Frequently asked questions from the previous class survey

- Virtual addresses

Topics covered in this lecture

- Contiguous memory allocations
- Fragmentations
  - External and Internal
- Segmentation
- Paging

Partitioning of memory

- Main memory needs to accommodate the OS and user processes
- Divided into two partitions
  - Resident OS
    - Usually low memory
  - User processes

Memory Mapping and Protection

- Base register (also referred to as a relocation register)
  - Smallest physical address
- Limit register
  - Range of logical addresses
Memory Mapping and Protection
- When CPU scheduler selects a process for execution
  - Base and limit registers reloaded as part of context switch
- Every address generated by the CPU
  - Checked against the relocation/limit registers

Memory Allocation: Fixed Partition method
- Divide memory into several fixed-size partitions
  - Each partition contains exactly one process
- Degree of multiprogramming
  - Bound by the number of partitions

Memory allocation: Variable-partition method [1/2]
- Used in batch environments
- OS maintains table tracking memory utilization
  - What is available?
  - Which ones are occupied?
- Initially all memory is available
  - Considered a large memory gap
  - Eventually many memory gaps will exist

Memory allocation: Variable-partition method [2/2]
- OS orders processes according to the scheduling algorithm
- Memory allocated to processes until requirements of the next process cannot be met
  - Wait till a larger block is available
  - Check if smaller requirements of other processes can be met

Variable-partition method: Reclaiming spaces
- When process arrives if space is too large
  - Split into two
- When process terminates
  - If released memory is adjacent to other memory gaps
  - Fuse to form a larger space
Splitting and Fusing Memory spaces

Dynamic Storage Allocation Problem
- Satisfying a request of size \( n \) from the set of available spaces
  - First fit
  - Best fit
  - Worst fit

First fit
- Scan list of segments until you find a memory-gap that is big enough
- Gap is broken up into two pieces
  - One for the process
  - The other is unused memory

Best Fit
- Scan the entire list from beginning to the end
- Pick the smallest memory-gap that is adequate to host the process

Comparing Best Fit and First Fit
- Best fit is slower than first fit
- Surprisingly, it also results in more wasted memory than first fit
  - Tends to fill up memory with tiny, useless gaps

Worst fit
- How about going to the other extreme?
  - Always take the largest available memory-gap
  - Perhaps, the new memory-gap would be useful
  - Simulations have shown that worst fit is not a good idea either
In our discussions so far ...

- Logical/virtual memory is **one-dimensional**
  - Logical addresses go from 0 to some max value

- Many problems can benefit from having two or more **separate** logical address spaces

---

A compiler has many tables that are built up as compilation proceeds

- Source Text
- Symbol table
- Names and attributes of variables

- Constants Table
  - Integer and floating point constants

- Parse tree
- Syntactic analysis of program

- Stack
- Procedure calls within the compiler

---

One dimensional address space with growing tables

- Program has an exceptional number of variables

---

One dimensional address space with growing tables

- Program has an exceptional number of variables

---

Base and limits translation lacks many of the features needed to support modern programs

- Base and limits translation supports only **coarse-grained** protection at the level of the entire process
  - It is not possible to prevent a program from overwriting its own code, for example
  - It is also difficult to share regions of memory between two processes

- Since the memory for a process needs to be contiguous ...
  - Supporting dynamic memory regions, such as for heaps, thread stacks, or memory mapped files, becomes difficult to impossible
Options available to the compiler

- Say that compilation cannot continue
  - Not cool
- Play Robin Hood
  - Take space from tables with room
  - Give to tables with little room

What would be really cool ...

- Free programmer from having to manage expansion and contraction of tables

But how?

- Provide many completely independent address spaces
  - Segments
- Each segment has linear sequence of addresses
  - 0 to max

Segments and Base/Limit registers

- The hardware supports an array of pairs of base and bounds registers, for each process
  - Segment Table
  - Each entry in the array controls a portion, or segment, of the virtual address space
  - The physical memory for each segment is stored contiguously, but different segments can be stored at different locations
  - For example, code and data segments are not immediately adjacent to each other in either the virtual or physical address space

Other things about segments

- Different segments can and do have different lengths
- Segments grow and shrink independently without affecting each other.
  - For example, consider a segment for the stack
  - Size increase: something pushed on stack segment
  - Size decrease: something popped off of stack segment

Users view memory as a collection of variable-sized segments
Segmentation

- Logical address space is a collection of segments
- Segments have name and length
- Addresses specify
  - Segment name
  - Offset within the segment
- Tuple: <segment-number, offset>

