Overview Basic Concepts

- Process must be loaded into memory before being executed.
- *Input queue* – collection of processes on the disk that are waiting to be brought into memory for execution.
Addresses

- The **symbolic addresses** are the addresses used in a source program. The variable names, symbolic constants and instruction labels are the basic elements of the symbolic address space.
- The **compiler** converts a symbolic addresses into a **relative address**.
- The **physical address** consists of the final address generated when the program is loaded and ready to execute in physical memory; the **loader** generates these addresses.
Process Address Space

- The set of logical addresses that a process references in its code.
- The operating system provides a mechanism that maps the logical addresses to physical addresses.
- When memory is allocated to the process, its set of logical addresses will be bound to physical addresses.
Logical and Physical Addresses

P1

Process address space

P2

Process address space

P1

P2

Physical memory
Phases of a Program Implementation

1. **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.

2. **Linkage time**: Program is combined with other modules.

3. **Load time**: Program is loaded into memory.

4. **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).
Program Phases and Addresses

Diagram:

- Editor
  - Symbolic addresses
  - Compiler/Assembler
    - Relative addresses
    - Linker
      - Other compiled modules
      - Relative addresses
      - Loader
        - Physical addresses
Managing the Address Space

- The compiler or assembler generates the program as a relocatable object module.
- The linker combines several modules into a load module.
- During memory allocation, the loader places the load module in the allocated block of memory.
- The loader binds the logical address to a physical address.
Binding

- The association of instructions and data to memory addresses
- Can occur at any of the following steps
  - Compile time
  - Load time
  - Execution time
Dynamic Loading

- Routines or modules to be used by a program are not loaded until called.
- All routines are stored on a disk in relocatable form.
- Better memory-space utilization; unused routines are 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.
- Example: Dynamic Linked Libraries (DLL)
- 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.
Logical vs. Physical Address Space

- **Logical address** – generated by the compiler/assembler; also referred to as *virtual address*.
- **Physical address** – address seen by the memory unit.
- **Logical address space** is the set of all addresses of a program.
- **Physical address space** is the set of addresses used to store the program into memory.
- The logical *address space* is **bound** to a separate *physical address space*. 
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 directly references the *real* physical addresses.
Static Relocation

- A process executes at a specific address.
- Addresses are assigned at load time.
- **Relocation** modifies instructions that address locations in memory.
Dynamic Relocation

- Addresses can be changed at execution time.
- Additional benefit, program may be moved during execution.
Contiguous Memory Allocation

- Main memory is divided into several partitions.
- A partition is a contiguous block of memory that can be allocated to an individual process.
- The degree of multiprogramming is determined by the number of partitions in memory.
- When a process completes and terminates, memory is de-allocated and that amount of memory becomes available.
Memory Management

- Swapping
- Partitioning
- Paging
- Segmentation
- Virtual memory -- Only currently executing portion of program is loaded into memory
Swapping

- A process can be *swapped* temporarily out of memory to secondary *storage*, and then loaded into memory again to resume execution.

- Secondary storage – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
Swapping

- *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 and Microsoft Windows.
Multiple Partitions

- **Fixed partitions (static)** – the number and sizes of the partitions do not change
- **Variable partitions (dynamic)** – partitions are created dynamically according to
  - the available memory
  - the memory requirements of processes
Fixed Partitions

- Memory is divided into fixed-sized partitions. These partitions are not normally of the same size.
- The number and the size of the partitions are fixed.
- One partition is allocated to each active process in the multiprogramming set.
- There is one special partition, the system partition, in which the memory-resident portion of the operating system is always stored.
Fixed Partitions

- Partition 6
- Partition 5
- Partition 4
- Partition 3
- Partition 2
- Partition 1
- Operating system
Fixed Partitions: Multiple Input Queues

If Partitions are of different sizes: Less space is left unused to Internal Fragmentation.

When a job arrives, it can be put into the input queue for the smallest partition which is large enough to hold it.

Disadvantage of sorting incoming jobs into separate queues occurs when the queue for small partitions is full, but the queue for large partition is empty and consequently jobs await space when space is available.
Alternative is to have a single queue for all jobs. When a partition becomes available, assign the job closest to front of queue that will fit.

