Notice: The slides for this lecture have been largely based on those accompanying the textbook Operating Systems Concepts with Java, by Silberschatz, Galvin, and Gagne (2003). Many, if not all, the illustrations contained in this presentation come from this source.
Logical vs. Physical Address Space

- 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 and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.
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

CPU

logical address
346

+ relocation register
14000

MMU

physical address
14346

memory
Hardware Support for Relocation and Limit Registers

CPU

limit register
logical address
< yes
no

relocation register
physical address

<trap; addressing error

memory
Dynamic Loading

- Routine is not loaded until it is called.
- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the operating system is required implemented through program design.
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 needed to check if routine is in processes’ memory address.
- Dynamic linking is particularly useful for libraries.
Overlays

- Keep in memory only those instructions and data that are needed at any given time.

- Needed when process is larger than amount of memory allocated to it.

- **Implemented by user**, no special support needed from operating system, programming design of overlay structure is complex.
Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.

- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows).
Schematic View of Swapping

- Operating System
- Main Memory
- User Space
- Swap out
- Process $P_1$
- Swap in
- Process $P_2$
- Backing Storage
Contiguous Allocation

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

- Single-partition allocation
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data.
  - Relocation-register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register.
• 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 hole large enough to accommodate it.
  – 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.

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

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8192 bytes).
- Divide logical memory into blocks of same size called pages.
- Keep track of all free frames.
- 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.
- Internal fragmentation.
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.
Address Translation Architecture
Paging Example

```
<table>
<thead>
<tr>
<th>page 0</th>
<th>page 1</th>
<th>page 2</th>
<th>page 3</th>
<th>logical memory</th>
<th>frame number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td>3</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>7</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>page table</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>page 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>page 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>page 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>page 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>physical memory</td>
</tr>
</tbody>
</table>
```

03/10/2004  CSCI 315 Operating Systems Design  18
Paging Example
Free Frames

Before allocation | After allocation

03/10/2004       CSCI 315 Operating Systems Design
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 (PRLR)* 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)*.
Associative Memory

**Associative memory** – parallel search

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

Address translation (A’, A’’)
- If A’ is in associative register, get frame # out.
- Otherwise get frame # from page table in memory
Paging Hardware With TLB

CPU logical address

page number

frame number

TLB

TLB hit

physical address

TLB miss

page table

physical memory
Effective Access Time

- **Associative Lookup** $= \varepsilon$ time unit
- Assume memory cycle time is 1 microsecond
- **Hit ratio** – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers.
- Hit ratio $= \alpha$
- **Effective Access Time (EAT)**

$$EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$
Memory Protection

- Memory protection implemented by associating protection bit with each frame.
- *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.
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>
```

00000

page 0

page 1

page 2

page 3

page 4

page 5

10,468

12,287

page table

Valid (v) or Invalid (i) Bit in a Page Table
Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables
Hierarchical Page Tables

- Break up the logical address space into multiple page tables.

- A simple technique is a two-level page table.
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
- Thus, a logical address is as follows:

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

where $p_1$ is an index into the outer page table, and $p_2$ is the displacement within the page of the outer page table.
Two-Level Page-Table Scheme
Address-Translation Scheme

Address-translation scheme for a two-level 32-bit paging architecture:

![Diagram of address-translation scheme]
Hashed Page Tables

• Common in address spaces > 32 bits.

• The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.

• Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.
Hashed Page Table

logical address

p  d

hash function

hash table

physical address

r  d

q  s

...
Inverted Page Table

• One entry for each real page of memory.

• Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

• Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs.

• Use hash table to limit the search to one — or at most a few — page-table entries.
Inverted Page Table Architecture
Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.

- **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

```
03/10/2004    CSCI 315 Operating Systems Design    37
```
Segmentation

- Memory-management scheme that supports user view of memory.

- **A program is a collection of segments.** A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays
User’s View of a Program

logical address space

- subroutine
- stack
- symbol table
- main program

Sqrt
Logical View of Segmentation

user space

physical memory space
Segmentation Architecture

- **Logical address consists of a two tuple:**
  `<segment-number, offset>`,
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory.
  - *limit* – specifies the length of the segment.
- **Segment-table base register (STBR)** points to the segment table’s location in memory.
- **Segment-table length register (STLR)** indicates number of segments used by a program;
  segment number *s* is legal if *s* < STLR.
Segmentation Architecture

- **Relocation.**
  - dynamic
  - by segment table

- **Sharing.**
  - shared segments
  - same segment number

- **Allocation.**
  - first fit/best fit
  - external fragmentation
Segmentation Architecture

• **Protection.** With each entry in segment table associate:
  – validation bit = 0 ⇒ illegal segment
  – read/write/execute privileges

• Protection bits associated with segments; code sharing occurs at segment level.

• Since segments vary in length, memory allocation is a dynamic storage-allocation problem.

• A segmentation example is shown in the following diagram.
Segmentation Hardware

CPU -> s -> limit, base
segment table

<

yes

trap; addressing error

no

+ -> physical memory
Example of Segmentation

- subroutine
- stack
- segment 3
- symbol table
- segment 4
- sqrt
- main program
- segment 2
- segment 1
- segment 0

Logical address space

Limit: 0 1 2 3 4
Base: 1000 400 1100 1000

Segment table:
- segment 0: 0 1000 1400
- segment 1: 6000 6300
- segment 2: 4300 4700
- segment 3: 3200 3300
- segment 4: 4000 4300

Physical memory: 03/10/2004
Sharing of Segments
Segmentation with Paging
Intel 386

As shown in the following diagram, the Intel 386 uses segmentation with paging for memory management with a two-level paging scheme.
Intel 80x86 Address Translation

[Diagram showing the address translation process involving logical address, selector, offset, descriptor table, segment descriptor, linear address, directory, page, offset, page frame, page directory, page table, page table entry, directory entry, page directory base register, and physical address.]