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 discuss the principle of the working-set model

Background

- 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
Virtual-address Space

- Logical view of a process stored in memory.
- View the process memory as beginning in address 0 and occupying a contiguous block of memory.
- As seen before this view is mapped to real memory by the use of page tables.

Direct program references to this region of the address space will be considered invalid.
Valid references are only system operations to push a new stack frame or to allocate heap space per program requests.

Shared Library Using Virtual Memory

Share existing code in memory by mapping part of virtual spaces to frames where shared code resides in memory.

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 ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- Lazy swapper
  - Swapper that deals with pages is a pager
Transfer of a Paged Memory to Swap Space

Similar to paging with swapping, but at the page level instead of the whole process, pages are brought only as needed or properly guessed.

Valid Bit

- With each page table entry a valid bit is associated
  - $v \Rightarrow$ page is valid and is in-memory
  - $i \Rightarrow$ not-in-memory and may be invalid
- Initially, valid bit is set to $i$ on all entries of page table

<table>
<thead>
<tr>
<th>Frame #</th>
<th>Valid Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$v$</td>
</tr>
<tr>
<td></td>
<td>$v$</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>$i$</td>
</tr>
</tbody>
</table>

Pages set to $i$ are not part of the valid address space of the process and will not be referenced by the process. Hence, during address translation, if valid bit in page table entry is $i \Rightarrow$ page fault

Page Fault

- First reference to page will trap to operating system as a page fault
  - Operating system decides:
    - Invalid reference $\Rightarrow$ abort
    - If it is valid, but it is just that it is not in memory:
      1. Get empty frame
      2. Swap page into frame
      3. Reset tables
      4. Set validation bit $= v$
      5. Restart the instruction that caused the page fault
Steps in Handling a Page Fault

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

After Page Fault is Resolved

- **Restart instruction** - it may not be that easy….
  - Some systems have a block move machine instruction.
  - Assume a page fault in the middle of execution of one such instruction.
  - If destination and source blocks overlap, when the instruction is restarted, some portion of the source block may have been modified… so just to restart again may not be the solution.
  - One possible solution for this particular case: first verify that the blocks involved are in memory before the instruction is executed.

Performance of Demand Paging

- **Page Fault Rate** $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- **Effective Access Time (EAT)**
  \[
  EAT = (1 - p) \times (\text{memory access time}) + p \times (\text{page fault time})
  \]
- **Page fault time** at least requires:
  - Service the page-fault interrupt
  - Read in the page
  - Restart the process

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
  \[
  EAT = (1 - p) \times 200 + p \times 8,000,000
  = 200 + p \times 7,999,800
  \]
- If one access out of 1,000 causes a page fault, then $p = 0.001$
  - Hence, EAT = 8.2 microseconds.
  - This is a slowdown by a factor of 40!!
Process Creation

- Virtual memory allows other benefits during process creation:
  - Copy-on-Write
  - Memory-Mapped Files (later)

Copy-on-Write

- Copy-on-Write (COW) allows parent and child processes to *share* the same pages in memory
  - If either process modifies a shared page, only then is the page copied to a new frame and properly assigned to the modifier process.

Copy-on-Write

- COW allows more efficient process creation as only modified pages are copied
  - Free pages are allocated from a **pool** of zeroed-out pages.
    - pages are zeroed for security or privacy issues since those pages may have been previously used by other processes...

Before Process 1 Modifies Page C
Page Replacement

- After a page-fault, the target page needs to be swapped into main memory.
- If no frame is available, some page in memory needs to be swapped out.
- **Page replacement** – find page in memory (victim page), swap it out from its actual frame (victim frame) and use that frame to copy the recently referenced page.

Page Replacement

- Performance – want an algorithm which **minimizes** the **number of page faults**
- Same page may be brought into memory several times
- 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

- Page replacement completes separation between logical memory and physical memory
  - large virtual memory can be provided on a smaller physical memory
Need For 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
   - Otherwise, use a page replacement algorithm to select a victim frame
1. Bring the desired page into the (newly) free frame; update the page and frame tables
2. Restart the process

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
- In all examples, the reference string is: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
First-In-First-Out (FIFO) Algorithm

- The number in red represents a page-fault

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

- 3 frames

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

- 4 frames

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

- Belady’s Anomaly: more frames per process may cause more page faults!

