Recitation 10

What’s Next???

Table of Contents

1. What have you done so far?
2. Next steps for the Scheduler
3. Next steps for the File System
4. Final Integration

Milestone 1: You should be able to...

Scheduler:

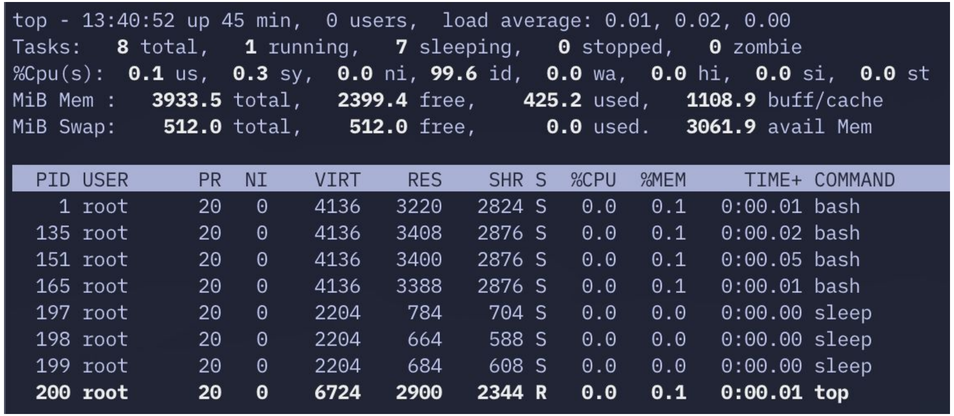
- Schedule processes with different priorities
- Send your own signals using k_process_kill
- Use the logger to debug

File System:

- mkfs and mount a file system
- Interact with the file system
    - What happens when you create a file? When you write to a file? ls? chmod?

Scheduler: Things to double check

- If all queues (-1, 0, 1) are empty, it should schedule “idle” process
  - Any empty queues should *not* be scheduled
  - Blocked/stopped processes should also *not* be scheduled
- p_waitpid should check for a state *change*
  - I.e. it’s not enough to just check if child’s state != Running
- p_waitpid, p_kill, etc. should error if provided pid is *not* a child process
- Check logs + `top` for scheduling



```
top - 13:40:52 up 45 min,  0 users,  load average: 0.01, 0.02, 0.00
Tasks:   8 total,   1 running,   7 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.3 sy,  0.0 ni, 99.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3933.5 total,   2399.4 free,    425.2 used,   1108.9 buff/cache
MiB Swap:    512.0 total,    512.0 free,      0.0 used.   3061.9 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    1 root      20   0    4136   3220   2824 S   0.0   0.1   0:00.01 bash
  135 root      20   0    4136   3408   2876 S   0.0   0.1   0:00.02 bash
  151 root      20   0    4136   3400   2876 S   0.0   0.1   0:00.05 bash
  165 root      20   0    4136   3388   2876 S   0.0   0.1   0:00.01 bash
  197 root      20   0    2204    784    704 S   0.0   0.0   0:00.00 sleep
  198 root      20   0    2204    664    588 S   0.0   0.0   0:00.00 sleep
  199 root      20   0    2204    684    608 S   0.0   0.0   0:00.00 sleep
  200 root      20   0    6724   2900   2344 R   0.0   0.1   0:00.01 top
```

Scheduler: Next Steps

- Sleep process (if not already implemented)
  - Should not consume CPU
  - Should work with multiple sleeping processes
- Add mounting of file system
  - E.g. `./pennos fatfs [schedlog]`
  - Keep track of file descriptor tables for each process you spawn
- Implement W_WIF...(status) macros for p_waitpid if you haven’t already
  - Will be needed for shell
- Replace read(), fprint(), etc. with f_write(), f_read(), etc.
- `nice` command to change process priority

File System: Next Steps

- Each process will have an “open file descriptor table”
    - Reserved fds for STDIN, STDOUT (at least)
- Somehow globally keep track of currently open files and their permissions
    - FILE structs?
    - Linked List?
- A user level program should be able to write to stdout using the same interface as it would write to a PennFAT file.
- **If a user level program is calling read(2), then you are doing something wrong.**

File System: System Calls

- Your own system calls!
- Mimic the behavior of C system calls in <cstdio.h> library
  - https://cplusplus.com/reference/cstdio/
- Calls to STDIN, STDOUT or your PennFAT filesystem

File System: System Calls Example

```
$ cat
```

What should be done?

1. Create a process for cat
2. Read from STDIN
3. Write to STDOUT

What system calls should be used?

f_read(int fd, int n, char *buf), f_write(int fd, const char *str, int n)

File System: File Corruption

- Write-write lock
  - If a process opens a file with write permissions, any other processes will be blocked (call error) from opening the same file with write permissions
  - Processes can read, though
- Cannot remove a file that is currently being used by another process
  - Make use of the flag name[0] = 2
  - What happens to the open FILEs struct?
  - How about the file descriptor table?

