CSE 30341
Operating System Principles

Virtual Memory

Virtual Memory

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

Memory Pages

- Why do we have pages?
- What are advantages & disadvantages of using “page model”?
Background

• Is all of your code used?
  – Error code
  – Unusual routines, certain options/features
  – Large data structures, lists, arrays, ...
• Entire program code not needed at same time!

What if we could only partially load what we need?

Virtual Memory

Separation of user logical memory from physical memory

• Can be implemented via demand paging & demand segmentation
Benefits

• Huge accessible space
  – Map virtual to physical
• Can easily share things
  – Open once, point to the same spot in memory
• Allows for more efficient process creation
  – Only load partial process
• More programs running concurrently
  – Actually physical memory used per process smaller
• Less I/O needed to load or swap processes
  – Some parts of process will never be needed

Virtual Address Space
Virtual Memory >> Physical Memory

Virtual Address Space

- Enables **sparse** address spaces
  - Holes are OK
  - Dynamically linked libraries
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation
Shared Library Using Virtual Memory

Real memory is the same, point to the same thing

Demand Paging

• Load page ONLY WHEN NEEDED:
  – Less I/O needed
  – Less memory needed
  – Faster response
  – More users

• Lazy swapper – never swaps a page into memory unless page will be needed
  – Swapper that deals with pages is a pager
Demand Paging

Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  - v ⇒ in-memory (memory resident)
  - i ⇒ not-in-memory
- Initial valid–invalid bit
  - Set to i on all entries
- During **address translation**, if valid–invalid bit in page table entry is i ⇒ **page fault**
Page Table When Some Pages Are Not in Main Memory

Virtual Memory

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 (kept with PCB) to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables to indicate page now in memory
   Set validation bit = v
5. Restart the instruction that caused the page fault
Aspects of Demand Paging

• Pure Demand Paging
  — Never bring page into memory until needed

• Multiple page faults could be triggered in one instruction (but unlikely due to “locality of reference”)
  — Example: block move (IBM, MVC, move up to 256 bytes)
Stages in Demand Paging

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

- 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)

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

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

- EAT = (1 – p) x 200 + p (8 milliseconds)
  = (1 – p) x 200 + p x 8,000,000
  = 200 + p x 7,999,800

- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!

- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
  - 20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

Process Creation: Copy-on-Write

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

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

- In general, free pages are allocated from a pool of zero-fill-on-demand pages
  - Why zero-out a page before allocating it?

- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call exec()
  - Very efficient
Before Process 1 Modifies Page C

After Process 1 Modifies Page C
What Happens if There is no Free Frame?

• **Page replacement** – find some page in memory, but not really in use, page 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

• 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

Assumption: Have Swap Space to Get It Back
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 page replacement algorithm to select the victim frame
     - Write victim frame to disk if dirty

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Page Replacement Dilemma

- Raw Performance
  - If it is dirty, need to write it to swap space (expensive)
  - Need to write the whole page

- Future Page Faults
  - The one swapped out could be used soon
Page Replacement

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  
  $$7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1$$
Page Faults vs Number of Frames

![Graph showing the relationship between number of page faults and number of frames.](image)

FIFO Page Replacement

3 pages available

7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

<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 0 0 1 1 1 0 0 3 3 3 2 2 2</td>
</tr>
</tbody>
</table>

Do more available pages always improve performance?

Possibly to actually make things worse – Belady’s Anomaly
Belady’s Anomaly

- 1,2,3,4,1,2,5,1,2,3,4,5
- 3 frames (9 page faults) versus 4 frames (10 page faults)

![Graph showing number of page faults versus number of frames]

Optimal Algorithm (OPT)

- Replace page that will not be used for longest period of time
- How do you know this?
- Used for measuring how well your algorithm performs
Op,imal Page Replacement

FIFO

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

page frames
LRU Algorithm

- 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
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of stack algorithms that don’t have Belady’s Anomaly

Stack Approach

reference string

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

stack before a

```
2
1
0
7
4
```

stack after b

```
7
2
1
0
4
```

a b
LRU Approximation Algorithms

- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however

- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - Reference bit = 0 -> replace it
    - Reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules
Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common

- **LFU Algorithm**: replaces page with smallest count

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

Allocation of Frames

- Each process needs *minimum* number of frames
- 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

- **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
  - Dynamic as degree of multiprogramming, process sizes change

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

\[
m = 64 \\
s_1 = 10 \\
s_2 = 127 \\
a_1 = \frac{10}{137} \times 64 \approx 5 \\
a_2 = \frac{127}{137} \times 64 \approx 59
\]
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
  – But then process execution time can vary greatly
  – But greater throughput, so more common

• **Local replacement** – each process selects from only its own set of allocated frames
  – More consistent per-process performance
  – But possibly underutilized memory
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - 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** = a process is busy swapping pages in and out

Thrashing (Cont.)

![Graph showing CPU utilization vs. degree of multiprogramming with a peak indicating thrashing](image-url)
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?
  \[ \sum \text{size of locality} > \text{total memory size} \]
  – Limit effects by using local or priority page replacement
Working-Set Model

- \( \Delta \) = working-set window = a fixed number of page references
  Example: 10,000 instructions
- \( WSS_i \) (working set of Process \( P_i \)) = total number of pages referenced in the most recent \( \Delta \) (varies in time)
  - if \( \Delta \) too small will not encompass entire locality
  - if \( \Delta \) too large will encompass several localities
  - if \( \Delta = \infty \) \( \Rightarrow \) will encompass entire program
- \( D = \sum WSS_i \) = total demand frames
  - Approximation of locality
- if \( D > m \) \( \Rightarrow \) Thrashing
- Policy if \( D > m \), then suspend or swap out one of the processes

Working-Set model

Page reference table

\[
\ldots 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 4 4 1 3 2 3 4 4 3 4 4 4 \ldots
\]

\( \Delta \)

\( f_1 \)

\( \Delta \)

\( f_2 \)

\( WS(t_1) = \{1,2,5,6,7\} \)

\( WS(t_2) = \{3,4\} \)

What are the things that need to be in memory for me to get work done?
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?

- Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency** rate and use local replacement policy
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
- Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared
- But when does written data make it to disk?
  - Periodically and/or at file close() time
  - For example, when the pager scans for dirty pages

Memory-Mapped File Technique for all I/O

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via mmap() system call
  - Now file mapped into process address space
- Memory mapped files can be used for shared memory
Memory Mapped Files

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
    - i.e. for device I/O
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
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into $A_L$ and $A_R$ of 128KB each
    - One further divided into $B_L$ and $B_R$ of 64KB
      - One further into $C_L$ and $C_R$ of 32KB each – one used to satisfy request
- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

Buddy System Allocator

![Buddy System Allocator Diagram](image-url)
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
- 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

---

![Slab Allocator Diagram](image-url)
Other Considerations -- 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
  - Assume $s$ pages are prepaged and $\alpha$ of the pages is used
    - Is cost of $s \alpha$ saved page faults $>$ or $<$ than the cost of prepaging $s (1-\alpha)$ unnecessary pages?
    - $\alpha$ near zero $\Rightarrow$ prepaging loses

Other Issues – Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- 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
- 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
- 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
    
    ```
    for (j = 0; j <128; j++)
      for (i = 0; i < 128; i++)
        data[i,j] = 0;
    ```

    128 x 128 = 16,384 page faults
  - Program 2
    
    ```
    for (i = 0; i < 128; i++)
      for (j = 0; j < 128; j++)
        data[i,j] = 0;
    ```

    128 page faults