Memory Management

• 3 issues to consider
  ▪ More than 1 process can’t own all physical memory same time
  ▪ Processes shouldn’t be allowed to read/write memory of another
    • Unless explicitly allowed (remember IPC?)
  ▪ Process may contain more data than physical memory can store

• OS must manage memory to address these issues
• Recall a process’s memory...

![Diagram of process memory areas: Stack, Heap, Static Data, Code]
In A Nutshell – Virtual Memory

• Mechanism
  ▪ Programs & processes reference “virtual” addresses
    • Cannot directly access physical memory addresses
  ▪ OS converts process virtual address into physical mem address
    • For every memory access from a process

• Implications
  ▪ Multiple processes can co-reside in memory at once
    • OS ensures that their addresses will not conflict
  ▪ OS is able to enforce memory protection
    • Since it will observe & translate address for every memory reference
      ▪ Allows running a program that is larger than physical memory

• Now let’s look at the details
Address Spaces

• Logical (Virtual) Address Space
  ▪ This is what a program sees (0x00...0 to 0xFF...F)
    ▪ Size depends on the CPU architecture (e.g. 32-bit vs. 64-bit)
    ▪ Compile + link + load generates logical addresses

• Physical Address Space
  ▪ This is what the main memory (and cache hierarchy) sees

• Memory Management Unit (MMU)
  ▪ Hardware that converts logical to physical addresses
  ▪ OS interacts w/ MMU
Process Memory Allocation

- **Programs can access memory**
  - Instructions are fetched from memory
  - Load and store operations address memory
  - No direct interface to disk
- **So programs & data must be in mem**
  - Executing processes allocated in memory
- **Possible solution:**
  - Load whole programs into main memory
  - Store base mem address of each program
    - “Base Register”
  - Store size of process for each program
    - “Limit Register”
  - CPU address generated by a process is:
    - Compared to limit register (for protection)
    - Added to base register to get physical addr
- **Any problems with this?**
Issues with Simple Solution

• How to decide where to load a new process in memory?
  ▪ Similar problem to:
    • Dynamic memory management (malloc / free)
    • Allocation of disk blocks
  ▪ Can use an allocation policy like: first fit, best fit, worst fit

• Fragmentation is a big problem
  ▪ 50-percent rule:
    • Statistical analysis shows that for N allocated blocks, 0.5N lost
    • 1/3 of memory is unusable
  ▪ This would be very costly for main memory
Paging

• Use non-contiguous physical memory for a process
  ▪ Avoids problem of external fragmentation
  ▪ Analogous to some disk block allocation schemes we studied

• Mechanism
  ▪ Physical memory divided into fixed-size blocks (frames)
  ▪ Virtual address space also divided into same-size blocks (pages)
  ▪ Addresses generated by the processor have 2 parts:
    • Page number
    • Page offset (byte address within the page)
  ▪ Typically page / frame size is a power of 2 (e.g. 4KB)
    • A range of page sizes may be supported (hugepages)
  ▪ A table stores the physical frame number for each virtual page
    • Called the page table
    • Per-process data structure (need one per virtual address space)
Paging Mechanism

Assume: 4KB pages, 32-bit virtual address space
Then: 12 page offset bits, 20 bits for page number

CPU Executing Process A

Id $r1, 0x00...02080

page offset

page number

0 0x3000
1 0x1000
2 0x6000
3 0x4000

Process A Page Table

Disk

Move pages between memory frames and disk

Assume: 4KB pages, 32-bit virtual address space
Then: 12 page offset bits, 20 bits for page number
More on Paging

• Internal fragmentation is still possible
  ▪ Some pages from address space do not use a full frame

• Page table entry size
  ▪ If 4 bytes w/ 4KB pages, then can map a $2^{32} \times 4$KB memory

• Address translation mechanism also provides protection
  ▪ User process generates virtual addresses in its address space
  ▪ OS + hardware translate to physical addresses
  ▪ Thus, no mechanism to access memory of another process
    • Except when explicitly enabled (IPC)

• OS tracks physical memory frames in a frame table
  ▪ Which are free and which are allocated (to which page / process)
A Microarchitecture Note

• Every load, store, i-fetch requires 2 mem accesses?
  ▪ One for page table entry
  ▪ One for the physical address translated using this entry?

• Hardware caches page table entries in TLB
  ▪ Translation lookaside buffer: often a fully associative cache
  ▪ Cache of virtual to physical page translation
  ▪ A TLB miss for a virtual address requires a separate mem access
  ▪ TLB often stores process ID owning the page address
    • Extends the protection mechanism for process address space isolation
    • TLB hit also compares the process ID to the running process ID
      ▪ Treated as a TLB miss if mismatch
    • Allows the TLB to service multiple processes at once
Protection

• Extra bits in the PT entry denote access rights for a page
  ▪ read-only
  ▪ read-write,
  ▪ execute only

• Compared against the operation causing the access
  ▪ Load
  ▪ Store
  ▪ Instruction fetch
Page Table

• As described so far, we have assumed some things
  ▪ Flat table with virtual-to-physical page translations
  ▪ Page table for a process exists at a fixed physical memory location

• We do have a problem with size
  ▪ Assume:
    • A 32 or 64 bit virtual address space
    • 4 byte page table entries
    • 4096 byte pages
  ▪ For 32-bit address space: \(\frac{2^{32}}{2^{12}} \times 4B = 4MB\)
  ▪ For 64-bit address space: \(\frac{2^{64}}{2^{12}} \times 4B = 2^{54}\) bytes

• Clearly we don’t want to (for 32b) or cannot (for 64b) use a dedicated, contiguous region of main memory for the PT
Hierarchical Page Table

• One solution: page the page table
  ▪ Add another level of indirection

• Split the page table entries up into groups
  ▪ The size of a page or physical memory frame
  ▪ With 4K pages & 4B PT entries, we have $2^{10}$ entries per group
  ▪ Allow these pages of PT entries to be swapped to/from disk
    ▪ i.e., paged, just like regular process data
  ▪ How do we find a PT entry in physical memory now?

• A top-level page table used to page the page table

Example: 32-bit address space, 4KB pages, 4B per PT entry

<table>
<thead>
<tr>
<th>index into top-level PT</th>
<th>index into PT page</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 bits</td>
<td>10 bits</td>
<td>12 bits</td>
</tr>
</tbody>
</table>
Hierarchical Page Table (2)

• What about 64-bit address spaces?
  ▪ Even with 2-level page table, outer page table has $2^{44}$ bytes
  ▪ Try 3 levels of page tables:
    • 12 bits of page offset
    • 10 bits for 1\textsuperscript{st} level PT index
    • 10 bits for 2\textsuperscript{nd} level PT index
    • 32 bits for top-level PT index
  ▪ Still requires $2^{34}$ bytes contiguously allocated in memory
• We need alternative solutions
Inverted Page Table

• The page table we’ve discussed...
  ▪ Has an entry per logical page of every active process
  ▪ Each entry stores the physical page frame for the logical page
    • Or is invalid, meaning the page is not in memory
    ▪ This is an intuitive way to do mapping, but requires huge space

• To solve the size problem, we can invert the page table
  ▪ One entry per physical page frame in memory
  ▪ Each entry stores the logical page it stores (and the PID)
  ▪ Finding a virtual-to-physical page mapping requires searching
    • Possibly every entry in the inverted page table!
  ▪ Used by 64b UltraSPARC and Power Architectures
Inverted Page Table Mechanism

1. CPU Executing Process A
   - Ld $r1, 0x00...05080

2. PID+ page number

3. Index of hit in inverted PT indicates the physical frame #
   - 0x00...02080

4. Move pages between memory frames and disk

- Page 0
- Page 1
- Page 2
- Page 3

CPU

Physical Memory

Disk
Inverted Page Table – Search Time

• Searching the entire inverted table would take too long
  ▪ Recall, it would only be searched on a TLB miss

• To solve this, inverted page tables are often hashed
  ▪ Hash input is the logical page number
  ▪ Each hash table entry is a list of physical frames
    • Contain the frame number, PID, pointer to next in list
  ▪ Reduces the TLB miss to (hopefully) only a few extra memory accesses to search the hashed entry in the inverted page table
Benefits of Virtual Memory

- Programs can be larger than the physical memory
- Multiple programs can execute at once
- Reduced I/O: only used portions of a program are loaded
- System libraries may be shared by many processes
  - A library is loaded into one location in memory
  - Virtual pages of multiple processes can map to this space
- Allows processes to share memory
  - Via mapping virtual pages to a single physical page frame
- Provides protection across process address spaces
More on Paging

- We’ve discussed the basic of paging
  - Fixed-sized chunks of data move between disk & mem
  - There are ways to convert virtual addresses to physical
    - Address translation via page tables
- Next we’ll discuss movement of pages between mem & disk
  - This is a management function of the OS
  - Demand paging brings pages into memory as needed
Overview

• Case #1: TLB hit
  ▪ Common case; VA to PA translation is complete quickly by CPU
  ▪ Note many CPUs have multi-level TLB cache hierarchy

• Case #2: TLB miss, Page Table hit
  ▪ Need to access the software page table structures
  ▪ Can be done by HW (page table walker) or SW (trap to OS)
  ▪ PT entry valid bit is set; load entry into TLB and use for translation

• Case #3: TLB miss, Page Table miss (called a Page Fault)
  ▪ Causes trap into OS (if not already there for SW TLB miss handler machines)
  ▪ OS initiates disk operation(s) to retrieve page
  ▪ OS identifies a physical memory frame to store the page
    • Either a free frame (e.g. via a free-frame list)
    • Or must pick valid frame to “evict” from memory and write back to disk
  ▪ OS modifies the page table entry: sets valid bit, PID field, physical frame
  ▪ OS returns from the trap (like a return from interrupt) to re-execute the instruction

• Work through example of access time...
Page Replacement

• As processes execute and access pages...
  ▪ Physical memory frames may fill
  ▪ New pages required by a process will need a free frame
  ▪ Must replace (evict) an existing page from a frame to make room

• Optimizations
  ▪ Keep a dirty bit in hardware with each memory frame
    • Only write page back to disk if it is dirty
    • Otherwise, can simply overwrite the frame with new page data
    • Significantly reduces disk I/O
  ▪ Write evicted, dirty page back to disk **swap space**
    • Will discuss swap space more in a bit

• Need to decide on a page replacement algorithm
  ▪ Many choices
FIFO Page Replacement

• Replace page brought into memory furthest in past
  ▪ Could keep a timestamp of load time of each frame
  ▪ Or could keep a separate FIFO queue of page frame numbers

• Easy to implement, but performance is often poor
  ▪ Some pages may have been allocated long ago, but used often

• We can evaluate page replacement algorithms by:
  1. Assuming some fixed number of memory page frames
  2. Simulate page replacement decisions for some sequence of page accesses (only concerned with the page number)
     • Metric of interest is page faults

• Belady’s Anomaly
  ▪ For some algorithms, page faults may *increase* with more frames
Evaluating Page Replacement

• Process
  ▪ Assume some number of memory page frames
  ▪ Simulate replacements for an access sequence of page numbers

• Example:
  ▪ 4 page frames
  ▪ Page accesses: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

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

10 page misses out of 20 accesses
FIFO Page Replacement Example

• Page number reference sequence:
  ▪ 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

• 1 page frame:
  ▪ 12 page faults (all 12 references)

• 2 page frames:
  ▪ 12 page faults

• 3 page frames:
  ▪ 9 page faults

• 4 page frames:
  ▪ 10 page faults (more than w/ 3 page frames! Belady’s Anomaly)

• 5 page frames:
  ▪ 5 page faults
Optimal (OPT) Page Replacement

• Replace the page that will not be accessed until furthest in the future
  ▪ Produces guaranteed minimal page faults
  ▪ Of course...cannot be implemented

• Serves a good point of comparison for other algos
  ▪ What is the # of page faults relative to OPT
LRU Page Replacement

- Approximate OPT by looking backwards
- Replace page that has not been accessed in longest time
- Generally gives low page faults; often implemented
- Easy to understand, but not to implement
  - Expensive per-frame timestamp counters & comparisons
  - Separate stack of page frames; page access moves it to top
- Not susceptible to Belady’s Anomaly
  - Type of stack algorithm
  - Set of pages in memory with N frames is always a subset of the pages that would be in memory with N+1 frames
Some LRU Approximations

• **Reference Bits Algorithm**
  - Associate some # of bits with each page frame (e.g. 8)
  - When a page is accessed, set its leftmost bit to 1
  - Periodically, the OS will shift bits of all pages to the right by 1
  - Page frame with the lowest value is the LRU page

• **Second Chance Algorithm**
  - Keep a single reference bit per page frame (set on page access)
  - Use FIFO replacement, but if reference bit is set, don’t replace it
    - Instead move on to look at second FIFO page
    - Also clear FIFO page reference bit & set arrival time to current time
    - Continue this until a page frame with a 0 reference bit is found
Swap Space

- Dedicated portion of the hard disk
- Not part of the file system
  - No regular files can be stored there
  - Raw device format
- Often used for storing pages moved to/from memory
- May be faster or more efficient than file system portion