1. Various ways of organizing memory hardware.
2. Memory-management techniques:
   1. Paging
   2. Segmentation.
Introduction

• Memory consists of a large array of words, each have its own address.

• A typical instruction-execution life cycle:
  • **Fetches**(load) an instruction from specific memory address.
  • **Decode** the instruction and fetch operands from specific memory address.
  • **Execute** the instruction on the operands.
  • **Store** the results back in memory.

• Program (set of instructions) must be brought (from Hard disk) into memory and placed within a process for it to be run.

• Memory unit only sees:
  • A stream of addresses + read requests (**READ Operation**).
  • A stream of address + data and write requests (**Write Operation**).

• Memory unit does not know:
  • how they are generated (instruction counter, indexing, indirection, …)
  • What they are for (instructions or data).
Basic Hardware

• Concerned with relative speed of accessing physical memory,
  • CPU can access directly the registers built into it (within one CPU clock) and main memory (may take many cycles of the CPU clock).
  • Any manipulated data (instructions/Data) used in execution must be in one of these direct-access storage devices.
  • If the data are not in memory, they must be moved before the CPU can operate on them (may take many cycles of the CPU clock).
  • The CPU must be wait until the data is present on the direct-access storage devices.
  • Solution: add fast memory between the CPU and main memory called a cache memory.

• Ensure correct operation:
  • To protect the OS from access by user processes.
  • Protect user processes from one another (any process has its own address space).
  • It can be implemented in several ways.
• **Basic Hardware**
  • Used to protect the OS.
  • Make sure that each process has a separate memory space (Process Address Range) that can access.
  • Ensure that the process can access only these legal addresses.

• **Using two registers:**
  • **Base register** holds the smallest legal physical memory address;
  • **Limit register** specifies the size of the range.

• **The OS can:**
  • load the base and limit registers using OS special privileged kernel instruction
  • Load users’ programs into users’ memory.
  • Prevent out those programs in case of errors.
  • access and modify parameters of system calls.

• **example,**
  • if the base register = 300040
  • the limit register is 120900
  • Then the program can legally access all addresses from 300040 through 420939 (inclusive).
Basic Hardware

A base and a limit register of a logical address space.

Hardware Address Protection with Base and Limit registers
Address Binding

• A program:
  • Resides on a disk as a binary executable file.
  • Must brought into main memory and replaced within a process to be.
  • The process may be moved between disk and memory during its execution.
  • Processes on the disk are waiting to be brought into memory for execution.
  • One of the processes in the disk is selected to be loaded into memory.
  • As the process is executed, it accesses instructions and data from memory.
  • Eventually, the process terminates, and its memory space is declared available.

• A user program will go through several steps (optional).

• Addresses may be represented in different ways during these steps.
  • **Source program:** addresses are generally **symbolic Address** (such as count).
  • **A compiler** will typically bind these symbolic addresses to **relocatable addresses** (such as “14 bytes from the beginning of this module”).
  • **linkage editor/loader** will bind the relocatable addresses to **absolute addresses** (such as 74014).
  • Each binding is a mapping from one address space to another
Binding of Instructions and Data to Memory

• Address binding of instructions and data to memory addresses can happen at three different stages
  
  • **Compile time:**
    • If memory location known a priori, **Symbolic 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)
Logical and Physical Address Space

• **logical address (virtual address):**
  • Is an address generated by the CPU.
  • **Logical address space:** set of all logical addresses generated by a program.

• **Physical address:**
  • Seen by the memory unit (loaded in memory-address register MAR).
  • **Physical address space:** the set of all physical addresses corresponding to these logical addresses.

• **logical and physical addresses** are:
  • Identical in the compile-time and load-time.
  • Different in the execution time.