Optimal Algorithm

- Replace page that will not be used for longest period of time

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

- 3 frames

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

- Problem: How do we know this?

*** Not possible most of the time.

- Used for measuring how well any algorithm performs
Least Recently Used (LRU) Algorithm

- **LRU** - replace the page that has the largest time since last reference to it.

  \[
  \begin{array}{ccccccccc}
  1 & 2 & 3 & 4 & 1 & 2 & 5 & 1 & 2 \\
  1 & 1 & 4 & 4 & 5 & 5 & 3 & 3 & 3 \\
  2 & 2 & 1 & 1 & 1 & 1 & 4 & 4 & 4 \\
  3 & 3 & 2 & 2 & 2 & 2 & 2 & 5 & 5 \\
  \end{array}
  \]

- **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
- One major limitation is the need for extra hardware (registers, …) to accelerate the management of the stack.
  - The stack may be updated on every memory reference!

LRU Stack Implementation

reference string

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

stack before a

\[
\begin{array}{cccc}
2 & 7 & 4 \\
1 & 2 & 0 \\
0 & 1 & 7 \\
7 & 0 & 4 \\
\end{array}
\]

stack after b

\[
\begin{array}{cccc}
2 & 7 & 4 \\
1 & 2 & 0 \\
0 & 1 & 7 \\
7 & 0 & 4 \\
\end{array}
\]

Stack Algorithms

- A **stack algorithm** satisfies the following property:
  - The **set of pages in memory** for **n frames** is always a **subset** of the pages that would be in memory with **n+1 frames**.
- **Stack algorithms** have the following important property: they **do not suffer the Belady’s anomaly**.
- LRU and Optimal replacement are stack algorithms.
LRU Approximation Algorithms

- Keep a reference bit on each page entry in page table
  - With each page associate a bit, initially = 0
  - When page is referenced, reference bit is set to 1
  - Replace the one which is 0 (if one exists)
- We do not know the order, however

Second-Chance (clock) Page-Replacement Algorithm

- Second chance (clock algorithm)
  - Need reference bit
  - Clock replacement - pages in memory are explored in a circular manner. Apply: `pageToReplace()` returns index of victim page

```java
// cp is assumed global
int pageToReplace() {
    while (true) {
        if (pagesInMem[cp].referenceBit == 1) {
            pagesInMem[cp].referenceBit = 0;
            cp = (cp + 1) % NUM_FRAMES;
        } else
            return cp;
    }
}
```

Enhanced Second-Chance Algorithm

- For each page, have both, a reference bit and a modify bit.
- Use the pair (rb, mb) of each page to decide replacement
- Replacement does the same circular scan as in 2nd Chance, clearing reference bits. Eventually pick a victim page from the lowest nonempty class found.
**Enhanced Second-Chance Algorithm**

- At any moment the pair for any page is one of the following classes:
  - \((0,0)\) - neither recently referenced not modified
    - ***best to replace***
  - \((0,1)\) - not recently used but modified
    - ***not good to replace - needs write back to disk***
  - \((1,0)\) - recently used but not modified
    - ***perhaps will be used again in near future***
  - \((1,1)\) - recently used and modified
    - ***not good to modify***

**Counting Algorithms**

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm** (Least-Frequently-Used): replaces page with smallest count - the one that has been referenced the least.
  - has the problem of keeping heavily used pages of the past, that may not be used for a while or not be used anymore
  - solution - shift counters to the right from time to time

**MFU Algorithm** (Most-Frequently-Used): based on the argument that the page with the smallest count was probably just brought in and has yet to be used

- These replacement schemes are rarely used - their implementation is kind of expensive and do not approximate well to optimal replacement.

