Topics as per syllabus

- Memory Management Requirements,
- Memory Partitioning, Virtual memory: Paging; Segmentation;
- Page replacement policies, page faults.
Memory Management

- Subdividing memory to accommodate multiple processes
- Memory needs to be allocated efficiently to pack as many processes into memory as possible
Memory Management

Requirements

• Relocation
  • Programmer does not know where the program will be placed in memory when it is executed
  • While the program is executing, it may be swapped to disk and returned to main memory at a different location (relocated)
  • Memory references must be translated in the code to actual physical memory address
Addressing Requirements for a Process

- Process control information
- Entry point to program
- Increasing address values
- Current top of stack
- Program
- Data
- Stack
- Branch instruction
- Reference to data
Memory Management Requirements

- Protection
  - Processes should not be able to reference memory locations in another process without permission
  - Impossible to check absolute addresses in programs since the program could be relocated
  - Must be checked during execution
    - Operating system cannot anticipate all of the memory references a program will make. So memory protection requirement must be satisfied by the processor hardware.
Memory Management Requirements

- Sharing
  - Allow several processes to access the same portion of memory
  - Better to allow each process (person) access to the same copy of the program rather than have their own separate copy
Memory Management Requirements

- Logical Organization
  Programs are written in modules. If OS & h/w can deal with user programs in form of modules than a no. of advantages can be realized:
  - Modules can be written and compiled independently
  - Different degrees of protection given to modules (read-only, execute-only)
  - Share modules
Memory Management

Requirements

- Physical Organization
  - Memory is organized in 2 levels: main & secondary
  - Main Memory available for a program plus its data may be insufficient
    - Overlaying allows various modules to be assigned the same region of memory
  - Programmer does not know how much space will be available in a multiprogramming environment.
  - The task of moving info between 2 levels of memory should be system responsibility.
Basic Hardware

- Main memory and registers are only storage CPU can access directly.
- Registers are accessed in one CPU clock (or less).
- Completing Main memory access can take many cycles.
- **Cache** sits between main memory and CPU registers which is a buffer used to accommodate speed differential.
- Protection of memory required to ensure correct operation.
Basic Hardware

- To provide protection a pair of base and limit registers define the logical address space, are used.
HW address protection with base and limit registers

- CPU address
- base ≥ yes > base + limit < yes
- trap to operating system monitor—addressing error
- memory
Address Binding

- Program must be brought into memory and placed within a process for it to be executed.
- Input Queue - collection of processes on the disk that are waiting to be brought into memory for execution.
- User programs go through several steps before being executed.
Names and Binding

- **Symbolic names → Logical names → Physical names**
  - **Symbolic Names**: known in a context or path
    - file names, program names, printer/device names, user names
  - **Logical Names**: used to label a specific entity
    - inodes, job number, major/minor device numbers, process id (pid), uid, gid..
  - **Physical Names**: address of entity
    - inode address on disk or memory
    - entry point or variable address
    - PCB address
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:** Must generate relocatable code if memory location is not known at compile 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).
Binding time tradeoffs

- Early binding
  - compiler - produces efficient code
  - allows checking to be done early
  - allows estimates of running time and space

- Delayed binding
  - Linker, loader
  - produces efficient code, allows separate compilation
  - portability and sharing of object code

- Late binding
  - VM, dynamic linking/loading, overlaying, interpreting
  - code less efficient, checks done at runtime
  - flexible, allows dynamic reconfiguration
Multistep Processing of a User Program

Address Binding

Symbolic code

Relocatable code

Absolute code

source program

compiler or assembler

object module

linkage editor

load module

loader

in-memory binary memory image

dynamic linking

dynamically loaded system library

system library

other object modules

compile time

load time

execution time (run time)

Chapter 4 Slides by: Ms. Shree Jaswal
MEMORY MANAGEMENT

Binding Logical To Physical

```c
void main()
{
    printf("Hello, from main\n");
b();
}

void b()
{
    printf("Hello, from 'b:\n");
}
```
MEMORY MANAGEMENT

