Lab 2

Part 2

Admin

- Lab 2 out!
  - Design Doc due 10/21/24 @11:59pm
  - Lab Code + Questions due 10/28/24 @11:59pm
- You will write design docs starting with lab 2
  - The better you fill it out, the more helpful we can be with our feedback!
  - Graded on effort basis, NOT correctness (although correctness would be nice)

Agenda

- Monitors Overview
- Lab 2 - Pipe
- Lab 2 - Exec
- Setting up the Stack exercises

Monitors

What the heck is a monitor?

- A monitor is made up of a lock and at least one condition variable

Why do we use monitors?

What the heck is a monitor?

- A monitor is made up of a lock and at least one condition variable

Why do we use monitors?

- Similar to locks but…
    - Allow processes to wait for certain conditions to become true while “holding lock” (waiter atomically releases the lock and reacquires the lock on wakeup).

Monitors in xk

- Lock
  - xk condition variable API only supports spinlock (an impl. choice)
- Condition
  - the shared data that threads are synchronizing on
  - E.g. for wait/exit this would be child's state
- Condition Variable
  - the waiter list is tracked by the process table
  - proc in SLEEPING state with the same `chan` are part of the same CV
  - `chan` is a pointer, can be anything (think of it as a cv identifier)

“Condition variable? I saw no mention of those in the provided code.” ~ You, a free thinker.

No Condition Variables in xk

The starter code does *not* provide the object-oriented std::condition_variable API you can find in C++: LINK

Instead it provides the sleep and wakeup helper functions (which together can implement the monitor pattern)

- sleep ~= wait
- wakeup ~= broadcast

Sleep

- sleep(void* chan, struct spinlock* lk)
  - atomically release your current lock and grabs the process table (ptable) lock
    - if your current lock is the ptable lock do nothing
    - why might your current lock be the ptable lock?
  - sets myproc()->state to SLEEPING
  - sets myproc()->chan to whatever channel we are waiting on
  - yields so that scheduler can run another process

Wakeup

- wakeup(void* chan)
  - acquires the process table lock
  - looks for all SLEEPING processes with the given channel (chan)
    - sets each proc->state to RUNNABLE (ready)
    - proc->chan is also cleared to NULL

Monitors in xk

- You will use monitors to implement wait(), exit(), and pipe() for lab2!

wait(), exit()

- Coordinating children and parent processes

pipe()

- Coordinating reader and writer processes

Lab 2 - Pipe

What is a Pipe?

A pipe is essentially a queue of bytes with two ends:

- One end designated for input, the other for output

When you type **‘ls | wc’** into the shell, you are using a pipe!!!

- ‘ls’ lists the directory contents
- ‘wc’ counts the number of lines output from the ls command
- The pipe joins the output from ‘ls’ to the input of ‘wc’

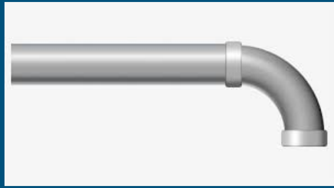
pipe(fds)

- Creates a pipe (kernel buffer) that can be read from/written to.
- From the user perspective: returns two new file descriptors
  - fds[0] = “read end”, O_RDONLY
  - fds[1] = “write end”, O_WRONLY
- Pipes allow processes to communicate with each other
  - Parent opens a pipe, forks a child (now they both have access to the pipe ends)
  - Typically each process only leaves one end open (closes the read end or the write end)

