Previous class…

- What is logical address? Who use it?
  - Describes a location in the logical memory address space
  - Compiler and CPU use it

Some slides are courtesy of Dr. Thomas Anderson and Gustavo Duarte
What will happen if the RAM is less than the total size of the working sets of the processes?

Thrashing. Consider a simple example: \texttt{memcpy(dst, src, 4096)}. Assume \texttt{dst} and \texttt{src} reside in page 0 and 1, respectively; obviously, the working set during the call is (at least) two pages, but what if the physical memory allocated to the process is only one page? There will be continuous swapping: to access page 1, you have to first swap out page 0, and vice verse.
Questions

• With segmentation, what is updated at process switch to assist address translation?
  – Register that points to the segment table

• Recall that a segment can be swapped in/out memory. If a segment is swapped in and its location changes, what information in the table should be updated?
  – Base field of the entry describing the location of that segment in physical memory

• Can it keep program from accidentally overwriting its own code?
  – Yes. Code segment can be set as read-only

• Can it share code/data with other processes?
  – Yes. A physical segment can be pointed to by multiple processes
UNIX fork() and Copy-on-Write (CoW)

- Efficient implementation of fork(): copy-on-write
  - Copy the segment table
  - Set the read only flag for all segments, for both parent and child, and increment the reference count (initially, 0)
  - When child or parent writes to any segment (e.g., stack, heap)
    - An exception will be triggered, and the control flow traps into kernel
    - Kernel copies the segment, set the new copy as r/w
    - Decrement the reference count for the original segment; if it is 0, remove the write-protection
  - Lazy copy of segments in order to avoid unnecessary copy and a slow return from fork()
Paging

• Divide physical memory into fixed-sized blocks called frames
  – Size is power of 2, between 512 bytes and 16 Mbytes (or larger), depending on specific systems
• Divide logical memory into blocks of same size called pages
• A page can be put at any frame
Virtual Address

- Address generated by CPU is divided into:
  - **Page number** – used as an index into a page table, which is an array of entries for translating page # to frame #
  - **Page offset** – combined with frame # addr to define the physical address that is sent to the memory unit

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>d</td>
</tr>
</tbody>
</table>

m -n

n
Page Table

• Each process has its own page table, which is an array of page table entries for address translation

• A page table entry
  – present bit (1 if the page is in physical memory)
  – frame number
  – protection bits: read, write, exec
  – modified bit (1 if the page has been modified)
  – reference bit (1 if page was recently referenced)
Steps of address translation using a page table

– Use the page number in the address to search in page table
– If the entry’s present bit is 1, build physical address: (frame #, offset)
– Otherwise, a page fault
Paging Hardware

CPU

logical address

physical address

f0000 ... 0000

f1111 ... 1111

physical memory

page table

f
Questions

• What is updated at a process switch to assist address translation?
  – Register pointing to the page table, …

• Given 4KB page size and a 32-bit virtual address, calculate m (# of bits for virtual address space), n (# of bits for offset within a page), and the number of bits for representing page #
  – m = 32; n = 12
  – So page # is represented by (32 – 12 =) 20 bits
Virtual address space

Virtual address space: 16 bytes \( (m = 4) \)
Memory size per page: 4 bytes \( (n = 2) \)

Write the page table and use it to translate the virtual address 0xa

<table>
<thead>
<tr>
<th>Page #</th>
<th>Frame #</th>
<th>Presence</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(present = 0)</td>
<td></td>
</tr>
</tbody>
</table>
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 => $2^{32}/2^{12} = 1M$ entries, 4 bytes for each entry => 4M memory
  - 64-bit machine currently uses 48 bits address space
    - $2^{48}/2^{12} = 64G$ entries => 256 G memory
Multilevel Page Table

Virtual Address

10 bits 10 bits 12 bits

Root page table (contains 1024 PTEs)

Frame # Offset

4-kbyte page table (contains 1024 PTEs)

Page Frame

Program Paging Mechanism Main Memory
Questions

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

• 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

1GB

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

Random stack offset

0xc0000000 == TASK_SIZE

Random mmap offset

RLIMIT_STACK (e.g., 8MB)

3GB

Stack (grows down)

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

program break
brk

start_brk
Random brk offset

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)

Virtual Page
Page Frame
Access

Matching Entry

Page Table Lookup

Physical Memory

Virtual Address
Page# Offset

Physical Address
Frame Offset

CSCE 311 – Operating Systems
TLB and Page Table Translation

Processor \(\rightarrow\) TLB

Virtual Address \(\rightarrow\) TLB

Hit \(\rightarrow\) Frame

Offset \(\rightarrow\) Physical Address

\(+\) \(\rightarrow\) Physical Memory

Data \(\rightarrow\) Processor

Virtual Address \(\rightarrow\) Page Table

Virtual Address Miss \(\rightarrow\) Page Table

Invalid \(\rightarrow\) Raise Exception

Valid \(\rightarrow\) Frame

Frame \(\rightarrow\) TLB

Data \(\rightarrow\) Physical Memory
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

Physical Memory

Translation Lookaside Buffer (TLB)

Matching Entry

Matching Superpage

Page Table Lookup

Virtual Address

Page# Offset

SP Offset

Superpage Superframe
(SP) or Page#
(SF) or Frame
Access

Physical Address

Frame Offset

SF Offset

CSCE 311 – 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
Implementation

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

CSCE 311 – 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>Processor 1 TLB</th>
<th>Process ID</th>
<th>VirtualPage</th>
<th>PageFrame</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0053</td>
<td>0x0003</td>
<td>R/W</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0x40FF</td>
<td>0x0012</td>
<td>R/W</td>
<td></td>
</tr>
</tbody>
</table>

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

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

CSCE 311 – Operating Systems
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