ECE 531 – Advanced Operating Systems
Lecture 16

Vince Weaver

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

vincent.weaver@maine.edu

31 October 2023

Announcements

- HW#6 was posted

Hand Back and Go Over Midterms

- Average was an 83

Describe Project Topics

- Don't forget due soon
- There's a list at the end of the project pdf
- Note that especially in this class it can be hard to judge difficulty. Things that might sound easy can turn out not to be.
- Note that "this was much harder than we thought and we only got it partly done" can be an OK result as long as you document along the way what you did and what parts ended up being difficult

Operating System without Virtual Memory

- Can you have an operating system without Virtual Memory?
- Historically, yes, many did (original UNIX, MacOS, Win3.1, etc)
- Linux typically relies on MMU (virtual memory).
- uCLinux is a version of Linux for micro-controllers without VM
- Our OS in the homework is similar in design to this.

Starting a Process on Linux/UNIX style operating systems

fork()

With VM you can run `fork()`

- `pid_t fork(void);`
- Makes complete copy of parent, only pid is different
- Return value is pid of child (in parent) or 0 (in child)
- On VM systems does this with Copy-on-Write (COW) magic

vfork()

Can't fork w/o OS, instead do a `vfork()`

- vfork() makes a new process (PCB) but doesn't allocate memory, the child runs with the parent's memory/stack
- The parent is put to sleep until the child finishes
- the *only* thing the child is allowed to do is either call exec or _exit() (not even plain exit() as that does various cleanups that could break the parent)

vfork() on ECE531 OS

- Complicated
- Allocates new process with `process_create()`
- Copies current process block over, including stack state (Did I mention, each process has own kernel stack, with the PCB taking up the top part of it)
- Update the pid values, parent info, also increment file-in-use counts
- point kernel stack info to that of child
- put child in process linked list

- put parent to sleep and force a reschedule (then return pid of child)
- child returns 0, and is *only* allowed to execve() or _exit()

Loading an Executable

Executable Formats

- ELF is complicated for what we need and hard to parse
- HW#6 uses bare executables made with `objcpy` but that has limitations, the main one being you can't use BSS variables
- A good compromise is bFLT

Flat File Format

- http://retired.beyondlogic.org/uClinux/bflt.htm

- “bFLT” or 0x62, 0x46, 0x4C, 0x54

-
```
struct flat_hdr {
    char magic[4];
    unsigned long rev;          /* version */
    unsigned long entry;        /* Offset of first executable instruction
                                   with text segment from beginning of file */
    unsigned long data_start;   /* Offset of data segment from beginning of
                                   file */
    unsigned long data_end;     /* Offset of end of data segment
                                   from beginning of file */
    unsigned long bss_end;      /* Offset of end of bss segment from beginning
                                   of file */

    /* (It is assumed that data_end through bss_end forms the bss segment.) */
    unsigned long stack_size;   /* Size of stack, in bytes */
    unsigned long reloc_start;  /* Offset of relocation records from
                                   beginning of file */
    unsigned long reloc_count;  /* Number of relocation records */
    unsigned long flags;
    unsigned long filler[6];    /* Reserved, set to zero */
};
```

bFLT documentation

- Spec isn't worth much
  Your best bet is various Wikis and blog postings (TI-nspire?)
- Actual code more useful
- `fs/binfmt_flat.c` in Linux kernel source.

Building bFLT

- Making the binaries hard.
- Not just a simple matter of telling gcc or linker (would need a linker script, as far as I know doesn't exist. Good project idea?)
- There was an "elf2flt" project but non-standard and hard to find a working version
- I wrote my own for this class. It broke recently when the cross compiler moved from gcc-7 to gcc-12, trying to fix it for next homework

exec()

- exec() is the syscall that loads a binary
- We use `execve(char *pathname, char *argv[], char *envp[])`
- First argument is the full path of the executable
- The second is the command line argument structure. (TODO: diagram).
  - This is an array of pointers (with the last entry NULL) that point to each command line argument.
  - It's up to the caller to set this up properly, often the shell

  - Remember argv[0] is the name of the program
- The third argument is similar to the command line args, but for environment variables

Loading a flat binary (531 OS)

- execve()
- Remember, exec() overwrites current process so the process has already been allocated and put in the process list
- TODO: should check permissions, executable bit
- read header — note big-endian so need ntohl()
- parse header, check magic number (falls back to raw executable if not bFLT)
- parse header, get sizes

- allocate stack
- allocate area for text/data/bss
- read data from disk into allocated space
- relocate the addresses
- set the name
- construct the command line arguments (these were passed to execve, but we need to copy to final location on stack)
- set up register save state to appear as if we had a context switch

  saved_sp should be allocated stack pointer just after argv

saved_pc should be entry point (?)

r0 = argc, r1 = argv

PIC/PIE – Position Independent Code

- Without VM, we can't know in advance where our code is loaded

  Something else might already be there
- Instead of loading from absolute address, uses an offset, usually in a register or PC-relative.
- gcc has an option `-fPIC` to generate

Relocation

- List of offsets to pointers
- PIC compiles things with zero offset
- At load time the pointers are fixed up to have the load address
- Separate relocation for GOT (global offset table) which is a list of pointers at the beginning of the data segment, ending with -1

Flat Shared Libraries

- Like mini executables, can have up to 256 of them
- Libraries loaded in place, then the callsites are fixed up to have the right address.
- Also at start time the various library init routines are called

Execute in Place

- On embedded system, might want our read-only text segment in ROM
- Why? Save space, save copying.
- Why not? ROM often slow, more complicated binaries (data not follow text)

Memory Allocation w/o Virtual Memory

- sbrk() doesn't work (due to fragmentation can't have an auto-growing heap)
- can use mmap() to allocate chunks
- overhead

Other Things our OS Does

Create an Idle Task

- Can fake up a process but it lives in kernel not userspace
- What does the system do if no jobs are ready to run?
- wfi vs msr (ARM1176 wfi is a nop)
- What happens if forget to setup a stack for the idle task? Not an issue unless you try to add a printk to track down a problem

Scheduler / Idle Thread

- How does the scheduler work?
- Simple, nothing fancy. There's a doubly linked-list of all processes and when a timer interrupt happens the list is walked to find the next one that's runnable.
- What if none available? Then run the idle thread.

Waitqueues

- See last lecture
- If you are sleeping (because of a vfork) or waiting on I/O (waiting for keypress) you get put to sleep and put on a linked-list waitqueue. Then when I/O comes in, you are woken up, removed from the queue, and marked as ready.
- This is tricky as in theory you are sort of sleeping in the kernel and that's how we implement it, so we need to save our kernel register state as well as the user space.

There's probably better ways to do this.

Aside on the Shell

- TODO: diagram
- Shell in main loop reads string and parses
- If not built-in command, assume you're running an executable
- Run vfork() so now two copies of shell
- In the child, try execve() on the program name
  - If it works, we've started a new program
  - If it fails, _exit()
- By default this would launch a program in the

background, how do we instead wait for the child to finish? waitpid()

- Then return to reading from loop

waitpid()

- waitpid() call will block waiting for child to finish
- When children run exit() they wait until parent has acknowledged their return value before completely exiting/freeing resources
- How do you acknowledge background tasks (started with ampersand in bash-type shells)? Non-blocking waitpid(NOHANG) needs to be run occasionally to reap finished children

Other Process Terms

- Orphaned children, what happens when parent dies?
  In theory init inherits them
- Might you intentionally orphan a child?
  How do you start background daemons?
- Zombie processes, ones that for whatever reason you can't kill (often they are stuck waiting for the kernel to do something that will never finish, for example reading from a disk that was removed)