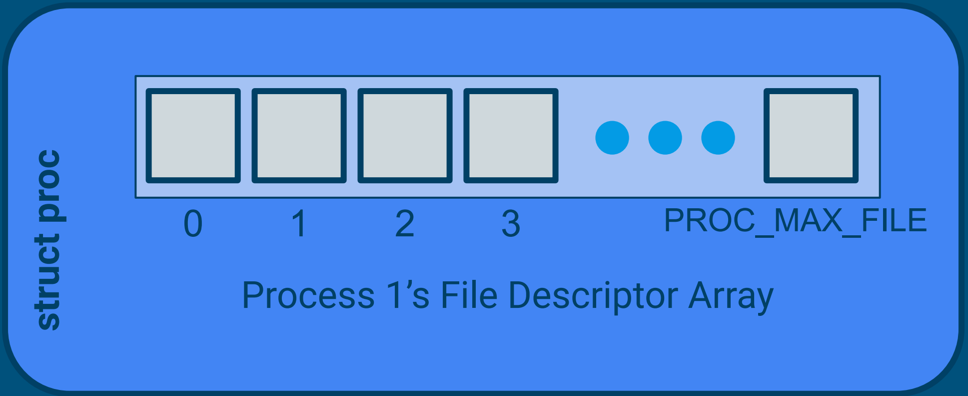
An Example to Illustrate Pipes

Now let's go through a demonstration of what happens as a sample user uses the pipe API (in the context of multiprocessing)!