Incoming jobs are queued until suitable partition is available.

Then, the process is loaded into the partition and run until terminated.
Fragmentation in Fixed Partition

Fragmentation problem

- Internal fragmentation - A partition is only partially used.
- A partition is available, but not large enough for any waiting progress.
Memory Allocation Problem

- An important problem in OS is finding a fit between the partition sizes and the actual memory requirements of processes.
- The goal is to minimize the fragmentation.
Variable Partition Allocation

- The partitions are created dynamically.
- The OS maintains a table of partitions allocated that indicates which parts (location and size) of memory are available and which have been allocated.
At any given time, there is a list of available blocks of contiguous memory of various sizes (holes) and a queue of processes requesting memory.

Memory is allocated contiguously to processes until there is no available block of memory large enough.
Memory Allocation

The memory manager can:
- Wait until a large enough block of memory is available, or
- Skip down the queue to find a process with smaller requirements for memory.
Holes in Variable Partitions

- **Hole** – a block of available memory; holes of various size are scattered throughout memory.
- When a process requests memory, it is allocated memory from a hole large enough to accommodate it.
- Operating system maintains data about:
  - allocated partitions
  - Available memory blocks (holes)
Holes and Allocation

• When a process is to be loaded, the OS searches for a hole (a block of free memory) large enough for this process and allocates the necessary space.

• When a process terminates, the OS frees its block of memory.

• In general, there is at any time, a set of holes, of various sizes, scattered throughout memory.

• If a new hole is adjacent to other holes, they will be merged to form one larger hole.

• The minimal hardware to support this scheme is the same as with fixed partition schemes.
Advantages of Variable Partition

- Memory utilization is generally better for variable-partition schemes.
- There is little or no internal fragmentation.
- There can be external fragmentation.
### Memory States

<table>
<thead>
<tr>
<th>OS</th>
<th>process 5</th>
<th>process 8</th>
<th>process 2</th>
</tr>
</thead>
</table>

- OS
- process 5
- process 8
- process 2

- OS
- process 5
- process 2

- OS
- process 5
- process 9
- process 2

- OS
- process 5
- process 9
- process 10
Compaction: Solution to External Fragmentation

- External fragmentation is a serious problem.
- The goal is to shuffle the memory contents to place all free memory together in one large block.
- This is only possible if relocation is dynamic (binding is done at execution time), using base and limit registers.
- Can be quite expensive (overhead).
Memory Allocation

How to satisfy a process 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 - Recap

- **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.
Non-contiguous Memory Allocation

- Paging
- Segmentation
Pages

- A page is a unit of logical memory of a program
- A frame is a unit of physical memory
- All pages are of the same size
- All frames are of the same size
- A frame is of the same size as a page
Paging

- Physical memory is divided into fixed-sized blocks called frames (size is power of 2).
- Logical memory is divided into blocks of same size called pages.
- A page of a program is stored on a frame, independently of other pages.
- A logical address on the page is converted to a physical address on the corresponding frame.
<table>
<thead>
<tr>
<th>f31</th>
<th>P6 - page 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>f30</td>
<td>P5 - page 0</td>
</tr>
<tr>
<td>f29</td>
<td>P6 - page 4</td>
</tr>
<tr>
<td>f28</td>
<td>P5 - page 3</td>
</tr>
<tr>
<td>f26</td>
<td>P6 - page 2</td>
</tr>
<tr>
<td>f25</td>
<td>P6 - page 2</td>
</tr>
<tr>
<td>f24</td>
<td>P6 - page 0</td>
</tr>
<tr>
<td>f23</td>
<td>P5 - page 1</td>
</tr>
<tr>
<td>f22</td>
<td>P6 - page 1</td>
</tr>
<tr>
<td>f21</td>
<td>P5 - page 1</td>
</tr>
<tr>
<td>f19</td>
<td>P6 - page 2</td>
</tr>
<tr>
<td>f18</td>
<td>P5 - page 2</td>
</tr>
<tr>
<td>f17</td>
<td>P6 - page 2</td>
</tr>
<tr>
<td>f16</td>
<td>P3 - page 3</td>
</tr>
<tr>
<td>f15</td>
<td>P4 - page 2</td>
</tr>
<tr>
<td>f14</td>
<td>P4 - page 0</td>
</tr>
<tr>
<td>f13</td>
<td>P4 - page 1</td>
</tr>
<tr>
<td>f12</td>
<td>P2 - page 1</td>
</tr>
<tr>
<td>f11</td>
<td>P3 - page 1</td>
</tr>
<tr>
<td>f10</td>
<td>P3 - page 2</td>
</tr>
<tr>
<td>f9</td>
<td>P2 - page 0</td>
</tr>
<tr>
<td>f8</td>
<td>P3 - page 0</td>
</tr>
<tr>
<td>f7</td>
<td>Operating system</td>
</tr>
</tbody>
</table>
Paging(2)

