ECE 531 – Advanced Operating Systems
Lecture 14

Vince Weaver

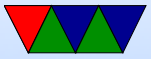
https://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

3 October 2025

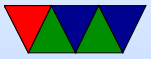
Announcements

- Homework #3 grades sent out
- Don't forget Homework#4 due Monday
- Homework #5 will be posted



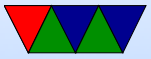
Static vs Dynamic Libraries

- Static: includes all code in one self-contained binary
  - This includes all libraries, including C library
  - Can make your executable 100x bigger
- Dynamic: library routines linked at load time
  - Smaller binaries (less disk and RAM)
  - Can share centralized versions of libraries

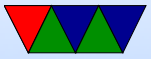


Static / Dynamic Tradeoffs

- Lots of debate about what is better
- Some people want Debian like where packages do not include common libraries
- Alternative is macos/windows/flatpack style where packages can include all libraries
- Benefits of static: no "DLL Hell" where you depend on unavailable library
- Downside: lots more disk space
  Also security, when a library has security bug need to

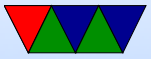


replace all apps rather than just update one central library



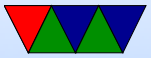
How Executable Loading Works (Linux)

- For more info on how this all works on Linux I recommend reading the following articles
- `https://lwn.net/Articles/630727/` How programs get run
- `https://lwn.net/Articles/631631/` How programs get run: ELF binaries
- And maybe `https://lwn.net/Articles/604515/` Anatomy of a system call, part 2

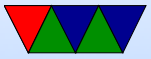


Linux – Getting to Processes

- Kernel Boots
- `init` (process 1) started, switch to userspace
- `init` calls `fork()`
  creates a child that is exact copy of init
- child calls `exec()`
- Kernel checks if valid ELF. Passes to loader
  Possibly not ELF. Shell scripts (Linux can also have custom file loaders for things like java, cross-arch via qemu, etc.)

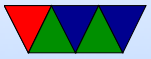


- Loader loads it.
- Program runs until complete (exit/return)
- Parent waits for child to finish and once parent notified, process can be freed
- Usually init starts up login and then a shell, or, alternately, the GUI interface. These are all started with fork/exec though with the original parent being init.



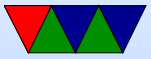
Linux Process Management – fork

- `fork()` used to create exact copy of program
- `pid=fork()`
- If `pid` is a number, we are in parent and it's the child's process ID (pid)
- If `pid` is zero, it means we are the child
- If trying to launch a new program we will immediately call `exec()` if pid is 0



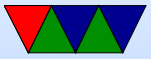
Linux Process Management – notes on fork

- Why split up process creation like this? In theory it lets user code do some setup of the new process before exec, otherwise the kernel does it all and the call to exec ends up having huge numbers of parameters
- Not all of the state is copied over on fork() but it's complicated what is and isn't
- On systems with virtual memory often no copying happens at all, rather read-only COW mappings are set up



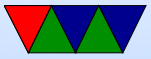
Linux Process Management – exec()

- `exec()` replaces currently running process with a new one from disk
- This also involves shutting down a lot of stuff that was part of the old executable
- On Linux lots of variants: execl, execlp, execle, execv, execve, execvp, execvpe
- Note: Linux kernel only implements `execve()` and `exevceat()` the rest are made by C library
- l/v: l means command line arguments passed one by



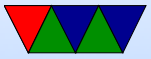
one, where v they are passed as a single pointer to a vector

- e: e means include a pointer to environment variables
- p: p means search the PATH for the executable



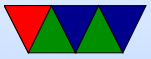
Linux Process Management – exiting

- Some process control of children gets a bit morbid
- Processes can exit with `exit()` or `exit_group()`
- They can also `return` at the end of `main()` which will also call exit
- You can also run `kill(pid)` to force it to exit from another process
- Processes can also exit if some sort of error/crash happens



