Virtual Memory

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

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Program and programs could be larger than physical memory

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
  - 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 > Physical Memory

Virtual-address Space
Virtual Address Space

- Enables sparse address spaces with holes left for growth, dynamically linked libraries, etc
- 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

- stack
- shared library
- heap
- data
- code
- shared pages
- stack
- shared library
- heap
- data
- code
Demand Paging

- Load entire process into memory at load time
- Or 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 $\Rightarrow$ 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

Swapping Pages
Page Table – Some Pages Are Not in Main Memory

Page Fault
Aspects of Demand Paging

- Extreme case – start process with no pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - Same for every other process pages on first access
  - Pure demand paging
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Pain decreased because of locality of reference
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with swap space)
  - Instruction restart

Instruction Restart

- Consider an instruction that could access several different locations
  - block move

  - auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?
**Performance of Demand Paging**

- Stages in Demand Paging
  - Trap to the operating system
  - Save the user registers and process state
  - Determine that the interrupt was a page fault
  - Check that the page reference was legal and determine the location of the page on the disk
- Issue a read from the disk to a free frame:
  - Wait in a queue for this device until the read request is serviced
  - Wait for the device seek and/or latency time
  - Begin the transfer of the page to a free frame
- While waiting, allocate the CPU to some other user
- Receive an interrupt from the disk I/O subsystem (I/O completed)
- Save the registers and process state for the other user
- Determine that the interrupt was from the disk
- Correct the page table and other tables to show page is now in memory
- Wait for the CPU to be allocated to this process again
- 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)
  \[
  EAT = (1 - p) \times \text{memory access} + p (\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})
  \]
Demand Paging Example

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

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

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

- If want performance degradation < 10 percent
  \[
  200 + 7,999,800 \times p < 220
  \]
  \[
  7,999,800 \times p < 20
  \]
  \[
  p < .0000025
  \]
  \[
  < \text{one page fault in every 400,000 memory accesses}
  \]

Demand Paging Optimizations

- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix

- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
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

```
<table>
<thead>
<tr>
<th>process_1</th>
<th>physical memory</th>
<th>process_2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>page A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>page B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>page C</td>
<td></td>
</tr>
</tbody>
</table>
```
After Process 1 Modifies Page C

What Happens if There is no Free Frame?

- Memory used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?

- 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

- 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
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; update the page and frame tables

4. 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

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

Page Faults Vs Number of Frames

![Graph showing the relationship between number of page faults and number of frames.](image)
First-In-First-Out (FIFO) Algorithm

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

<table>
<thead>
<tr>
<th>1</th>
<th>7</th>
<th>2</th>
<th>4</th>
<th>0</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**

- How to track ages of pages?
  - Just use a FIFO queue

FIFO Page Replacement

```
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 4 4 4 0 0 0 7 7 7
0 0 0 3 3 3 2 2 2 1 1 1 1 0 0
1 1 1 1 0 0 0 3 3 3 2 2 2 1
```
**FIFO Illustrating Belady’s Anomaly**

![Graph showing FIFO Illustrating Belady’s Anomaly](image)

**Optimal Algorithm**

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example on the next slide

- How do you know this?
  - Can’t read the future

- Used for measuring how well your algorithm performs
Optimal Page Replacement

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

Reference string:

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| 7 | 7 | 7 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 7 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 |
|   |   | 1 | 1 | 3 | 3 | 3 | 1 | 1 | 1 |   |   |   |   |   |   |   |   |   |   |

Page frames:

- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?
LRU Algorithm (Cont.)

- 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 find smallest value.
    - Search through table needed.
- 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.

Use Of A Stack to Record The Most Recent Page References

<table>
<thead>
<tr>
<th>reference string</th>
<th>4 7 0 7 1 0 1 2 1 2 7 1 2</th>
</tr>
</thead>
</table>

| stack before a   | 2 1 0 7 4             |
| stack after b    | 7 2 1 0 4             |

- Example stack update:
  - a: 7 to top
  - b: 1 to top

CIS 415 Virtual Memory
LRU Approximation Algorithms

- LRU needs special hardware and still slow

**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

Second-Chance (clock) Page-Replacement Algorithm

![Diagram](image-url)
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

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim

- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty

- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected
Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - Raw disk mode
- Bypasses buffering, locking, etc

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

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

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

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
  - 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

Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are NUMA – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
  - And modifying the scheduler to schedule the thread on the same system board when possible
  - Solved by Solaris by creating lgroups
    - Structure to track CPU / Memory low latency groups
    - Used my schedule and pager
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup
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)
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
  
  - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern

![Graph showing locality in a memory-reference pattern](image)
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 $\Delta = \infty$ ⇒ will encompass entire program

- $D = \sum WSS_i$ = total demand frames
  
  - Approximation of locality

- if $D > m$ ⇒ Thrashing

- Policy if $D > m$, then suspend or swap out one of the processes

---

Working-set model

page reference table

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

- $WS(t_1) = \{1,2,5,6,7\}$
- $WS(t_2) = \{3,4\}$

- $\Delta$
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
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
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 use 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
- For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway
  - But map file into kernel address space
  - Process still does `read()` and `write()`
    - Copies data to and from kernel space and user space
  - Uses efficient memory management subsystem
    - Avoids needing separate subsystem
- COW can be used for read/write non-shared pages
- Memory mapped files can be used for shared memory (although again via separate system calls)

Memory Mapped Files

![Diagram of memory mapping](image)
Memory-Mapped Shared Memory in Windows

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 Diagram]
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

Slab Allocation
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 \( s \times \alpha \) save pages faults > or < than the cost of prepaging \( s \times (1 - \alpha) \) 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
  - Resolution
  - 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)
- On average, growing over time
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
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**

[Diagram showing buffer and disk drive connected with an arrow]