CSCE 311 - Operating Systems
Memory Management –
Address Translation

Qiang Zeng, Ph.D.
Fall 2018
Outline

• Fixed partitions
• Dynamic partitions
• Buddy system

• Segmentation
• Paging

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

Non-contiguous allocation:
Each process occupies multiple memory regions scattered in the physical memory.
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 segments or pages
  – 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
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.
What will happen if the RAM is less than the total size of the working sets of the processes?

Thrashing. Consider a simple example: \textit{memcpy}(dst, src, 4096). Assume \textit{dst} and \textit{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

<table>
<thead>
<tr>
<th>VPage 0</th>
<th>VPage 1</th>
<th>VPage N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>Data</td>
<td>Stack</td>
</tr>
<tr>
<td>Heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Physical Memory

<table>
<thead>
<tr>
<th>Frame 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code0</td>
</tr>
<tr>
<td>Data0</td>
</tr>
<tr>
<td>Heap1</td>
</tr>
<tr>
<td>Code1</td>
</tr>
<tr>
<td>Heap0</td>
</tr>
<tr>
<td>Data1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frame M</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack0</td>
</tr>
<tr>
<td>Stack1</td>
</tr>
<tr>
<td>Heap2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
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
Registers for Contiguous Allocation

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

CPU

limit register

<

limit register

yes

+ physical address

no

trap: addressing error

memory
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?
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
Sharing segments

Processor’s View

Process 1’s View

Virtual Memory

Processor

Virtual Address 0x0500

Code

Data

Heap

Stack

Processor

Virtual Address 0x0500

Code

Data

Heap

Stack

Segment Table

Base	Bound	Access

Code		Read

Data	R/W

Heap	R/W

Stack	R/W

Physical Address

Process 2’s View

Virtual Memory

Processor

Virtual Address 0x0500

Code

Data

Heap

Stack

Processor

Virtual Address 0x0500

Code

Data

Heap

Stack

Segment Table

Base	Bound	Access

Code		Read

Data	R/W

Heap	R/W

Stack	R/W

Physical Address

CSCE 311 - Operating Systems
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
Summary

- Thrashing
- Logical view vs. Physical view
- Address translation for
  - Contiguous memory allocation
  - Segmentation
  - Paging (next lecture)
- fork() and cow