Chapter 4: Memory Management

Part 1: Mechanisms for Managing Memory
Memory management

- Basic memory management
- Swapping
- Virtual memory
- Page replacement algorithms
- Modeling page replacement algorithms
- Design issues for paging systems
- Implementation issues
- Segmentation
In an ideal world…

- The ideal world has memory that is:
  - Very large
  - Very fast
  - Non-volatile (doesn’t go away when power is turned off)

- The real world has memory that is:
  - Very large
  - Very fast
  - Affordable!

⇒ Pick any two…

- Memory management goal: make the real world look as much like the ideal world as possible
Memory hierarchy

What is the memory hierarchy?
- Different levels of memory
- Some are small & fast
- Others are large & slow

What levels are usually included?
- Cache: small amount of fast, expensive memory
  - L1 (level 1) cache: usually on the CPU chip
  - L2 & L3 cache: off-chip, made of SRAM
- Main memory: medium-speed, medium price memory (DRAM)
- Disk: many gigabytes of slow, cheap, non-volatile storage

Memory manager handles the memory hierarchy
Basic memory management

- Components include
  - Operating system (perhaps with device drivers)
  - Single process
- Goal: lay these out in memory
  - Memory protection may not be an issue (only one program)
  - Flexibility may still be useful (allow OS changes, etc.)
- No swapping or paging
Fixed partitions: multiple programs

- Fixed memory partitions
  - Divide memory into fixed spaces
  - Assign a process to a space when it’s free
- Mechanisms
  - Separate input queues for each partition
  - Single input queue: better ability to optimize CPU usage
How many programs is enough?

- Several memory partitions (fixed or variable size)
- Lots of processes wanting to use the CPU
- Tradeoff
  - More processes utilize the CPU better
  - Fewer processes use less memory (cheaper!)
- How many processes do we need to keep the CPU fully utilized?
  - This will help determine how much memory we need
  - Is this still relevant with memory costing less than $1/GB?
Modeling multiprogramming

- More I/O wait means less processor utilization
  - At 20% I/O wait, 3–4 processes fully utilize CPU
  - At 80% I/O wait, even 10 processes aren’t enough
- This means that the OS should have more processes if they’re I/O bound
- More processes => memory management & protection more important!
Multiprogrammed system performance

- Arrival and work requirements of 4 jobs
- CPU utilization for 1–4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - Numbers show amount of CPU time jobs get in each interval
  - More processes => better utilization, less time per process

<table>
<thead>
<tr>
<th>Job</th>
<th>Arrival time</th>
<th>CPU needed</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10:00</td>
<td>4</td>
<td>0.80</td>
<td>0.64</td>
<td>0.51</td>
<td>0.41</td>
</tr>
<tr>
<td>2</td>
<td>10:10</td>
<td>3</td>
<td>0.20</td>
<td>0.36</td>
<td>0.49</td>
<td>0.59</td>
</tr>
<tr>
<td>3</td>
<td>10:15</td>
<td>2</td>
<td>0.20</td>
<td>0.18</td>
<td>0.16</td>
<td>0.15</td>
</tr>
<tr>
<td>4</td>
<td>10:20</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 10 15 20 22 27.6 28.2 31.7

Time

1
2
3
4
Memory and multiprogramming

- Memory needs two things for multiprogramming
  - Relocation
  - Protection
- The OS cannot be certain where a program will be loaded in memory
  - Variables and procedures can’t use absolute locations in memory
  - Several ways to guarantee this
- The OS must keep processes’ memory separate
  - Protect a process from other processes reading or modifying its own memory
  - Protect a process from modifying its own memory in undesirable ways (such as writing to program code)
**Base and limit registers**

- **Special CPU registers: base & limit**
  - Access to the registers limited to system mode
  - Registers contain
    - Base: start of the process’s memory partition
    - Limit: length of the process’s memory partition

- **Address generation**
  - Physical address: location in actual memory
  - Logical address: location from the process’s point of view
  - Physical address = base + logical address
  - Logical address larger than limit => error

---

Logical address: 0x1204
Physical address: 0x1204 + 0x9000 = 0xa204
Swapping

- Memory allocation changes as
  - Processes come into memory
  - Processes leave memory
    - Swapped to disk
    - Complete execution
- Gray regions are unused memory
Swapping: leaving room to grow

- Need to allow for programs to grow
  - Allocate more memory for data
  - Larger stack
- Handled by allocating more space than is necessary at the start
  - Inefficient: wastes memory that’s not currently in use
  - What if the process requests too much memory?
Tracking memory usage: bitmaps

- Keep track of free / allocated memory regions with a bitmap
  - One bit in map corresponds to a fixed-size region of memory
  - Bitmap is a constant size for a given amount of memory regardless of how much is allocated at a particular time

- Chunk size determines efficiency
  - At 1 bit per 4KB chunk, we need just 256 bits (32 bytes) per MB of memory
  - For smaller chunks, we need more memory for the bitmap
  - Can be difficult to find large contiguous free areas in bitmap

