ECE 531 – Advanced Operating Systems
Lecture 23

Vince Weaver

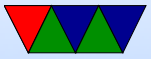
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

28 November 2023

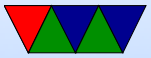
Announcements

- ThreadX being released open source
- HW#7 was due (and will be discussed below)
- HW#8 posted (short, look at it before the midterm)
- Tentative project schedule sent out

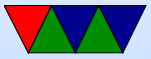


Second Midterm

- Thursday November 30th
- Cumulative, but heavily concentrating on topics since the 1st midterm
- Topics:
  - Memory allocation
  - Virtual Memory: what it's good for, how two addresses can have the same address
  - Filesystems: Fat, ext4, others. Why use fat?
  - Multi-core/Locking/Deadlock/IPC

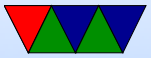


- Maybe a brief graphics question

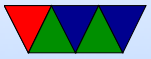


Project Presentations

- Aim for 8 minute presentation with 2 min for questions/setup
- Can present with slides if you want
- Mostly just showing off your project idea and what you accomplished.
- Topics to include
  - Intro/Overview: what you did and why
  - Brief Related Work: any similar projects, how your project differs (it's OK if it doesn't)

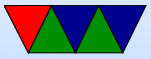


- Hardware: describe any hardware used in the project If it's interfaced via built-in BCM2835 hardware briefly describe that interface too.
- Software: describe the software you wrote and how it ties into an operating system (is it a device driver? is it userspace and talks via syscalls? Does it bypass the OS? is it a proof-of-concept outside the OS?)
- Challenges: Describe any challenges you encountered
- Future Work: if you had more time, what else would you do
- Demo: show off what you did, if possible



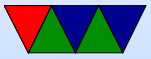
Project Writeup

- Described in the document
- Ideally IEEE format (this is grad course)
- Like a mini published paper. 6 pages or so is fine.
- Document on website describes which sections to include



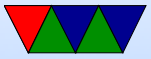
HW#6 Review – Code

- Scheduling algorithm – round robin (it walks list, wraps around on end, stops if we get back to original)
- Memory Algorithm – first fit
- Changing to next-fit, mostly involves static variable to hold last found address and wrapping around when hit end



HW#6 Review – Questions

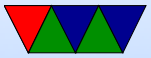
- Memory free, 11 bits 0, times 4k=44k
- First fit, at 0x3 as it's the first block with 4 consecutive blocks free
- Best fit would go at 0xc as that's exactly 16k
- Best fit might be better, reduce fragmentation
- Not possible to allocate 32k, even though 44k free
  Could you de-frag the RAM? Not if C pointers involved



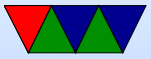
HW#7 Review – Virtual Memory

- Features of virtual memory?
  - Giving illusion of more RAM than we have
  - Demand paging
  - Needed for caching? not always, only on machines with VIPT (virtually-indexed) caches like on ARM/Pi
- What hides page lookup overhead? TLB
- What is it called when a page is not found in the page tables?

  Page fault. It's the OS's job to handle this

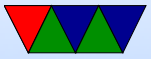


- If physical memory is full, room can be made by swapping out a page to disk. Can also kick out pages that aren't dirty (like executable data). Also things like disk cache.
- Aside, what happens if can't free up memory? Linux at least has OoM killer that will kill processes to free up memory. related: memory over-commit



HW#7 Review – Filesystems

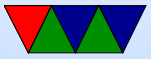
- ext2 is traditional UNIX-style fs where filename is in directory entry, which points to an inode (which has the rest of the metadata and block pointers). Why are they separate?
  - Primarily for hard links (multiple filenames can point to same actual file)
  - Also lets inodes be more compact and not have to hold a filename which can be long and of varying length
- ext2: 12 direct pointers, 1 indirect, 1 double indirect, 1



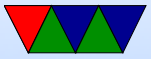
triple indirect

- direct pointer = 12*1k = 12k of data
- 1 indirect = 12k+(1k*256) = 268k of data
- 2 indirect = 12k+256k+(1k*256*256) = 65535k+268k= 65M
- 3 indirect = 12k+256k+65535k+16777216k=16GB
- Overhead, 1+256+65536 = 65M of indirect blocks

- How is ext2 better than fat: permissions, less filename restrictions, bigger files
- How is fat better than ext2: available on all OSes, code for accessing relatively simple/compact

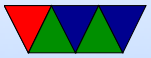


- Holes in the filesystem
- Other filesystems supported by Linux: people chose a variety



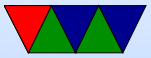
HW#7 Review – Device Types

- char device: bytes streaming in, can't seek
- block devices: data comes in chunks, can usually more or less have random access to all the data



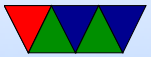
HW#7 Review – Graphics

- attribute aligned, gives aligned memory allocation
- We need this for GPU mailbox access as the bottom bits used to indicate channel



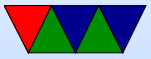
531-OS Locking Example

- Need to protect any place where multiple processes can be in the kernel at the same time accessing shared state
- For example, memory allocation. Multiple CPUs could be running code to access at same time
- If both do a `find_free()` at same time (before the other marks as reserved) and then return it, two processes could end up using the same memory (bad)
- Where to lock?
- You could lock the whole thing, but ideally want to lock



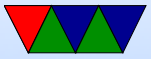
the smallest amount of code possible.