• **Memory Management Unit**
  • HW device maps logical(virtual) address into physical address at run time.
  • Simply every address generated by a user process is added to value in the relocation register(Base Register) at the time it is sent to memory.
  • The user program deals with logical addresses (never sees physical addresses).
Memory Management Unit

- A HW device that maps logical(virtual) address into physical address at run time.
- Simply every address generated by a user process is added to value in the relocation register(Base Register) at the time it is sent to memory.
- The user program deals with logical addresses (never sees physical addresses).
- The program can create a pointer to location 346, store it in memory, manipulate, and compare it with other addresses (ex. 348).
- We now have two different types of addresses:
  - **logical addresses** by user programs (in the range 0 to max)
  - **Physical addresses** (in the range R + 0 to R + max for a base value R).
Swapping

• If the total physical memory space of processes is greater than physical memory.
• A process can be swapped temporarily out of memory to a HD, and vice versa.
• **Roll out:** a lower priority process can be swapped from main memory to the HD If a higher-priority process arrives and wants service.
• **Roll in:** When the higher-priority process finished, the lower-priority process can be swapped back in To the memory and execution is continued.
• Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  • Swapping normally disabled
  • Enabled if more than threshold amount of memory allocated
  • Disabled again once memory demand reduced below threshold
The effect of Swapping on the context switch time

- If next processes to be put on CPU is not in memory, **swap out** a process and **swap in** target process
- Context switch time can then be very high

**Example:**
- 100MB process swapping to hard disk with transfer rate of 50 MB/sec.
- The actual transfer to or from main memory = \(100 \text{ MB} \div 50 \text{ MB/s} = 2\) seconds.
- Disk latency is 8 s.
- **Swap out time** = actual transfer + disk latency = \(2 \text{ s} + 8 \text{ ms} = 2008 \text{ ms}\)
- **Swap in time** of the same process = 2008 ms
- Total context switch swapping = **Swap out time + Swap in time** = 4016 ms (> 4 s)
Contiguous Memory Allocation
Contiguous Allocation

- Main memory usually divided into two partitions:
  - **OS Part** (low memory)
  - **User Processes Part** (high memory).
- Each process contained in single contiguous section of memory.

**Memory Mapping and Protection**

- The **relocation register (Base)** contains the value of the smallest physical address;
- the **limit register** contains the range of logical addresses.
- Example: relocation = 100040, limit = 74600.
- Each logical address must be less than the limit register;
- The MMU maps the logical address *dynamically* by adding the value in the relocation register.
- This mapped address is sent to memory.

*Figure 8.6  Hardware support for relocation and limit registers.*
Contiguous Allocation (Cont.)

- **Multiple-partition allocation**
  - Degree of multiprogramming limited by number of partitions
  - **Hole**: A block of available memory of various size scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Terminated Process frees its partition.
  - Adjacent free partitions is combined
  - OS 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 (Hole Size $\geq$ Requested Size).
    • Searching can started to find a suitable hole.
    • And stopped as soon as a free hole that is large enough is founded.
  
  • **Best Fit:**
    • Allocate the smallest hole that is big enough (Hole Size $\approx$ Requested Size).
    • An entire list of holes must be searched, unless the list is ordered by size.
    • This strategy produces the **smallest leftover hole**.
  
  • **Worst fit:**
    • Allocate the largest hole (Hole Size $\gg$ Requested Size).
    • The entire list must be searched, unless it is sorted by size.
    • This strategy produces the **largest leftover hole** (more useful than the smaller leftover hole from a best-fit approach).

• **First-fit and best-fit better than worst-fit in terms of speed and storage utilization.**
Fragmentation

• External Fragmentation:
  • As processes are loaded and removed from memory the free memory space is broken into little space.
  • External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous;
    • storage is fragmented into a large number of small holes.

• Internal Fragmentation:
  • Consider a hole size= 18,464 bytes and next process requests 18,462 bytes.
  • A hole of 18,464 - 18,462 = 2 bytes are created.
  • The overhead to keep track of this hole will be substantially larger than the hole itself.
    • unused memory that is internal to a partition.