BINDING LOGICAL TO PHYSICAL

ASSEMBLY LANGUAGE LISTING

```
000000B0: 6BC23FD9  stw  %r2,-20(%sp) ; main()
000000B4 37DE0080  ldo  64(%sp),%sp
000000B8 E8200000  bl   0x000000C0,%r1   ; get current addr=BC
000000BC D4201C1E  depi 0,31,2,%r1
000000C0 34213E81  ldo  -192(%r1),%r1   ; get code start area
000000C4 E8400028  bl   0x000000E0,%r2   ; call printf
000000C8 B43A0040  addi 32,%r1,%r26   ; calc. String loc.
000000CC E8400040  bl   0x000000F4,%r2   ; call b
000000D0 6BC23FD9  stw  %r2,-20(%sp) ; store return addr
000000D4 4BC23F59  ldw  -84(%sp),%r2
000000D8 E840C000  bv   %r0(%r2) ; return from main
000000DC 37DE3F81  ldo  -64(%sp),%sp

STUB(S) FROM LINE 6

000000E0 E8200000  bl   0x000000E8,%r1
000000E4 28200000  addi  L%0,%r1
000000E8 E020E002  beq  0x00000000(%sr7,%r1)

000000EC 08000240  nop
000000F0 6BC23FD9  stw  %r2,-20(%sp)
000000F4 37DE0080  ldo  64(%sp),%sp
000000F8 E8200000  bl   0x00000100,%r1   ; get current addr=F8
000000FC D4201C1E  depi 0,31,2,%r1
0000100 34213E81  ldo  -256(%r1),%r1   ; get code start area
0000104 E85F1FAD  bl   0x000000E0,%r2   ; call printf
0000108 B43A0010  addi 8,%r1,%r26
000010C 4BC23F59  ldw  -84(%sp),%r2
0000110 E840C000  bv   %r0(%r2) ; return from b
0000114 37DE3F81  ldo  -64(%sp),%sp

void    b()  
```

Chapter 4  Slides by: Ms. Shree Jaswal
MEMORY MANAGEMENT

Binding Logical To Physical

EXECUTABLE IS DISASSEMBLED HERE

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>00002000</td>
<td>sw</td>
<td>000900DF</td>
</tr>
<tr>
<td>00002004</td>
<td>ldo</td>
<td>0800240</td>
</tr>
<tr>
<td>00002008</td>
<td>d48656C6C</td>
<td></td>
</tr>
<tr>
<td>0000200C</td>
<td>6F2C2066</td>
<td></td>
</tr>
<tr>
<td>00002010</td>
<td>726F6D20</td>
<td></td>
</tr>
<tr>
<td>00002014</td>
<td>620A0001</td>
<td></td>
</tr>
<tr>
<td>00002018</td>
<td>48656C6C</td>
<td></td>
</tr>
<tr>
<td>0000201C</td>
<td>6F2C2066</td>
<td></td>
</tr>
<tr>
<td>00002020</td>
<td>726F6D20</td>
<td></td>
</tr>
<tr>
<td>00002024</td>
<td>6D61696E</td>
<td></td>
</tr>
<tr>
<td>000020B0</td>
<td>sw</td>
<td>6BC23FD9</td>
</tr>
<tr>
<td>000020B4</td>
<td>ldo</td>
<td>37DE0008</td>
</tr>
<tr>
<td>000020B8</td>
<td>bl</td>
<td>E8200000</td>
</tr>
<tr>
<td>000020BC</td>
<td>depi</td>
<td>D4201C1E</td>
</tr>
<tr>
<td>000020C0</td>
<td>ldo</td>
<td>34213E81</td>
</tr>
<tr>
<td>000020C4</td>
<td>bl</td>
<td>E84017AC</td>
</tr>
<tr>
<td>000020C8</td>
<td>addi</td>
<td>B43A0040</td>
</tr>
<tr>
<td>000020CC</td>
<td>bl</td>
<td>E8400040</td>
</tr>
<tr>
<td>000020D0</td>
<td>sw</td>
<td>6BC23FD9</td>
</tr>
<tr>
<td>000020D4</td>
<td>ldw</td>
<td>4BC23F59</td>
</tr>
<tr>
<td>000020D8</td>
<td>bv</td>
<td>E840C000</td>
</tr>
<tr>
<td>000020DC</td>
<td>ldo</td>
<td>37DE3F81</td>
</tr>
<tr>
<td>000020E0</td>
<td>bl</td>
<td>E8200000</td>
</tr>
<tr>
<td>000020E4</td>
<td>addil</td>
<td>28203000</td>
</tr>
<tr>
<td>000020E8</td>
<td>be, n</td>
<td>E020E772</td>
</tr>
<tr>
<td>000020EC</td>
<td>nop</td>
<td>08000240</td>
</tr>
</tbody>
</table>

8: Memory Management
MEMORY MANAGEMENT

EXECUTABLE IS DISASSEMBLED HERE

8: Memory Management

8: Memory Management
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 -> relocation register (14000) -> MMU -> memory

logical address: 346

physical address: 14346
Program Loading

- 1\textsuperscript{st} step in creation of an active process is to load a program into main memory and create a process image
- The application consists of a no. of assembled or compiled modules in object-code form
- These are linked to resolve any references between modules
- There are 3 approaches for loading:
  - Absolute loading
  - Relocatable loading
  - Dynamic run-time loading
Program Loading

