Lab 2 Discussion

CSE451 21Sp - 22 Apr 2021

Admin

- Lab 2 due next Friday (4/30)

Agenda

- I have some discussion questions related to lab 2
- I also have the section 3 slides to refer back to if you need a high-level overview of anything
  - Feel free to yell at me
- Time at the end for open Q&A

Debugging in the Trap

- You will likely encounter an error from the trap like the below image during your xk endeavors
  - Do not fear!
- Trap frame contains registers saved before jumping into the kernel
  - See kernel/trapasm.S for the mechanism there
- This can be useful!
- tf->trapno, “trap number”
  - Gives the “reason” why kernel was invoked
  - See inc/trap.h for trapno indicators
- Trap frame registers can help give context for why you’re in the kernel
  - In gdb, `x tf->rip` can give program counter that caused fault
  - On a page fault, `addr` contains address attempted to be accessed
    - Loaded from cr2 register

```
pid 6 : trap 14 err 5 on cpu 0 rip 0x491 addr 0x80000000--kill proc
```

Lab 2 - Processes

IF IN DOUBT: DO WHAT LINUX DOES

fork()

- Create a new process by duplicating the calling process.
- Returns twice!
  - 0 in the child (newly created) process
  - Child's PID in the parent
- How does fork() "return twice"?
  - I.e. when the child process is scheduled for the first time, it returns from the fork system call with a return value of 0
  - Thoughts?

wait()/exit()

- wait(): Sleep until a child process terminates, then return that child's PID.
- exit(): Halts program and sets state to have its resources reclaimed

- Why can't a process not clear out it's own proc struct in `exit`?
  - Who is responsible?

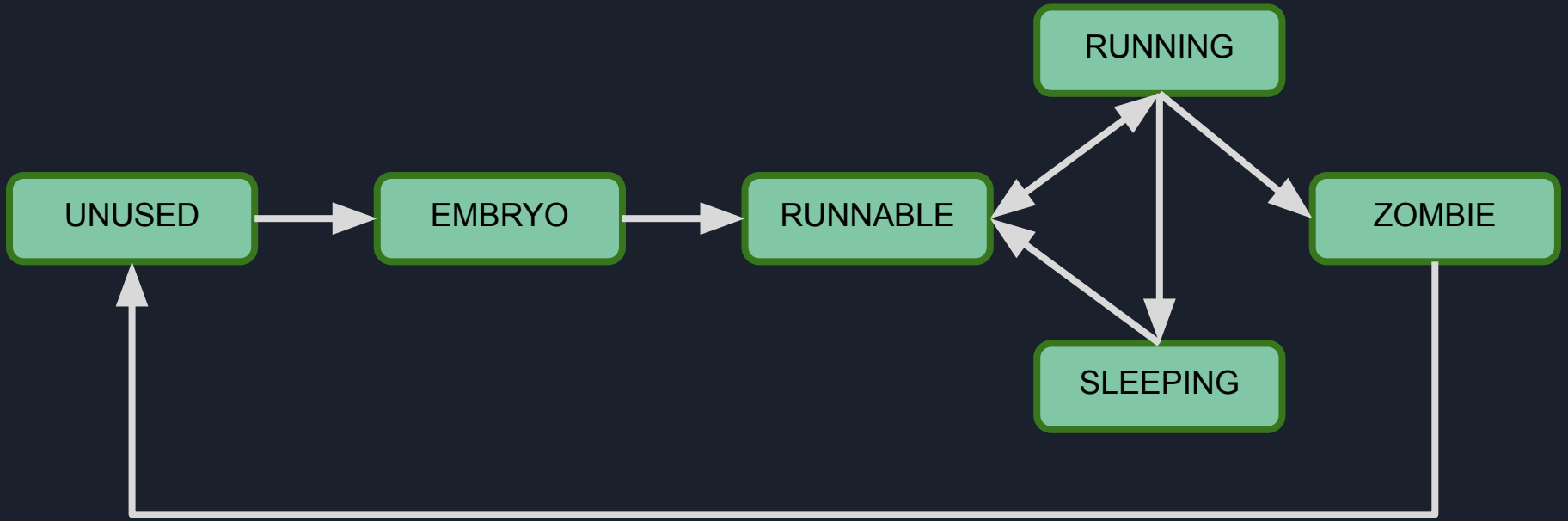
- If a parent process calls `exit` before its child finishes executing, how does the child process need to be modified to guarantee that someone will wait for the child?

