Objectives

- 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 examine the relationship between shared memory and memory-mapped files
- To explore how **kernel memory** is managed
Background

- The instructions being executed must be brought into physical memory.
- The first approach to meeting this requirement is to place the entire logical address of the program in physical memory.
- Dynamic loading can help to ease this restriction, but it generally requires special precautions and extra work by the programmer.
- Code needs to be in main memory to execute, but entire program rarely used due to:
  - Error code, unusual routines, large data structures
  - Entire program code not needed at same time
  - Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements.
Background

Consider ability to **execute partially-loaded program** in memory has the following benefits:

- A *user program doesn't need to be constrained by the size of physical memory that is available.*
- User programs would be allowed for an extremely large *virtual address space*, simplifying the programming task.
  - Each program takes less memory while running \(\rightarrow\) more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time
  - Less I/O needed to load or swap programs into memory \(\rightarrow\) each user program runs faster

Thus, running a program that is not entirely in main memory would benefit both the system and the user.
Virtual memory involves the separation of logical address space (address of a user program in a disk space) from physical memory (main memory).

This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Figure 9.1).

Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the availability of physical memory space.
Virtual Memory That is Larger Than Physical Memory

Figure 9.1 Diagram showing virtual memory that is larger than physical memory.
Background

- **Virtual Memory (VM)** – separation of *user logical memory* from physical memory
  - Only a part of the user program needs to be in main memory for execution
  - Logical address space (disk space) can be much larger than physical address space (main memory space)
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs can run concurrently
  - Less I/O needed to load or swap processes
Background

- **Virtual address space** ⇒ *logical view* of how an executable file is stored in the disk memory
  - Usually start at **address 0**, contiguous addresses until end of space

- **Physical memory** (Main Memory) organized in **page frames**
  - MMU must map *logical memory* to *physical memory*

- **Virtual memory** access implemented via:
  - Demand paging
  - Demand segmentation
Virtual-address Space

- Figure 9.2 shows the heap to grow upward in memory as it is used for dynamic memory allocation
  - Dynamic memory allocation: allocating user code from its disk folder into MM during the execution time.

- Similarly, the stack will grow downward in memory through successive function calls.

- The large blank space (or hole) between the heap and the stack is part of the virtual address space (such as page table) but will be used for sharing the space with other processes.

- Virtual address spaces that include holes are known as sparse address spaces.
Figure 9.2  Virtual address space.
Shared Library Using Virtual Memory

Figure 9.3 Shared library using virtual memory.
Demand Paging

Consider how an executable file (.exe file) might be loaded from its compiler folder into main memory space.

- **One option** is to load the entire program in main memory at program execution time:
  - However, a problem with this approach is that it may not initially *needs the entire program in memory* for the program to be executed.

- **An alternative strategy** is to load pages only as they are needed.
  - This technique is known as **demand paging** and is commonly used in virtual memory systems.
Demand Paging

- With demand-paged virtual memory, pages are loaded into MM only when they are demanded during the program execution.
  - Pages that are never accessed (not needed) are thus never loaded into physical memory.

- A demand-paging system is similar to a paging system with swapping (Figure 9.4) where processes reside in VM.

- When we want to execute a process, swap it into memory, rather than swapping the entire process into memory, though, we use a lazy swapper.
  - A lazy swapper never swaps a page into memory unless that page will be needed.
  - We thus use “pager,” rather than “swapper,” in connection with demand paging.
Demand Paging

- Bring a page into memory only when it is needed
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)
- Page is needed \(\Rightarrow\) reference to it
  - not-in-memory (invalid) \(\Rightarrow\) bring to memory
- Lazy swapper – “never swaps a page into memory unless page will be needed”
  - Swapper that deals with pages is a pager

![Figure 9.4 Transfer of a paged memory to contiguous disk space.](image-url)
Valid-Invalid Bit

- Instead of swapping-in a whole process, the pager (the page swapper) brings only those in needed pages into memory.
  - Thus, it avoids unwanted pages and hence decreasing the swap time and the amount of physical memory needed.