The Loading Function

Process Control Block

Program

Data

Object Code

Program

Data

Stack

Process image in main memory
Linker => Loader

A Loading Scenario
Loading

- **Absolute loading**: An absolute loader requires that a given load module always be loaded into the same location in main memory.
- Disadvantage is that the resulting load module can only be placed in one region of main memory. But since many programs share main memory, it may not be desirable.
- **Relocatable Loading**: the assembler or compiler produces addresses that are relative to some known point such as start of the program.
Absolute and Relocatable Load Modules

(a) Object module

(b) Absolute load module

(c) Relative load module
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 and is implemented through program design
Linking

- Linker creates a single load module that is the contiguous joining of all of the object modules
- Each intra-module reference must be changed from a symbolic address to a reference to a location within the overall load module
The Linking Function

(a) Object modules

(b) Load module
Dynamic Linking

- It is similar to dynamic loading
- Linking postponed until execution time
- Dynamic linking is particularly useful for libraries
- 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 another processes’ memory address
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 from operating system; programming design of overlay structure is complex.
Overlaying
Overlaying

Program

load one overlay and run until other overlay is needed

RAM
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.
Schematic View of Swapping

1. Swap out
2. Swap in
Swapping

- 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)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- If we want to swap a process, we must be sure that it is completely idle.
Contiguous Allocation

- Main memory is usually divided into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- 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 – each logical address must be less than the limit register
- MMU maps logical address \textit{dynamically}
Contiguous Allocation (Cont.)

- 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)
Managing allocated and free partitions

- Example: memory with 5 processes and 3 holes:
  - tick marks show memory allocation units.
  - shaded regions (0 in the bitmap) are free.
Memory Management with Linked Lists

Before X terminates

(a) A X B

(b) A X

(c) X B

(d) X

After X terminates

becomes

A

becomes

A

becomes

A

becomes

A
Types of Partitioning

- Fixed Partitioning
- Dynamic Partitioning
- Simple Paging
- Simple Segmentation
- Virtual Memory Paging
- Virtual Memory Segmentation
Fixed Partitioning

- Equal-size partitions
  - any process whose size is less than or equal to the partition size can be loaded into an available partition
  - if all partitions are full, the operating system can swap a process out of a partition
  - a program may not fit in a partition. The programmer must design the program with overlays
Fixed Partitioning

- Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. Thus there is wastage of space internal to a partition. This is called *internal fragmentation*. 
Example of Fixed Partitioning of a 64 Mbyte Memory

(a) Equal-size partitions

(b) Unequal-size partitions
Placement Algorithm with Partitions

- Equal-size partitions
  - because all partitions are of equal size, it does not matter which partition is used

- Unequal-size partitions
  - can assign each process to the smallest partition within which it will fit
  - queue for each partition
  - processes are assigned in such a way as to minimize wasted memory within a partition
Memory Assignment for Fixed Partitioning

(a) One process queue per partition

(b) Single queue
Dynamic Partitioning

- Partitions are of variable length and number
- Process is allocated exactly as much memory as required
- Eventually get holes in the memory. This is called *external fragmentation*
- Thus in external fragmentation total memory space exists to satisfy a request, but it is not contiguous
- Must use compaction to shift processes so they are contiguous and all free memory is in one block
The Effect of Dynamic Partitioning
The Effect of Dynamic Partitioning
Dynamic Partitioning Example

- **External Fragmentation**
- Memory external to all processes is fragmented
- Can resolve using *compaction*
  - OS moves processes so that they are contiguous
  - Time consuming and wastes CPU time

Refer to Figure 7.4
Dynamic Partitioning Placement Algorithm

- Operating system must decide which free block to allocate to a process

- **First-fit algorithm**
  - Allocate the *first* hole that is big enough.
  - Fastest
  - May have many process loaded in the front end of memory that must be searched over when trying to find a free block
Dynamic Partitioning Placement Algorithm

- **Best-fit algorithm**
  - Allocate the *smallest* hole that is big enough.
  - Chooses the block that is closest in size to the request
  - Worst performer overall
  - Since smallest block is found for process, the smallest amount of fragmentation is left, memory compaction must be done more often
Dynamic Partitioning Placement Algorithm

- **Next-fit/ Worst-fit**
  - Allocate the *largest* hole.
  - More often allocate a block of memory at the end of memory where the largest block is found
  - The largest block of memory is broken up into smaller blocks
  - Compaction is required to obtain a large block at the end of memory

- First-fit and best-fit better than worst-fit in terms of speed and storage utilization
Example Memory Configuration Before and After Allocation of 16 Mbyte Block

(a) Before

(b) After

Increasing address value

Allocated block
Free block

Last allocated block (14K)

Chapter 4 Slides by: Ms. Shree Jaswal
Memory Allocation Policies

Example: *Parking Space Management*

- A scooter, car and a truck are looking out for space for parking. They arrive in the order mentioned above. The parking spaces are available for each one of them as per their size. Truck parking space can accommodate, a car and a scooter or even two scooters. Similarly, In a car parking space two scooters can be parked.
Memory Allocation Policies

Alongside is shown the partition in the parking area for Truck, Car and Scooter.

