Today's plan

- Minix Exec
- Linux Modules
- Signals

Minix `exec`

- part of exec is implemented in the exec library call, `lib/posix/_execve.c` (or `minix/lib/libc/sys/execve.c` in the latest version)
  - details in .../`stack_utils.c`
  - this builds the initial stack in a buffer
    - allocated using the `sbrk` system call
  - it might be dangerous to trust a user library to build the stack right, but the stack is (over)writable by the user code anyway, so there is no loss of security

Minix `exec` implementation

(for the version of Minix in the textbook, from line 18744)

- check to see that the stack size is reasonable
- copy the stack into an internal buffer (before freeing the memory)
- check to see that the file is accessible, by asking the file system, including using system calls -- the file system server has some special code to make file system calls from the PM look like they are from the process itself
- read the executable file header (line 18886), which also checks to see whether this is a script (line 18938)
- scripts require executing a different file, with this one as an argument (line 18801)
- look to see if the text can be shared (`find_share`, line 19253)
- allocate the new memory (19019 on p. 891, called from 18812 on p. 887) -- this is the commit point, if this succeeds, we can never return, because there is no longer an old process image to return to
- relocate the stack (see below -- patch_ptr, not patch_stack) and copy it to the newly allocated stack segment
- read in the text (unless shared) and the initialized data
- change the effective user/group ID if executing a setuid/setgid file
- set default signal handlers
- ask FS to close "close-on-exec" files
- ask the system to initialize the new stack pointer, with the initial valid return address and to make the process ready to execute (`sys_exec`, line 18878, and `kernel/system/do_exec.c`, line 10300)
- return, no need to send any messages

Minix `exec` allocation

- `new_mem`, line 18977
- compute needed sizes, all in multiples of clicks (look at the arithmetic carefully -- ceiling computation)
- make sure sizes are reasonable, e.g. the data+bss segment does not overlap the stack segment (gap < 0, line 19016)
- allocate the new memory -- if not, fail. This is overly conservative, since it may be that the new memory would be available if we returned the old memory first, but if we return the memory first, we will not be able to fail.
- free the current text (unless shared) and data + bss + stack segments
- initialize the three segment descriptors (T, D, and S)
- ask the kernel to map these segments
- clear the uninitialized parts of these segments

Minix `exec` stack relocation

- `patch_ptr`, line 19071
- stack built by user library assumes the initial stack pointer will be zero
- the initial stack pointer is not at virtual address zero -- the virtual address depends on the size of the text, data, bss, gap, and stack segments
- PM assumes that anything in the initial stack is either a 0 -- a null pointer, used e.g. to mark the end of an arguments list -- or a relative pointer
  - starting from the top of the stack
  - patch_ptr adds the segment base to any nonzero entry (pointer)
  - until it has seen two null pointers, one to terminate the end of the args list, the other to terminate the environment list
- sanity check is done to make sure this update doesn't run off the end of the stack

C program execution

- when a C program begins execution, it starts at a C runtime system call `crtso`, whose entire function is to call main
- `crtso` pushes the addresses of the three arguments to `main`: `argc`, `argv`, and `envp`, then calls `main`
- the final stack is as in Figure 4-38d (page 436), remembering that the stack grows downward

Linux Modules

- exec and the stack relocation are somewhat similar to what is needed for a module
- a module is a loadable piece of code that executes in kernel space
- the module must be relocated when loaded, or must be position-independent code, since it is loaded dynamically at an arbitrary location in kernel space
- entry points for each module must be recorded inside the kernel, for example an initialization routine which sets everything up, including e.g. interrupt handlers and read/write functions
- however, a Linux module is not a separate process/task, so it does not need its own stack -- it executes on the kernel stack that is active when it is called
- if the kernel is multithreaded, however, the module must be coded in a thread-safe way, e.g. locking global data structures before modifying them
- because it executes with kernel privilege, a module can be a correctness or security risk
- modules make it much easier to install new drivers, since no kernel recompilation is needed

Unix Signals

- any process may set a signal handler for a given signal

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

- the argument to the signal handler is the signal number, so the same signal handler can handle multiple signals
- the return value of `signal` is the old signal handler
- the signal handlers default to `SIG_DFL`, which:
  - aborts if the signal is HUP, INT, PIPE, ALARM, TERM, USR1, USR2 (and other non-Posix signals such as POLL)
  - aborts and dumps core if the signal is QUIT, ILL, ABRT, FPE, SEGV, etc
  - returns (ignoring the signal) if the signal is CHLD, URG, or others
- See `signal(7)` for details
- the signal handlers can also be set to SIG_IGN, which ignores the signal (except KILL and STOP cannot be blocked or ignored)
- "Unix" signal handling varies among systems, but typically the signal is blocked during execution of the signal handler

sigaction: another way of handling signals in Unix/Posix

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int,
           siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer)(void);
}
int sigaction(int signum,
    const  struct sigaction  *act,
    struct sigaction *oldact);

int  sigprocmask(int  how,
     const  sigset_t *set,
     sigset_t *oldset);

int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

- `sigaction` is defined portably across Unix-like systems
- the second parameter (if non-null) is used to define a signal handler, either a sa_handler or a sa_sigaction, but not both
- the `sa_handler` can be set to SIG_DFL or SIG_IGN
- the mask specifies which signals should be blocked during execution of the handler
- the flags can be used to specify
  - whether we are specifying a sigaction or a handler
  - that certain signals should be ignored (e.g. child stop signals)
  - whether the signal handler should be permanently installed, or executed at most once
  - whether system calls should continue across a signal
  - whether further instances of the signal being handled should be deferred while the signal handler is executing
- `sigprocmask` lets us block or unblock the given signals
- `sigpending` returns raised signals that are currently blocked
- `sigsuspend` suspends the process after temporarily installing the given signal mask

Signal Handling and System Calls

- a process making a system call is suspended (in Minix, blocked on receive)
- what is the appropriate behavior of a long-running system call (e.g. read from a pipe or a socket) when its process gets a signal?
- some OSs think the system call should be ended with errno = EINTR, so the caller can (or must) call it again
- some OSs execute the signal handler while the main part of the process is blocked on the system call -- a system call return then has to make sure it is returning to the right place
- Linux and Solaris (at least) give a choice via the `SA_RESTART` flag (in `sigaction`), which if set, means slow system calls are automatically restarted
- Minix completes/aborts the system call, by sending it a message with result code EINTR

- how about other OSs?