ECE 598 – Advanced Operating Systems
Lecture 24

Vince Weaver

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

vincent.weaver@maine.edu

28 April 2015

Announcements

- HW#8 was posted; very short multiple choice, due Thursday before class
- HW#9 will be coding, but simple.
- There will be no HW#10 in the end

Review of the HW#9 Code

- Currently we have a glorified bootloader.
- What was needed to get it running applications?
- Needed to move our "shell" to a separate application.
  - Needed to write a C library
  - C library had to use only syscalls (no longer can just call kernel routines, printk, etc).
  - Compile a -fPIC binary. Use objcopy to create raw binary. No flat file format, too hard.

- Include it as a binary blob in the executable (use xxd -i and #include it)

- Need to load our shell
  - Need to allocate memory!
  - Detect memory at boot, break into chunks
  - Have a memory free bitmap for all of memory
  - Mark the kernel as reserved
  - Have an allocate routine that finds free memory.

- Need to execute our shell
  - Copy the blob to the memory we allocated

- Allocate a userspace stack for it to use
- Point the userspace PC to the memory we allocated
- Switch to userspace!

- Now we have something about as powerful as DOS!

- What was done next?

- Need to go multi-tasking
  - Most issues have to do with stack being wrong. Really hard to debug w/o memory protection, and mostly works while silently corrupting things.

– Another issue was properly saving/restoring registers on context switch. Turns out to be a pain to do this in C, had to re-write the IRQ handler to be all in assembly
– Need to speed up timer interrupt
Not strictly necessary, currently
– Implement system time
time syscall
– Need to set up process table and have list of processes
Done
– Need to implement scheduler to switch between them
Done, simple round-robin

- Need way to make process runnable.
  Custom run syscall, also stop syscall.
- Switching to userspace leaks stack, how to avoid that.
- Idle task that just runs `wfi` forever. Need to have something to run if all other tasks are blocked.

- Still TODO
  - Properly only schedule idle task when idle
  - Blocking I/O
    How to implement?
  - Interrupt driven serial port

– PS/2 GPIO keyboard support
– Implement nanosleep system call
– Some sort of filesystem
– Exiting a process (and freeing memory)

Multi-Processing

- Multi-processing
  Symmetric, Asymmetric
  SMP vs CMP (Symmetric and Chip Multi-processing)

- Multi-threading
  (Hyperthreading, SMT)

- Bus (small amounts) – for memory just puts request on the bus. If busy it waits. Why can this be bad if large

CPUs?
Cache – each CPU has local cache. Have to keep cache coherent though. Large (¿16?) traditional cache coherence doesn't scale well. Then use crossbars, switching networks. Gets more complex.

- Shared memory vs Distributed
  Shared memory, a CPU can write a value to memory, read it back and it will be different (another CPU can write to it)

- UMA, NUMA, CC-NUMA (cache-coherent)

Non-uniform memory access

- How many copies of the OS? One per core or single image? One per core is more like a cluster.

Multi-Processor Resource Sharing

- How are resources shared in SMP system?
- Any core can access any of the devices. Need locking.
- What about interrupts?
  - Have one core handle all interrupts?
    Might have better cache behavior
  - Round-robin interrupts to each core?
    Reduces load on core0 but hurts others.

– Balance interrupt load across processors?

OS Support for SMP

- How can we have multiple cores share one OS-image?
- Big-kernel-lock, but poor performance
- Only parts of OS happen at once. Scheduler can run at same time as say serial driver or filesystem read or page fault
- Split up with fine-grained critical sections.
- Suddenly deadlocks are a problem.

- What kinds of locks?
  - Spinning easiest, but poor performance.
  - Switch threads. Multi-threading OS?
  - Linux has kernel threads (look in top for things starting with k or rcu). Interrupt handlers have fast handler and worker threads.

SMP Scheduling

- 4 processors, 5 jobs
  How to avoid ping-ponging? Better to make two processes slow or all of them?

- Gang scheduling – if you have processes that are using IPC (or multithreads) you want to schedule all at the same time so can communicate without having to wait through multiple context switches.

- Keeping jobs on same CPU started on (why is this

good?) Cache behavior. TLB, NUMA.
Why might you want to move them?

- When might you want to run everything on one core even though lots available? Power! Can put rest of CPUs to sleep.

- How do you online/offline hotswap processors.

Initializing SMP on ARM

- Detecting the processors
- Need to power them up
- Then need to somehow (implementation dependent) set the PC for each
- Typically leave them waiting in WFE (similar to WFI but also will wait for SEV event). SEV sends event to all cores waking any in WFE state.

- On x86 IPI (inter-processor interrupts) are used during bringup