Virtual Memory – Outline

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
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

- Benefits of not requiring the entire program in memory before execution
  - programs can be larger than the physical address space
  - more programs can be resident in memory at the same time
  - faster program startup
Virtual-address Space

- Space between heap and stack is not used, unless the heap or stack grows
  - virtual memory does not require physical pages for such *holes*

- Shared memory
  - pages can be easily shared by virtual memory
  - shared libraries can be mapped read-only into address space of each process
  - shared memory IPC can be implemented easily
  - parent pages can be shared with child during *fork()*
Shared Library Using Virtual Memory
Demand Paging

- Bring a page into memory only when it is needed
  - less I/O needed
  - less memory needed
  - faster response
  - more applications simultaneously resident in memory

- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

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

- Each page table entry associated with a valid–invalid bit
  - \( v \rightarrow \text{in-memory}, \ i \rightarrow \text{not-in-memory} \)
  - initially set to invalid for all page table entries

- Example of a page table snapshot:

<table>
<thead>
<tr>
<th>Frame #</th>
<th>valid-invalid bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>v</td>
</tr>
<tr>
<td></td>
<td>i</td>
</tr>
<tr>
<td>....</td>
<td></td>
</tr>
<tr>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>i</td>
</tr>
</tbody>
</table>

- Valid–invalid bit in page table set to \( i \) during address translation causes a page fault
Page Table When Some Pages Are Not in Main Memory
Steps in Handling a Page Fault
Steps in Handling a Page Fault (2)

1. First reference to data or instructions on a page occurs
2. Memory access traps to the OS with a page fault
   - operating system looks at another table to decide:
     - invalid reference ⇒ abort, or just not in memory
3. Find a free frame in physical memory
4. Read page from disk into free frame
5. Update page table valid-invalid bit to 'v'
6. Restart the instruction that caused the page fault
Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - $p = 0$ no page faults
  - $p = 1$, every reference is a fault

- Effective Access Time ($EAT$)

  $$EAT = (1 - p) \times \text{memory access}$$
  $$+ p (\text{page fault overhead})$$
  $$+ \text{swap page out}$$
  $$+ \text{swap page in}$$
  $$+ \text{restart overhead}$$
Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

\[
EAT = (1 - p) \times 200 + p \times 8 \text{ milliseconds}
\]

\[
= (1 - p) \times 200 + p \times 8,000,000
\]

\[
= 200 + p \times 7,999,800
\]

- If one access out of 1,000 causes a page fault, then
  \[
  EAT = 8.2 \text{ microseconds}
  \]
  This is a slowdown by a factor of 40!!
Process Creation – Copy-on-Write

■ Original process creation
  ● fork() system called duplicates parent address space for child
  ● demand paging with copy-on-write allow sharing address spaces

■ Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
  ● shared pages marked as copy-on-write pages
  ● if either process modifies a shared page, then the page is copied
  ● only the modified pages are copied

■ COW allows more efficient process creation as only modified pages are copied

■ Free pages are allocated from a pool of zeroed-out pages
Process 1 Modifies Page C

Before

After
Need For Page Replacement

- Review of demand paging
  - separates logical memory from physical memory
  - allows logical address space > physical address space
  - enables a greater degree of multiprogramming
    - higher CPU utilization and throughput
  - allows fast process startup

- Drawbacks of demand paging
  - may increase later individual memory access times
  - potential for over-allocation of physical memory

- Over-allocation of memory
  - currently active processes may reference more pages than room in physical memory
  - pages from active processes may need to be replaced
Page Replacement

- Page replacement is needed when a page fault occurs, and there are no free frames to allocate
  - terminate user process?
  - swap out an entire process?
  - find some page in memory, hopefully not really in use, and swap it out?

- Same page may be brought into memory several times
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 \textit{victim} frame
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process

- Reducing page fault overhead
  - use \textit{modify bit} to with each frame
  - only copy the replacement page to memory, if modified
Basic Page Replacement (2)
Page Replacement Algorithms

■ Criteria
  ● get the lowest page fault rate

■ Page replacement schemes
  ● first in first out (FIFO)
  ● optimal page replacement
  ● least recently used (LRU)
  ● LRU approximation algorithms
  ● counting based replacement

■ Evaluation of replacement algorithm
  ● simulate it on a particular string of memory page references
  ● compute the number of page faults on the reference string

■ In all our examples, the reference string is:

  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
Expected Graph of Page Faults Versus The Number of Frames
First-In-First-Out (FIFO) Algorithm

- Simplest page replacement algorithm
  - each page associated a time when it was brought in memory
  - replace the oldest page, during page replacement
  - implemented using a FIFO queue of all pages
  - replacement page found from the head of the queue
  - new page added to the tail of the queue

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

- For some page replacement algorithm, the page fault rate may increase as the number of frames increases!
  - can happen with FIFO replacement scheme

Example

- reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- FIFO replacement algorithm

![Graph showing page fault rate vs. number of frames]
Optimal Page Replacement

- Algorithm with the lowest page fault rate
  - replace the page that will not be used for the longest period
  - provably optimal
  - does not suffer from Belady's anomaly
  - needs future information!