**Allocation of Frames**

- Each process needs *minimum* number of pages
- Example: IBM 370 required up to 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*
Allocation of Frames

- 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

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

\[
\begin{align*}
  m &= 64 \\
  s_1 &= 10 \\
  s_2 &= 127 \\
  a_1 &= \frac{10}{137} \times 64 \approx 5 \\
  a_2 &= \frac{127}{137} \times 64 \approx 59
\end{align*}
\]

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; one process can take a frame from another.
  
  **** This is the most commonly used scheme.
- **Local replacement** – each process selects from only its own set of allocated frames
Thrashing

- If a process does not have “enough” frames assigned, 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 = a process is busy swapping pages in and out

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  $\Sigma$ (size of locality) > total memory size ($m$)
**Working-Set Model**

- \( \Delta \equiv \text{working-set window} \) (fixed number of page references) Example: 10,000 instructions
- Let \( WSS_i \) be the working set size of Process \( P_i \): total number of pages referenced in the most recent \( \Delta \) references.
  - Varies with time
  - \( \Delta \) too small will not encompass entire locality
  - \( \Delta \) too large will encompass several localities
  - \( \Delta = \infty \Rightarrow \text{will encompass entire program} \)

**Working-set model**

- \( D = \Sigma (WSS_i) \equiv \text{total demand frames} \)
- if \( D > m \Rightarrow \text{Thrashing} \)
- Policy if \( D > m \), then suspend one of the processes
- Example using \( \Delta = 10 \). Let \( WS(t) \) be the working set of the running process at time \( t \).

**Keeping Track of the Working Set**

- Approximate with interval timer + a reference bit
- Example: \( \Delta = 10,000 \)
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 \( \Rightarrow \) page in working set
- Why is this not completely accurate?

**Page-Fault Frequency Scheme**

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
Working Sets and Page Fault Rates

- Working set changes as time passes during program execution.
- At different parts of the program, different locality requirements arise.

Memory-Mapped Files

- **Memory-mapped file I/O** allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory.

- A file is initially read using **demand paging**. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.

Memory-Mapped Files

- Simplifies file access by treating file I/O through memory rather than **read**, **write** system calls.

- Also **allows several processes to map the same file** allowing the pages in memory to be **shared**.
Memory-Mapped Shared Memory in Windows

Memory-Mapped Files in Java (1)

```java
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MemoryMapReadOnly {
    // Assume the page size is 4 KB
    public static final int PAGE_SIZE = 4096;

    public static void main(String args[]) throws IOException {
        FileInputStream inFile = new FileInputStream(args[0], "r");
        FileChannel in = inFile.getChannel();
       MappedByteBuffer mappedBuffer = in.map(FileChannel.MapMode.READ_ONLY, 0, inFile.size());
        long numPages = inFile.size() / (long)PAGE_SIZE;
        if (inFile.size() % PAGE_SIZE > 0)
            ++numPages;

        // we will "touch" the first byte of every page
        int position = 0;
        for (long i = 0; i < numPages; i++) {
            byte item = mappedBuffer.get(position);
            position += PAGE_SIZE;
        }

        inFile.close();
    }
}
```

Memory-Mapped Files in Java (2)

Opens the file to map.

Memory-Mapped Files in Java (3)

Create buffer to read from file
Memory-Mapped Files in Java (4)

```java
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MemoryMapReadOnly {
    // Assume the page size is 4 KB
    public static final int PAGE_SIZE = 4096;

    public static void main(String[] args) throws IllegalArgumentException {
        RandomAccessFile inFile = new RandomAccessFile(args[0], "r");
        FileChannel in = inFile.getChannel();
       MappedByteBuffer mappedBuffer = in.map(FileChannel.MapMode.READONLY, 0, inFile.size());
        long numPages = inFile.size() / (long)PAGE_SIZE;
        if (inFile.size() % PAGE_SIZE > 0) ++numPages;
        // we will "touch" the first byte of every page
        int position = 0;
        for (long i = 0; i < numPages; i++) {
            byte item = mappedBuffer.get(position);
            position += PAGE_SIZE;
        }
    }
}
```

