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

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 That is Larger Than 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
Demand Paging

- Could bring 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 ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

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

Transfer of a Paged Memory to Contiguous Disk Space
Valid-Invalid Bit
- With each page table entry a valid–invalid bit is associated
  \( v \Rightarrow \text{in-memory} \) – \( i \Rightarrow \text{not-in-memory} \)
- Initially valid–invalid bit is set to \( i \) on all 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>( i )</td>
</tr>
<tr>
<td></td>
<td>( v )</td>
</tr>
<tr>
<td></td>
<td>( i )</td>
</tr>
</tbody>
</table>

During address translation, if valid–invalid bit in page table entry is \( i \Rightarrow \text{page fault} \)

Page Table When Some Pages Are Not in Main 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 to decide:
     - Invalid reference \( \Rightarrow \text{abort} \)
     - Just not in memory
  2. Get empty frame
  3. Swap page into frame via scheduled disk operation
  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

- **Extreme case** – start process with no pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And 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**)
Performance of Demand Paging

- 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:
     a. Wait in a queue for this device until the read request is serviced
     b. Wait for the device seek and/or latency time
     c. Begin the transfer of the page to a free frame
     d. While waiting, allocate the CPU to some other user
  6. Receive an interrupt from the disk I/O subsystem (I/O completed)
  7. Save the registers and process state for the other user
  8. Determine that the interrupt was from the disk
  9. Correct the page table and other tables to show page is now in memory
  10. Wait for the CPU to be allocated to this process again
  11. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging (Cont.)

- 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
  \[
  EAT = (1 - p) \times 200 + p \times (8 \text{ milliseconds}) = (1 - p \times 200 + p \times 8,000,000) = 200 + p \times 7,999,800
  \]
- If one access out of 1,000 causes a page fault, then EAT is 8.2 us
  - 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
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
After Process 1 Modifies Page C

What Happens if There is no Free Frame?

- 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 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
    \[7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1\]

![Graph of Page Faults Versus The Number of Frames](image)

**First-In-First-Out (FIFO) Algorithm**

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

- 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 2 1 2 0 1 7 0 1
page frames

FIFO Illustrating Belady’s Anomaly

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

LRU Approximation Algorithms

- True 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
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

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.)

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?

  $\sum$ size of locality $>$ total memory size
  - Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern
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 $\Delta \to \infty$ ⇒ will encompass entire program

- $D = \Sigma WSS_i$ = total demand frames
  - Approximation of locality
  - if $D > m$ ⇒ Thrashing
  - Policy if $D > m$, then suspend or swap out one of the processes

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 ⇒ page in working set

- Why is this not completely accurate?

- Improvement = 10 bits and interrupt every 1000 time units
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).