Now when a scooter, car and truck come in order, parking space is allocated according to algorithm policies.
Memory Allocation Policies

Now take another theoretical example.

- Given the partition of 100K, 500K, 200K, 300K, 600K as shown, the different algorithms will place the processes 212K, 417K, 112K, 426K respectively.

- The request for 426K will be rejected in case of next fit and worst fit algorithm because any single partition is less than 426K.
Next Fit (212k, 417k, 112k)

Similarly, we cannot implement for the other two policies.

The request for 426k will be rejected.

Similarly, we cannot implement for the other two policies.
212K-Green
417K-Blue
112K-Pink
426K-Yellow
External Fragmentation-Gray
Unused Partitions-White

Next Fit

Best Fit

Worst Fit
User selects the algorithm in order of worst fit, best fit and, next fit.

- **Worst Fit**: 300K
- **Best Fit**: 300K and 400K
- **Next Fit**: 300K, 400K, and 600K

User input = 300K process sizes

Chapter 4          Slides by: Ms. Shree Jaswal
Empty Memory

User entered 5 processes which allotted in memory

300 k
400 k
500 k
800 k
User Selected Best Fit

User entered 5 processes allotted in memory

300 k
400 k
500 k
800 k

New process given by user

450 k

Best Fit

300 k
400 k
450 k
800 k

External Fragmentation

Chapter 4  Slides by: Ms. Shree Jaswal
User Selected Worst Fit

User entered 5 processes allotted in memory

- 300 k
- 400 k
- 500 k
- 800 k

New process given by user
- 450 k

Worst Fit
- 300 k
- 400 k
- 500 k
- 450K
- 350K

External Fragmentation

Chapter 4        Slides by: Ms. Shree Jaswal
User Selected Next Fit

User entered 5 processes allotted in memory

Next Fit

New process given by user

External Fragmentation
First-fit and next-fit can allocate faster than best-fit and worst-fit (no need to scan the complete list).

First-fit favors allocation near the beginning and tends to create less fragmentation than next-fit.

Worst-fit is the worst method both in terms of fragmentation and in allocation speed.

There is no clear winner between first-fit and best-fit in terms of external fragmentation.
Relocation

- When program loaded into memory the actual (absolute) memory locations are determined.
- A process may occupy different partitions which means different absolute memory locations during execution (from swapping).
- Compaction will also cause a program to occupy a different partition which means different absolute memory locations.
Addresses

- **Logical**
  - reference to a memory location independent of the current assignment of data to memory
  - translation must be made to the physical address
- **Relative**
  - address expressed as a location relative to some known point
- **Physical**
  - the absolute address or actual location in main memory
Hardware Support for Relocation

- Base Register
- Bounds Register
- Adder
- Comparator
- Interrupt to operating system
- Process Control Block
- Program
- Data
- Stack
- Process image in main memory

Relative address
Absolute address

Chapter 4        Slides by: Ms. Shree Jaswal
 Registers Used during Execution

• Base register
  • starting address for the process
• Bounds register
  • ending location of the process
• These values are set when the process is loaded and when the process is swapped in
Registers Used during Execution

- The value of the base register is added to a relative address to produce an absolute address
- The resulting address is compared with the value in the bounds register
- If the address is not within bounds, an interrupt is generated to the operating system
Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- It avoids external fragmentation & need for compaction
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
Paging

- 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

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

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

CPU

logical address

physical address

f0000 ... 0000

f1111 ... 1111

physical memory

page table

p d

f d
Paging Model of Logical and Physical Memory

[Diagram showing paging model with logical memory (page 0, page 1, page 2, page 3) and physical memory (frames labeled 0 to 7).]

Chapter 4       Slides by: Ms. Shree Jaswal
In logical address, \( n=2 \) & \( m=4 \). Using page size of 4 bytes & physical memory of 32 bytes ->
Figure 7.9 Assignment of Process Pages to Free Frames

(a) Fifteen Available Frames
(b) Load Process A
(b) Load Process B
Figure 7.9 Assignment of Process Pages to Free Frames
Free Frames

(a) free-frame list
14
13
18
20
15

page 0
page 1
page 2
page 3

new process

(b) free-frame list
15

page 0
page 1
page 2
page 3

0
1
2
3

14
13
18
20

new-process page table 21
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.
Translation Lookaside Buffer