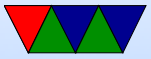
Linux Process Management – wait()

- Parents are supposed to wait()/waitpid() for their children to finish
- wait() will tell you the exit status of the child
- If a parent doesn't wait, a child can become a "zombie" (it is dead but it can't go away because its parent is ignoring it)
- If a parent exits before a child process, the child becomes an "orphan" and generally init (process 1) will inherit all the orphan processes



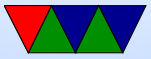
Processes (from the kernel side)

- We have so far been looking at processors from the userspace interface
- What do things look like to the kernel?
- What is a process?
  - It is the program running in memory
  - It's also all the state the kernel has to store in addition to keep it running



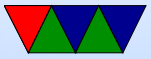
Kernel Process State

- Hardware state (for context switch)
  - registers (r0-r14), PC, status register
  - Floating Point / Vector? Performance Counters?
- Software/OS State
  - pid (process id), uid (user id)
  - Memory ranges, stack location, page tables
  - Process accounting / time stats
  - Open files (all open files, file offsets, etc)



Process Control Block

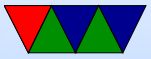
- The structure that holds all the info on a kernel process is sometimes called the Process Control Block (PCB)
- All of the processes in the system are usually kept together in some sort of data structure (often an array or linked list)



Internally How do you Make a Process

`create_process()` or similar (used by `fork()`

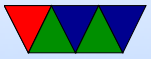
- Allocate memory for a new process structure
- Initialize it
- Assign a process ID
- Insert into current process array or linked list
- Allocate a stack
- Initialize the registers
- If called by fork() copy over the info from parent (including stack contents?)



Loading an Executable

`load_process()` or similar (used by `exec()`

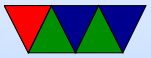
- Clear out any previous program state (if coming from fork)
- Load executable from disk
- Parse the executable headers
- Allocate memory for machine code and data
- Allocate/zero the BSS memory
- Set up the saved program counter to point to entry point



Freeing a Process

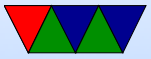
`delete_process()` or similar

- Used by exit() / exit_group()
- Close all open files
- Free all memory
- Possibly let parent know and wait until acknowledged
- Pass exit/return value back to parent
- Remove process from list

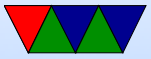


Kernel Process Creation

- Involves assembly language trickery
- Kernel can create threads
  - idle thread, pid 0, often does nothing (might halt/wfi to enter sleep mode when nothing else running)
  - Linux has a bunch of kernel worker threads it creates to help out with things, that look like processes but are parts of the kernel
- Kernel also does a bit of extra work to get init going because it has to be started in kernel space before

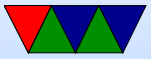


switching to user space



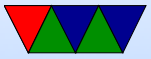
Kernel Process Creation

- First set up user registers. How do you do this from kernel/supervisor mode? Tricky, ARM created a special "system" mode (user+permissions) to make this easier.
- Set up stack
- Set the SPSR and link register to act as if we were returning from an exception, but with the return address the start of our user program.
- Return



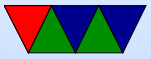
How Dynamic Linking Works

- ELF executable can have interp section, which says to load `/lib/ld-linux.so` first
- This loads things up, then initialized dynamic libraries.
- Links things in place, updates function pointers and shared variables, offset tables, etc.
- Lazy-Linking is possible. Function calls just call to a stub that calls into linker. Only resolves the link if you actually use it. Why is this a benefit (faster startup, not load things not need). Does add indirection every time



you call.

- Can use `ldd /bin/ls` to see what dynamic libraries a program is using



More info on Loader

- TODO: more info on this
- `/lib/ld-linux.so.2`
- Can launch dynamic libs on this
- Also gcc -static compiled. Possibly because it includes dynlib() (name resolver?)
- Hand-assembled assembly calling only syscalls will not load with this