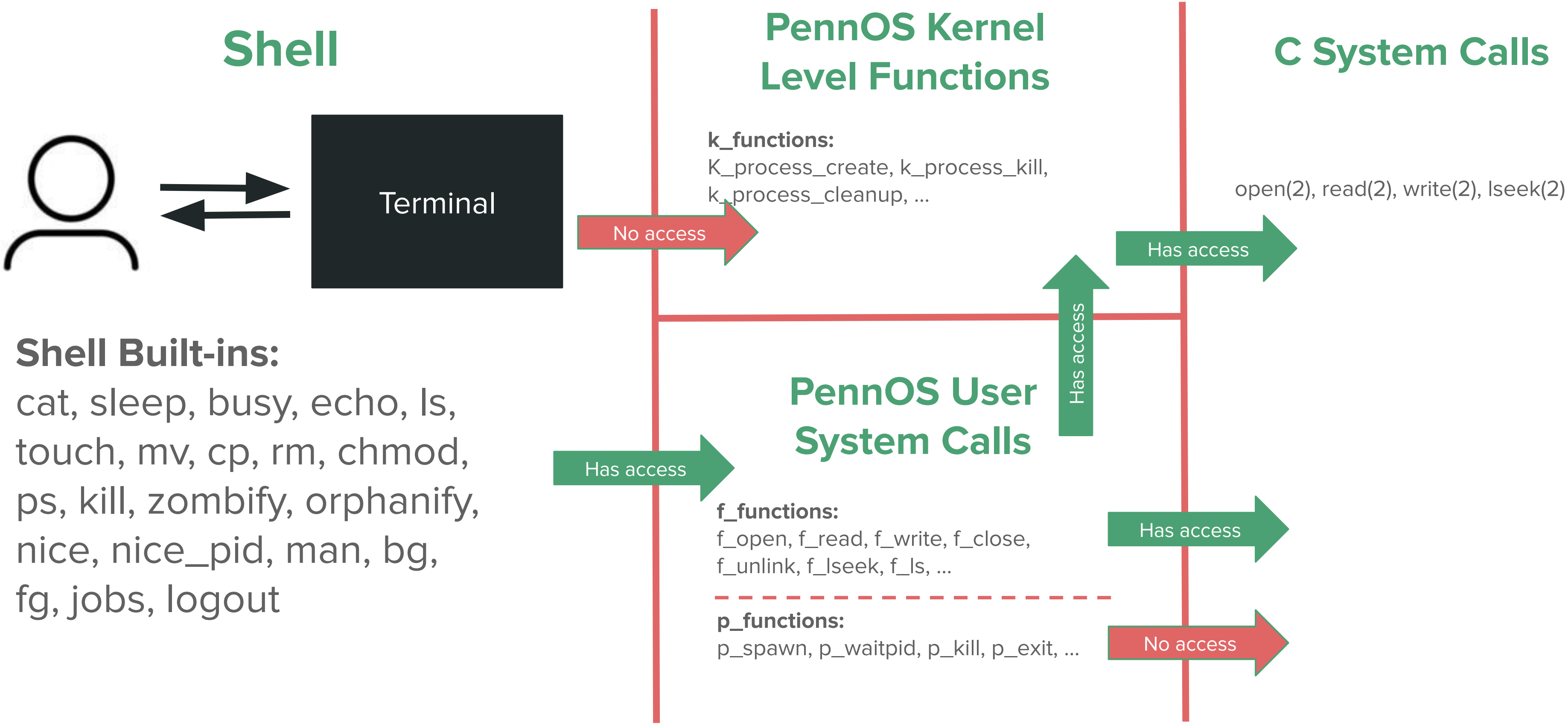
Final Touches: Shell

- Synchronous Child Waiting
  - Shell attempts to wait on ALL children using p_waitpid before reprompt
- Redirections
- Parsing
  - May use parser.o
- Terminal Signal Handling
  - Ctrl-Z, Ctrl-C should not stop or terminate PennOS
  - Relay the signal to the proper thread via user-system calls
  - Ctrl-D or `logout` will exit PennOS
- Terminal Control of stdin
  - If a process tries to take control of stdin when it should not, send a S_SIGSTOP to it
  - Should not be using tcsetpgrp(2)

Final Touches: Error Handling

- errno.h, p_perror
- Have global ERRNO macros
- Call p_perror for PennOS System call errors like f_open, p_spawn
- Call perror(3) for any host OS System call error like malloc(3) or open(2)

Final Touches - Abstraction!



Shell

Shell Built-ins:
cat, sleep, busy, echo, ls, touch, mv, cp, rm, chmod, ps, kill, zombify, orphanify, nice, nice_pid, man, bg, fg, jobs, logout

PennOS Kernel Level Functions

k_functions:
K_process_create, k_process_kill, k_process_cleanup, ...

PennOS User System Calls

f_functions:
f_open, f_read, f_write, f_close, f_unlink, f_lseek, f_ls, ...

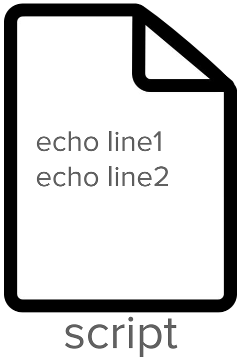
p_functions:
p_spawn, p_waitpid, p_kill, p_exit, ...

C System Calls

open(2), read(2), write(2), lseek(2)

Final Touches: Shell Scripts

```
$ echo echo line1 >script
$ echo echo line2 >>script
$ cat script
echo line1
echo line2
$ chmod +x script
$ script > out
$ cat out
line1
line2
```



```
echo line1
echo line2
```

script

Final Touches: Companion Document

Doxygen:

https://www.doxygen.nl/

https://www.gnu.org/software/gsasl/doxygen/gsasl.pdf

Or just write your own

- Include functions for shell builtins, PennOS system calls, but not every single helper function needs to be there
- Include Global Variables, structs, enums, and macros you create and use

Any Questions?