ECE 598 – Advanced Operating Systems
Lecture 12

Vince Weaver

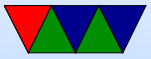
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

1 March 2018

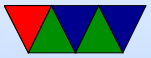
Announcements

- Next homework will be due after break.
- Midterm next Thursday
  We'll review for it on Tuesday
  Material will be similar to the homeworks.



Various Types of Memory Management

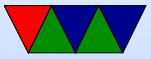
- Application
- Operating System
- Hardware



Application Memory Allocation on C/Linux

```c
#include <stdio.h>

int global_x=5; /* data */
int global_y=0; /* bss */

int main(int argc, char **argv) {

    int local_x=5; /* stack */
    static int static_y=5; /* data */
    static int static_x=0; /* bss */

    char *heap_x=malloc(1024); /* heap or mmap() */

    printf("Hello␣world\n");

    return 0;
}
```

- Local variables on the stack



Stack auto-grows down

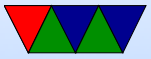
```
int q[1000];

sub sp,sp,#0x3f0
stp x29,x30,[sp,#-16]!  // store pair, fp and lr
...
ldp x29,x30,[sp],#16
adc sp,sp,#0x3f0
```

Can you dynamically allocate on stack? alloca(). Also variable defined arrays (gcc extension?)

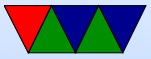
Downsides: stack overflow attacks (show example)

What happens to pointers once return

Contents at startup if not initialized.

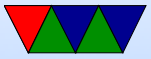


- Global and static variables that are initialized in the data segment, loaded directly from disk.
- Global and static variables initialized to zero in the bss segment.
- Dynamic variables allocated on the heap or via mmap
  - `malloc()` is not a syscall, but a library call.
  - Kernel interface is the `brk()` system call which moves the end of the data segment (essentially making the heap bigger)
  - `mmap()` initially was map file into memory so can be accessed with memory allocations rather than



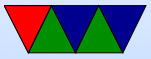
read/write

anonymous mmap will allocate memory range with no backing file.

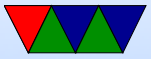


How Heap/Malloc Works

- Memory allocation libraries can vary underneath. Basically a big chunk of RAM is grabbed from the OS, and then split into parts in a custom way.
- Do you just grab a chunk of mem and return a pointer? Or is there extra info you need to track?
- The biggest problem is fragmentation, which happens when memory is freed in non-contiguous areas.
- dlmalloc – Doug Lea – glibc uses ptmalloc based on it. Memory allocated in chunks, with 8 or 16-byte header

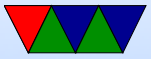


bins of same sized objects, doubly linked list
Small allocations (256kB?) closest power of two used
Larger, mmap used, multiple of page size.



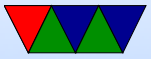
Manual vs Automatic

- With C you can manually allocate and free memory. Prone to errors:
  - Use-after-free errors
  - Buffer overflows
  - Memory Leaks
  - ALL OF THE ABOVE CAN LEAD TO ROOT EXPLOITS
- High-level languages such as Java will automatically



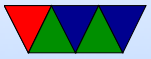
allocate memory for objects. The user never sees memory pointers. Unused memory areas are periodically freed via "garbage-collection". At the same time the memory can be compacted, avoiding fragmentation. Problem? Slow, not real-time, can be complex detangling complex memory dependency chains.

- Finding bugs: Valgrind

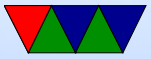


Memory Allocation w/o Virtual Memory

- sbrk() doesn't work
- mmap
- overhead

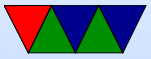


Brief History of Memory Handling in Operating Systems



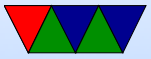
Mono-Programming

- Simple mono-programming: just OS and one program in memory at once (like DOS)

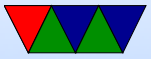


Fixed Multi-Programming

- Multiprogramming: let you run multiple tasks at once.
- Fixed Partitions of memory available. Jobs queued. When spot frees up job can run. Can have complex scheduling rules out which size and priority to give to jobs. Older mainframes (OS/MFT) used this.
- Relocations a problem
- Memory protection. Permissions on pages.

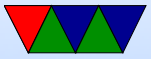


- Solution to both protection and permission in segments (with base offset and range that are valid to access)



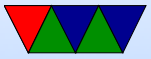
Swapping

- Timesharing systems. All jobs not fit in RAM?
- Swapping: bring in each program in entirety, run it a while, then when done writing all back out to disk.
- Paging: virtual memory.



Tracking Memory

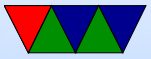
- How do we know how much memory we have?
  Proing, firmware, assume
- What granularity should be used?
- Bitmap – chunks of memory, bitmap indicating free space
  Have to search bitmap and find N consecutive empty areas for each allocation (also used by some filesystems)
- Free-lists, linked list of memory areas



Example

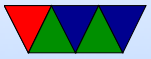
```
Each block size 1k bytes.  1 means used, 0 free.
0xf501f080
1111 0101 0000 0001 1111 0000 1000 0000
```

- Want to allocate size 18? (1 block)
- Want to allocate size 1500? (2 blocks)
- Want to allocate size 6000? (6 blocks)
- Want to allocate size 8192? (8 blocks)
- How can you quickly find how much free? Brute force, Clever/complex C, popcnt instruction



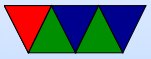
Fragmentation

- Enough memory available, but split up. How can fix?
- Memory compaction. Swap everything out, bring it back in (Relocating). Is this always possible? On Java? In C?

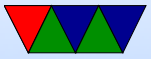


Fit Algorithms

- **First Fit**: scans bitmap, returns first block big enough to meet request. Fastest.
- **Next Fit**: Picks up where the last first fit case left off (optimization)
- **Best fit**: search entire map and find hole that fits it best. Actually can cause more fragmentation, end up with lots of tiny holes
- **Worst Fit**: always biggest hole. Not so great either.

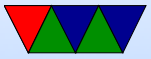


- **Quick Fit**: separate lists for more common sizes



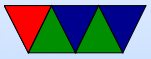
Fixed Sized Allocation

- Memory Pool – fast – blocks of pre-allocated memory in power of two sizes that can be handed out
  fast to allocate/free
  fragmentation

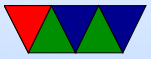


Buddy Allocator

- Used by Linux
- Pick a low size, say 4k, and a high size, say 1MB
- When allocate, round up to the next power of two
- Search for free area that size. If not, scale up. If you find one, split it into chunks until you reach the size being looked for. Give it.
- When freeing, not only free but see if neighboring blocks



also free, if so, re-join them to bigger sized memory.



Memory Protection

- x86 segmentation
- ARM MPU Memory Protection Unit
  - Base and size register
  - 16 of them, can be overlapped
  - types: normal, device (for mmio), strongly ordered
  - data access: full, pri only, read-only, no access
  - insn access: no-execute. Explain how ROP (return-oriented programming) gets around this
- Virtual memory