Outline

• Fixed partitions
• Dynamic partitions

Contiguous allocation:
Each process occupies a contiguous memory region in the physical memory.

• Segmentation
• Paging

Non-contiguous allocation:
Each process occupies multiple memory regions scattered in the physical memory.
Fixed vs Dynamic partitioning

- **Fixed partitioning**: street parking with meters
  - **Internal Fragmentation**: Parking a small car in a parking slot leads to internal fragmentation

- **Dynamic partitioning**: street parking w/o meters
  - **External Fragmentation**: When small cars drive out, those non-contiguous “holes” left cannot be used by a big SUV
Previous class: why is non-contiguous allocation better?

- **Reason 1**: Contiguous allocation does not exploit the Working Set theory
- **Non-contiguous** allocation: the memory needed by a process is cut into
  - A couple of parts (this is called segmentation)
  - Many equal-sized parts (this is called paging)
- Non-contiguous allocation only **swaps in** the parts (segments or pages) corresponding to the current Working Set into memory, so that your memory allows more active processes
- **Reason 2**: the fragmentation problem is mitigated
  - “Small chunks” can now be used by segments or pages
Segmentation vs. Paging

• Segmentation
  – Causes external fragmentation
  – Does not handle dynamic components (e.g., stack, heap) very well
  – Swapping a segment is still slow
  – Working Set is exploited at the segment granularity

• Paging
  – No external fragmentation
  – Allows dynamic components grow and shrink flexibly
  – Swapping occurs at the page granularity
  – Working Set is exploited at the page granularity
  – Cons: Complex and expensive address translation
### Paging Example

#### Internal fragmentation?
- Yes, but it only occurs when the requested size is not a multiple of pages. E.g., a process that requests 3.1 pages of space will get 4 pages

#### External fragmentation?
- No, any “holes”, i.e., page frames, left by the exited process can be reused happily

---

<table>
<thead>
<tr>
<th>Frame number</th>
<th>Main memory</th>
<th>Frame number</th>
<th>Main memory</th>
<th>Frame number</th>
<th>Main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A.0</td>
<td>0</td>
<td>A.0</td>
<td>0</td>
<td>A.0</td>
</tr>
<tr>
<td>1</td>
<td>A.1</td>
<td>1</td>
<td>A.1</td>
<td>1</td>
<td>A.1</td>
</tr>
<tr>
<td>2</td>
<td>A.2</td>
<td>2</td>
<td>A.2</td>
<td>2</td>
<td>A.2</td>
</tr>
<tr>
<td>3</td>
<td>A.3</td>
<td>3</td>
<td>A.3</td>
<td>3</td>
<td>A.3</td>
</tr>
<tr>
<td>4</td>
<td>B.0</td>
<td>4</td>
<td>B.0</td>
<td>4</td>
<td>B.0</td>
</tr>
<tr>
<td>5</td>
<td>B.1</td>
<td>5</td>
<td>B.1</td>
<td>5</td>
<td>B.1</td>
</tr>
<tr>
<td>6</td>
<td>B.2</td>
<td>6</td>
<td>B.2</td>
<td>6</td>
<td>B.2</td>
</tr>
<tr>
<td>7</td>
<td>C.0</td>
<td>7</td>
<td>C.0</td>
<td>7</td>
<td>C.0</td>
</tr>
<tr>
<td>8</td>
<td>C.1</td>
<td>8</td>
<td>C.1</td>
<td>8</td>
<td>C.1</td>
</tr>
<tr>
<td>9</td>
<td>C.2</td>
<td>9</td>
<td>C.2</td>
<td>9</td>
<td>C.2</td>
</tr>
<tr>
<td>10</td>
<td>C.3</td>
<td>10</td>
<td>C.3</td>
<td>10</td>
<td>C.3</td>
</tr>
<tr>
<td>11</td>
<td>D.0</td>
<td>11</td>
<td>D.0</td>
<td>11</td>
<td>D.0</td>
</tr>
<tr>
<td>12</td>
<td>D.1</td>
<td>12</td>
<td>D.1</td>
<td>12</td>
<td>D.1</td>
</tr>
<tr>
<td>13</td>
<td>D.2</td>
<td>13</td>
<td>D.2</td>
<td>13</td>
<td>D.2</td>
</tr>
<tr>
<td>14</td>
<td>D.3</td>
<td>14</td>
<td>D.3</td>
<td>14</td>
<td>D.3</td>
</tr>
</tbody>
</table>

(a) Fifteen Available Frames  
(b) Load Process A  
(c) Load Process B  
(d) Load Process C  
(e) Swap out B  
(f) Load Process D
What will happen if the RAM is less than the total size of the working sets of the processes?

