ECE 531 – Advanced Operating Systems
Lecture 12

Vince Weaver

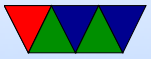
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

5 October 2023

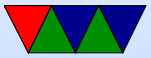
Announcements

- Midterm the 17th
- HW#5 due date extended until Monday

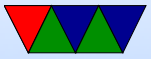


HW#5 Timer note

- The timer we are using is based on the CPU core clock This was 250MHz on older Pis
- Newer Pis this clock is variable, and can be at 250MHz, 400MHz, or 500MHz depending on frequency scaling
- Ideally this wouldn't affect us (or we could set it to fixed in boot config file)
- If you do notice your LED doesn't blink at the right rate because of this, don't worry about it
- This affects i2c and SPI too!

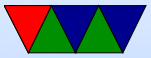


- The real solution would be to find a timer that doesn't change frequency. Possibly GPU Timer #3



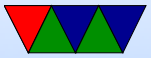
Various Types of Memory Management

- Application
- Operating System
- Hardware



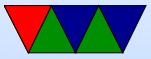
Application Memory Allocation on C/Linux

```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;
}
```



Compiler Optimization Note

- The compiler can (and will) optimize away memory accesses whenever possible
- At -O2 optimization if you don't use a pointer to a variable it might only ever live in a register
- The old `register` keyword used to enforce this

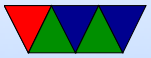


Variables on the Stack

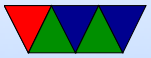
- Local variables go on the stack
- Stack auto-grows down

```
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/Issues
  - stack overflow attacks (show example)

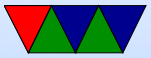


- What happens if you return a pointer to a local variable
- Contents of uninitialized variables might have old data / not zero



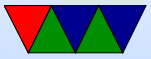
Variables in Data

- Global and static variables that are initialized go in the data segment
- Loaded directly from the executable



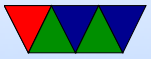
Variables in BSS

- Global and static variables initialized to zero go in the bss segment.
- Uninitialized global/static variables also go in BSS
  On Linux at least these will be initialized to zero even if you don't request it
  You wouldn't want actually uninitialized data on process start up, huge security risk.
- These aren't in executable, it just holds the total BSS size request, and the OS allocates and zeros it at start



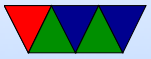
Dynamic Memory Allocation

- `malloc()` is not a syscall, but a library call
- Generally the C library will request chunks of memory from the OS, then hand it out in smaller pieces as requested



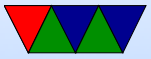
The Heap

- Kernel interface is the `brk()` system call which moves the end of the data segment (essentially making the heap bigger)



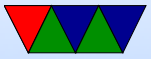
mmap()

- Widely used modern method of getting memory to use
- `mmap()` initially mapped file into memory so can be accessed with load/store memory accesses rather than disk read/write
- You can specify ANONYMOUS access and it will back with zeroed out memory instead, essentially letting you allocate arbitrary sizes of memory
- In addition you can set extra constraints like READ / WRITE / EXEC to have it read only, read write,



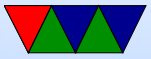
executable (shared libraries are loaded this way, map memory, copy in as executable)

- Can mark as SHARED to share pages between processes for inter-process communication (IPC)
- MAP_FIXED can be used to request it be loaded at a specific address if possible
- MAP_LOCKED can request not be swapped out



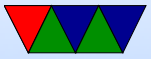
How malloc() Works

- Many ways to write malloc(), each C library has own
- 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.



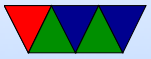
dlmalloc – Doug Lea

- glibc uses ptmalloc based on dlmalloc
- 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.

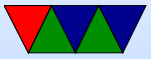


Manual vs Automatic

- 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.

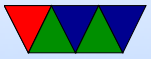


- The user never sees memory pointers.
- Unused memory areas are periodically freed via "garbage-collection".
- At the same time the memory can be compacted, avoiding fragmentation.
- Problem? Slow, not real-time, can be complex detangling complex memory dependency chains.



Kernel Memory Handling

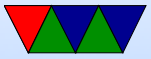
- The kernel also needs to manage memory
- Needs to allocate memory for processes and such
- Might need to dynamically allocate stuff in general operation though that's sometimes discouraged (local variables on stack much easier to deal with than fragmentation issues from malloc() like allocations)



Detecting Memory

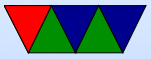
- How do we know how much memory we have?
  - Firmware – ask the bootloader/firmware (but how does it know?)
  - Assume – if only running on a machine with a fixed amount
  - Probing – try writing a value to memory, then read it back and see if it's the same

    Might read a bunch before writing to avoid false positives from bus capacitance



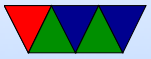
Tracking Memory – Granularity

- What granularity should be used?
- Too small (byte granularity) too much overhead to track it all
- Too large, then hand out much bigger chunks than actually need

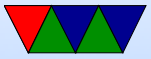


Tracking Memory – Data Structures

- Bitmap
  - Simplest
  - Bitmap of chunks of memory, each bit indicated free/used
  - Have to search bitmap and find N consecutive empty areas for each allocation
  - Lots of bit-fiddling though some architectures have instructions (popcnt) that make this easier
  - We'll see this is also used by some filesystems

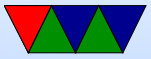


- Free-lists, linked list of memory areas
- Trees



Bitmap Example

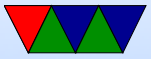
- 2GB of memory, memory broken up into 1k chunks
- 32-bit system so each word 4 bytes
- 2GB/1k = 2MB entries to track, each one a bit, so 2MB/32 = 64k of integers array (256k total size)
- To find if memory address is used/free, take address, /1024/32 to get index in array, then use bottom 5 bits to pick which bit to look at



Bitmap Example Continued

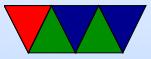
```
Each block size 1k bytes.  1 means used, 0 free.
0xf501f080
1111 0101 0000 0001 1111 0000 1000 0000
```

- 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)
- ((size-1)/1024)+1 to find out how many blocks you need



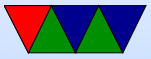
Fragmentation

- 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