Reference string:

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

Page frames:

```
7 7 7 2 2 2 2 2 2 7
0 0 0 0 4 0 0 0 0
1 1 3 3 3 1 1
```
Least Recently Used (LRU) Algorithm

- Approximate the optimal page replacement algorithm
  - detect and store when each page is used
  - replace page that has not been used for the longest period
  - a very good replacement policy
  - requires hardware support to be accurate and fast
  - does not suffer from Belady's anomaly

<table>
<thead>
<tr>
<th>Reference string</th>
<th>Page frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
<td>7 7 7 2 2 4 4 4 0 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 3 3 3 3 3 3 3 3</td>
<td>3 0 0 0 0 0 0 0 2 2 2 2 2 2 2 2</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 7 7 7 7</td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>
LRU Implementation

- Counter implementation
  - every page table entry has a *time-of-use* field
  - copy the clock into the time-of-use field on every page access
  - for replacement, find the page with the smallest time-of-use field
  - replacement algorithm requires search of the full page table
  - each memory access needs additional write to time-of-use field
  - counter overflow?
Stack implementation

- keep a stack of page numbers in a linked list
- move page to top of stack when referenced
- find replacement page from bottom of stack
- each update is more expensive due to stack operations
- no search for replacement

Reference string

<table>
<thead>
<tr>
<th>4</th>
<th>7</th>
<th>0</th>
<th>7</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>2</th>
<th>7</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack before

Stack after

a

b
LRU Approximation Algorithms

■ Reference bit algorithm
  ● associate reference bit with each page
  ● set bit when page is references; clear all bits occasionally
  ● replace page that has reference bit = 0 (if one exists)
  ● cannot store the order of accesses in one bit!

■ Record reference bit algorithm
  ● each page associated with a 8-bit field for recording reference bit
  ● at regular intervals
    ‣ right shift 8-bit field
    ‣ move reference bit into highest order bit
  ● page with lowest value in 8-bit field is the LRU page
  ● Example: (reference bit, 8-bit field)
    ‣ (1, 10101100) → (0, 1101110)
    ‣ (0, 11001101) → (0, 01100110)
**Second-chance algorithm**
- Single reference (RF) bit
- Basic FIFO replacement algo
- Find replacement page
  - Proceed, if RF=0
  - Give page 2\textsuperscript{nd} chance, if RF=1 and move onto next page, clear reference bit
- Circular queue of pages
- Pointer indicates next replacement page
- What is all bits are set?
Counting-Based Algorithms

- Keep a counter of the number of references that have been made to each page.

- Least frequently used
  - replace page with the smallest count
  - (most active page should have largest count)

- Most frequently used
  - replace page with the largest count
  - (page with smallest count was brought in last and will be accessed next)

- Not very popular algorithms
Page Buffering Algorithms

- Optimizations by maintaining a list of **victim** frames

- Scheme 1
  - bring new page into one of victim frame slots
  - then, copy-out the replacement page to memory
  - memory access to new page does not wait for copy-out

- Scheme 2
  - copy-out modified pages to disk when paging device is idle
  - replacement is now faster

- Scheme 3
  - remember which page is in each victim frame
  - if new page is in one of victim frame, then use directly
  - works well if replacement algorithm replaces a page that is needed after afterwards
Allocation of Frames

- How many frames in physical memory should the OS allocate to each process?
- Minimum number of frames
  - is there a minimum number of frames a process needs?
  - depends on the maximum number of pages any instruction in the ISA can reference at a time
    - what can happen if we do not provide this minimum?
  - direct addressing load/store need 2 frames
    - one for instruction, and one for memory reference
  - one-level indirect addressing needs 3 frames
    - one for instruction, one for address, and one for memory reference
  - PDP-11 needs 6 frames, IBM 370 needs 8 frames
  - should there be a maximum number?
Frame Allocation Algorithms

- **Equal allocation**
  - divide $m$ frames equally among $n$ processes
  - each process gets about $m/n$ frames
  - may give some process more frames than it needs!

- **Proportional allocation**
  - each process may need different amounts of memory
  - allocate memory proportional to process size
  - Example:
    \[ s_i = \text{size of process } p_i \]
    \[ S = \sum s_i \]
    \[ m = \text{total number of frames} \]
    \[ a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m \]
    \[ m = 64 \]
    \[ s_i = 10 \]
    \[ s_2 = 127 \]
    \[ a_1 = \frac{10}{137} \times 64 \approx 5 \]
    \[ a_2 = \frac{127}{137} \times 64 \approx 59 \]
Other Frame Allocation Issues

- 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
    - process cannot control its own page-fault rate
  - **local replacement** – each process selects from only its own set of allocated frames
    - may hinder progress by not allowing high-priority process access to frames of low-priority process

- Non-uniform memory access
  - memory may be distributed on multi-processor systems
  - how to assign frames in such situations?
  - assign memory which is close to where a process is running
Thrashing

A process is **thrashing** if it spends more time paging than executing

- process keeps swapping active pages in and out
- leads to a very high page fault rate

**Causes of thrashing behavior**

- process does not have “enough” pages

**Vicious cycle of thrashing**

- not having enough frames cause more page faults
- lowers CPU utilization
- OS thinks that it needs to increase the degree of multiprogramming
- another process added to the system
- even less frames per process leads to more page faults
- cycle continues
Thrashing becomes more likely with increasing degree of multiprogramming

Can we prevent thrashing by using a local replacement algorithm?
Preventing Thrashing Behavior

■ Give a process as many frames as it needs
  ○ how to know how many frames a process needs ?!

