CS 241 Section Week #9
(11/05/09)

## Topics

- MP6 Overview
- Memory Management
- Virtual Memory
- Page Tables

## MP6 Overview

## MP6 Overview

- In MP6, you create a virtual memory system with memory mapped IO.
  - We provide you with two parameters:
    - PAGESIZE: The size (bytes) of each memory page.
    - MEMSIZE: The total number of memory pages.
    - Use these variables and NOT the numbers they correspond with (eg: use PAGESIZE not 1024) .

## MP6 Overview

- In MP6, you have a fixed amount of pre-defined memory for storing memory mapped I/O contents.
  - memory[PAGESIZE * MEMSIZE]
    - This memory will likely not be large enough to store the entire contents of files.
    - You will need to program a page table to manage the paging of data in and out of this memory.

## MP6 Overview

- In MP6, you need to implement four functions:
  - my_mmap()
    - Initializes the memory mapping structure / space.
  - my_mread() / my_mwrite()
    - Reads and writes to the memory mapped space.
  - My_munmap()
    - Destroys the memory mapped structure, flushes any pages still in memory back to disk.

## MP6 Overview

- Besides the functions and storage space defined above, we only require three additional things:
  - Pages must be replaced by a "Least Recently Used" algorithm
  - Pages must only be written out to disk when they are paged out if their contents have changed.
  - When writing out the THIRD page (index 2) to disk, you should print some output.

## MP6 Overview

- Everything else is left for you to decide how to implement.

- Like MP5, we provide a simple tester. You should program other testers to test fully test the robustness of your library.

# Memory Management

## Memory

- Contiguous allocation and compaction
- Paging and page replacement algorithms

## Fragmentation

- External Fragmentation
  - Free space becomes divided into many small pieces
  - Caused over time by allocating and freeing the storage of different sizes
- Internal Fragmentation
  - Result of reserving space without ever using its part
  - Caused by allocating fixed size of storage

## Contiguous Allocation

- Memory is allocated in monolithic segments or *blocks*
- Public enemy #1: external fragmentation
  - We can solve this by periodically rearranging the contents of memory

## Storage Placement Algorithms

- Best Fit
  - Produces the smallest leftover hole
  - Creates small holes that cannot be used

## Storage Placement Algorithms

- Best Fit
  - Produces the smallest leftover hole
  - Creates small holes that cannot be used
- First Fit
  - Creates average size holes

## Storage Placement Algorithms

- Best Fit
  - Produces the smallest leftover hole
  - Creates small holes that cannot be used
- First Fit
  - Creates average size holes
- Worst Fit
  - Produces the largest leftover hole
  - Difficult to run large programs

## Storage Placement Algorithms

- Best Fit
  - Produces the smallest leftover hole
  - Creates small holes that cannot be used
- First Fit
  - Creates average size holes
- Worst Fit
  - Produces the largest leftover hole
  - Difficult to run large programs

**First-Fit** and **Best-Fit** are *better* than **Worst-Fit** in terms of *SPEED* and *STORAGE UTILIZATION*

## Exercise

- Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of (a) 12KB, (b) 10KB, (c) 9KB for
  - First Fit?

## Exercise

- Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of (a) 12KB, (b) 10KB, (c) 9KB for
  - First Fit? 20KB, 10KB and 18KB

## Exercise

- Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of (a) 12KB, (b) 10KB, (c) 9KB for
  - First Fit? 20KB, 10KB and 18KB
  - Best Fit?

## Exercise

- Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of (a) 12KB, (b) 10KB, (c) 9KB for
  - First Fit? 20KB, 10KB and 18KB
  - Best Fit? 12KB, 10KB and 9KB

## Exercise

- Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of (a) 12KB, (b) 10KB, (c) 9KB for
  - First Fit? 20KB, 10KB and 18KB
  - Best Fit? 12KB, 10KB and 9KB
  - Worst Fit?

## Exercise

- Consider a swapping system in which memory consists of the following hole sizes in memory order: 10KB, 4KB, 20KB, 18KB, 7KB, 9KB, 12KB, and 15KB. Which hole is taken for successive segment requests of (a) 12KB, (b) 10KB, (c) 9KB for
  - First Fit? 20KB, 10KB and 18KB
  - Best Fit? 12KB, 10KB and 9KB
  - Worst Fit? 20KB, 18KB and 15KB

## malloc Revisited

- Free storage is kept as a list of free blocks
  - Each block contains a size, a pointer to the next block, and the space itself

## malloc Revisited

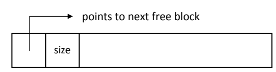
- Free storage is kept as a list of free blocks
  - Each block contains a size, a pointer to the next block, and the space itself
- When a request for space is made, the free list is scanned until a big-enough block can be found
  - Which storage placement algorithm is used?

# malloc Revisited

- Free storage is kept as a list of free blocks
  - Each block contains a size, a pointer to the next block, and the space itself
- When a request for space is made, the free list is scanned until a big-enough block can be found
  - Which storage placement algorithm is used?
- If the block is found, return it and adjust the free list. Otherwise, another large chunk is obtained from the OS and linked into the free list

# malloc Revisited (continued)

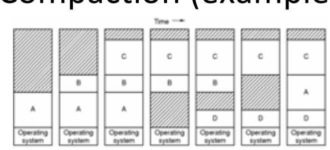
```
typedef long Align;     /* for alignment to long */

union header {          /* block header */
  struct {
    union header *ptr;  /* next block if on free list */
    unsigned size;      /* size of this block */
  } s;
  Align x;              /* force alignment of blocks */
};

typedef union header Header;
```




# Compaction

- After numerous `malloc()` and `free()` calls, our memory will have many holes
  - Total free memory is much greater than that of any contiguous chunk
- We can *compact* our allocated memory
  - Shift all allocations to one end of memory, and all holes to the other end
- Temporarily eliminates of external fragmentation

# Compaction (example)




- Lucky that A fit in there!  To be sure that there is enough space, we may want to compact at (d), (e), or (f)
- Unfortunately, compaction is problematic
  - It is very costly.  How much, exactly?
  - How else can we eliminate external fragmentation?

## Paging

- Divide memory into *pages* of equal size
  - We don't need to assign contiguous chunks
  - Internal fragmentation can only occur on the last page assigned to a process
  - External fragmentation cannot occur at all
  - Need to map contiguous logical memory addresses to disjoint pages

## Page Replacement

- We may not have enough space in physical memory for all pages of every process at the same time.
- But which pages shall we keep?
  - Use the history of page accesses to decide
  - Also useful to know the *dirty* pages

## Page Replacement Strategies

- It takes two disk operations to replace a dirty page, so:
  - Keep track of dirty bits, attempt to replace clean pages first
  - Write dirty pages to disk during idle disk time
- We try to approximate the optimal strategy but can seldom achieve it, because we don't know what order a process will use its pages.
  - Best we can do is run a program multiple times, and track which pages it accesses

## Page Replacement Algorithms

- **Optimal**: last page to be used in the future is removed first
- **FIFO**: First in First Out
  - Based on time the page has spent in main memory
- **LRU:** Least Recently Used
  - Locality of reference principle again
- **MRU**: most recently used = removed first
  - When would this be useful?
- **LFU:** Least Frequently Used
  - Replace the page that is used least often

## Example

- Physical memory size: 4 pages
- Pages are loaded on demand
- Access history: **0 1 2 3 4 0 1 2 3 4 …**
  - Which algorithm does best here?

- Access history: **0 1 2 3 4 4 3 2 1 0 …**
  - And here?