CS370 Operating Systems
Colorado State University
Yashwant K Malaiya
Spring 2019 L17
Main Memory

Slides based on
• Text by Silberschatz, Galvin, Gagne
• Various sources
FAQ

• Where are registers, cache, main memory in a system?
  – Registers: in the processor
  – Cache: outside the processor, but generally on the same chip
  – Memory: Separate chips (except for System-on-a-chip)

• A program is compiled.linked and executed using a remote terminal. Where does compilation and execution take place?

• Symbolic addresses: data or instructions

• Local variables on stack?
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, causing a stall.

Cache sits between main memory and CPU registers making main memory appear much faster.

Ch 8

Ch 9

Ch 10,11,12: Disk, file system    Cache: CS470
Partitioning: Base and Limit Registers

- Base and Limit for a process
  - **Base**: Smallest legal physical address
  - **Limit**: Size of the range of physical address
- A pair of **base** and **limit registers** define the logical address space for a process
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
- Base: **Smallest** legal physical address
- Limit: Size of the **range** of physical address
- Eg: Base = 300040 and limit = 120900
- Legal: 300040 to \((300040 + 120900 -1) = 420939\)
• The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  – **Logical address** – generated by the CPU; also referred to as **virtual address**
  – **Physical address** – address seen by the memory unit

• **Logical address space** is the set of all logical addresses generated by a program

• **Physical address space** is the set of all physical addresses
Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
  - Many methods possible, we will see them soon
- Consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called relocation register
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with logical addresses; it never sees the real physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical address bound to physical addresses
Dynamic relocation using a relocation register
Linking: Static vs Dynamic

• **Static linking** – system libraries and program code combined by the loader into the binary image
  – Every program includes library: wastes memory

• **Dynamic linking** – linking postponed until execution time
  • Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
  • Stub replaces itself with the address of the routine, and executes the routine
  • Operating system checks if routine is in processes’ memory address
    – If not in address space, add to address space
  • Dynamic linking is particularly useful for
    – shared libraries
Dynamic loading of routines

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- OS can help by providing libraries to implement dynamic loading
Swapping a process

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
Schematic View of Swapping

1. swap out
2. swap in

operating system

user space

main memory

process $P_1$

process $P_2$

backing store
Context Switch Time including Swapping

• If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
• Context switch time can then be very high
• 100MB process swapping to hard disk with transfer rate of 50MB/sec
  – Swap out time of 100MB/50MB/s = 2 seconds
  – Plus swap in of same sized process
  – Total context switch swapping component time of 4 seconds + some latency
• Can reduce if reduce size of memory swapped – by knowing how much memory really being used by a process
• Standard swapping not used in modern operating systems
  – But modified version common
    • Swap only when free memory extremely low
Memory Allocation Approaches

• **Contiguous allocation:** entire memory for a program in a single contiguous memory block. Find where a program will “fit”. earliest approach

• **Segmentation:** program divided into logically divided “segments” such as main program, function, stack etc.
  – Need table to track segments.

• **Paging:** program divided into fixed size “pages”, each placed in a fixed size “frame”.
  – Need table to track pages.
Contiguous Allocation
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vectors
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory
• **Registers** used to protect user processes from each other, and from changing operating-system code and data
  
  – **Relocation (Base) register** contains value of smallest physical address
  
  – **Limit register** contains range of logical addresses – each logical address must be less than the limit register

• **MMU** maps logical address \textit{dynamically}
Hardware Support for Relocation and Limit Registers

MMU maps logical address *dynamically*.

Physical address = relocation reg + valid logical address
Multiple-partition allocation

- Degree of multiprogramming limited by number of partitions
- **Variable-partition** sizes for efficiency (sized to a given process’ needs)
- **Hole** – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Operating system maintains information about:
  a) allocated partitions  
  b) free partitions (hole)
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

**Simulation studies:**

- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
- 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 holes
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

- Simulation analysis reveals that given $N$ blocks allocated, $0.5 \ N$ blocks lost to fragmentation
  - $1/3$ may be unusable -> **50-percent rule**
• 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
Paging vs Segmentations
Pages

• Pages and frames
• Addresses: page number, offset
• Page tables: mapping from page # to frame #
• TLB: page table caching
• Memory protection and sharing
• Multilevel page tables
Paging

- Divide physical memory into fixed-sized blocks called **frames (or page frames)**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- 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
- Still have Internal fragmentation
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
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

- For given logical address space $2^m$ and page size $2^n$
Paging Hardware

Page number p to frame number f
$n=2$ and $m=4$

Logical add. space = $2^4$ bytes,

$2^2=4$-byte pages

32-byte physics memory with 8 frames

8 frames
Frame number 0-to-7

Example:
Logical add: 00 10
Physical Add: 101 10

Page 0 maps to frame 5
Paging (Cont.)