- Each virtual memory reference can cause two physical memory accesses
  - One to fetch the page table
  - One to fetch the data
- To overcome this problem a high-speed cache is set up for page table entries
  - Called a Translation Lookaside Buffer (TLB)
- Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
Associative Memory

- The TLB is Associative, high speed memory

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

CPU

logical
address

p
d

page
number

frame
number

TLB

TLB hit

f
d

physical
address

physical
memory

TLB miss

p

f

page table
Figure 8.9  Direct Versus Associative Lookup for Page Table Entries
Translation Lookaside Buffer

- Given a virtual address, processor examines the TLB
- If page table entry is present (TLB hit), the frame number is retrieved and the real address is formed
- If page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table
Translation Lookaside Buffer

- First checks if page is already in main memory
  - If not in main memory a page fault is issued
- The TLB is updated to include the new page entry
Figure 8.7 Use of a Translation Lookaside Buffer
Figure 8.8  Operation of Paging and Translation Lookaside Buffer (TLB) [FURH87]
Translation Lookaside Buffer

Figure 8.10 Translation Lookaside Buffer and Cache Operation
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)**

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

$$= 2 + \varepsilon - \alpha$$
Example

- Hit ratio is 80%. It takes 20 ns to search the TLB & 100 ns to access memory. If we fail to find the page no in TLB(20 ns) then we must 1\(^{st}\) access memory for the page table and frame no. (100 ns) and then access the desired byte in memory(100 ns)

- Thus, \( EAT = 0.80 \times 120 + 0.20 \times 220 \)
  \[ = 140 \text{ ns} \]
Example (contd.)

- In this example, we suffer a 40% slowdown in memory-access time (from 100 to 140 ns).
- For a 98% hit ratio, we get,
- \[ EAT = 0.98 \times 120 + 0.02 \times 220 \]
  \[ = 122 \text{ ns} \]
- Thus increased hit rate produces only a 22% slowdown in access time
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>

<table>
<thead>
<tr>
<th>Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>n</td>
</tr>
</tbody>
</table>
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

- **Process $P_1$**
  - **Data 1**: Page 1
  - **Data 2**: Page 2

- **Process $P_2$**
  - **Data 1**: Page 3
  - **Data 2**: Page 4

- **Process $P_3$**
  - **Data 3**: Page 6

Page Table

- **$P_1$**
  - Page 0: Data 1
  - Page 1: Data 3
  - Page 2: Ed 1
  - Page 3: Ed 2
  - Page 4: Ed 3

- **$P_2$**
  - Page 0: Data 1
  - Page 1: Data 3
  - Page 2: Ed 1
  - Page 3: Ed 2
  - Page 4: Ed 3

- **$P_3$**
  - Page 0: Data 1
  - Page 1: Data 3
  - Page 2: Ed 1
  - Page 3: Ed 2
  - Page 4: Ed 3
Page Size

- Smaller page size, less amount of internal fragmentation
- Smaller page size, more pages required per process
- More pages per process means larger page tables
- Larger page tables means large portion of page tables in virtual memory
Page Size

- Secondary memory is designed to efficiently transfer large blocks of data so a large page size is better.
Page Size

- Small page size, large number of pages will be found in main memory
- As time goes on during execution, the pages in memory will all contain portions of the process near recent references. Page faults low.
- Increased page size causes pages to contain locations further from any recent reference. Page faults rise.
## Table 8.3 Example Page Sizes

<table>
<thead>
<tr>
<th>Computer</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlas</td>
<td>512 48-bit words</td>
</tr>
<tr>
<td>Honeywell-Multics</td>
<td>1024 36-bit word</td>
</tr>
<tr>
<td>IBM 370/XA and 370/ESA</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>VAX family</td>
<td>512 bytes</td>
</tr>
<tr>
<td>IBM AS/400</td>
<td>512 bytes</td>
</tr>
<tr>
<td>DEC Alpha</td>
<td>8 Kbytes</td>
</tr>
<tr>
<td>MIPS</td>
<td>4 Kbytes to 16 Mbytes</td>
</tr>
<tr>
<td>UltraSPARC</td>
<td>8 Kbytes to 4 Mbytes</td>
</tr>
<tr>
<td>Pentium</td>
<td>4 Kbytes or 4 Mbytes</td>
</tr>
<tr>
<td>IBM POWER</td>
<td>4 Kbytes</td>
</tr>
<tr>
<td>Itanium</td>
<td>4 Kbytes to 256 Mbytes</td>
</tr>
</tbody>
</table>
Structure of the Page Table

- 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 Page Table Scheme

- Outer-page table
- Page of page-tables
- Physical memory

1

100

929

500

708

900
Two Level Paging Example

- A logical address (32bit machine, 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 consists of a 10-bit page number, a 10-bit page offset
- Thus, a logical address is organized as (p1,p2,d) where
  - p1 is an index into the outer page table
  - p2 is the displacement within the page of the outer page table