- The OS keeps track of all free (available) frames, and allocated frames in the page table.
- To run a program of size $n$ pages, the OS needs $n$ free frames to load program.
- The OS sets up a page table and converts logical to physical addresses.
- There is a small amount of internal fragmentation.
Memory Allocation with Paging

- The frames allocated to the pages of a process need not be contiguous; in general, the system can allocate any empty frame to a page of a particular process.
- There is no external fragmentation.
- There is potentially a small amount of internal fragmentation that would occur on the last page of a process.
Logical Address

- Any address referenced in a process is defined by the page that the address belongs to and the relative address within that page.
- A logical address of a process consists of a page number and an offset.
Logical Address

Address generated by the compiler/assembler 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** \((d)\) – the relative address in the page.

This pair of numbers will be converted to the physical memory address that is sent to the memory unit.
Example of a Logical Address

<table>
<thead>
<tr>
<th>Page number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000010</td>
<td>0111011110</td>
</tr>
</tbody>
</table>
Physical Address

- When the system allocates a frame to this page, it translates this logical address into a physical address that consists of a frame number and the offset.
- For this, the system needs to know the correspondence of a page of a process to a frame in physical memory and it uses a page table.
Example of a Physical Address

Frame number: 0000000100
Offset: 0111011110
Address Translation Architecture

Logical Address

Page Number

Page Offset

Page Table

Page Table Entry

Physical Memory

Frame 3

Frame 1

Frame 4

Frame 0

Frame 2

Page Offset

Page Number is index into Page Table
## Page Table Example

<table>
<thead>
<tr>
<th>Page</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>f5</td>
</tr>
<tr>
<td>1</td>
<td>f10</td>
</tr>
<tr>
<td>2</td>
<td>f8</td>
</tr>
<tr>
<td>3</td>
<td>f16</td>
</tr>
</tbody>
</table>
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 registers* or *translation look-aside buffers* (TLBs)
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/Invalid Bit

- Page table:
  - Page 0
  - Page 1
  - Page 2
  - Page 3
  - Page 4
  - Page 5

- Frame number:
  - 0: v
  - 1: v
  - 2: v
  - 3: v
  - 4: v
  - 5: v
  - 6: i
  - 7: i

- Page numbers:
  - Page 0
  - Page 1
  - Page 2
  - Page 3
  - Page 4
  - Page 5
  - Page n
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>$p_i$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
Address-Translation Scheme

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

- **Process $P_1$**
  - Page Table for $P_1$
  - Ed 1
  - Ed 2
  - Ed 3
  - Data 1

- **Process $P_2$**
  - Page Table for $P_2$
  - Ed 1
  - Ed 2
  - Ed 3
  - Data 2

- **Process $P_3$**
  - Page Table for $P_3$
  - Ed 1
  - Ed 2
  - Ed 3
  - Data 3
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
Segments

- subroutine
- stack
- symbol table
- Sqrt
- main program

logical address space
Logical View of Segmentation

(user space)

1
2
3
4

(physical memory space)

1
4
2
3
Segmentation Example

- Segment 0: subroutine
- Segment 1: Sqrt
- Segment 2: main program
- Segment 3: stack
- Segment 4: symbol table