Process States

Fill out the process state diagram below. Draw arrows from one state to another with the action that would result in that transition

Process States

Fill out the process state diagram below. Draw arrows from one state to another with the action that would result in that transition



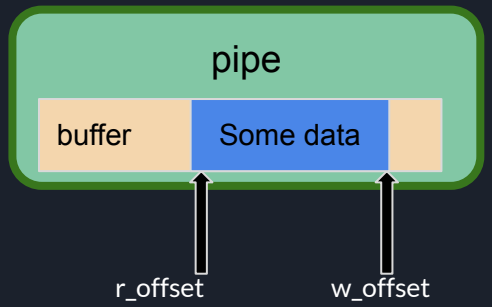
pipe(pipefds)

- Creates a pipe (internal buffer) for reading from/writing to
- From the user perspective: two new files
  - One (“read end”) is not writable
  - Other (“write end”) is not readable
- You’ll want to somehow make this compatible with the read/write(fd) interface

Pipes

- Managing a circular buffer with read end and write end

- If reading and pipe is empty,
  - Reader should wait until some data is available
- If writing and pipe is full,
  - Writer should wait until it has some room to write
- How do you tell if a pipe is full/empty?



- Suppose a reader of a pipe is sleeping waiting for the writer to write some data
  - If writer process is killed before it gets a chance to write data, how does the reader get woken up?
  - What should the read call return?
- Open readers, no writers: return remaining data (if any), then 0 (EOF)
- Open writers, no readers: return -1

exec(progname, args)

Replaces the process' state by executing the given program with the given arguments.

This will require you to (carefully!) set up the process' stack memory and register state.

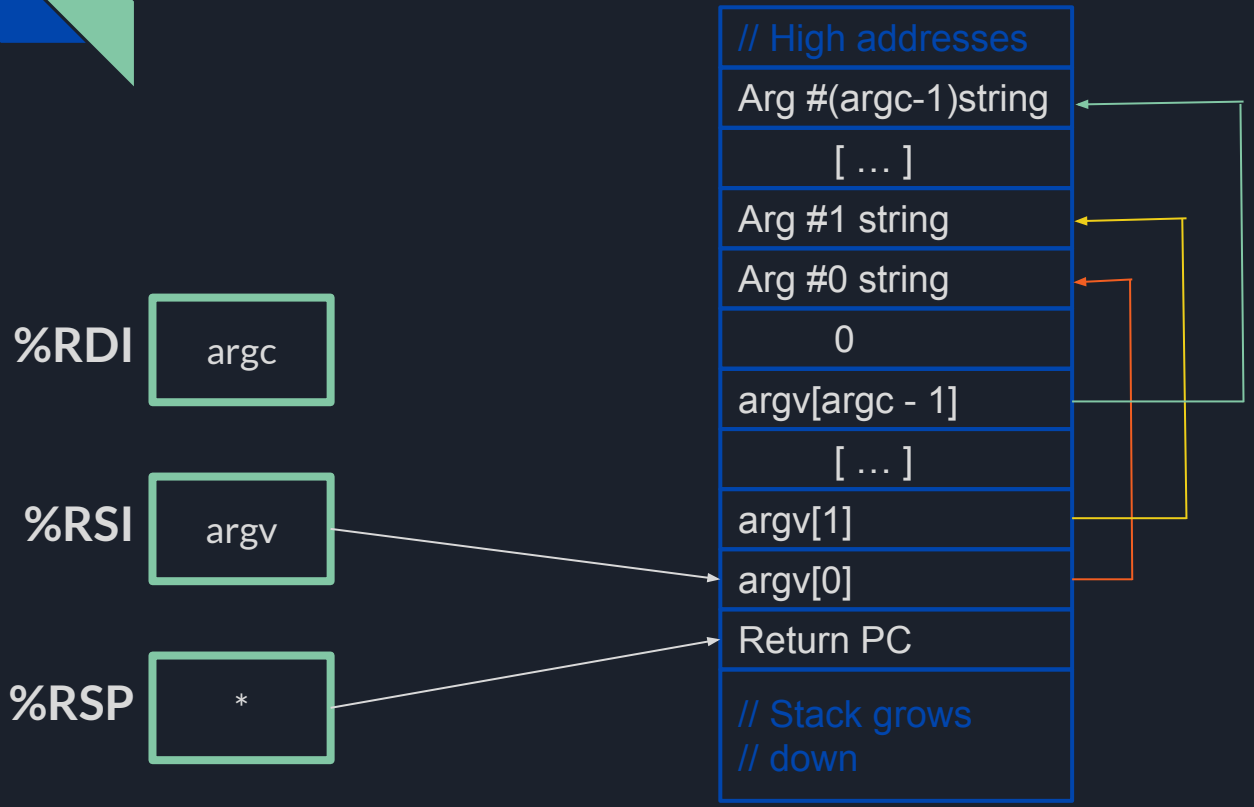
This will be tricky! You'll be using a number of vspace___ functions

- init, loadcode and initstack may be helpful for initializing a new memory space (in that order)
- use vspacewritetova to export data to a page table that isn't currently installed
- once the memory space is ready, use vspaceinstall(myproc()); to engage
- and free the old vspace!