- With this scheme, we need some form of HW to distinguish between the pages that are in memory (MM) and the pages that are on the VM (it is a part of disk act as MM during the time of process execution).

- The valid–invalid bit scheme associated with a page table can be used for this purpose.

- When this bit is set to “valid,” the associated page is both legal and in main memory.

- If the bit is set to “invalid,” means that the page is not in MM or is valid but is still on the VM (see the Figure 9.5).
Figure 9.5 Page table when some pages are not in main memory.
Page Fault

What happens if the process tries to access a page that was not brought into main memory?

- Access to a page marked invalid causes a page fault.

The paging HW, the one which translates the process logical address through the page table, will notice that the invalid bit is set in the page table. This causes a trap (System call) to the OS – Indicates a page fault!

- This will result the execution of page-fault handler routine (PFHR) by the OS.
Page Fault

- If there is a reference to a page missing in the page table, that will **trap to OS** and is known as a **page fault**.

- During a **page fault**:
  1. OS looks at the **page reference**:
     - If the page status in **page table** is invalid (valid-invalid bit = 0), that means the page is not in the MM frame, then
  2. Find a free memory frame for the referenced page
  3. Swap the page into the MM frame via disk swap operation
  4. Update **page table** to indicate the page now in memory and set its page **valid-invalid bit** = v (by asserting ‘1’)
  5. Restart the instruction that caused the **page fault**.
  6. Update the page table

- The detailed procedure for handling this **page fault** is straightforward (**Figure 9.6**):
Steps in Handling a Page Fault

Figure 9.6 Steps in handling a page fault.
Steps in Handling a Page Fault

The procedure for handling this page fault is straightforward (Figure 9.6):

1. First, check the page table for this process to determine whether the reference was a valid or an invalid memory access.

2. If the reference was invalid, we terminate the process. If it was valid but have not yet brought in that page in MM from disk.

3. Find a free frame (by evicting one of the frames – called page replacement).

4. Schedule a disk operation to read the desired page and swap that into the newly allocated frame.

5. When the disk read is complete, modify the page table to indicate that the new page is now in memory.

6. Then restart the instruction that was interrupted by the trap (PFHR).
Performance of Demand Paging

- How to Compute the effective memory access time (EMAT) for a demand-paged memory?
  - For most computer systems, the memory-access time, ranges from 10 to 200 nanoseconds.

- If there is no page fault, then the EMAT is equal to the memory access time (MAT).

- Let $p$ be the probability of a page fault ($0 \leq p \leq 1$). We would expect $p$ to be close to zero—that is, we would expect to have only a few page faults. The EMAT is then:

$$EMAT = (1 - p) \times MAT + (p \times \text{page fault time})$$

- With an average page-fault service time of 8 ms and a memory access time of 200 ns, the EMAT in ns is $(1 - p) \times (200) + p \times (8 \text{ ms})$

$$= (1 - p) \times 200 + p \times 8,000,000 \quad \{1\text{ms} = 1,000,000\text{ns}\}$$

$$= 200 + 7,999,800 \times p.$$
A Page Fault Causes The following

- A page fault causes the following:

1. Trap to the OS
2. Save the user registers and process state
3. Perform PFHR
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:
   - Wait in a queue for this device until the read request is serviced
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (indicates page transfer completed)
8. Update the page table and other tables to show page is now in memory
9. Wait for the CPU to be allocated to this process again
10. Restore the user registers, process state, and new page table, and then resume the interrupted instruction
What Happens if There is no Free Frame?

- **Page replacement** – find some page in memory frame, but not really in use, then swap it out
  - **Algorithm** – terminate? swap out? replace the page?
  - **Performance** – want an algorithm which will result in minimum number of *page faults*

- *Same page may be brought into memory several times*
Page Replacement

- Consider that the system memory is not used only for holding all of the pages from the user program but also it acts as buffers for I/O operation.
- This can increase the strain on memory-placement algorithms.
- During a program execution, deciding how much memory to allocate to I/O and how much to program pages is a significant challenge of the OS.
- Some systems allocate a fixed percentage of memory for I/O buffers, whereas others allocate all system memory for both user processes and I/O operations.
Page Replacement

- Over-allocation of memory manifests itself as follows:
  - Assume that a user process causes a page fault. The OS determines where the desired page is residing on the disk. In case, there are no free frames on the MM for the missing page, then OS should free one of the memory frames for the new page (Figure 9.9).

