reference
    • 2 levels for 32 bits
    • 4 levels for 48 bits (in current 64-bit systems)
TLB (Translation Lookaside Buffer)

- TLB: hardware cache for the page tables
  - each entry stores a mapping from page # to frame #
- Address translation for an address “page # : offset”
  - If page # is in cache, get frame # out
  - Otherwise get frame # from page table in memory
  - Then load the (page #, frame #) mapping in TLB
TLB Lookup

Translation Lookaside Buffer (TLB)

Matching Entry

Virtual Address

Page#  Offset

Virtual Page  Page Frame  Access

Physical Address

Frame  Offset

Physical Memory

Page Table Lookup

CIS 3207 - Operating Systems
TLB and Page Table Translation

Processor → Virtual Address → TLB → Virtual Address

Hit → Frame → TLB → Virtual Address

Miss → Invalid → Raise Exception

Valid → Frame → Page Table

Invalid → Raise Exception

Offset → + → Physical Memory

Physical Address → Physical Memory

Data → Physical Memory

Data → Processor

CIS 3207 - Operating Systems
Cost

• Hit ratio: percentage of times that the cache has the needed data; denoted as $h$; so Miss ratio: $1 - h$
• Cost of TLB look up: $T_{tlb}$
• Cost of Page Table look up $T_{pt}$
• Cost of translation =
  \[ h \times T_{tlb} + (1-h) \times (T_{tlb} + T_{pt}) = T_{tlb} + (1-h) \times T_{pt} \]
• Assume
  – Cost of TLB lookup = 1ns
  – Cost of page table lookup = 200ns
  – Hit ratio = 99% (percentage of times that)
  – Cost = $1 + 0.01 \times 200 = 3$ns; a boost compared to 200ns
Improvement 1: Superpages

• On many systems, a TLB entry can be for
  – A superpage
  – It saves a lot of TLB entries compared to small pages

• x86: superpage
  – 2MB
  – 4MB
  – 1GB (x86-64)
Superpages

Virtual Address

<table>
<thead>
<tr>
<th>Page#</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>Offset</td>
</tr>
</tbody>
</table>

Translation Lookaside Buffer (TLB)

Matching Entry

Matching Superpage

Page Table Lookup

Physical Memory

CIS 3207 - Operating Systems
Inverted Page Table

• One page table for each process may be too costly
• A hash table that maps virtual page # to frame #
• The whole system uses a single Inverted Page Table regardless of the number of processes or the size of virtual space
• Structure is called inverted because it indexes page table entries by frame number rather than by page number
Inverted Page Table

Each entry in the page table includes:

- Process identifier
  - the process that owns this page frame

- Page number
  - Because there may be hash collision

- Control bits
  - includes flags and protection and locking information

- Chain pointer
  - To resolve hash collision
Inverted Page Table

Note that \( j \) is used as the frame #
Process switch

• Upon process switch what is updated in order to assist address translation?
  – Contiguous allocation: base & limit registers
  – Segmentation: register pointing to the segment table (recall that each process has its own segment table)
  – Paging: register pointing to the page table; TLB should be flushed if the TLB does not support multiple processes
Summary

• Address translation for contiguous allocation
  – Base + bound registers
• Address translation for segmentation
  – Segment table
  – Copy-on-write
  – Sharing
• Memory-efficient address translation for paging
  – Multi-level page tables
• Accelerated address translation for paging
  – TLB
Writing assignment

• Why do we use multi-level page tables for address translation?
• What is TLB? What is it used for?
Backup pages
Improvement 2: Tagged TLB

• An address mapping only makes sense for a specific process. One solution to dealing with process switch is to flush TLB.
  – But, it takes time for filling TLB entries for the newly switched-in process
• Some entries can be pinned, e.g., those for kernel
• Tagged TLB
  – Each TLB entry has process ID
  – TLB hit only if process ID matches current process
  – So no need to flush TLB
Physical Memory

Processor

Virtual Address

Page# Offset

Process ID

Matching Entry

Translation Lookaside Buffer (TLB)

Process ID Page Frame Access

Physical Address

Frame Offset

Page Table Lookup

CIS 3207 - Operating Systems
Improvement 3: TLB Consistency

• The OS should discard the related TLB entry, e.g.,
  – When swapping out a page
  – When R/W pages become read-only due to fork()
  – These events can be summarized as Permission Reduction

• To support shared memory, upon permission reduction
  – all the corresponding TLB entries should be discarded

• On a multicore, upon permission reduction, the OS must ask each CPU to discard the related TLB entries
  – This is called TLB shootdown
  – It leads to Inter-processor Interrupts. The initiator processor has to wait until all other processors acknowledge the completion, so it is costly
## TLB Shootdown

<table>
<thead>
<tr>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor 1 TLB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x0053</td>
<td>0x0003</td>
<td>R/W</td>
</tr>
<tr>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
<tr>
<td>Processor 2 TLB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x0053</td>
<td>0x0003</td>
<td>R/W</td>
</tr>
<tr>
<td>0</td>
<td>0x0001</td>
<td>0x0005</td>
<td>Read</td>
</tr>
<tr>
<td>Processor 3 TLB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
</tr>
<tr>
<td>0</td>
<td>0x0001</td>
<td>0x0005</td>
<td>Read</td>
</tr>
</tbody>
</table>
Question

• Why upon permission reduction only?
  – In the case of permission increment, a page fault will be triggered and the TLB has chance to be updated then (we will cover page fault handling next class)
  – Lazy update
If swapping is not used, do the schemes of segmentation/paging still have advantages over contiguous memory allocation?

Yes. With segmentation or paging, a process occupies multiple, rather than one, partitions. When a process exits, those “small chunks” that are otherwise “useless” in dynamic partitioning may be used to accommodate segments or pages of one or more processes. So it is a better use of space. You can see other advantages, e.g., sharing.
Terms (X86)

• **Real mode**: a legacy operating mode of x86-compatible CPUs.
  – 20-bit address space, i.e., 1M
  – physical_address = segment_part × 16 + 16-bit offset
  – No support for memory protection: each process can access any physical location

• **Protected mode**
  – Enables memory protection (recall privilege rings) by setting the Protection Enable bit in CR0
  – Segmentation and Paging
Terms (X86)

• GDT (Global Descriptor Table) and LDT (Local Descriptor Table)
  – They contain Segment Descriptors; each defines the base, limit, and
    privilege of a segment
  – A Segment Register contains a Segment Selector, which determines
    which table to use and an index into the table
  – Each logical address consists of a Segment Register and an offset.
    However, the Segment Selector is usually specified implicitly:
      • E.g., an instruction fetch implies the Code Segment; a data access implies DS

• Privilege check:
  – CPL ≤ DPL
  – where CPL is the current Privilege Level (found in the CS register),
    and DPL is the descriptor privilege level of the segment to be
    accessed
  – This is how Protection
Segmentation in Linux/x86

- Segmentation cannot be disabled on x86-32 processors; it used to translate a two-part logical address to a linear address
- Linux has to pretend that it is using segmentation
- For each segment, bases = 0 and limit = 4G, which means that a logical address = a linear address
- However, those Segment Selector registers, especially the Code Segment register, are critical to implement the Protection Ring idea
- Linux defines four Segment Selector values:
  - __USER_CS, __USER_DS
  - __KERNEL_CS, __KERNEL_DS
  - For user space and kernel space, respectively
  - Stack Segment register uses the data segment