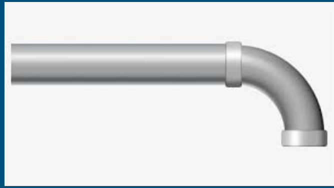
Pipe usage



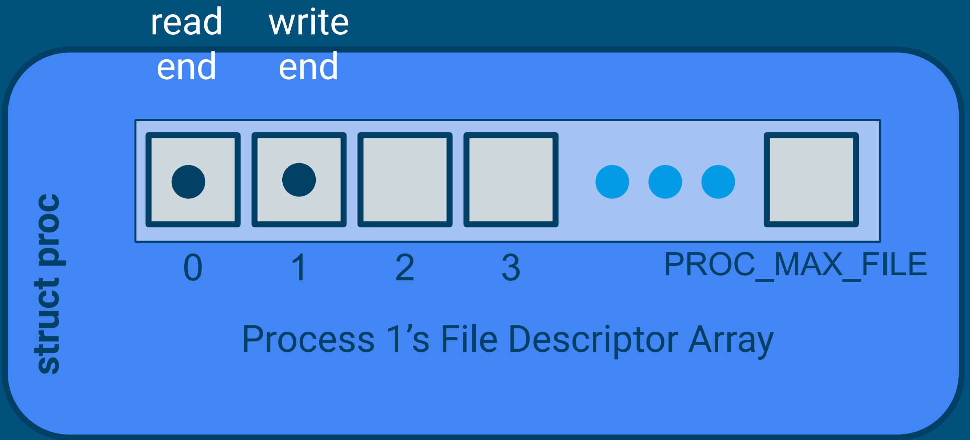
- Process 1 starts with no open files



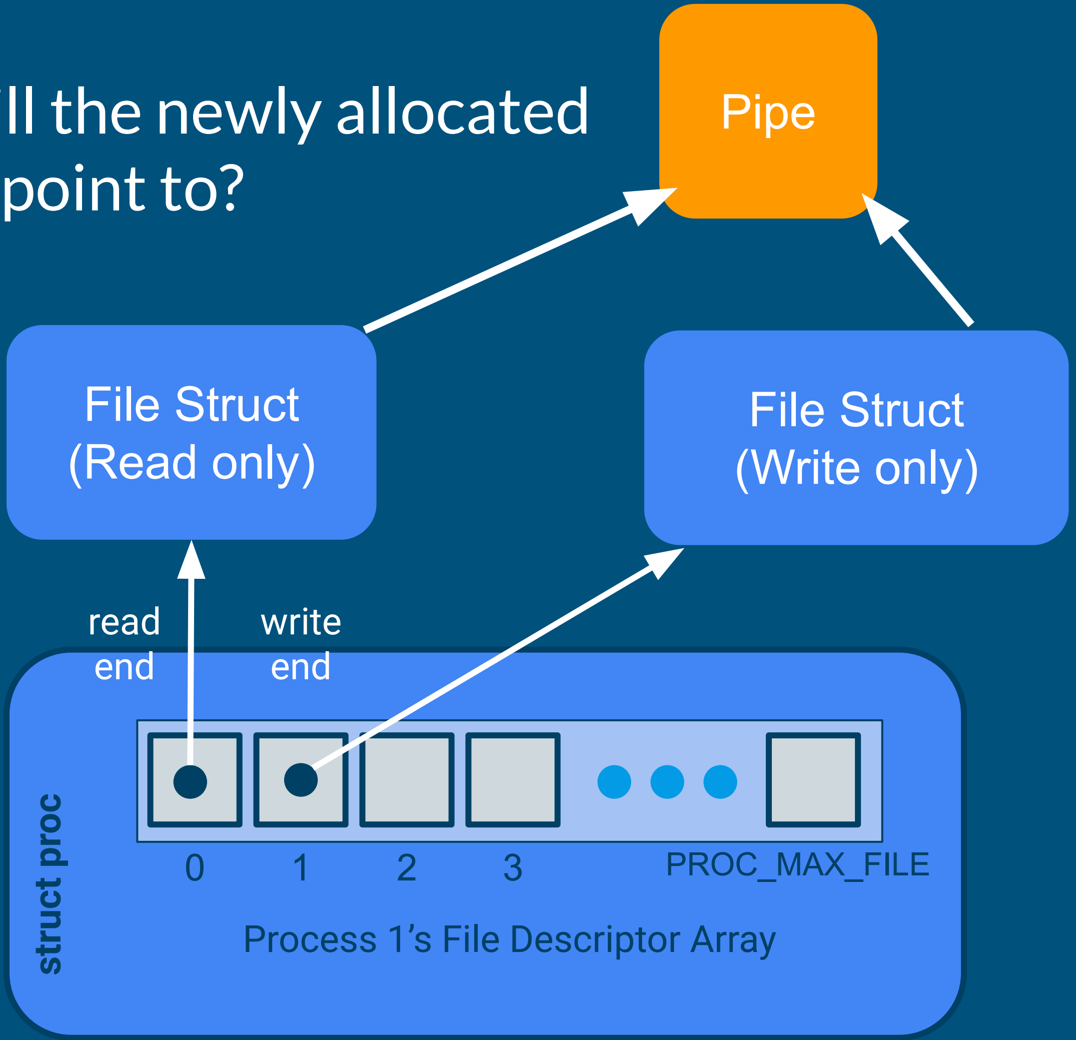
Pipe usage



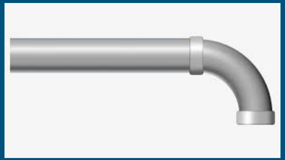
- Process 1 calls `pipe()`



What will the newly allocated pipe fds point to?