- Be sure when a lock is taken that all possible ways to exit the code release the lock, or could end up with deadlock
- Note that functions can be run by multiple cores as local variables end up on stack and each thread of execution has its own stack



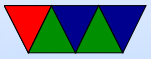
531-OS Locking Example – Code

```
void *memory_allocate(uint32_t size) {

    int first_chunk,num_chunks,i;
// Lock here?  (No, why?)
    if (size==0) size=1;
    num_chunks = ((size-1)/CHUNK_SIZE)+1;
// Lock here? (yes, why?)
    first_chunk=find_free(num_chunks);
    if (first_chunk<0) {
        printk("Error!\n");
// Unlock here? (yes, why?)
        return NULL;
    }
    for(i=0;i<num_chunks;i++) memory_mark_used(first_chunk+i);
// Unlock here? (yes, why?)
    memset((void *)(first_chunk*CHUNK_SIZE),0,num_chunks*CHUNK_SIZE);
// Unlock here? (no, why?)
    return (void *)(first_chunk*CHUNK_SIZE);

}
```



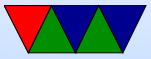
IPC – Inter-Process Communication

- Processes want to communicate with each other
- Two issues:
  getting the message across
  synchronizing



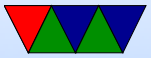
Linux IPC Methods

- There are nearly 20. Other OSes like windows also have a lot.
- Many are historical



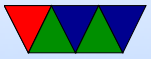
Linux IPC – Files

- Just write to a file, all can see it.
- What happens when multiple readers/writers?
- `fcntl()` syscall, with one of its many features being file locking
- For example: mail daemon writing mail to mailspool while your client is also trying to delete mail from the middle, what happens
- Things get exciting over network filesystems (NSF)

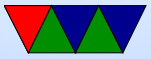


Linux IPC – Signals

- set up signal handler, sort of like a userspace interrupt.
- Can catch many things, such as segfault, control-C, control-Z (sleep), hangup
- A few user-allowed ones like USR1 you can send.
- Has many of the problems of interrupts (locking).
- Many functions are not signal safe.
- Crazy ways (setjmp) to exit signal handler without returning to where signal happened.
- Send with a kill() syscall (or kill/killall command line).

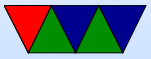


- What happens if system call interrupted?

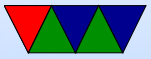


Linux IPC – Pipes

- Anonymous – parent opens fd, forks child, child reads in from fd. One way communication (half-duplex).
- `ls -la | sort | uniq`
- Linux actually has a `pipe()` system call
  - uses a magic invisible pipe filesystem
  - Creates two fds, one in, one out.
  - Write to in, appears on out.
  - There is a maximum size before it blocks waiting for other side to consume (10k or so?).

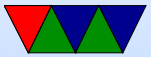


- Shell command line pipes
  - Shell forks children
  - Over-writes stdout of one to be a fd
  - Over-writes stdin of other to be same fd
  - Can use `close()` and `dup()` syscalls to set this up
  - Can use the `popen()` c library call to do something similar from C.
  - TODO: lookup how vmwOS does this



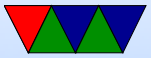
Linux IPC – Named Pipes (FIFOs)

- mkfifo creates a file on disk that's a pipe.
- Multiple writers can write to this file and it is queued up and then a process can read it.
- Processes that aren't a child can access these
- Can open nonblocking
- Also get SIGPIPE if write to a closed pipe.



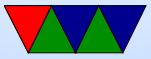
Linux IPC – Message Queues

- Sort of like a mailbox.
- Unlike pipes don't have to wait until other side connect (think phone-call vs text)



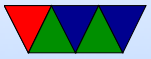
Linux IPC – SysV IPC

- Supports channeling (extra data that can be used to filter)
- Can read things out of order.
- Use `ipcs` to see.
- Use `ipcrm` to remove
- Use `msgset()` system call
- POSIX – supports priorities



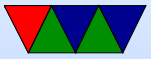
Linux IPC – Shared Memory

- Make a region of memory common between two processes
  - SysV – part of SysV IPC. Historical. Use `shmget()` `syscall`
  - POSIX –
  - Anonymous `mmap()` memory – how Linux implements POSIX? Can do this with a file, or anonymous if want to be visible only in process.



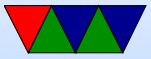
Linux IPC – Sockets

- Regular network sockets – can open a regular network connection to localhost (see ECE435)
- UNIX Domain Sockets – fast. Like network sockets, but local so without going over the network. Live on the filesystem (usually under /tmp or /var/run) so can have file permissions. Can send file descriptors across.
- Netlink Sockets – designed for fast communication between userspace and kernel. Also allow for broadcast in user to user



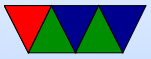
Linux IPC – Sockets

- Inotify – can monitor filesystem and notice when someone else changes a file



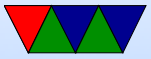
Linux IPC – FUSE

- FUSE – used for implementing a filesystem in userspace, but can also be used to communicate



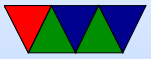
Linux IPC – Locks

- Locks can be thought of as a way of communicating between processes/threads
- SysV semaphores – Use `semget() syscall`
- POSIX semaphores
- FUTEX – kernel accelerated locking, used by pthreads() and such. You're not really supposed to use these manually.



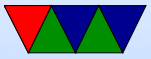
Speeding up IPC

- Splice – move data from fd to pipe w/o a copy? VM magic?
- Sendfile. zero copy?



IPC in the news / DBUS

- kdbus – dbus into kernel to make it faster
  - Desktop Bus, D-Bus
  - allows communication on a desktop bus between apps and kernel
  - example – incoming skype call can notify system, and apps like the audio adjust and mp3 player can pause music and set up microphone
  - multicast
  - example – battery low notification, apps can listen,



save state, prepare to shut down.

- Kernel developers resist it
  - Most because who has to maintain it?
  - Also how well designed is it? can it be used by other tools?
  - We already have a lot of IPC in the kernel, can it be made generic?
  - It is faster, but what if user programs just bloat to negate this?