![Bitmap Diagram]

```
11111100
00111000
01111111
11110100
```
Tracking memory usage: linked lists

- Keep track of free / allocated memory regions with a linked list
  - Each entry in the list corresponds to a contiguous region of memory
  - Entry can indicate either allocated or free (and, optionally, owning process)
  - May have separate lists for free and allocated areas
- Efficient if chunks are large
  - Fixed-size representation for each region
  - More regions => more space needed for free lists

![Memory regions diagram]

- A: 0-6
- B: 10-3
- C: 17-9
- D: 26-3
Allocating memory

- Search through region list to find a large enough space
- Suppose there are several choices: which one to use?
  - First fit: the first suitable hole on the list
  - Next fit: the first suitable after the previously allocated hole
  - Best fit: the smallest hole that is larger than the desired region (wastes least space?)
  - Worst fit: the largest available hole (leaves largest fragment)
- Option: maintain separate queues for different-size holes

Allocate 20 blocks first fit
Allocate 13 blocks best fit
Allocate 12 blocks next fit
Allocate 15 blocks worst fit
Freeing memory

- Allocation structures must be updated when memory is freed.
- Easy with bitmaps: just set the appropriate bits in the bitmap.
- Linked lists: modify adjacent elements as needed.
  - Merge adjacent free regions into a single region.
  - May involve merging two regions with the just-freed area.
Limitations of swapping

- Problems with swapping
  - Process must fit into physical memory (impossible to run larger processes)
  - Memory becomes fragmented
    - External fragmentation: lots of small free areas
    - Compaction needed to reassemble larger free areas
  - Processes are either in memory or on disk: half and half doesn’t do any good
- Overlays solved the first problem
  - Bring in pieces of the process over time (typically data)
  - Still doesn’t solve the problem of fragmentation or partially resident processes
Virtual memory

- Basic idea: allow the OS to hand out more memory than exists on the system
- Keep recently used stuff in physical memory
- Move less recently used stuff to disk
- Keep all of this hidden from processes
  - Processes still see an address space from 0 – max address
  - Movement of information to and from disk handled by the OS without process help
- Virtual memory (VM) especially helpful in multiprogrammed system
  - CPU schedules process B while process A waits for its memory to be retrieved from disk
Virtual and physical addresses

- Program uses *virtual addresses*
  - Addresses local to the process
  - Hardware translates virtual address to *physical address*
- Translation done by the *Memory Management Unit*
  - Usually on the same chip as the CPU
  - Only physical addresses leave the CPU/MMU chip
- Physical memory indexed by physical addresses
Paging and page tables

- Virtual addresses mapped to physical addresses
  - Unit of mapping is called a page
  - All addresses in the same virtual page are in the same physical page
  - Page table entry (PTE) contains translation for a single page
- Table translates virtual page number to physical page number
  - Not all virtual memory has a physical page
  - Not every physical page need be used
- Example:
  - 64 KB virtual memory
  - 32 KB physical memory

<table>
<thead>
<tr>
<th>Virtual address space</th>
<th>Physical memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>60–64K</td>
<td></td>
</tr>
<tr>
<td>56–60K</td>
<td></td>
</tr>
<tr>
<td>52–56K</td>
<td></td>
</tr>
<tr>
<td>48–52K</td>
<td>6</td>
</tr>
<tr>
<td>44–48K</td>
<td>5</td>
</tr>
<tr>
<td>40–44K</td>
<td>1</td>
</tr>
<tr>
<td>36–40K</td>
<td></td>
</tr>
<tr>
<td>32–36K</td>
<td></td>
</tr>
<tr>
<td>28–32K</td>
<td>3</td>
</tr>
<tr>
<td>24–28K</td>
<td></td>
</tr>
<tr>
<td>20–24K</td>
<td></td>
</tr>
<tr>
<td>16–20K</td>
<td>0</td>
</tr>
<tr>
<td>12–16K</td>
<td></td>
</tr>
<tr>
<td>8–12K</td>
<td></td>
</tr>
<tr>
<td>4–8K</td>
<td>4</td>
</tr>
<tr>
<td>0–4K</td>
<td>7</td>
</tr>
</tbody>
</table>
What’s in a page table entry?

Each entry in the page table contains
- Valid bit: set if this logical page number has a corresponding physical frame in memory
  - If not valid, remainder of PTE is irrelevant
- Page frame number: page in physical memory
- Referenced bit: set if data on the page has been accessed
- Dirty (modified) bit: set if data on the page has been modified
- Protection information

![Diagram showing the structure of a page table entry](image-url)
Mapping logical => physical address

- Split address from CPU into two pieces
  - Page number ($p$)
  - Page offset ($d$)

- Page number
  - Index into page table
  - Page table contains base address of page in physical memory

- Page offset
  - Added to base address to get actual physical memory address

- Page size = $2^d$ bytes

Example:
- 4 KB (=4096 byte) pages
- 32 bit logical addresses

$2^d = 4096$  \[ \Rightarrow \]  $d = 12$

32 - 12 = 20 bits  12 bits

32 bit logical address
Address translation architecture
Memory & paging structures