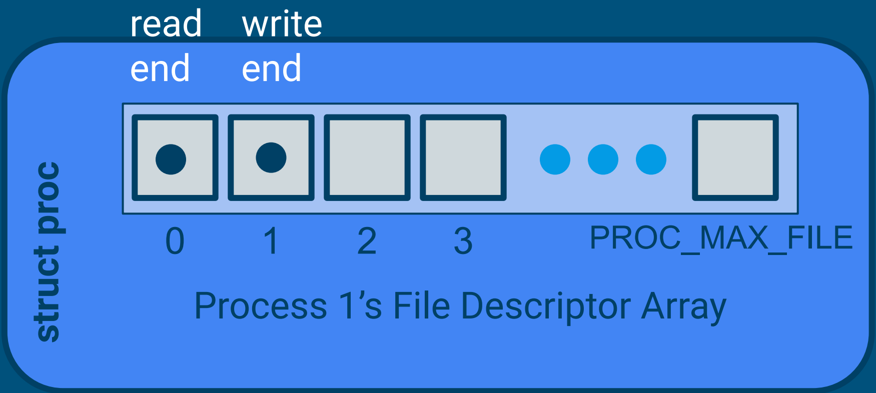


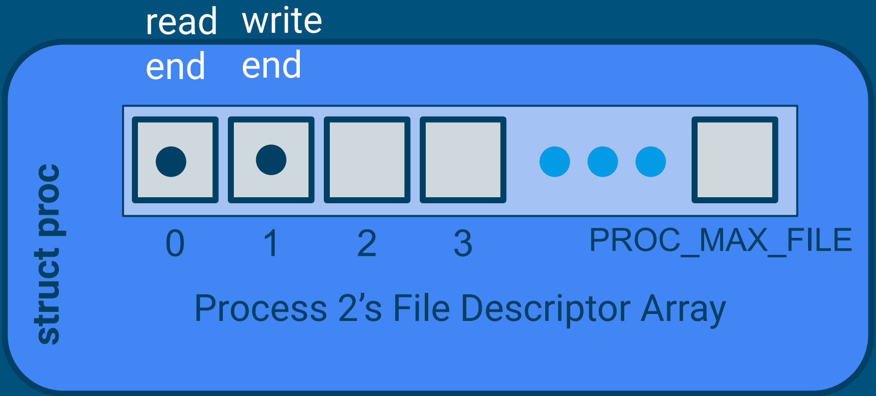
Pipe usage



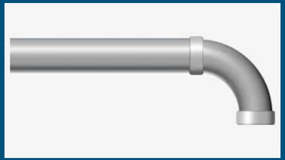
- Note: fork() is called by user and should not be called within the actual pipe() call

- Process 1 calls `fork()`, fd table is duplicated

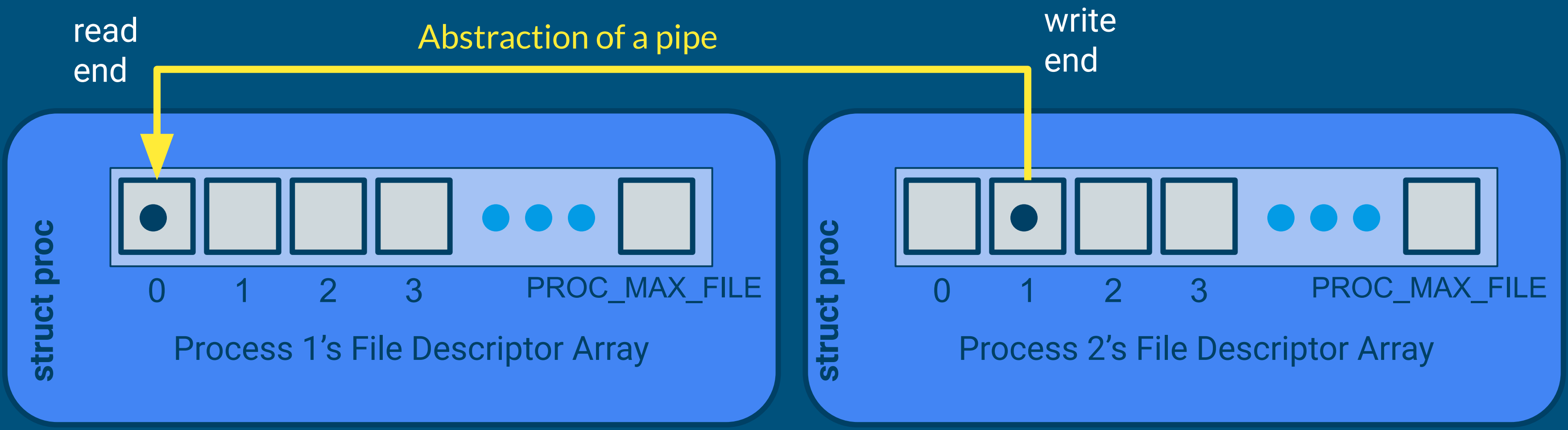




Pipe usage



- Process 1 `close(1)`, process 2 `close(0)`
- The process with the write end open is a writer, and the one with the read end open is a reader