<table>
<thead>
<tr>
<th>Page number</th>
<th>Page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
</tr>
</tbody>
</table>
Address-Translation Scheme

logical address
\[ p_1 \quad p_2 \quad d \]

- outer page table
  \[ p_1 \]

- page of page table
  \[ p_2 \]

\[ d \]
Multilevel paging

- Each level is a separate table in memory
  - converting a logical address to a physical one may take 4 or more memory accesses.
- Caching can help performance remain reasonable.
  - Assume cache hit rate is 98%, memory access time is quintupled (100 vs. 500 nanoseconds), cache lookup time is 20 nanoseconds
  - Effective Access time = $0.98 \times 120 + 0.02 \times 520 = 128$ ns
  - This is only a 28% slowdown in memory access time...
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

Diagram showing the process of hashing a logical address to a physical address using a hash function and a hash table.
Inverted Page Table

- One entry for each real page of memory
  - Entry consists of virtual address of page in real memory with information about process that owns page.
- Decreases memory needed to store page table
- Increases time to search table when a page reference occurs
  - Table sorted by physical address, lookup by virtual address
- Use hash table to limit search to one (maybe few) page-table entries.
Inverted Page Table Architecture
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
  - local variables, global variables, common block
  - stack, symbol table, arrays
  - Protect each entity independently
  - Allow each segment to grow independently
  - Share each segment independently
User’s View of a Program

![Diagram showing components of a program]

- subroutine
- stack
- symbol table
- sqrt
- main program

logical address
Logical view of segmentation

User Space

Physical Memory
Segmentation Architecture

- Logical address consists of a two tuple 
  \(<\text{segment-number, offset}>\)

- Segment Table
  - Maps two-dimensional user-defined addresses into one-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.
  - \textit{Segment-table base register} (STBR) points to the segment table’s location in memory.
  - \textit{Segment-table length register} (STLR) indicates the number of segments used by a program; segment number is legal if \(s < \text{STLR}\).
Segmentation Hardware

CPU \[ \rightarrow \] \text{s, d} \rightarrow s \rightarrow \text{segment table} \rightarrow + \rightarrow \text{physical memory}

\text{trap: addressing error}
Example of Segmentation

- Subroutine
- Stack
- Symbol Table
- Main Program
- Logical address space

<table>
<thead>
<tr>
<th>segment</th>
<th>limit</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>1400</td>
</tr>
<tr>
<td>1</td>
<td>400</td>
<td>6300</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
<td>4300</td>
</tr>
<tr>
<td>3</td>
<td>1100</td>
<td>3200</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>4700</td>
</tr>
</tbody>
</table>

Physical memory:
- Segment 0
- Segment 1
- Segment 2
- Segment 3
- Segment 4

Chapter 4        Slides by: Ms. Shree Jaswal
Segmentation Architecture (cont.)

- Relocation is dynamic - by segment table
- Sharing
  - Code sharing occurs at the segment level.
  - Shared segments must have same segment number.
- Allocation - dynamic storage allocation problem
  - use best fit/first fit, may cause external fragmentation.
- Protection
  - protection bits associated with segments
    - read/write/execute privileges
    - array in a separate segment - hardware can check for illegal array indexes.
Segment Table Entries

Virtual Address

Segment Table Entry

<table>
<thead>
<tr>
<th>Segment Number</th>
<th>Offset</th>
</tr>
</thead>
</table>

| P | M | Other Control Bits | Length | Segment Base |

(b) Segmentation only
Figure 8.12  Address Translation in a Segmentation System
Shared segments

Logical Memory
process P1

Logical Memory
process P2

Segment Table
process P1

Segment Table
process P2

<table>
<thead>
<tr>
<th>Limit</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25286, 43602</td>
</tr>
<tr>
<td>1</td>
<td>4425, 68348</td>
</tr>
</tbody>
</table>

0
1

43062
68348
72773
90003
98553

editor
data 1
segment 0
segment 1

data 1
editor
segment 0
segment 1

data 2
editor
segment 0
segment 1

data 2
Combined Paging and Segmentation

- Paging is transparent to the programmer
- Segmentation is visible to the programmer
- Each segment is broken into fixed-size pages
Segmented Paged Memory

- Segment-table entry contains not the base address of the segment, but the base address of a page table for this segment.
  - Overcomes external fragmentation problem of segmented memory.
  - Paging also makes allocation simpler; time to search for a suitable segment (using best-fit etc.) reduced.
  - Introduces some internal fragmentation and table space overhead.
- Multics - single level page table
- IBM OS/2 - OS on top of Intel 386
  - uses a two level paging scheme
Combined Paging and Segmentation