- When creating the user stack in xk, what should the stack pointer start at?
  - (this would be an argument to pass to vspaceinitstack)


+------------------+ <- 0xFFFFFFFFFFFFFFFF (18 exabytes)
| |
| Kernel |
| |
+------------------+ <- KERNBASE = 0xFFFFFFFF80000000
| |
| Unused |
| |
+------------------+ <- 2GB (vspace.regions[VR_USTACK].va_base)
| |
| Stack |
| |
+------------------+ <- vspace.regions[VR_USTACK].va_base - vspace.regions[VR_USTACK].size
| |
| |
| Unused |
| |
| |
+------------------+ <- vspace.regions[VR_HEAP].va_base + vspace.regions[VR_HEAP].size
| Heap |
+------------------+ <- vspace.regions[VR_HEAP].va_base
| |
| Text |
| |
+------------------+ <- vspace.regions[VR_CODE].va_base

Main's Stack

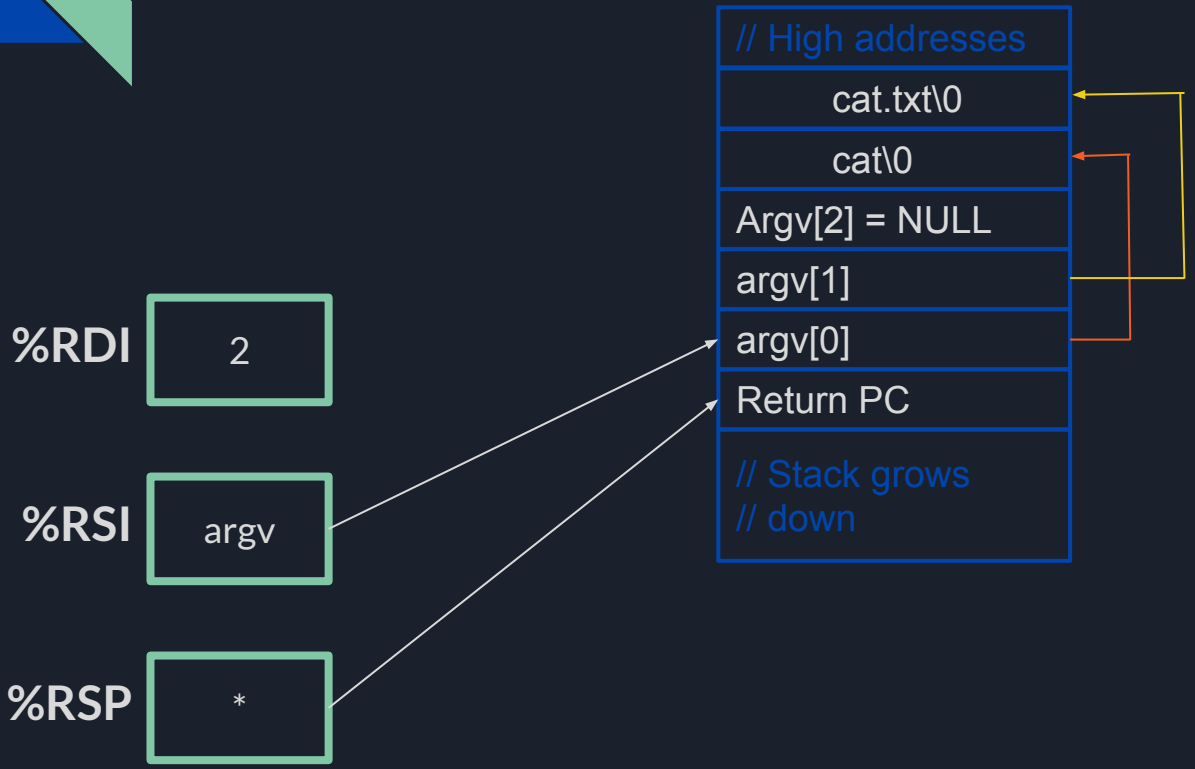


- Since argv is an array of pointers, %RSI points to an array on the stack
- Since each element of argv is a char*, each element points to a string elsewhere on the stack

Practice Exercise 1

TODO:
Draw stack layout and determine register values for exec called with “cat cat.txt”

Practice Exercise 1: soln



- RDI holds argc, which is 2
- RSI holds argv: the beginning of the argv array
- RSP is properly set to the bottom of the stack.
- The specific value of the return PC doesn't matter (program exits from main without returning)

Good luck on Lab 2!