• Compaction: used to reduce External fragmentation:
  • Move 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
The scheduling algorithm will be FCFS
Memory Paging
Paging

• Is a Memory Management scheme that enables the physical address space of a process to be non-contiguous.
• Avoids external fragmentation and the need for compaction.
• Paging is more advantageous and used in most OS.
• Paging is handled by hardware.
• In recent OS Paging is handled by integrating the hardware and OS, especially on 64-bit CPU.
Paging – Basic Method

• Dividing logical memory into blocks of the same size called *pages*.
• Dividing physical memory into fixed-sized blocks called *frames*.
• Their size of page/frame is defined by the hardware:
  • Is typically a power of 2,
  • varying between 512 bytes and 16 MB per page/frame.
• When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store - is divided into fixed-sized blocks that are of the same size as the memory frames).
• Every address generated by the CPU is divided into two parts:
  • Page number (**p**) is used as an index into a *page table*. The page table contains the base address of each page in physical memory.
  • Page offset (**d**). Is combined with the base address to define the physical memory address that is sent to the memory unit.
• A fixed page size makes the translation of a logical address into a page number and page offset easy.
Paging – Basic Method

**Figure 8.7** Paging hardware.

**Figure 8.8** Paging model of logical and physical memory.
Paging – Basic Method

• If the logical address space is $2^m$, and a page size is $2^n$ addressing units (words) then number of pages is $2^m/2^n = 2^{m-n}$.

• The high-order $m - n$ bits of a logical address designate the page number, and the $n$ low-order bits designate the page offset.

• $p$ is an index into the page table and $d$ is the displacement within the page.

\[
\begin{align*}
\text{Page Number} & \quad \text{Page Offset} \\
p & \quad d \\
&m - n & \quad n
\end{align*}
\]

\[
P \quad \text{Page Size} \\
L \quad \text{Logical Address} \\
L \Rightarrow p, d \\
p = L \div P \\
d = L \mod P \\
\text{Ph. Addr.} = (f - 1) \times P + d
\]
Paging – Example

Example: \( n = 2 \) (page size=4 bytes) and \( m = 4 \). Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages).

- \( 32=2^5 \) (5-bit address, bits (1,2,30) used to get page number, bits(4,5) used to get byte within a page).
- Logical address 0 is page 0, offset 0, maps to physical address \((5 \times 4) + 0=20\).
- Logical address 3 (page 0, offset 3) maps to physical address \((5 \times 4) + 3=23\).
- Logical address 4 is page 1, offset 0, maps to physical address \((6 \times 4) + 0=24\).
- Logical address 13 maps to physical address 9.

<table>
<thead>
<tr>
<th>Logical Address ((p,d))= (0,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
</tr>
<tr>
<td>Page No. = 0</td>
</tr>
<tr>
<td>Offset=0</td>
</tr>
<tr>
<td>Frame No. =5</td>
</tr>
<tr>
<td>5 * page size(4) =5*4=20</td>
</tr>
<tr>
<td>Physical Address = 20 + 0 =20</td>
</tr>
</tbody>
</table>
Paging – Usage

• When using a paging scheme, **no external fragmentation**.
• any free frame can be allocated to a process that needs it.
• **Example:**
  – page size is 2,048 bytes
  – a process of 72,766 bytes will need 35 pages plus 1,086 bytes.
  – It will be allocated 36 frames, resulting in internal fragmentation of 2,048 – 1,086 = 962 bytes.
  – **In the worst case**, a process would need n pages plus 1 byte (ex. 2049). It would be allocated n + 1 frames, resulting in internal fragmentation of almost an entire frame.
• The small page sizes are desirable. (overhead is involved in each page-table entry, and this overhead is reduced as the size of the pages increases.
• Generally, page sizes have grown over time as processes, data sets, and main memory have become larger.
• Today, pages typically are between 4 KB and 8 KB in size, and some systems support even larger page sizes.
• Some CPUs and kernels even support multiple page sizes. For instance, Solaris uses page sizes of 8 KB and 4 MB, depending on the data stored by the pages.
Paging – Usage

