ECE 598 – Advanced Operating Systems
Lecture 21

Vince Weaver

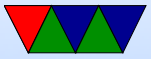
http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

17 April 2018

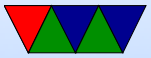
Announcements

- Still working on HW#10
- Second Midterm (non-cumulative) Tuesday May 1st



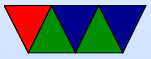
Project Updates

- No one wants to go a week early?



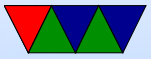
HW#10 Challenges

- Starting the CPUs not too bad (remember, just write address to jump to to mailbox, send SEV to wake them up)
- Need to set up Virtual Memory and caches on the CPUs. Why? Turns out you can't do locking (ldrex/strex) without the caches running. A pain, as how can you printk() for debugging if you need locking in printk()?
- Setting all the pagetable stuff correctly is hard. Need to have pagetables set with share bits, and also probably

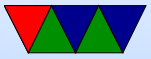


outer/inner cache writeback/allocate all set right Also need to set the SMPEN bit or else the core won't take place in sharing, but the way described for ARMv8 chips doesn't work (locks up in 32-bit mode?)

- Getting shared memory going a pain. Did all the above, had a volatile struct holding whether CPU ready, in theory the main core can start them one by one by waiting for the bit to go high in while loop.
- Turns out I forgot to disable hypervisor mode on all of the secondary cores
- Unclear how Linux does it, actual SMP setup on Linux

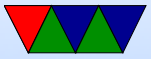


is in a bunch of hacked up assembly, function-redirect, and ifdef mess that is completely nuts to try to follow



Multiprocessor Scheduling

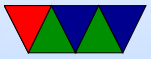
- Currently
  - Timer interrupt (or yield waiting for I/O comes in)
  - We scan the linked-list of processes seeing if any other process is ready to run
  - If it is, run it. If not, keep running current.
  - If no process is ready, run idle task
- Now
  - Timer interrupt: does the timer interrupt go to each core? Do you have separate timers? Do you have one



timer and it broadcasts an IPI to all cores?

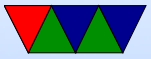
- Multiple cores inside scheduler at once. Is that an issue? Need locking.
- Each core looks at the list to see if anything ready to run.

- Issues
  - affinity – Ideally, a process stays on same core if at all possible.
    Maybe even have separate per-processor queue of jobs to run
  - smart scheduling – if a process has a spinlock held



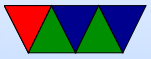
let it have a bit more time to clear so other processes aren't stuck on it

- space scheduling – a job needs say 8 threads, wait until 8 cores are available to run it
- gang scheduling – time and space scheduling if doing IPC with other processes, doesn't make sense to schedule the other side of these at different times



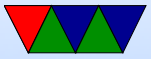
Another Locking example

```
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);

}
```



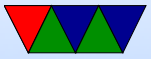
IPC – Inter-Process Communication

- 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.
- File – 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
- Signals –set up signal handler, sort of like a userspace interrupt. Can catch many things, such as segfault, control-C, control-Z (sleep), hangup, USR. Has many of the problems of interrupts (locking). Many functions

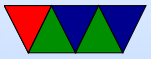


are not signal safe. Crazy ways (setjmp) to exit signal handler without returning to where signal happened. Send with a kill() syscall (or kill/killall command line). What happens if system call interrupted?

- 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 that uses a magic invisible pipe filesystem. Creates two fds, one in, one out. Write to in, appears on out. There is a

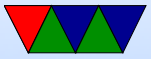


maximum size before it blocks waiting for other side to consume (10k or so?).

Shell creates processes, but over-writes the stdin/stdout as appropriate with the pipe fd.

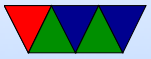
Use `close()` / `dup()` system calls to over-write the file descriptor. Can use the popen() c library call to do something similar from C.

- Named Pipes (FIFOs) – mkfifo creates a file on disk that's a pipe. Then multiple writers can write to this file and it is queued up and then a process can read it. Also a process that's not a child can access it.



Can open nonblocking. Also get SIGPIPE if write to a closed pipe.

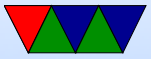
- Message Queues – sort of like a mailbox. Unlike pipes don't have to wait until other side connect (think phone-call vs text)
  - SysV – supports channeling (extra data that can be used to filer). Can read things out of order.
    Use `ipcs` to see. Use `ipcrm` to remove
    Use `msgset()` system call
  - POSIX – supports priorities
- Shared Memory – make a region of memory common



between two processes

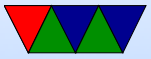
- SysV – part of SysV IPC. Historical. Use `shmget()` `syscall`
- POSIX –
- Anonymous `mmap()` memory – how Linux implements POSIX? Can do this with a file, or anonymous if want to be visible only in process.

- Locks
  - SysV semaphores – Use `semget()` `syscall`
  - POSIX semaphores
  - FUTEX – kernel accelerated locking, used by



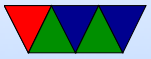
pthreads() and such. You're not really supposed to use these manually.

- Sockets
  - 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
  - Regular network sockets – can open a regular network



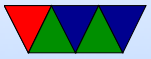
connection to localhost

- Inotify – can monitor filesystem and notice when someone else changes a file
- FUSE – used for implementing a filesystem in userspace, but can also be used to communicate
- D-Bus – a high level IPC mechanism used on the desktop. There have been many calls to kernel accelerate this



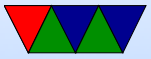
Speeding up IPC

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



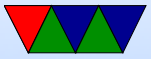
IPC in the news

- 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,



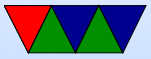
save state, prepare to shut down.

- 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?

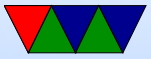


Worse is Better

- Worse is better / the right thing
- Richard Gabriel, 1989, made famous by JWZ sending it around
- New Jersey vs MIT
- UNIX/C vs LISP
- Is it better to spend lots of time coming up with a perfect specification on paper, then implement it?
- Is quicker/easier to understand better than complex and perfect?



- Should you come up with something "good enough" and let it grow naturally?
- Is it possible to ever design a perfect interface?
- Original example
  - PC loser-ing problem
  - Signal comes in during a system call. Right thing would be to somehow back everything out, return to userspace, and restart, all transparently to the user
  - UNIX example was to just cancel the syscall and return an error
  - This means that if you have bad luck on UNIX any



system call can fail. You should check each for EINTR

- Pretty much no-one does this
- Is it right to make the kernel simpler/easier to understand at the expense of userspace?
- Eventually "the right thing" was done, and you can add SA_RESTART to your syscall to automatically restart