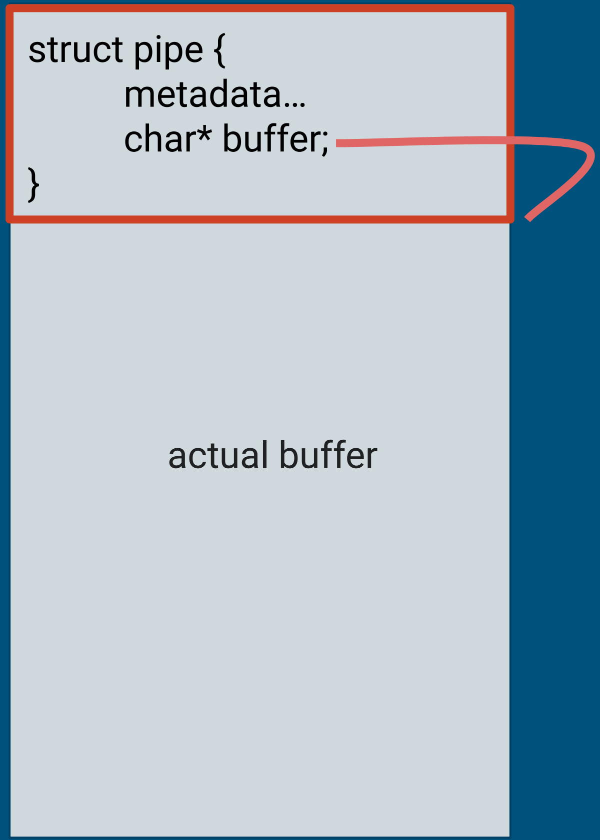
pipe FAQs

- When should pipe be allocated?
  - dynamically! when pipe() is called!
- How does xk do dynamic memory allocation?
  - hint: kstack is also dynamically allocated
  - `kalloc` allocates a page (4096 bytes) of memory from the kernel heap
    - wait, but how do I put a pipe onto the page?

struct pipe* p = kalloc();

p->buffer = ???

should be right past the struct, and what would that be?



a page of memory
(4096 bytes)

pipe FAQs

- When can you free the pipe and its buffer?
  - remember there may be multiple references to read end and write end
- Can we always write to or read from the buffer? (Hint: bounded buffer sync)
  - What if there's no room to write, or no data to read?
  - What happens if all read/write ends are closed?
- How will pipes integrate with the file syscalls?
  - Need a way to determine if a struct file is an inode or a pipe

Interaction with File API

Pipes are accessed through file descriptors.

This means you need to think through how the lab 1 syscalls will work when called on pipe file descriptors:

- `close`
- `dup`
- `read`
- `write`
- `stat`

What should pipe contain?

- What metadata/information do you need for pipe?

What should pipe contain?

- What metadata/information do you need for pipe?
    - Read offset
    - Write offset
    - # of bytes available in the buffer
    - Whether the read end is still open
    - Whether the write end is still open
    - Lock and condition variables
    - A way to track the active writer [why?]
- Similar to the bounded buffer problem

And that's pipe!

… But wait! There's more! (that you have to do in lab 2 part 2)

But wait! .... There's more! (in lab 2 part 2)

In lab 2 part 2 you are also implementing exec

exec(3) — Linux manual page

Lab 2 - exec

Motivation

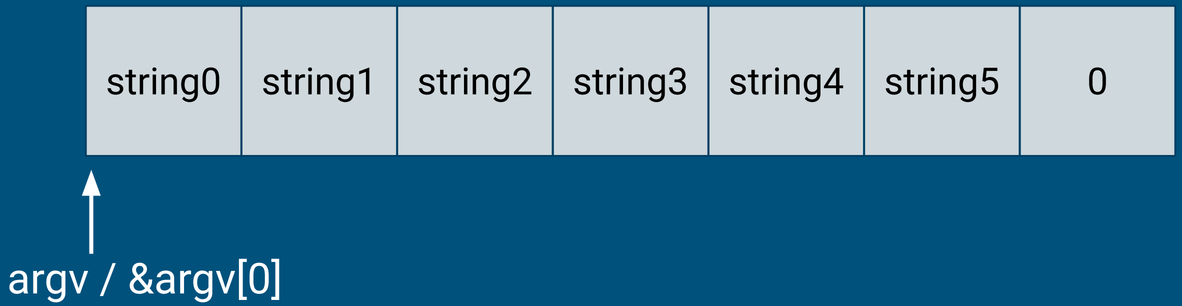
Why do we have exec?

