Lecture 13: Main Memory
(Chapter 8)
Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
- **Logical** vs. **Physical address space**:
  - Logical address: generated by the CPU. Also called virtual address.
  - Physical address: the address seen by the memory unit.
Address binding of instructions and data to memory addresses can happen at three different stages:

- **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes.
- **Load time**: If not, the compiler must generate **relocatable code**. Final binding happens at load time.
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers).

How are logical and virtual addresses in each of these cases? (i.e., identical or different?)
Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register
A pair of **base** and **limit** registers define the logical address space
How To Protect Processes?

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses
  - MMU maps logical address \textit{dynamically}
Swapping

1. swap out
2. swap in
Memory Allocation

- Problem: how to allocate memory for multiple processes (in a multi-programming environment)

- Solutions:
  - Contiguous allocation
    - Fixed partitions
    - Dynamic partitions
  - Paging
Contiguous Allocation (Fixed Partitions)

(a) Separate input queues for each partition
(b) Single input queue
Contiguous Allocation (Dynamic Partitions)

- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes, held in high memory

- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a large enough hole
  - Operating system maintains information about:
    a) allocated partitions
    b) free partitions (hole)
Dynamic Storage-Allocation Problem

How to satisfy a request of size $n$ from a list of free holes

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

Statistics:

- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- Fragmentation: for $N$ allocated blocks, another $N/2$ wasted due to fragmentation
Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Reduce external fragmentation by **compaction**

- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time
- I/O problem
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers
- Can be very slow: 256MB of memory, copy 4 bytes in 40ns → compacting memory in 2.7 sec
  - Almost never used
Paging

- **Key idea:** physical address space of a process can be noncontiguous

- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 and 8,192 bytes)

- Divide logical memory into blocks of same size called **pages**

- To run a program of size $n$ pages, need to find $n$ free frames and load program

- Set up a page table to translate logical to physical addresses
### Paging Example

32-byte memory and 4-byte pages
Address Translation Scheme

- Address generated by CPU is divided into:
  
  - **Page number** \((p)\) – used as an index into a page table which contains base address of each page in physical memory
  
  - **Page offset** \((d)\) – combined with base address to define the physical memory address that is sent to the memory unit

  \[
  \begin{array}{|c|c|}
  \hline
  \text{page number} & \text{page offset} \\
  \hline
  p & d \\
  m - n & n \\
  \hline
  \end{array}
  \]

- For given logical address space \(2^m\) and page size \(2^n\)
Paging Hardware

The diagram illustrates the process of translating a logical address to a physical address in a paging system. The logical address is broken down into page and directory components, which are then used to access the page table. The page table entry maps the logical page number to the physical page number. This process ensures that the correct page is loaded into memory when a process references a page.
In practice, why is it preferred that the number of frames in memory is a power of two?

Write brief answers to the following questions (on memory allocation):
- What is the difference between fixed partitioning and dynamic partitioning?
- What do fixed partitioning and dynamic partitioning suffer from, respectively?
- What does paging allow to do that was not possible with partitioning?
- Cite important differences between virtual paging and simple paging.
When a New Process is Created...

How large can the logical address space be?

Before allocation

After allocation
Virtual Memory That is Larger Than Physical Memory

Key Idea: Only part of the program needs to be in memory for execution.

Needs: Demand paging (or demand segmentation)
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

- **Lazy swapper** – never swaps a page into memory unless page needed
  - Swapper that deals with pages is a pager
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated ($v \Rightarrow \text{in-memory}, \ i \Rightarrow \text{not-in-memory}$)
- Initially valid–invalid bit is set to $i$ on all entries

- During address translation, if valid–invalid bit in page table entry is $i \Rightarrow \text{page fault}$
Page Fault

If there is a reference to a page, first reference to that page will trap to the operating system: page fault.
Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault

- Effective Access Time (EAT)
  \[
  EAT = (1 - p) \times \text{memory access} + p \times \left( \text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead} \right)
  \]
Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

\[ EAT = (1 - p) \times 200 + p \times (8 \text{ milliseconds}) \]
\[ = (1 - p) \times 200 + p \times 8,000,000 \]
\[ = 200 + p \times 7,999,800 \]

- If one access out of 1,000 causes a page fault, then
  \[ EAT = 8.2 \text{ microseconds}. \]
  This is a slowdown by a factor of 40!!
Copy-on-Write

- Copy-on-Write (COW) allows processes to initially share the same pages in memory.
- If a process modifies a shared page, only then is the page copied.
After Process 1 Modifies Page C
What happens if there is no free frame?

1. Swap out the victim page.
2. Change the valid bit to invalid.
3. Swap in the desired page.
4. Reset the page table for the new page.
Page Replacement Algorithms

- Performance objective: lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

Common page replacement algorithms:
- FIFO
- Least Recently Used
- Optimal (theoretical only)
- Counting: LFU and MFU
- ... Countless others in the literature
Each process needs *minimum* number of pages

Two major allocation schemes
- fixed allocation
- priority allocation

Global vs. local page replacement
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement** – each process selects from only its own set of allocated frames
Multiplexing memory between guest OSes:
- What if OS1 needs more than OS2?
- What if OS3 is not using its share of memory and OS1 needs more?
- What if OS1, ..., OS10 all use the same (large) OS libraries – 10 copies in memory?

... while:
- Not modifying the guest OS (unlike Xen)
- Not knowing how each OS does its memory management
Memory Protection

Valid (v) or Invalid (i) Bit In A Page Table

```
<table>
<thead>
<tr>
<th>Frame Number</th>
<th>Valid-Invalid Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>v</td>
</tr>
<tr>
<td>1</td>
<td>v</td>
</tr>
<tr>
<td>2</td>
<td>v</td>
</tr>
<tr>
<td>3</td>
<td>v</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
</tr>
<tr>
<td>5</td>
<td>v</td>
</tr>
<tr>
<td>6</td>
<td>i</td>
</tr>
<tr>
<td>7</td>
<td>i</td>
</tr>
</tbody>
</table>
```

```plaintext
000000
0 2  v
1 3  v
2 4  v
3 7  v
4 8  v
5 9  v
6 0  i
7 0  i
```