Virtual Address

| Segment Number | Page Number | Offset |

Segment Table Entry

| Control Bits | Length | Segment Base |

Page Table Entry

| P | M | Other Control Bits | Frame Number |

(c) Combined segmentation and paging

P = present bit
M = Modified bit
Figure 8.13  Address Translation in a Segmentation/Paging System
MULTICS address translation scheme

Segment Table Base Register

Virtual Address

Segment Page Displacement

Segment Table

S + b

b

s

Segment Table

Page Table

Main Memory

Physical Address

b

s

pt

pt + p

f

f + d

f + d

pt

p

p

f

f + d
Virtual Memory

- Background
- Demand paging
  - Performance of demand paging
- Page Replacement
  - Page Replacement Algorithms
- Allocation of Frames
- Thrashing
- Demand Segmentation
Need for Virtual Memory

- **Virtual Memory**
  - Separation of user logical memory from physical memory.
  - Only *PART* of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Need to allow pages to be swapped in and out.

- **Virtual Memory can be implemented via**
  - Paging
  - Segmentation
Support Needed for Virtual Memory

- Hardware must support paging and segmentation
- Operating system must be able to do the management the movement of pages and/or segments between secondary memory and main memory
Virtual Memory

- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation
Virtual Memory That is Larger Than Physical Memory

- page 0
- page 1
- page 2
- memory map
- physical memory

virtual memory
Virtual-address Space

[Diagram showing a virtual address space with layers labeled 'code', 'data', 'heap', 'stack', and 'Max']
Benefits of virtual memory

- System libraries can be shared by several processes
- Enables processes to share memory
- Can allow pages to be shared during process creation with fork() system call, thus speeding up process creation.
Shared Library Using Virtual Memory

- stack
- shared library
- heap
- data
- code

- stack
- shared library
- heap
- data
- code

shared pages
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 $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**, which is used in demand paging
Transfer of a Paged Memory to Contiguous Disk Space

Diagram:

- **Program A**
  - Main Memory
- **Program B**
  - Main Memory

Swap Out:
- Pages 0 to 3

Swap In:
- Pages 16 to 19
- Pages 0 to 3

Disk Space:
- Pages 0 to 23
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
- Example of a page table snapshot:

```
Frame # | valid-invalid bit
--------|-------------------
         | v
         | v
         | v
         | v
         | v
         | i
         | ....
         | i
         | i
```

- During address translation, if valid–invalid bit in page table entry
  is \(i \Rightarrow \text{page fault}\)
Page Table When Some Pages Are Not in Main Memory
Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: page fault

1. Operating system looks at an internal table (kept with PCB) to decide:
   - Invalid reference → abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables (internal & page)
5. Set validation bit = v
6. Restart the instruction that caused the page fault
Steps in Handling a Page Fault

1. Load M
2. Trap
3. Page is on backing store
4. Bring in missing page
5. Reset page table
6. Restart instruction

Diagram:
- Operating system
- Reference
- Page table
- Free frame
- Physical memory
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times
Need For Page Replacement

Logical memory for user 1:
- Frame 0: H
- Frame 1: load M
- Frame 2: J
- Frame 3: M

Page table for user 1:
- Page 3: valid
- Page 4: valid
- Page 5: valid
- Page 6: invalid

Physical memory:
- Frame 0: monitor
- Frame 1: D
- Frame 2: H
- Frame 3: load M
- Frame 4: J
- Frame 5: A
- Frame 6: E
- Frame 7: M

Logical memory for user 2:
- Frame 0: A
- Frame 1: B
- Frame 2: D
- Frame 3: E

Page table for user 2:
- Page 6: valid
- Page 2: invalid
- Page 7: valid

Valid-invalid bit

Chapter 4 Slides by: Ms. Shree Jaswal
Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a victim frame
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process
Page Replacement

1. Swap out victim page
2. Change to invalid
3. Swap desired page in
4. Reset page table for new page

Frame | Valid-Invalid Bit
0 | i
f | v

Page Table

Physical Memory
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
Page Replacement Algorithms

- Want 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
Page Replacement Algorithms

- We can generate reference strings:
  1. Artificially (by using random number generator) or
  2. We can trace a given system & record the address of each memory reference

- For the 2\textsuperscript{nd} case we use 2 facts:
  1. For a given page size, we need to consider only the page number, rather than the entire address
  2. If we have a reference to a page $p$, then any references to page $p$ that immediately follow will never cause a page fault (as $p$ will already be in memory after the first reference)
Page Replacement Algorithms

- Eg., if we trace a particular process, we might record the following address sequence:
  
  0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

- At 100 bytes per page, this sequence is reduced to the following reference string:
  