- The OS has several options at this point. It could terminate one of the user process:
  - However, demand paging is the OS’s attempt to improve the computer system’s utilization and throughput.
  - paging should be logically transparent to the user.

- So this option is not the best choice. The OS could instead swap out a process, freeing all its frames and reducing the level of multiprogramming.
Figure 9.9 Need for page replacement.
Page replacement from the MM takes the following steps:

- If no frame is free, find out one of the frames that is not currently being used for a long time and free it.
- Free a frame by moving it to the disk space and updates the page table (and all other tables) to indicate that the page is no longer in memory (Figure 9.10).
- Then use the freed frame to hold the missing page for which the process needed.
Figure 9.10 Page replacement.
Basic Page Replacement

- **Page replacement** is described as follow:

  1. Find the location of the desired page on the disk.
  2. Find a free frame:
     a. If there is a free frame, use it.
     b. If there is no free frame, use a **page-replacement algorithm** to select a **victim frame** (*the frame which is going to be replaced from a MM frame is called the victim frame*).
     c. Write the **victim frame** to the disk (**swap-out**); update the page table accordingly.
  3. Read the desired page into the newly freed frame
  4. Continue the user process from where the page fault occurred.
Page-replacement Algorithms

- There are many different page-replacement algorithms.
  - Every OS probably has its own page replacement scheme.

- How do we select a particular replacement algorithm?
  - In general, we want the one with the lowest page-fault rate.

- We evaluate a page replacement algorithm by running it on a particular string of memory references and computing the number of page faults.
  - The string of memory references is called a reference string.
Page-replacement Algorithms

- **Page replacement** is basic to **demand paging**.
- It completes the separation between **logical memory** (Disk) and **physical memory** (MM).
- With this mechanism, an **enormous virtual memory** can be provided for programmers on a **smaller physical memory**.

**With no demand paging**, user addresses (*logical addresses*) are mapped into physical addresses:
  - That is, all the pages of a process still must be in physical memory!

**With demand paging**, the size of the logical address space (**program size**) is no longer constrained by physical memory size.
For example, assume a user process has twenty pages, the system can execute it in ten frames simply by using demand paging and using a replacement algorithm to find a free frame whenever necessary:

- If a page that has been modified (updated) is to be replaced, its contents are copied to the disk.
- A later reference to that modified page will cause a page fault (because the modified page is moved to disk).
  - At that time, the page will be brought back into memory, perhaps replacing some other page in the process.
There are many page replacement algorithms available but this topic focuses only the three of them:

- Optimal
- Least recently used (LRU)
- First-in-first-out (FIFO)
The optimal page replacement policy selects the page for which the time to the next reference is the longest (if the newly resident page no. is less than the arrived page, then replace that one or replace the previous big one with a page fault).

The Figure 8.15 gives an example of the optimal policy. The example assumes a fixed frame allocation for this process of three frames.

The execution of the process requires reference to five distinct pages.

The page reference string is 2 3 2 1 5 2 4 5 3 2 5 2 (means that the first page referenced is 2, the second page is referenced is 3, the third page referenced is 2, and so on).

The optimal policy produces three page faults after the frame allocation has been filled (see Figure 8.15).

This policy is very complex to implement, because it would require the OS to have perfect knowledge of future page addresses.
Page-replacement Algorithm - Optimal

Figure 8.15 gives an example of the optimal policy
The least recently used (LRU) policy replaces the page in memory that has not been referenced for longest time.

The LRU policy does nearly as well as the optimal policy.

To tag each page with the time of its last reference; this would have to be done at each memory reference, both instruction and data.

The problem with this approach is the complexity in implementation.

Even if, the HW would support such scheme, the overhead would be tremendous.