Logical address space:

<table>
<thead>
<tr>
<th>Limit</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1400</td>
</tr>
<tr>
<td>1</td>
<td>6300</td>
</tr>
<tr>
<td>2</td>
<td>4300</td>
</tr>
<tr>
<td>3</td>
<td>3200</td>
</tr>
<tr>
<td>4</td>
<td>4700</td>
</tr>
</tbody>
</table>

Physical memory:

- Segment 0
- Segment 1
- Segment 2
- Segment 3
- Segment 4
Segmentation Architecture

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

- *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 \(s < \text{STLR}\).
Segmentation Architecture (Cont.)

- Relocation.
  - dynamic
  - by segment table

- Sharing.
  - shared segments
  - same segment number

- Allocation.
  - first fit/best fit
  - external fragmentation
### Segmentation Architecture (Cont.)

- Protection. With each entry in segment table associate:
  - validation bit = 0 $\Rightarrow$ 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
Sharing Segments

Logical memory process $P_1$

- Segment 0
  - Editor
  - Data 1

Logical memory process $P_2$

- Segment 0
  - Editor
  - Data 2

Segment table process $P_1$

<table>
<thead>
<tr>
<th>limit</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>25286</td>
<td>43062</td>
</tr>
<tr>
<td>4425</td>
<td>68348</td>
</tr>
</tbody>
</table>

Segment table process $P_2$

<table>
<thead>
<tr>
<th>limit</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>25286</td>
<td>43062</td>
</tr>
<tr>
<td>8850</td>
<td>90003</td>
</tr>
</tbody>
</table>

Physical memory

- Editor
- Data 1
- Data 2
Segmentation with Paging – MULTICS

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments.
- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a page table for this segment.
Virtual Memory Principles

A process can execute without having all its pages in physical memory. Some advantages are:

- A user process can be larger than physical memory
- Less I/O for loading and unloading for individual user processes
- Higher CPU utilization and throughput.
Virtual Memory Techniques

- Overlays (old technique)
- Paged virtual memory
- Segmented virtual memory
Paged Virtual Memory

When to swap pages into memory:

- **Demand paging** - a page is not swapped in until it is referenced
- **Prepaging** - a page is swapped in before it is referenced
Demand Paging

- Load 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 the page)
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ load into memory
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated:
  - 1 (in-memory)
  - 0 (not-in-memory)
- Initially valid–invalid but is set to 0 on all entries.
- Example of a page table snapshot.

<table>
<thead>
<tr>
<th>Frame #</th>
<th>valid-invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>0</td>
</tr>
<tr>
<td>n+1</td>
<td>0</td>
</tr>
</tbody>
</table>

*page table*
Page Fault

Each page table entry has a resident bit, it indicates whether the corresponding page is in memory.

- If the page is not in memory, a page fault has occurred and the control is trapped to the OS.
- During address translation, if valid–invalid bit in page table entry is 0, it is a page fault.
Demand Paging

In demand paging, a **page fault** occurs when a reference is made to a page **not in memory**. The page fault may occur while:

- fetching an instruction, or
- fetching an operand of an instruction.
Page Replacement

- When there is a page fault, the referenced page must be loaded but there is no available frame in memory so one page is selected for replacement.

- If the selected page has been modified, it must be copied back to disk.
Performance and Page Replacement

- Page replacement
  - performance – select a replacement algorithm that will result in minimum number of page faults.
- Same pages may be referenced several times.
Handling Of A Page Fault

1. For every page reference, the page table entry for the page referenced is examined. If the access is invalid, the process is terminated.

2. If the page is not in memory, a page fault occurs and the OS interrupts the process.

3. The OS handles the page fault
   1. carries out the page replacement.
   2. Swaps out the replaced page
   3. Swaps in the referenced page
   4. Reschedules the process that caused the page fault

4. The instruction which caused the page fault is restarted.
Performance Goal in VM

- Directly dependent on the number of page faults
- It is very important to keep the page fault rate low.
- How can this be accomplished?