• When a process arrives in the system to be executed:
  – Express its size pages.
  – Each page of the process needs one frame.
  – Thus, if the process requires n pages, at least n frames must be available in memory.
  – If n frames are available, they are allocated to this arriving process.
  – For each page, Insert record in the page table (page number, frame

• The user program:
  – Views memory as one single space, containing only this one program.
  – In fact, the user program is scattered throughout physical memory, which also holds other programs.
  – The difference between the user’s view of memory and the actual physical memory is reconciled by the address-translation hardware.
  – This mapping is hidden from the user and is controlled by the OS.
  – The user process by definition is unable to access memory it does not own.
  – It has no way of addressing memory outside of its page table, and the table includes only those pages that the process owns.
Frame Table

– A data structure used by the OS to keep track of:
  • Allocated frames.
  • Available Frames.

– Has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

– The OS maintains a copy of the page table for each process.

– This copy is used to translate logical addresses to physical addresses whenever the OS must map a logical address to a physical address manually.

– Paging therefore increases the context-switch time.
Figure 8.10 Free frames (a) before allocation and (b) after allocation.
Page Table – HW Implementation

- Each OS has its own methods for storing page tables.
- OS allocate a page table for each process.
- A pointer to the page table is stored in the PCB registers.
- When the dispatcher starts a process:
  - it must reload the PCB registers
  - define the correct hardware frames.

1. **Page Table implementation Using Registers:**
   - A set of dedicated registers with very high-speed logic to make the paging-address translation efficient.
   - Every access to memory must go through the paging map.
   - Instructions to load or modify page-table registers are run only by OS.
   - Is satisfactory if the page table is reasonably small (for example, 256 entries).
   - For Large page table the use of registers is not feasible (1 million entries).
2. **Storing the page Table in main memory:**
   - The page table is kept in main memory, and a page-table base register (PTBR) points to the page table.
   - Changing page tables requires changing only this one PTBR register.
     - reducing context-switch time
   - **The problem,**
     - Accessing the main memory tow times
       - One for the page table
       - One for accessing the data.
     - Thus, memory access is slowed by a factor of 2
3. **Translation look-aside buffer (TLB)**

- A small fast lookup hardware cache, associative high-speed memory.
- Each entry in the TLB consists of a key and a value.
- For any CPU address search the page number in TLB:
  - If found (**TLB hit**) and return frame number.
  - If it is not found in the TLB (**TLB miss**)—Add an entry so that they will be found quickly on the next reference.
- If the TLB is already full of entries, the OS must select one for **replacement**.
- Replacement policies range from least recently used (LRU) to random.
- Sometimes Kernel Code not allowed to be removed.
Figure 8.11  Paging hardware with TLB.
• **hit ratio**: The percent of times that a particular page number is found in the TLB.

• **Example**:  
  • Hit ratio=80%.
  • If the item is found in TLB, (it takes 20 ns to search the TLB, 100 ns to access memory=120 ns).
  • If the item is not found in the TLB (20 ns search LTB + 100 ns to access memory to obtain Frame + 100 to access memory for data= 220 ns).

• **effective access time** = \(0.80 \times 120 + 0.20 \times 220 = 140 \text{ ns}\)
1. **Using protection bits (RD / WR - Read only).**
   - Each time the physical address is accessed the protection bits can be checked.
   - Any write to a read-only page will cause an OS memory-protection violation.

2. **Use valid–invalid bit.**
   - Valid means legal address.
   - Invalid (the page is not in the process’s legal address space).
   - The OS sets this bit for each page to allow or disallow access to the page.

