Memory Management

To do ...

- Basic memory management
- Swapping
- Kernel memory allocation
- Next Time: Virtual memory
Memory management

- Ideal memory for a programmer – large, fast, non-volatile *and* cheap – not an option

- Best alternative → memory hierarchy
  - Small amount of fast, expensive memory – cache
  - Some medium-speed, medium price main memory
  - Gigabytes of slow, cheap and non-volatile disk storage

- To handle the memory hierarchy – memory manager
  - Allocates scarce resource among competing requests to maximize (memory) utilization and system throughput
  - Offers a convenient abstraction for programming
  - Provides isolation between processes
Basic memory management

- Simplest memory abstraction – no abstraction at all
  - Early mainframes (before ‘60), minicomputers (before ‘70) and PCs (before ‘80)
  - MOV REG1, 1000 #REG1 $\leftarrow$ (physical memory 1000)
  - Logically, only one program running at a time Why?
  - Still here, some alternatives for organizing memory

Mainframes & minicomputers

Some palm tops & embedded systems

Early PCs

User program

Operating system in RAM

Operating system in ROM

Devices in ROM

User program

Operating system in RAM

BIOS

MSDOS
Multiprogramming and swapping

- Even with no memory abstraction, you can run multiple programs concurrently
- Switching processes with swapping – Corbató’s CTSS
  - Simple
  - Bring each process entirely
  - Move another one to disk
  - Process swapped back in restarts from where it was

- Clearly, moving the whole thing is not too good for performance
Multiprogramming with fixed partitions

- With some additional hardware, avoid swapping
- IBM OS/360
  - Split memory in $n$ parts (possible $\neq$ sizes)
  - Single or separate input queues for each partition
  - ~MFT: Multiprogramming with Fixed number of Tasks

**Protection?**

```
<table>
<thead>
<tr>
<th></th>
<th>Operating System</th>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
<th>Partition 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protection?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
```

**Relocation?**

```
<table>
<thead>
<tr>
<th></th>
<th>Operating System</th>
<th>Partition 1</th>
<th>Partition 2</th>
<th>Partition 3</th>
<th>Partition 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relocation?</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
```

- Multiple input queues
- Single input queue
Size of the allocation unit

- Important design issue
- A pervasive problem – fragmentation
  - Internal fragmentation – the available portion is larger than what was requested
  - External fragmentation – two small partitions left, but one big partition needed!
Multiprogramming – Protection

- What stops a process from writing outside its allocated memory?

- IBM 360 – Check it for each reference
  - Split memory into 2KB blocks
  - Each block was assigned a 4b protection key kept in CPU
  - The PSW (program status word) also kept a 4b key
  - OS trapped any process trying to access memory with protection key != the PSW key
Multiprogramming – Relocation

- With absolute physical memory – the program must always run in the exact same location
  - What a little JMP X can do!
- IBM 360 stop-gap solution – Change it at loading time
  - Static relocation
  - If a program was reloaded starting at 131,072, the constant 131,072 was added to every address at load time
  - Not particularly fast!
A new abstraction – Address space

- Address space – all the memory a process can reference
  
  MOV REG1, ????

- What does it include
  - Code – static, so it won’t change while the program runs
  - Stack – to keep track of where it is in the function call chain
  - Heap – for dynamically allocated memory
  - We placed them opposite so they can both grow (but this is just a convention)
Multiprogramming with variable partitions

- A simple solution – partition physical memory into dynamically allocated blocks
  - CDC 6600 and Intel 8088

- Base and limit values
  - Address locations + base value → physical address
  - Ideally, base and limit registers can only be modified by OS
  - A disadvantage – Comparisons can be done fast but additions can be expensive

- Transparency
- Protection
- Efficiency
And now a short break ...

MAN, I SUCK AT THIS GAME.
CAN YOU GIVE ME
A FEW POINTERS?

I HATE YOU.