Logical memory (P0)
- Page 0
- Page 1
- Page 2
- Page 3
- Page 4

Page table (P0)
- Page frame number:
  - 0
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 8
  - 9

Physical memory
- Page 0 (P0)
- Page 1 (P1)
- Page 1 (P0)
- Page 2 (P0)
- Page 4 (P0)
- Page 0 (P0)
- Page 0 (P1)
- Page 3 (P0)

Free pages
- 0
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
Two-level page tables

- Problem: page tables can be too large
  - $2^{32}$ bytes in 4KB pages need 1 million PTEs
- Solution: use multi-level page tables
  - “Page size” in first page table is large (megabytes)
  - PTE marked invalid in first page table needs no 2nd level page table
- 1st level page table has pointers to 2nd level page tables
- 2nd level page table has actual physical page numbers in it
More on two-level page tables

- Tradeoffs between 1st and 2nd level page table sizes
  - Total number of bits indexing 1st and 2nd level table is constant for a given page size and logical address length
  - Tradeoff between number of bits indexing 1st and number indexing 2nd level tables
    - More bits in 1st level: fine granularity at 2nd level
    - Fewer bits in 1st level: maybe less wasted space?

- All addresses in table are physical addresses
- Protection bits kept in 2nd level table
Two-level paging: example

- System characteristics
  - 8 KB pages
  - 32-bit logical address divided into 13 bit page offset, 19 bit page number
- Page number divided into:
  - 10 bit page number
  - 9 bit page offset
- Logical address looks like this:
  - $p_1$ is an index into the 1st level page table
  - $p_2$ is an index into the 2nd level page table pointed to by $p_1$
2-level address translation example

- Page number
  - $p_1 = 10$ bits
  - $p_2 = 9$ bits
- Page offset
  - Offset = 13 bits

Physical address: 19 13

Frame number: 0 1

Main memory
Implementing page tables in hardware

- Page table resides in main (physical) memory
- CPU uses special registers for paging
  - Page table base register (PTBR) points to the page table
  - Page table length register (PTLR) contains length of page table: restricts maximum legal logical address
- Translating an address requires two memory accesses
  - First access reads page table entry (PTE)
  - Second access reads the data / instruction from memory
- Reduce number of memory accesses
  - Can’t avoid second access (we need the value from memory)
  - Eliminate first access by keeping a hardware cache (called a translation lookaside buffer or TLB) of recently used page table entries
Translation Lookaside Buffer (TLB)

- Search the TLB for the desired logical page number
  - Search entries in parallel
  - Use standard cache techniques
- If desired logical page number is found, get frame number from TLB
- If desired logical page number isn’t found
  - Get frame number from page table in memory
  - Replace an entry in the TLB with the logical & physical page numbers from this reference

Example TLB

<table>
<thead>
<tr>
<th>Logical page #</th>
<th>Physical frame #</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>unused</td>
<td></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>
Handling TLB misses

- If PTE isn’t found in TLB, OS needs to do the lookup in the page table
- Lookup can be done in hardware or software
- Hardware TLB replacement
  - CPU hardware does page table lookup
  - Can be faster than software
  - Less flexible than software, and more complex hardware
- Software TLB replacement
  - OS gets TLB exception
  - Exception handler does page table lookup & places the result into the TLB
  - Program continues after return from exception
  - Larger TLB (lower miss rate) can make this feasible
How long do memory accesses take?

- Assume the following times:
  - TLB lookup time = a (often zero - overlapped in CPU)
  - Memory access time = m

- Hit ratio (h) is percentage of time that a logical page number is found in the TLB
  - Larger TLB usually means higher h
  - TLB structure can affect h as well

- Effective access time (an average) is calculated as:
  - EAT = (m + a)h + (m + m + a)(1-h)
  - EAT = a + (2-h)m

- Interpretation
  - Reference always requires TLB lookup, 1 memory access
  - TLB misses also require an additional memory reference
Inverted page table

- Reduce page table size further: keep one entry for each frame in memory
- PTE contains
  - Virtual address pointing to this frame
  - Information about the process that owns this page
- Search page table by
  - Hashing the virtual page number and process ID
  - Starting at the entry corresponding to the hash result
  - Search until either the entry is found or a limit is reached
- Page frame number is index of PTE
- Improve performance by using more advanced hashing algorithms
Inverted page table architecture

- **Process ID**: \( p \) = 19 bits
- **Offset**: Offset = 13 bits

**Inverted Page Table**

- **Search**:
  - \( k \) steps
- **Physical Address**:
  - \( 19 \) for page number
  - \( 13 \) for offset
- **Main Memory**

**Page Frame Number**:
- \( 0 \)
- \( 1 \)
- \( k \)

**Example**: For \( \text{pid}_0 \) and \( p_0 \), the physical address is determined by the inverted page table.