ECE 531 – Advanced Operating Systems
Lecture 15

Vince Weaver

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

`vincent.weaver@maine.edu`

24 October 2023

Announcements

- HW#6 will be posted
- Midterms will be graded by Thursday

Notes on Waitqueue from Last Time

- We'll overexplain things a bit

Process Control Blocks

- Each process has some sort of process control block structure that holds all the info in a process
- There is a “processes” linked list that can be iterated by the OS to find processes
  - The scheduler will do this
  - The OS might have other reasons to look for a process struct (killing it, etc)

Process Control Structure

Here are the things you might find in a PCB, others are possible. This list is from vmwOS

- Saved State
  - Register saved state (for context switch, r0..r15, spsr)
  - Kernel saved state (maybe needed to restart process blocking in I/O)
- Linked list pointers (note, fancy kernels might use better data structs)
  - Next/Prev Process linked list pointers

- Waitqueue linked list pointer

- Process info
  - status (valid/running)
  - time accounting (user/kernel)
  - pid – process id
  - name – for printing
- Parent info
  - exit value, parent pointer – when program ends it needs to stick around until parent acknowledges it and gets the exit value
- Memory info – useful for virtual memory during page

faults

  - stack pointer, size
  - text pointer, size
  - data pointer, size
  - bss pointer, size

- Open file info
  - open_files array
  - current working directory

Open Files Array

- Each process tracks its open files
- Indexed via the filedescriptor
- This is UNIX/Linux so “everything is a file”
- Offset/inode/count/flags/name
- Also VFS struct with function pointers to the driver responsible for I/O
  - read()
  - write()
  - llseek()

- getdents()
- ioctl()
- open()

Wait Queue

- There is often separate waitqueue linked lists that can hold everything waiting on a certain kind of I/O
- The process itself doesn't move, it just is added/removed from these lists as needed
- Often waitqueue is per device, so you'll have a serial one, a disk one, a network one, etc

Wait Queue Example – Console Code (vmwos)

- your code read(stdin,buffer,size)
- syscall looks up file descriptor, sees fd maps to console
- calls console_read()
- console driver has a buffer that gets filled in the background by serial port data
  - if more data avail then requested, return that many bytes
  - if less data avail then requested, return all available

    - if no data available, put on waitqueue for console data
- next time serial port sends more data to console, it will put in buffer, but also wake up everyone in waitqueue
- that marks each process is READY and then removes all from queue

HW#6 Troubles

- Memory detection in device tree changed in the 2020-2021 time frame
  Found someone reporting issue 2 years ago online (was me)
- Lots of corner cases, especially on machines with more than 1GB handled properly (non-contiguous, cheat for now)
- Other issue was just general Pi4 working, mostly interrupts

- Also issue with Flattened Device Tree breaking with GCC 12 which I think most of the cross compilers are

Virtual Memory Wrapup

Quick run-through, the path of a load

- OoO, load buffer, etc
- VIPT. So on access it looks up the physical tag in TLB while reading out the tags from each way with the index. Also keep in mind MESI is going on at this level.
- If tag from TLB matches a tag from cache, hit! Good! Cache hit!
- If tag in TLB but not in cache, cache miss.
- If tag not in TLB, TLB miss. Won't know if cache hit until later.

- Now let the hardware walk the page tables.
- If hardware finds the page, great! Return it back up to the TLB
- If hardware can't find the page, time to get the Operating System involved. Page fault.
- OS has a list of what should be in memory where (from the executable). Typically these are demand-loaded
  - Text/code — read from disk
  - Data — read from disk
  - BSS — allocate zeros
  - Stack — if near top growing down, auto-grow

- ◦ Heap — similar to stack
- ◦ Shared page — could already be in memory (shared lib?) Just need to point to it.
- ◦ Zeros — just have one page of zeros you can point to
- ◦ Paged out to disk — have offset in page file, need to load it
- ◦ What if page is invalid/not part of process? segfault!

- Time to bring in the page! Need to find room in Physical RAM. If no room, need to make room. Possibly paging out to disk (this is what LRU/dirty bits are used for).
- What kind of issues come up when low on RAM and

constantly paging same pages in and out (thrashing?)

- Page now in physical RAM, time to go backwards. Update the page table
- Fill in the TLB. Return to memory.
- If page fault occurred, usually re-execute the instruction.
- Issues
  - Could you have race where you re-execute it and the page had gotten swapped out again?
  - Can we page out the page tables? What can go wrong there? Double faults? How many nested page faults

Aside: what if you have unused bits at top?

- People use them, causes problems later. See M68k/MacOS, IBM 390, ARM1
- AMD64 canonical addresses to avoid this (top bits have to be all zeros or all ones)
- Though recent systems support this, have a special mode to "ignore" top bits
  - ARM64 Memory Tagging Extension (MTE), Top Byte Ignore (TBI)
  - AMD64 Upper Address Ignore (UAI)

◦ Intel Linear Address Masking (LAM)

Large Pages

- Another way to avoid problems with 64-bit address space
- Larger page size (64kB? 1MB? 2MB? 2GB?)
- Less granularity. Potentially waste space
- Fewer TLB entries needed to map large data structures

Transparent Huge Pages

- Compromise: multiple page sizes.
  Complicate O/S and hardware. OS have to find free blocks of contiguous memory when allocating large page.
- Transparent usage? Transparent Huge Pages? Alternative to making people using special interfaces to allocate.

Having Larger Physical than Virtual Address Space

- 32-bit processors cannot address more than 4GB
  x86 hit this problem a while ago, ARM just now
- Real solution is to move to 64-bit
- As a hack, can include extra bits in page tables, address more memory (though still limited to 4GB per-process)
- Linus Torvalds hates this.
- Hit an upper limit around 16-32GB because entire low 4GB of kernel addressable memory fills with page tables

ARMv7 Virtual Memory

- ARM virtual memory is *really* complicated
- It's a lot more complicated than x86
- TODO, write this up better

ARMv7 Page Tables

- ARMv7 supports two pagetables, one for kernel-type thing that's fixed and always there, one for process that you can swap in/out
- Pagetable has lots of fields
  - Address (31-20)
  - NS - not secure
  - nG - not global
  - S - shared
  - AP[2:0] access permissions (kernel r/w, kernel r/o,

anyone r/w, anyone r/o, no access)

- TEX - caching (cacheable, no cache)
- Domain - up to 16 domains with different checking permissions

ARMv7 VM on our OS

- Our OS we set up a 1:1 Virtual to Physical "Section" Mapping with 1MB pages
- Set up a pagetable, 4k table that is 14-bit aligned
- Setup pagetable, point to it
- Setup domains (want 0x55555555 not 0xffffffff or won't check)
- Flush TLB/Caches?
- Enable MMU

ARMv7 Caches

- Caches are small, fast memories that mirror parts of DRAM for speed
- Important for performance, really hard to set up on ARMv7