**Example**

- System 14-bit address (0 to 16383), Page size=2KB.
- A program with addresses 0 to 10468 (pages 0, 1, 2, 3, 4, and 5 are mapped normally through the page table).
- Any attempt to generate an address in pages 6 or 7, however, will find invalid, and the computer will trap to the OS (invalid page reference).
• Example:
  • A system of 40 users executes a text editor.
  • The text editor needs 150 KB of code and 50 KB of data space, we need 8,000 KB.
  • The code can be shared, but data can not.
  • The code is shared among 40 processes.
  • Each process has its own data page. has its own copy of registers and data storage.
  • Only one copy of the editor need be kept in physical memory.
  • Each user’s page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
  • We need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user = 2150 KB.
  • Other heavily used programs can also be shared compilers, window systems, runtime libraries, database systems.
Structure of the page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB ($2^{12}$)
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don’t want to allocate that contiguously in main memory

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

- Divide the page table into smaller pieces.
- Use a two-level paging algorithm, in which the page table itself is also paged.
- Example:
  - a system with a 32-bit address and a page size of 4 KB.
  - One level (20 page number+12 offset d)
  - The 20 bit page number divided to a 10 page number and a 10 page offset.
  - P1 is an index into the outer page table and p2 is the displacement within the page of the outer page table.
64-bit Logical Address Space

• Even two-level paging scheme not sufficient

• If page size is 4 KB ($2^{12}$)
  • Then page table has $2^{52}$ entries
  • If two level scheme, inner page tables could be $2^{10}$ 4-byte entries
  • Address would look like

• Outer page table has $2^{42}$ entries or $2^{44}$ bytes
• One solution is to add a 2$^{nd}$ outer page table
• But in the following example the 2$^{nd}$ outer page table is still $2^{34}$ bytes in size
  • And possibly 4 memory access to get to one physical memory location
Hashed Page Tables

- The hash value = virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location.
- Each element consists of three fields:
  - The virtual page number
  - the value of the mapped page frame
  - a pointer to the next element in the linked list.
- The algorithm works as follows:
  - The virtual page number = hash value.
  - The virtual page number is compared with field 1 in the first element in the linked list.
  - If there is a match, the corresponding frame (field 2) is used to form the desired physical address.
  - If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.
Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages.
- Track all physical pages
- 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
Segmentation

• Paging allow the separation of the user’s view of memory from the actual physical memory.
• The user’s view of memory is not the same as the actual physical memory.
• The user’s view is mapped onto physical memory.
• This mapping allows differentiation between logical memory and physical memory.
Segmentation – Basic Method

• is a memory-management scheme that supports this user view of memory.
• A logical address space is a collection of segments.
• Each segment has a name and a length.
• The user specifies each address by two quantities: a segment name and an offset.
Segmentation

• 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
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`

• 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

Example of Segmentation

Diagram showing the process of address translation in a segmented memory system. The CPU sends a tag 's' to the segment table, which is used to determine if the address is valid. If the address is less than the segment limit, it is added to the base address to get the physical memory address. If the address is not valid, a trap for addressing error is generated. The diagram also shows the logical address space and the segment table with specific segment limits and bases.
End of Chapter 8
Chapter 9: Virtual Memory

• To describe the benefits of a virtual memory system

• To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames

• To discuss the principle of the working-set model
Background

• All memory-management strategies have the same goal:
  • to keep many processes in memory simultaneously to allow multiprogramming it must keep the entire process in memory before it can execute.

• The first approach is to place the entire logical address space in physical memory, but it generally requires special precautions and extra work by the programmer.

• The requirement that instructions must be in physical memory to be executed seems both necessary and reasonable;

• but it is also unfortunate, since it limits the size of a program to the size of physical memory.

• In many cases, the entire program is not needed. For instance, consider the following:
  • Programs often have code to handle unusual error conditions.
  • Arrays, lists, and tables are often allocated more memory than they actually need.
  • Certain options and features of a program may be used rarely.
Virtual memory

• Is the separation of user logical memory from physical memory
  • Only part of the program needs to be in memory for execution
  • Logical address space can be much larger than physical address space
  • Allows address spaces to be shared by several processes
  • Allows for more efficient process creation
  • More programs running concurrently
  • Less I/O needed to load or swap processes.