Thrashing. Consider a simple example: `memcpy(dst, src, 4096)`. Assume `dst` and `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 versa.
Logical view and physical view

- The layout of processes in physical memory forms a **physical view**
- A compiler, however, compiles a program based on some **logical view**
Logical view vs. physical view

Processor’s View

Physical Memory

VPage 0
VPage 1
VPage N

Stack
Stack
Stack

Frame 0
Frame M

Code
Data
Heap

Code0
Data0
Heap0
Code1
Data1
Heap1

Code
Data
Heap

Heap2
Stack1
Stack0

CIS 5512 - Operating Systems
Addresses

• Logical address
  – Reference to a memory location in the logic view
  – Used by processor and compiler
  – In Segmentation and Paging, also called Virtual Address
  – Intel: logical address + segment base = virtual address.
    • In Linux, the segment base is 0, so still logical address = virtual address

• Physical address
  – Actual location in main memory
  – Sent to the memory bus
Address Translation

• Typically, a hardware component MMU (Memory Management Unit) does the translation

![Diagram of address translation]
Registers for Contiguous Allocation

- A pair of **base** and **limit (or, bound)** registers define the partition in physical memory
Address translation for Contiguous Allocation

[Diagram showing the process of address translation]

- CPU
- Logical address
- Limit register
- Base register
- Physical address
- Memory
- Trap: Addressing error
Address translation for Contiguous Allocation - Question

• What is updated at a process switch to assist address translation?
  – Base & limit registers
• Can it keep program from accidentally overwriting its own code?
  – No
• Can it share code/data with other processes?
  – No
Address translation for Segmentation

• A logical address consists of
  – Segment number/label
  – Offset (inside that segment)

• Each process has a segment table
  – Each entry in the table saves the information for a segment: base, bound, and access permission
Address translation for Segmentation

• Processes can share segments easily, how?

CIS 5512 - Operating Systems
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()

• UNIX fork
  – *Virtually* makes a complete copy of the parent’s memory
  – (1) any given variable (except for the return value of *fork()* has the same value between parent and child;
    (2) when you change a variable of a process, the copy in the other will not be changed, since the two processes are independent

• A very slow implementation:
  – Copy every segment

• A slightly improved implementation:
  – Share the code segment and copy others
UNIX fork() and Copy-on-Write (CoW)

- Efficient implementation: 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()
Questions

• How does the system distinguish it is a valid write to a CoW segment or just an invalid write?
  – The ground truth permission information for each segment is stored separately in the kernel
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>
<tr>
<td>m -n</td>
<td>n</td>
</tr>
</tbody>
</table>
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
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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
</tbody>
</table>

Physical Memory

<table>
<thead>
<tr>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
</tbody>
</table>

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>Page Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>frame # = 3</td>
</tr>
<tr>
<td>1</td>
<td>frame # = 2</td>
</tr>
<tr>
<td>2</td>
<td>frame # = 0</td>
</tr>
<tr>
<td>3</td>
<td>(present = 0)</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

Frame # Offset

Root page table ptr

Root page table (contains 1024 PTEs)

4-kbyte page table (contains 1024 PTEs)

Program Paging Mechanism Main Memory

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

Stack (grows down)

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

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

Virtual Address

Page#  Offset

Translation Lookaside Buffer (TLB)

Virtual Page  Page Frame  Access

Matching Entry

Physical Address

Frame  Offset

Page Table Lookup

Physical Memory
TLB and Page Table Translation
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 * 200 = 3ns; 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

Page# Offset
SP Offset

Translation Lookaside Buffer (TLB)

Superpage Superframe (SP) or Page# Frame Access

Matching Entry

Matching Superpage

Page Table Lookup

SF Offset

Physical Address

Frame Offset

Physical Memory
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
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
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

• What is the benefit of CoW?
• Compare conventional page table and inverted page table
• Why do we use multi-level page tables for address translation?
• What is TLB? What is it used for?
Backup pages
What is internal fragmentation? What is external fragmentation

**Fragmentation**: small useless chunks
**Internal fragmentation**: small useless chunks that are inside the allocated partitions
**External fragmentation**: ...outside...
How Buddy System Helps

• Three sentences:
  – Split-based allocation
  – Coalescing-buddies-based deallocation
  – Freelists-based implementation: an array of free lists, each storing the unallocated partitions of certain size

• Internal fragmentation:
  – Best fit every time
  – Upper bound: ~50%. E.g., 128.01k is rounded up to 256k

• External fragmentation:
  – Coalescing can only occur between buddies
  – Restriction about splitting, e.g., 64kB minimal
If swapping is not used, do the schemes of segmentation/paging still have advantages over contiguous memory allocation?

Yes. Segmentation and paging can be regarded as a special type of dynamic partitioning, as a process can occupy multiple, rather than one, partitions. Those “small chunks” that are probably “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 will see other advantages, e.g., sharing
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 #
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:
  – \( \text{CPL} \leq \text{DPL} \)
  – 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