0x3A28213A
0x6339392C,
0x7363682E.
Memory management’s other client

- The OS manages physical memory and allocates it to
  - User processes
  - Kernel subsystems
- Kernel needs small, odd sized chunks of memory
  - Proc structures, file descriptor blocks ... sizes << a page
  - Pre-allocates part of memory for the KMA
- Most OS manage memory as set of fixed-size pages, so another approach
  - Introduce a page-level allocator
  - Let the kernel manage the space in a set of pages – Kernel Memory Allocator (KMA)
Page-level allocator and KMA

- Page-level allocator has
  - Two main routines: e.g `get_page()` & `freepage()` in SVR4
  - Two main clients: Paging system & KMA

Diagram:
- Physical memory
- Provides odd-size buffers to various kernel subsystems
- Page-level allocator
  - Kernel memory allocator
  - Paging system
  - Network Buffers
  - Proc structures
  - inodes, file descriptors
  - User processes
  - Block buffer cache
Memory management

- With dynamically allocated memory
  - OS must keep track of allocated/free memory
  - Two general approaches – bit maps and linked lists

Bit maps
  - Divide memory into allocation units (size?), track usage with a bitmap

Linked list of allocated or free spaces
  - List ordered by address
Picking a place – basic algorithms

- First fit – simple and fast
  ![First fit diagram]

- Next fit - ~ First fit, starting where it left off
  - Slightly worst performance than First fit
  ![Next fit diagram]

- Best fit – try to waste the least,
  - but many tiny holes and a whole list search
  ![Best fit diagram]

- Worst fit – try to “waste” the most (making it easier to reuse)
  - Not too good either
  ![Worst fit diagram]
Tracking the size of allocated units

- Keep extra info in a header block
  - Size (this for sure)
  - Additional pointers for faster de-allocation
  - Magic number for additional integrity checks

```c
typedef struct _header_t {
    int size;
    int magic;
} header_t;

void free(void *ptr) {
    header_t *hptr = (void *) ptr - sizeof(header_t);
    ...
    assert(header_t->magic == MAGICN);
    ...
}
```
KMA – Resource map allocator

- Resource map – a set of \(<base, size>\) pairs
- Initially the pool is described by a single pair
- … after a few exchanges … a list of entries per contiguous free regions
- Allocate requests based on
  - First fit, Best fit, Worst fit
- A simple interface

```c
offset_t rmalloc(size);
void rmfree(base, size);
```

```
256,128
576,448
rmalloc(256)  rmalloc(320)
```

```
rmfree(256,128)
```
Resource map allocator

• **Pros**
  – Easy to implement
  – Not restricted to memory allocation
  – It avoid waste
  – Client can release any part of the region
  – Allocator coalesces adjacent free regions

• **Cons**
  – After a while maps ended up fragmented – low utilization
  – Higher fragmentation, longer map
  – Map may need an allocator for its own entries
  – To coalesce regions, keep map sorted – expensive
  – Linear search to find a free region large enough
KMA – Simple power-of-two free list

- A set of free lists
- Each list keeps free buffers of a particular size \(2^x\)
- Each buffer has one word header
  - Pointer to next free buffer, if free or to
  - Pointer to free list (or size), if allocated
KMA – Simple power-of-two free list

- Allocating(size)
  - allocating (size + header) rounded up to next power of two
  - Return pointer to first byte after header

- Freeing doesn’t require size as argument
  - Move pointer back header-size to access header
  - Put buffer in list

- Initialize allocator by pre-allocating buffers or get pages on demand; if it needs a buffer from an empty list ... 
  - Block request until a buffer is released
  - Satisfy request with a bigger buffer if available
  - Get a new page from page allocator
Power-of-two free lists

- **Pros**
  - Simple and pretty fast (avoids linear search)
  - Familiar programming interface (malloc, free)
  - Free does not require size; easier to program with

- **Cons**
  - Rounding means internal fragmentation
  - As many requests are power of two and we loose header; a lot of waste
  - No way to coalesce free buffers to get a bigger one
  - Rounding up may be a costly operation
Other algorithms – Buddy

- Coalescing is critical; buddy allocator
- Free memory is a $2^N$ size block
  - When requesting a block, split in half until is big enough
  - When block is freed, check if buddy is free & coalesce
    - Finding a buddy – address of each buddy pair differs only by a single bit!
- Repeat
Kernel memory allocation – Evaluation

- Several approaches to KMA
  - Slab, power-of-two lists, Buddy, ...

- Memory utilization
  - Physical memory is limited after all
  - Major cause of wasted memory – fragmentation

- Speed
  - It is used by various kernel subsystems (e.g. interrupt handler)
  - Better be fast – both in avg and worst case

- Simple API
  - A simple free & malloc is nice but you can’t release partial memory with too simple of a malloc

- Allow a two-way exchange with page-level allocator
Coming up …

- Virtual memory in all its beauty