Figure 8.15 shows an example of the LRU policy, using the same page reference string as for the optimal policy example. In this case, there are four page faults.
Page-replacement Algorithm - LRU

Figure 8.15 shows an example of the LRU policy
Page-replacement Algorithm - FIFO

- The logic behind **FIFO policy** is *replacing the page that has been in memory longest*.
  - The FIFO policy treats the page frames allocated to a process as a **circular buffer** and pages are removed in **round-robin** style.

- All that is required is a **pointer** that circles through the page frames of the process.

- This is therefore one of the **simplest page replacement policies** to implement.

- A page fetched into memory a long time ago may have now fallen out of use. This reasoning will often be wrong. **Pages will be repeatedly paged-in and page-out** by the FIFO algorithm.

- **Figure 8.15** shows the FIFO policy results **six page faults**.
  - Note that LRU recognizes that **pages 2 and 5** are referenced more frequently than other pages, whereas FIFO does not.
Figure 8.15 shows the FIFO policy results six page faults
Graph of Page Faults Versus The Number of Frames

The graph shows the relationship between the number of page faults and the number of frames. As the number of frames increases, the number of page faults decreases, indicating an improvement in the system's efficiency.
Allocation of Frames

- Each process needs a **minimum** number of memory **frames** in order to handle its instruction execution.
- 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**
- **Maximum** of course is total frames in the system.
- Two major allocation schemes:
  - fixed allocation
  - priority allocation
- Many variations.
Fixed Allocation of Frames

- **Equal allocation** – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool

- **Proportional allocation** – Allocate according to the size of process
Priority Allocation of Frames

- 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
Thrashing

- **Thrashing** ≡ a process is busy with swapping *pages in* and *out*
  - That is, the process does not have “enough” pages causes very high page-fault rate

- **Page fault** to get page from disk
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking 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 with a shaded area indicating thrashing.](image-url)
Other Issues – Page Size

- Sometimes OS designers have a choice about **page size**
  - Especially if running on custom-built CPU

- **Page size** selection must take into the following consideration:
  - Fragmentation
  - Page table size
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness

- Page size is always power of 2, usually in the range $2^{12}$ (4,096 bytes) to $2^{22}$ (4,194,304 bytes)
Other Issues – TLB Reach

- **TLB Reach** - The amount of memory accessible from the TLB
  - \( \text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size}) \)

- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults

- **Increase the Page Size**
  - This may lead to an increase in fragmentation as not all applications require a large page size

- **Provide Multiple Page Sizes**
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation
Other Issues – Program Structure

- Program structure
  - `int[128,128] data;`

- Each row is stored in one page
  - **Program 1**
    ```c
    for (j = 0; j < 128; j++)
      for (i = 0; i < 128; i++)
        data[i][j] = 0;
    ```
    - Execution will result $128 \times 128 = 16,384$ page faults

  - **Program 2**
    ```c
    for (i = 0; i < 128; i++)
      for (j = 0; j < 128; j++)
        data[i][j] = 0;
    ```
    - Execution will result $128$ page faults
Windows

- Uses **demand paging** with **clustering**. Clustering brings in pages surrounding the faulting page.

- Processes are assigned **working set minimum** and **working set maximum**.

- **Working set minimum** is the *minimum number of pages* the process is guaranteed to have in memory.

- A process may be assigned as many pages up to its **working set maximum**.

- When the amount of free memory in the system falls below a *threshold*, **automatic working set trimming** is performed to restore the amount of free memory.

- **Working set trimming** removes pages from processes that have pages in excess of their **working set minimum**.
Solaris

- Maintains a list of **free pages** to assign **faulting processes**
- **Lotsfree** – threshold parameter (amount of free memory) to begin paging
- **Desfree** – threshold parameter to increasing paging
- **Minfree** – threshold parameter to being swapping
- Paging is performed by **pageout** process
- **Pageout** scans pages using modified clock algorithm
- **Scanrate** is the rate at which pages are scanned. This ranges from **slowscan** to **fastscan**
- **Pageout** is called more frequently depending upon the amount of free memory available
- **Priority paging** gives priority to process code pages