- To let user code execute user programs!
  - E.g. Shell commands like ‘ls’ and ‘cat’ commands are exec’ed by the ‘sh’ program.

exec(program, args)

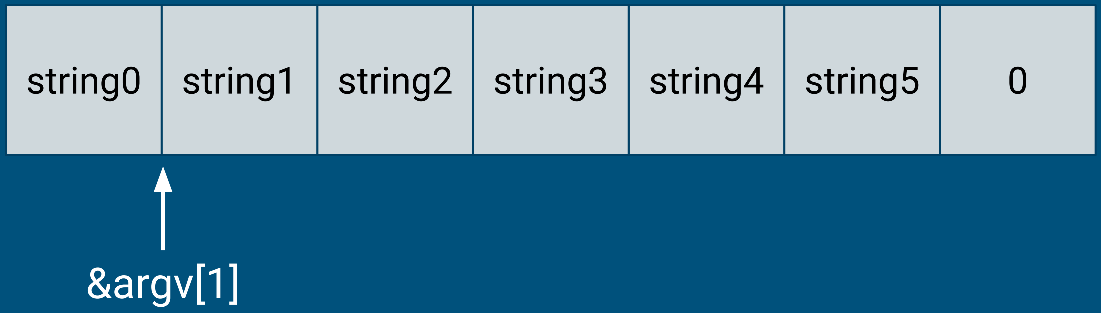
- Fully replaces the current program; it does not create a new process
- How do we replace the current program?
  - need to set up a new virtual address space and new registers states
  - and then switch to using the new VAS and register states
  - file descriptors and pid remain the same

exec(path, argv) arguments validation



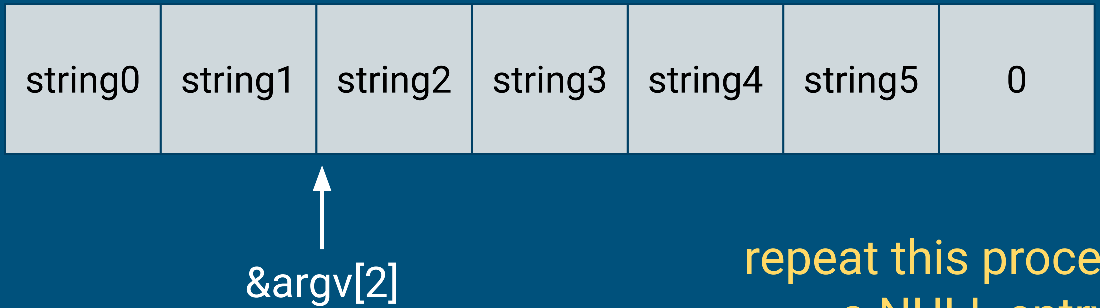
must be validated for an 8 byte pointer before we can access argv[0] and validate string0

exec(path, argv) arguments validation



must be validated for an 8 byte pointer before we can access argv[1] and validate string1

exec(path, argv) arguments validation



must be validated for an 8 byte pointer before we can access argv[2] and validate string2

repeat this process until

- a NULL entry is reached
- a validation error

`exec`(program, args)

- Setting up a new virtual address space (pseudocode)
  - `vspaceinit` for initialization
  - `vspaceloadcode` to load code
  - `vspaceinitstack` to allocate stack vregion
    - you still need to populate user stack with arguments
    - `vspacewritetova` to write data into the stack of the new VAS
  - `vspaceinstall` to swap in the new vspace
  - `vspacefree` to release the old vspace
- The swapover to the new vspace can be tricky to get right!
  - To swap: Assign the new vspace to current vspace

How are the args set up in exec?