- **Internal fragmentation**
  - Ex: Page size = 2,048 bytes, Process size = 72,766 bytes
    - 35 pages + 1,086 bytes
    - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
    - But each page table entry takes memory to track
  - Page size growing over time
    - X86-64: 4 KB (common), 2 MB (“huge” for servers), 1GB (“large”)

- Process view and physical memory now very different

- By implementation, a process can only access its own memory unless ..
Free Frame allocation

A new process arrives
That needs four pages

Before allocation

After allocation
Implementation of Page Table

Page table is kept in main memory

- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data/instruction

The *two memory access problem* can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

One page-table
For each process

TLB: cache for page Table
Caching: The General Concept

• Widely used concept:
  – keep small subset of information likely to needed in near future in a fast accessible place
  – Hopefully the “Hit Rate” is high

Examples:
  – Cache Memory (“Cache”):
    Cache for Main memory Default meaning for this class
  – Browser cache: for browser
  – Disk cache
  – Cache for Page Table: TLB

Challenges:
  – 1. Is the information in cache? 2. Where?
  – Hit rate vs cache size
Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
- Otherwise need to flush TLB at every context switch

TLBs typically small (64 to 1,024 entries)

On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered
  - Some entries can be **wired down** for permanent fast access
Associative Memory

- Associative memory – **parallel** search using hardware
  - Electronics is very expensive

<table>
<thead>
<tr>
<th>Page #</th>
<th>Frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Address translation (p, d)
  - If p is in associative register, get frame # out ("Hit")
  - Otherwise get frame # from page table in memory ("Miss")
Paging Hardware With TLB

TLB Miss: page table access may be done using hardware or software
Effective Access Time

- **Associative Lookup** = \( \varepsilon \) time units
  - Can be < 10% of memory access time
- **Hit ratio** = \( \alpha \)
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- **Effective Access Time (EAT)**: probability weighted
  \[
  \text{EAT} = (100 + \varepsilon) \alpha + (200+\varepsilon)(1 - \alpha)
  \]
- **Ex:**
  Consider \( \alpha = 80\% \), \( \varepsilon \) = negligible for TLB search, 100ns for memory access
  - \( \text{EAT} = 0.80 \times 100 + 0.20 \times 200 = 120\text{ns} \)
- **Consider more realistic hit ratio** -> \( \alpha = 99\% \),
  - \( \text{EAT} = 0.99 \times 100 + 0.01 \times 200 = 101\text{ns} \)
Memory Protection

• Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  – Can also add more bits to indicate page execute-only, and so on

• **Valid-invalid** bit attached to each entry in the page table:
  – “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  – “invalid” indicates that the page is not in the process’ logical address space

• Any violations result in a trap to the kernel
Valid (v) or Invalid (i) Bit In A Page Table

“invalid”: page is not in the process’s address space.
Shared Pages among Processes

- **Shared code**
  - One copy of read-only (reentrant non-self modifying) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed

- **Private code and data**
  - Each process keeps a separate copy of the code and data
  - The pages for the private code and data can appear anywhere in the logical address space
Shared Pages Example

ed1, ed2, ed3
(3, 4, 6) shared
Overheads in paging: Page table and internal fragmentation

Optimal Page Size:
page table size vs internal fragmentation tradeoff

• Average process size = \(s\)
• Page size = \(p\)
• Size of each entry in page table = \(e\)
  – Pages per process = \(s/p\)
  – \(se/p\): Total page table space for average process

• Total Overhead = Page table overhead + Internal fragmentation loss
  = \(se/p + p/2\)
• Total Overhead = $se/p + p/2$
• Optimal: Obtain derivative of overhead with respect to $p$, equate to 0
  
  $$-se/p^2 + 1/2 = 0$$
• i.e. $p^2 = 2se$ or $p = (2se)^{0.5}$

**Assume** $s = 128$KB and $e=8$ bytes per entry

• Optimal page size = 1448 bytes
  
  – In practice we will never use 1448 bytes
  – Instead, either 1K or 2K would be used
    • **Why?** Pages sizes are in powers of 2 i.e. $2^x$
    • Deriving offsets and page numbers is also easier