Segmentation Addressing Example

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Table</th>
<th>Segment 0</th>
<th>Segment 1</th>
<th>Segment 2</th>
<th>Segment 3</th>
<th>Segment 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limit</td>
<td>Base</td>
<td>Limit</td>
<td>Limit</td>
<td>Limit</td>
<td>Limit</td>
<td>Limit</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>400</td>
<td>1400</td>
<td>4300</td>
<td>4300</td>
<td>4300</td>
<td>4300</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
<td>4400</td>
<td>4400</td>
<td>4400</td>
<td>4400</td>
<td>4400</td>
</tr>
<tr>
<td>3</td>
<td>1000</td>
<td>4700</td>
<td>4700</td>
<td>4700</td>
<td>4700</td>
<td>4700</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>5800</td>
<td>5800</td>
<td>5800</td>
<td>5800</td>
<td>5800</td>
</tr>
</tbody>
</table>

Segmentation Hardware

- CPU
- Address
- Segment table
- Physical address
- YES: The offset must be between 0 and the segment limit
- NO: TRAP: Addressing Error

Contiguous Memory Allocation: Fragmentation

- As processes (and segments) are loaded/removed from memory
  - Free memory space is broken into small pieces
- External fragmentation
  - Enough space to satisfy request; BUT
  - Available spaces are not contiguous

Fragmentation: Example

- Process P3 cannot be loaded because memory space is fragmented
Fragmentation can be internal as well

- Memory allocated to process may be **slightly larger** than requested
- **Internal fragmentation**
  - Unused memory is internal to blocks

Compaction: Solution to external fragmentation

- **Shuffle** memory contents
  - Objective: Place free memory into large block
- Not possible if relocation is static
  - Load time
- Approach involves moving:
  1. Processes towards one end
  2. Gaps towards the other end

Compaction: Example

```
P1
P2
P3
P4
P5
```

Memory compaction is time intensive and is usually not done

- Let's consider a machine with 1 GB of RAM
- The machine can copy 4 bytes in 20 nanoseconds
- Time to compact all the memory:
  \[ 10^9 \times \left(\frac{20 \times 10^{-9}}{4}\right) = 5 \text{ seconds (approximately)} \]
  
  Note: 1 GB is approximately \(10^9\) bytes.

Overview of how mapping of logical and physical addresses is performed

```
CPU

Virtual address

Memory Management Unit (MMU)

Translation Lookaside Buffer (TLB)

Physical address

Physical Memory
```

[Page Table may be shared here]
The Paging memory management scheme
- Physical address space of process can be non-contiguous
- Solves problem of fitting variable-sized memory chunks to backing store
  - Backing store has fragmentation problem
    - Compaction is impossible

Basic method for implementing paging
- Break memory into fixed-sized blocks
  - Physical memory: frames
  - Logical memory: pages
- Backing store is also divided the same way

What will seem odd, and perhaps cool, about paging [1/2]
- While a program thinks of its memory as linear ...
  - It is usually scattered throughout physical memory in a kind of abstract mosaic
  - The processor will execute one instruction after another using virtual addresses
    - The virtual addresses are still linear
    - However, an instruction located at the end of a page will be located in a completely different region of physical memory from the next instruction at start of another page

What will seem odd, and perhaps cool, about paging [2/2]
- Data structures appear to be contiguous using virtual addresses
  - But a large matrix is scattered across many physical page frames

Paging: Analogy
- Shuffling several decks of cards together
  - A single process in its virtual address page sees the cards of a single deck in order
    - A different process sees a completely different deck, but it will also be in order
  - In physical memory, however, the decks of all processes currently running will be shuffled together, apparently at random
  - Page tables are the magician’s assistant in locating cards from the shuffled decks
CS370: Operating Systems [Fall 2018]  
Dept. Of Computer Science, Colorado State University

---

**Paging: Logical and Physical Memory**

<table>
<thead>
<tr>
<th>Logical Memory</th>
<th>Physical Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page 0</td>
<td>Page 0</td>
</tr>
<tr>
<td>Page 1</td>
<td>Page 1</td>
</tr>
<tr>
<td>Page 2</td>
<td>Page 2</td>
</tr>
<tr>
<td>Page 3</td>
<td>Page 3</td>
</tr>
</tbody>
</table>

---

**Paging Hardware: Performing address translation**

- Logical Address: f000…000, f111…111
- Page number
- Page offset
- Physical Address: (000…000)

---

The contents of this slide-set are based on the following references: