Outline

- Simple memory management: swap etc.
- Virtual memory and paging
  - Page table and address translation
  - Translation lookaside buffer (TLB)
- Multi-level page table
- Page replacement algorithms and modeling
- Working set of processes

Segmentation-base memory management

Protection of memory: using base and limit registers

Swapping

Consider a multi-programming environment:
- Each program must be in the memory to be executed
- Processes come into memory and
- Leave memory when execution is completed

Many Questions:
1. How to generate the addresses for a process?
2. How to assign physical memory?

Reject if! But if you want to support more processes,
Swap out an old process to a disk
(reading for long I/O, quantum expired etc)

What if no free region is big enough?
Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.
  - **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.
  - **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out to allow higher-priority process to be loaded and executed.
- Swapping would be needed to free up memory for additional processes.

Virtual Memory

- **Basic idea:** allow OS to allocate more memory than the real.
- **Program uses virtual addresses**
  - Addresses local to the process
  - Can be any size → limited by # of bits in address (32/64)
  - 32 bits: 4G
  - 64 bits:
- **Virtual memory >> physical memory**

Motivations for Virtual Memory

- **Use physical DRAM as cache for the disk**
  - Virtual pages of processes can exceed physical memory size
- **Simplify memory management**
  - Multiple processes resident in main memory
  - Each with its own address space
  - Only “active” code and data is actually in memory
- **Provide protection**
  - One process can’t interfere with another
    - Because they operate in different address spaces
  - User process cannot access privileged information
    - Different sections of address spaces have different permissions

Virtual and Physical Addresses

- **Virtual address space**
  - Determined by instruction width
  - Same for all processes
- **Physical memory indexed by physical addresses**
  - Limited by bus size (# of bits)
  - Amount of available memory
- **Paging**
  - A memory-management scheme that permits address space of process to be non-continuous.
Paging and Page Systems

- Virtual address
  - Divided into pages
- Physical memory
  - Divided into frames
- Page vs. Frame
  - Same size address block
  - Unit of mapping/allocation
- A page is mapped to a frame
  - All addresses in the same virtual page are in the same physical frame
  - offset in a page

An Example of Virtual/Physical Addresses

- Example:
  - 64 KB virtual memory
  - 32 KB physical memory
  - 4 KB page/frame size → 12 bits as offset (d)

Page Table

- Each process has one page table
- Map page number → physical page number
- Number of PTEs in page table
- Number of total pages in virtual space
- Not just the pages in use, why?
- Page table is checked for every address translation
- Where to store page table?
- Not all pages need map to frames at the same time
- Not all physical frame need be used

More on Page Table

- Different processes have different page tables
  - CR3 points to the page table
  - Change CR3 registers when context switches
- Page table resides in main (physical) memory
  - Continuous memory segment

Why?
**How big the page table is?**

- **Page Table Size**
  - Modern Systems/Applications
    - 32 bits virtual address
    - System with 1GB physical memory \(\rightarrow\) 30 bits physical address
    - Suppose the size of one page/frame is 4KB (12 bits)
  - Page table size
    - \# of virtual pages: \(32 - 12 = 20\) bits \(\rightarrow\) \(2^{20}\) PTEs
    - Page table size = PTE size \(\times\) \(2^{10}\) = 4 MB per process \(\rightarrow\) \(2^{10}\) frames
  - If there are 128 processes
    - Page tables occupy 128 \(\times\) 4MB = 512 MB
    - 50% of memory will be used by page tables?

- **How can we get smaller page table?!**

**Two-Level Page Tables**

- **Solution:** multi-level page tables
- Virtual address: three parts
  - Level-one page number (10 bits)
  - Level-two page number (10 bits)
  - Offset (12 bits)
- PTE in 1st level page table contains physical frame # for one
  2nd level page table
- 2nd level page table has actual physical frame numbers for the
  memory address

- **Why it is good?**
  - We don’t have to allocate all levels initially
  - They don’t have to be continuous
Example: 2-level Address Translation

- Page number
- Page offset
- p1 = 10 bits
- p2 = 10 bits
- Offset = 12 bits
- Frame number
- Physical address
- P0
- P1
- P2
- Main memory

Which tables should be in memory?

Memory Requirement of Page Tables

- Only the 1st level page table and the required 2nd level page tables need to be in memory

Example: a process access 32 MB (recall 32 bits virtual address, using 10-10-12), what is the minimum and maximum memory for the page table?

- 4KB / page → process has 8K (8 * 2^13) virtual pages
- One 2nd level page table maps 2^12 pages;
- Number (minimum) of 2nd level page table needed:
  \[ 8 = \frac{8 * 2^{10}}{2^{10}} = 8 \]
- Total (minimum) memory for page table: 1st level page table + 8; in total of 9 page tables → 9 x 4KB = 36 KB

Memory Requirement of Page Tables

- Only the 1st level page table and the required 2nd level page tables need to be in memory

Example: a process access 32 MB (recall 32 bits virtual address, using 10-10-12), what is the maximum memory for the page table?

- 4KB / page → process has 8K virtual pages
- 32MB = 8K pages, and the total number of 2nd level pages 2^10
- Thus, 8K pages could cover all of these entries.
- Total maximum memory for page table: 1st level page table + 1024 pages for 2nd level; → 1025 * 4KB
Linux’s 3 level page table

Linear Address converted to physical address using 3 levels

Index into Page Dir.  Index into Page Middle Dir.  Index into Page Table  Page Offset

What is the benefit to use 3-level page table?
What is the shortcoming?

Size of Page/Frame: How Big?

- Determined by number of bits in offset (12Bit→4KB)
- Smaller pages have advantages
  - Less internal fragmentation
  - Better fit for various data structures, code sections
- Larger pages are better because
  - Less overhead to keep track of them
  - More efficient to transfer larger pages to and from disk
- One principle: page table → fit into one frame
  - 32bits machine, 10 bits for each level

How can we make the address translation faster?

Integrating VM and Cache

- Most caches “physically addressed”
  - Accessed by physical addresses
  - Allows multiple processes to have blocks in cache at same time else context switch == cache flush
  - Allows multiple processes to share pages
  - Cache doesn’t need to be concerned with protection issues
  - Access rights checked as part of address translation
- Perform address translation before cache lookup
  - Could involve memory access itself (to get PTE)
  - No page table entries can also be cached

Translation Lookaside Buffer (TLB)

- Small Hardware: fast
  - Store recent accessed mapping of page → frame (64 ~ 1024 entries)
  - If desired logical page number is found, get frame number from TLB
  - If not, get frame number from page table in memory
  - Use standard cache techniques
  - Replace an entry in the TLB with the logical & physical page numbers from this reference
  - Contains complete page table entries for small number of pages

<table>
<thead>
<tr>
<th>Logical page #</th>
<th>Physical frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>29</td>
<td>6</td>
</tr>
<tr>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>

Example TLB
Address Translation with TLB

Integrating TLB and Cache

What happens when cpu performs a context switch?

Fragmentation

- **External Fragmentation**
  - total memory space exists to satisfy a request, but it is not contiguous

- **Internal Fragmentation**
  - allocated memory larger than requested; this size difference is called internal partition.

- **How can we reduce external fragmentation**
  - **Compaction**: migrate memory contents to place all free memory together in one large block
  - Compaction is possible only if relocation is dynamic, and is done at execution time

Designing of 2 level page table

1. Determine number of bits of page offset (12Bit→4KB)
2. Determine the number of bits for the page table entry
3. Determine the number of bits for 2nd level page table
   - **One principle**: page table → fit into one frame
4. Determine the number of bits for 1st level page table
Designing of Multi-level Page Table

Suppose that a system has a 24-bit logical address space and is byte-addressable. The amount of physical memory is 1MB (i.e., the physical address has 20 bits) and the size of a page/frame is 1K bytes. How to design a two-level page table?

1. The size of page table entry will be 2 bytes (larger than 13 bits)
2. One page can hold 512 entries (1K/2=512)
3. Thus, we need 9 bits for the 2nd level page table

---

Designing of Multi-level Page Table

Suppose that a system has a 24-bit logical address space and is byte-addressable. The amount of physical memory is 1MB (i.e., the physical address has 20 bits) and the size of a page/frame is 1K bytes. What about this design?

- 4 bits
- 10 bits
- 10 bits

Second level 10 bits, can’t be fitted into one page. Then we may need to have multiple pages (2 pages) that are continuous, which can’t guarantee or increase the complexity of OS design.

---

Designing of Multi-level Page Table (2)

Suppose that a system has a 16-bit logical address space and is byte-addressable. The amount of physical memory is 64KB (i.e., the physical address has 16 bits) and the size of a page/frame is 256 bytes. How to design a two-level page table?