■ Working-Set model
  ○ based on the **locality model** of process execution
    ▸ each process phase uses a small set of memory references / frames
    ▸ execution moves from one program phase to another
  ○ the number of frames required for each phase is called the **working set**

■ Implementation
  ▸ assume a particular working-set window, $\Delta$
  ▸ $WSS_i$ (working set of Process $P_i$) = total number of pages referenced in the most recent $\Delta$
  ▸ $D = \sum WSS_i \equiv$ total demand frames
  ▸ If $D > m \Rightarrow$ Thrashing, then suspend one of the processes
Working Sets and Page Fault Rates

- Page faults spike at the start of every new program working set (also called a phase)
  - page fault rate falls until the next program phase
  - behavior repeats
Preventing Thrashing Behavior (2)

- Page-Fault frequency scheme
  - working set model is needs several assumptions and is complicated
  - measure the page fault rate routinely
  - establish “acceptable” page-fault rate
    - If actual rate too low, process loses frame or increase processes
    - If actual rate too high, process gains frame, or suspend processes
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
  - removes the need to read(), write() system calls
  - converts disk access to memory access
  - simplifies disk access for the user

- Mechanism details
  - a file is initially read using demand paging
  - a page-sized portion of the file is read from the file system into a physical frame
  - subsequent reads/writes to/from the file are treated as ordinary memory accesses
Memory Mapped Files (2)

- Allows several processes to map the same file allowing the pages in memory to be shared
Examples of OS using memory-mapped files

- Solaris, Unix, and Linux provide the `mmap()` system call to create a memory-mapped file.
- Files accessed using `open()`, `read()`, `write()` are mapped to kernel address space in Solaris.
- Windows NT and XP use memory mapped files to accomplish shared memory.
Memory-Mapped I/O

- Special I/O instructions may be available to transfer data and control messages to the I/O controller

- Memory-mapped I/O
  - I/O device registers mapped to logical memory address spaces
  - convenient and fast

- There may be a control bit to indicate if data is available in the data register
  - if CPU polls the control bit, then method of operation is called **programmed I/O**
  - if device sends interrupt to CPU to indicate data availability, then mode of operation is **interrupt driven I/O**
Allocating Kernel Memory

- Kernel memory often allocated from a free memory pool
  - not using demand paging

- Reasons for treating kernel memory allocation differently
  - some kernel memory needs to be contiguous
  - attempt to minimize memory waste due to internal fragmentation
    - kernel requests memory for structures of varying size
    - do not use paging system

- Strategies for managing free memory
  - buddy system
  - slab allocator
Buddy System

■ **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

■ **Example:**
  - 21kb requested from 256kb
Slab Allocator

- **Slab** contains several physically contiguous pages.
- **Cache** consists of one or more slabs.
- Single cache for each unique kernel data structure
  - cache filled with instantiations of the data structure (objects)

**Slab allocation algorithm**
- create cache of objects in contiguous space, mark as free
- store objects in free slots, mark as used
- if current slab is full of used objects, allocate next object from empty slab
- if no empty slabs, go to free slots in the next slab

**Benefits**
- no fragmentation
- fast memory request satisfaction
Slab Allocation (2)

- Kernel objects
- Caches
- Slabs

3 KB objects

7 KB objects

Physical contiguous pages
Other Virtual Memory Issues

- **Prepaging**
  - reduce page faults at process startup
  - *prepage* all or some process pages before they are referenced
    - I/O and memory wasted if prepaged pages are unused

- **Issues in deciding a page size**
  - internal fragmentation
    - small page size preferred to reduce memory lost on the final page
  - page table size
    - large page size preferred to reduce size of page table
  - I/O overhead
    - large page size preferred to reduce disk latency and seek time
  - Page faults
    - large page size reduces number of generated page faults
Other Virtual Memory Issues (2)

■ TLB Reach
  ● 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
  ● improving TLB reach
    ▸ increase the page size
    ▸ provide multiple page sizes

■ 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
  ● provide a lock bit with every page frame
Program structure

- being aware of the paging structure can help performance
- example: Int[128, 128] data;
- Each row is stored in one page
- program 1: 128 * 128 = 16,384 page faults
  for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
      data[i, j] = 0;
- Program 2: 128 page faults
  for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
      data[i, j] = 0;