• Virtual memory can be implemented via:
  • Demand paging
  • Demand segmentation
Virtual Memory That is Larger Than Physical Memory
Virtual Address Space of a process

- Refers to the logical (or virtual) view of how a process is stored in memory.
- This view is that a process begins at a certain logical address, address 0—and exists in contiguous memory.
- Allow the heap to grow upward in memory through **dynamic memory allocation**.
- Allow the stack to grow downward in memory through successive **function calls**.
- The large blank space between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows.
Shared Library Using Virtual Memory

- Virtual Memory allows files and memory to be shared by two or more processes through page sharing.
- This leads to the following benefits:
  - System libraries can be shared by several processes through mapping of the shared object into a virtual address space. The actual pages where the libraries reside in physical memory are shared by all the processes.
  - Enables communicated processes to share memory. Virtual memory allows one process to create a region of memory that it can share with another process.
Demand Paging

• For executing a program (Could bring entire process into memory at load time)
  • the entire program in physical memory at program execution time.
  • problem: we may not initially need the entire program in memory
  • example: Suppose a program starts with a list of available options from which the user is to select.

• Solution:
  • is to load pages only as they are needed (demand paging technique and is commonly used in virtual memory systems).

• demand paging: bring a page into memory only when it is needed
  • Less I/O needed, no unnecessary I/O
  • Less memory needed
  • Faster response
  • More users

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

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

- HW are needed to distinguish between pages in memory and pages in the disk.
- With each page table entry a valid-invalid bit is associated:
  \( v \Rightarrow \text{ page in-memory – memory resident,} \)
  \( i \Rightarrow \text{ page not-in-memory}) \)
- Initially: valid-invalid bit is set to \( i \) on all entries
- During address translation, if valid-invalid bit in page table entry is \( i \Rightarrow \text{page fault} \)
Page Fault

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

1. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory (in HDD).

2. Get empty frame

3. Swap page into frame via scheduled disk operation

4. Reset tables to indicate page now in memory
   Set validation bit = $v$

5. Restart the instruction that caused the page fault
Performance of Demand Paging (stages)

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
   1. Wait in a queue for this device until the read request is serviced
   2. Wait for the device seek and/or latency time
   3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction
Performance of Demand Paging (Cont.)

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

• Effective Access Time (EAT)
  
  $\text{EAT} = (1 - p) \times \text{memory access} + p \times \text{(page fault service)}$

With an average page-fault service time of 8 milliseconds and a memory-access time of 200 nanoseconds, the effective access time in nanoseconds is

\[
\text{effective access time} = (1 - p) \times (200) + p \times 8,000,000 \\
= (1 - p) \times 200 + p \times 8,000,000 \\
= 200 + 7,999,800 \times p.
\]

We see, then, that the effective access time is directly proportional to the page-fault rate. If one access out of 1,000 causes a page fault, the effective access time is 8.2 microseconds. The computer will be slowed down by a factor of 40 because of demand paging! If we want performance degradation to be less than 10 percent, we need

\[
220 > 200 + 7,999,800 \times p, \\
20 > 7,999,800 \times p, \\
p < 0.0000025.
\]
What Happens if There is no Free Frame?

• **Page Replacement:**
  • Find some page in memory, but not really in use, page it out
  • Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
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
     - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame;
4. Update the page and frame tables
5. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT
Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace

- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access

Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

- String is just page numbers, not full addresses
- Repeated access to the same page does not cause a page fault

In all our examples, the reference string is

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

- Reference string: \(7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1\)
- 3 frames (3 pages can be in memory at a time per process)

15 page faults
Least Recently Used (LRU) Algorithm

• Use past knowledge rather than future
• Replace page that has not been used in the most amount of time
• Associate time of last use with each page
• 12 faults – better than FIFO but worse than OPT
• Generally good algorithm and frequently used
• But how to implement?
End of Chapter 9