Memory management

Memory (ideally)
- Ideally...
  - Extremely fast (faster than the CPU in executing an instruction)
  - Abundantly large
  - Dirt cheap

Memory (for real)

<table>
<thead>
<tr>
<th>Typical access time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 nsec</td>
<td>&lt; 1K</td>
</tr>
<tr>
<td>2 nsec</td>
<td>~ 1M</td>
</tr>
<tr>
<td>10 nsec</td>
<td>~ 1-4G</td>
</tr>
<tr>
<td>100 nsec</td>
<td>~ 5-100G</td>
</tr>
<tr>
<td></td>
<td>&gt; 20G</td>
</tr>
</tbody>
</table>

Memory cntd.

- Registers: typical 32 in a 32 bit CPU
- Cache: divided into cache lines (64 bytes each)
  - Cache hit – no main memory access, no bus involvement
  - Cache miss – costly
- Main memory
- Disk (multiple plates, heads, arms)
  - Logical structure: sectors, tracks, cylinders
- Magnetic tape: backup, cheap, removable

OS management of memory

- The part of the OS that handles the management of the memory is called:
  - Surprise, surprise: memory manager!
- Various levels of complicity
  - Depending on the type of OS
- E.g. mono-programming, multiprogramming, and so forth

Specifics of memory management

- Basic memory management strategies
  - Monoprogramming without swapping or paging
  - Multiprogramming with fixed partitions
  - Multiprogramming with variable partitions
  - Swapping
  - Virtual memory: paging
  - Virtual memory: segmentation
Mono-programming

- No swapping, a couple of options...

Some embedded systems (e.g., PALM)

Modeling multiprogramming

- Each process spends a fraction $p$ of its time waiting for I/O to complete
- If we have $n$ processes, the probability that all of them are simultaneously waiting for I/O is: $p^n$
- CPU utilization is thus: $1 - p^n$

Example

- A system has 32Mbyte of RAM, OS takes 16Mbytes
- Each process occupies on average 4Mbytes (4 processes simultaneously in memory) and has 20% utilization time (80% blocked on I/O)
- CPU utilization approx 60%
- Buying 16M additional RAM will allow to increase multiprogramming to 8, CPU utilization will get to about 83%
- Another 16M will get from 83 to 93%, depending on memory price we can make an informed choice

Relocation and protection

- Relocation when loading the code. The linker stores some additional information which is used at load time to relocate (rewrite) every single instruction referencing memory.
- HW support through the use of base and limit registers
- Partial support, only base but no limit
Swapping and virtual memory

- Swapping: whole process data/code in memory when running
- Virtual memory: only part of the data/code in memory

Swapping cndt.

- Memory compaction (remove holes)
- If processes could grow (by allocating memory on a heap like in many programming languages), how does the OS take care of it?
- Many different solutions: e.g. reserve room for growth or swap the process out and relocate it to a bigger memory partition, etc.

How is it implemented?

- Bitmaps
  - Memory is divided into allocation units
  - Each bit of the bitmap represents a unit ($1 = \text{used}, 0 = \text{free}$)
  - The size of allocation unit is an important design issue (less unused memory, bigger bitmap)
  - Search bitmaps when loading in a new process for $k$ consecutive free allocation units

How is it implemented?

- Linked lists
  - A linked list may store:
    - Information about a process or a hole
    - Address where it starts
    - Length
    - Pointer to the next element
  - Merging operation (e.g. two consecutive holes)
  - Process' table entry will contain a pointer to the element in the list relative to it

Different algorithms

- First fit: the first hole that fits the process is used (hole is broken down into two pieces)
- Next fit: it doesn't start from the beginning, simply restart from where it left the previous search
- Best fit: search the whole list for the smallest hole that fits
- Performance: best fit creates a lot of fragmentation in practice, first fit tends to leave larger holes (less fragmentation)
Fragmentation

- Internal: partition or page not fully used by a given process
- External: entire partitions or free space (holes) not used because no process fits in the size of any of the holes

MVT

- Curiosity: used by the IBM OS/360 (1960 or so) in a version called MVT (multiprogramming with variable number of tasks):
  - Dynamical partitions: sized as the size of processes
  - Swapping: as described earlier

Virtual memory

- Once upon a time there was the “overlay”
- In practice programmer divided the program (by hand) into many parts that could be swapped in and out from disk (overlaid onto unused parts)
- Why don’t we delegate this function to the machine itself?
- Virtual memory was born!

Demand paging

- Pages are loaded from disk only when needed (demanded)
- Process that causes the page fault can be considered blocked for I/O (and another process could run)
- Swapping (of pages), lazy backing store (e.g. “lazy” means that pages are only loaded when needed otherwise the system does nothing, it doesn’t swap entire processes)
Paging can be

- Pure: for every logical page there exists a physical page, always everything available in memory
- On demand: at any given instant only a subset of the virtual address space is in memory (but everything is still consistent)

Example

- MOV REG,0
  - CPU sends request for address 0
  - MMU looks for 0 and sees that the page containing virtual address 0 is at frame 2
  - It thus adds 8192 (frame 2 start address)
  - MMU finally sends 8192 on the bus

Since memory is finite

- Present/absent bit in the virtual page table (the X's in the picture)
- Same as before:
  - MOV REG,32780 what happens?
  - Page fault, the page is not in physical memory but rather on the disk
  - The OS needs to evict a page from main memory and to replace it with the missing page, to update the MMU’s tables, and to restart the instruction that caused the fault

Operation of the MMU

- The page table can be extremely large
  - 32bits systems with a 4K-page size has more than a million pages
  - 64bits 2^52 pages? Gosh!
- The mapping must be fast (VERY fast)
  - Every memory reference requires a virtual to physical conversion, a single instruction might have >1 reference

Is it a simple task?
Where’s the page table?

• Within the MMU
  – Every context switch requires loading the whole page table into the MMU registers, good because it doesn’t require more memory reference afterwards
• Memory
  – A single pointer to the table needs to be reloaded in a context switch, more memory references (to the page table) are required for mapping each memory reference

About the page table

• Present/absent: in memory?
• Protection bits: e.g. read/write/execute
• Modified: whether any address has been changed, rewrite to disk is required before evicting the page
• Referenced: used by the OS to decide which page to evict
• Caching: may be used to avoid caching pages required for I/O

What does the TLB do?

• Small number of entries, within the MMU, fast
• Association (direct) of virtual page to page frame
• Parallel compare over the whole table, if virtual page is not there, do the normal lookup (over memory) and then evict an entry and replace with the new one

Multi-level page tables

• Example: 32 bits could be partitioned as a 10-bit pointer to level 1 table, 10-bit to level 2 and 12-bit offset fields

TLBs

• Translation Lookaside Buffers
  – Page tables in memory require additional memory accesses, unpractical
  – Most programs tend to make a large number of references to a small number of pages
  – Use something called a TLB or Associative Memory

Additional issues

• Software TLB management
  – Some microprocessors don’t have the TLB completely in HW, the handling of the TLB fault is done in SW (i.e. the OS does it)
• Inverted Page Tables
  – Imagine a 64 bit computer: page tables would be too big
  – Inverted table, one per page frame rather than per page
  – It requires a search (potentially slow), needs a good implementation (fast) and a possibly large TLB