  1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

- If we had 3 frames, we would have only 3 faults: 1 fault for the 1\textsuperscript{st} reference to each page. With 1 frame the no. of faults would be 11.
Graph of Page Faults Versus The Number of Frames
First-In-First-Out (FIFO) Algorithm

- Most simplest algorithm wherein, when a page must be replaced, the oldest page is chosen
- A FIFO queue can be created to hold all pages in memory
- We replace the page at the head of the queue & when a page is brought into the memory, we insert into the tail of the queue.
- In all our examples, the reference string is 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1
FIFO Page Replacement

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

page frames

7 7 7 2 2 2 4 4 4 0 0 0 7 7 7
0 0 0 3 3 3 2 2 2 1 1 1 1 0 0
1 1 1 1 0 0 3 3 3 3 2 2 2 1
First-In-First-Out (FIFO) Algorithm

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
3 frames (3 pages can be in memory at a time per process)

3 frames

```
1 1 4 5
2 2 1 3
3 3 2 4
```

Belady’s Anomaly: more frames ⇒ more page faults
First-In-First-Out (FIFO) Algorithm

- Belady’s Anomaly: For some page replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. This most unexpected result is known as Belady’s anomaly.
- FIFO is easy to understand & program
- Performance is not always good
- A bad replacement choice (active pages) increases the page fault rate & slows process execution.
FIFO Illustrating Belady’s Anomaly

![Graph showing the number of page faults against the number of frames. The graph indicates a pattern that suggests a solution to the Belady’s Anomaly.](image)
Optimal Algorithm

- **Replace page that will not be used for longest period of time**
- **4 frames example**

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- It requires future knowledge of reference string, so difficult to implement
- Used for comparison studies measuring how well a new algorithm performs
## Optimal Algorithm

**Reference string**

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |

**Page frames**

| 7 | 7 | 7 | 2 | 2 | 2 | 2 | 2 | 7 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 1 |

| 1 | 1 | 3 | 3 | 3 | 1 | 1 | 7 | 0 | 0 |

**Chapter 4**  **Slides by: Ms. Shree Jaswal**
Least Recently Used (LRU) Algorithm

- Replace a page that has not been used for the longest period of time.
- It is considered as a good page replacement algorithm.
- It's difficult to implement LRU page replacement and the algorithm requires substantial hardware assistance.
- The problem is to determine an order for the frames defined by the time of last use. 2 implementations are possible: counters and stack.
Least Recently Used (LRU) Algorithm

<table>
<thead>
<tr>
<th>reference string</th>
<th>page frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
<td>7 7 7 2 0 0 1 1</td>
</tr>
<tr>
<td></td>
<td>2 4 4 4 0 0 3 2 2 2</td>
</tr>
<tr>
<td></td>
<td>7 3 3 2 2 3 3 0 0 7</td>
</tr>
</tbody>
</table>

Chapter 4        Slides by: Ms. Shree Jaswal
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change
LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement
Use Of A Stack to Record The Most Recent Page References

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

stack before

2
1
0
7
4

stack after

7
2
1
0
4

a
b
Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation
Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process

\[ s_i = \text{size of process } p_i \]
\[ S = \sum s_i \]
\[ m = \text{total number of frames} \]
\[ a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m \]

\[ m = 64 \]
\[ s_1 = 10 \]
\[ s_2 = 127 \]
\[ a_1 = \frac{10}{137} \times 64 \approx 5 \]
\[ a_2 = \frac{127}{137} \times 64 \approx 59 \]
Priority Allocation

- Use a proportional allocation scheme using priorities rather than size

- If process $P_i$ generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; even if that frame is currently allocated to another process. One process can take a frame from another. A Process may not be able to control its page fault rate.

- **Local replacement** – each process selects from only its own set of allocated frames. Process slowed down even if other less used pages of memory are available.

- Global replacement has better throughput
  - Hence more commonly used.
Thrashing

- Swapping out a piece of a process just before that piece is needed
- The processor spends most of its time swapping pieces rather than executing user instructions
- **Thrashing** ≡ a process is busy swapping pages in and out
Thrashing

• If a process does not have “enough” frames, the page-fault rate is very high. This leads to:
  • low CPU utilization
  • operating system thinks that it needs to increase the degree of multiprogramming
  • another process added to the system
Thrashing (Cont.)

![Graph showing CPU utilization vs. degree of multiprogramming]

- CPU utilization
- Degree of multiprogramming

thrashing
Thrashing (Cont.)

- To prevent thrashing provide a process with as many frames as it needs, which we can know by an approach called **locality model** of process execution.

- Eg. When a function is called, it defines a new locality.

- Locality model states that:
  - Process migrates from one locality to another.
  - Localities may overlap.

- If we do not allocate enough frames to accommodate the size of current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.
Thank You!