1. The size of page table entry will be 2 bytes (larger than 11 bits)
2. One page can hold 256 entries (256/2=128)
3. Thus, we need 7 bits for the 2nd level page table

---

Memory Accesses Time

Assuming:
- TLB lookup time = a
- Memory access time = m

Hit ratio (h) is percentage of time that a logical page number is found in the TLB

More TLB entries usually means higher h

Effective Access Time (EAT) is calculated (don’t include cache effect)

EAT = (m + a)h + (m + m + a(1-h)) = a + (2-h)m

Interpretation
- Reference always requires TLB lookup, 1 memory access
- TLB misses also require TLB lookup and 1 memory reference
**Performance of Demand Paging**

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

- Effective Access Time (EAT)
  $$EAT = (1 - p) \times memory\_access + p \times page\_fault\_time$$

- $page\_fault\_time$ depends on several factors:
  - Save user reg and proc state, check page ref, read from the disk, there might be a queue, (CPU can be given to another proc), get interrupt, save other user reg and proc state, correct the page table, put this process into ready queue.... Due to queues, the $page\_fault\_time$ is a random variable.

---

**Demand Paging Example**

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

$$EAT = (1 - p) \times 200 + p \times 8,000,000$$
$$= 200 + p \times 7,999,800$$

- If one out of 1,000 access causes a page fault, then EAT = 8.2 microseconds. This is a slowdown by a factor of 40!!
- If we want just 10% performance degradation, then $p$ should be 220 > $(1 - p) \times 200 + p \times 8$ milliseconds
  $$p < 0.0000025$$, i.e., 1 page fault out of 400,000 accesses.

---

**Thrashing**

- If a process does not have “enough” pages, the page-fault rate is very high.
- E.g., a process needs 6 pages, but only have 5 frames. Thus it will evict a page from existing 5 pages.

Frequent faults:
- This leads to:
  - low CPU utilization
  - OS increase the degree of multiprogramming
  - another process added to the system, worse case

---

**Page Replacement Algorithms**

- How to select the victim frame?
  - You can select any frame, the page replacement will work; but the performance???
  - Gives the lowest page-fault rate

- Evaluate an algorithm by running it on a particular string of memory references (reference string) and compute the number of page faults on that string.

In all our examples, we will have 3 frames and the following reference string:

```
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
```
First-In-First-Out (FIFO) Algorithm

- Maintain an FIFO buffer
  - The page used before may not be needed
  - An array used early, might be used again and again
- Easy to implement
- Belady’s Anomaly: more frames ⇒ more page faults

Easy to implement
Belady’s Anomaly: more frames ⇒ more page faults

Optimal Algorithm

- Replace a page that will not be used for longest time

Least Recently Used (LRU) Algorithm

- Use recent past as an approximation of the future
- Select the page that is not used for a long time…
  - OPT if you look at from backward
  - NO Belady’s Anomaly: so more frames ⇒ less page faults

FIFO Illustrating Belady’s Anomaly

Reference string (12 accesses)
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
Summary: Page Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out useful pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but hard to implement exactly</td>
</tr>
<tr>
<td>OPT (Optimal)</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
</tbody>
</table>

Buddy System (Dividing)

Free Page’s List

Page Allocation

Two continuous blocks with the same size the first one will start as $2^n$
Buddy De-Allocation

- When blocks of page frames are released the kernel tries to merge pairs of "buddy" blocks of size $b$ into blocks of size $2b$.
- Two blocks are buddies if:
  - They have equal size $b$.
  - They are located at contiguous physical addresses.
  - The address of the first page frame in the first block is aligned on a multiple of $2b^2$ (starting at $2b$ page).
- The process repeats by attempting to merge buddies of size $2b$, $4b$, $8b$ etc...

Slab Allocator

- Performs the following functions
  - Allocate memory
  - Initialize objects/structures
  - Use objects/structures
  - Deconstruct objects/structures
  - Free memory
- `/proc/slabinfo` – gives full information about memory usage on the slab level. (see also `/usr/bin/slabtop`)

Slab

- Slab is one or more physically contiguous pages
- Cache consists of one or more slabs
- Single cache for each unique kernel data structure (process descriptions, file objects, semaphores)
  - 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, next object is allocated from empty slab, if no empty slab, new slab allocated
- Benefits include
  - No fragmentation,
  - Memory request is satisfied quickly

Important Topics

- Address Translation
- Designing Multi-Level Page Table
- Page Table Size
- Memory Access Time
- Page Replacement
- Buddy Allocator

Homework and Project