Mode of the mapping. Three modes: READ_ONLY, WRITE_ONLY, PRIVATE

Memory-Mapped Files in Java (5)

```java
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MemoryMapReadOnly {
    // Assume the page size is 4 KB
    public static final int PAGE_SIZE = 4096;

    public static void main(String[] args) throws IllegalArgumentException {
        RandomAccessFile inFile = new RandomAccessFile(args[0], "r");
        FileChannel in = inFile.getChannel();
       MappedByteBuffer mappedBuffer = in.map(FileChannel.MapMode.READONLY, 0, inFile.size());
        long numPages = inFile.size() / (long)PAGE_SIZE;
        if (inFile.size() % PAGE_SIZE > 0) ++numPages;
        // we will "touch" the first byte of every page
        int position = 0;
        for (long i = 0; i < numPages; i++) {
            byte item = mappedBuffer.get(position);
            position += PAGE_SIZE;
        }
    }
}
```

Determine the number of pages that file has

Memory-Mapped Files in Java (6)

```java
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MemoryMapReadOnly {
    // Assume the page size is 4 KB
    public static final int PAGE_SIZE = 4096;

    public static void main(String[] args) throws IllegalArgumentException {
        RandomAccessFile inFile = new RandomAccessFile(args[0], "r");
        FileChannel in = inFile.getChannel();
       MappedByteBuffer mappedBuffer = in.map(FileChannel.MapMode.READONLY, 0, inFile.size());
        long numPages = inFile.size() / (long)PAGE_SIZE;
        if (inFile.size() % PAGE_SIZE > 0) ++numPages;
        // we will "touch" the first byte of every page
        int position = 0;
        for (long i = 0; i < numPages; i++) {
            byte item = mappedBuffer.get(position);
            position += PAGE_SIZE;
        }
    }
}
```

Forces the input of every page in file. (demand paging)

Memory-Mapped Files in Java (7)

```java
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MemoryMapReadOnly {
    // Assume the page size is 4 KB
    public static final int PAGE_SIZE = 4096;

    public static void main(String[] args) throws IllegalArgumentException {
        RandomAccessFile inFile = new RandomAccessFile(args[0], "r");
        FileChannel in = inFile.getChannel();
       MappedByteBuffer mappedBuffer = in.map(FileChannel.MapMode.READONLY, 0, inFile.size());
        long numPages = inFile.size() / (long)PAGE_SIZE;
        if (inFile.size() % PAGE_SIZE > 0) ++numPages;
        // we will "touch" the first byte of every page
        int position = 0;
        for (long i = 0; i < numPages; i++) {
            byte item = mappedBuffer.get(position);
            position += PAGE_SIZE;
        }
    }
}
```

The map exists until buffer object is garbage-collected
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

Buddy System Allocator

Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
Slab Allocator

- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Other Issues -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted

Other Issues -- Prepaging

- Assume *s* pages are prepaged and *α* of the pages is used
  - If cost of *s* * α save pages faults > or < than the cost of prepaging *s* * (1- α) unnecessary pages?
  - *α* near zero ⇒ prepaging loses
Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults

Other Issues – TLB Reach

- 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

  \[
  \text{for} \ (j = 0; \ j < 128; \ j++) \\
  \quad \text{for} \ (i = 0; \ i < 128; \ i++) \\
  \quad \quad \text{data}[i,j] = 0;
  \]\n
  \[128 \times 128 = 16,384 \text{ page faults}\]
Other Issues – Program Structure

- Program 2
  
  ```
  for (i = 0; i < 128; i++)
      for (j = 0; j < 128; j++)
          data[i,j] = 0;
  ```
- 128 page faults

Other Issues – I/O interlock

- I/O Interlock – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm

Reason Why Frames Used For I/O Must Be In Memory

Operating System Examples

- Windows XP
- Solaris
Windows XP

- 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.

Windows XP

- 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

Solaris

- 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
Solaris 2 Page Scanner

The diagram shows a graph comparing scan time (y-axis) with memory availability (x-axis). The memory availability is categorized into three levels: minfree, desfree, and lotsfree. The y-axis is marked with scan times: 8192 fastscan and 100 slowscan. The graph illustrates how scan time decreases as the amount of free memory increases from minfree to lotsfree.