Another look at main()

exec sets up the function arguments for main!

```
int main(int argc, char** argv)
```

- argc: The number of elements in argv
- argv: An array of strings representing program arguments
  - First is always the name of the program
  - Argv[argc] = 0

Setting up the Stack

Quick Review: X86_64 Calling Conventions

From 351:

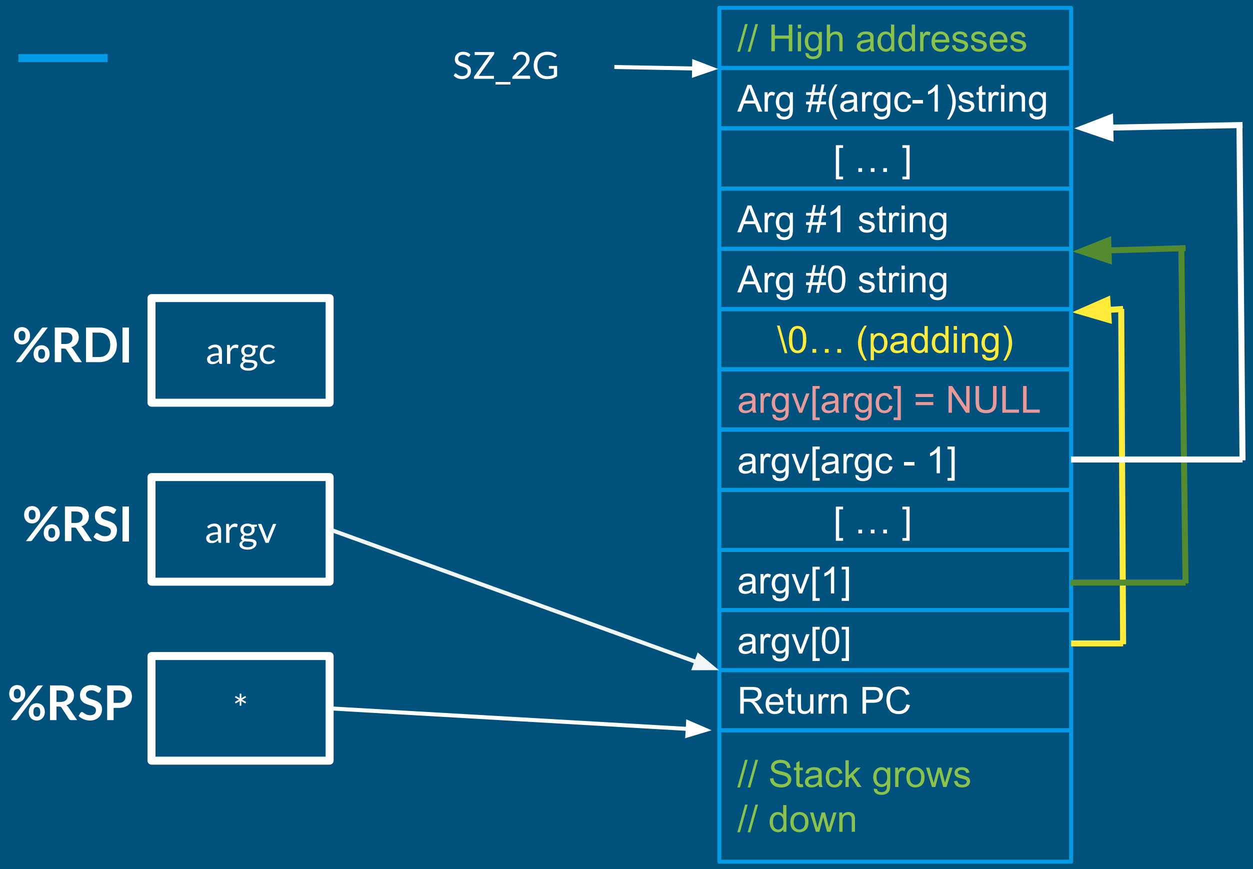
- `%rdi`: holds the first argument
- `%rsi`: holds the second argument
  - `%rdx`, `%rcx`, `%r8`, `%r9` comes next
  - overflows (arg7, arg8 …) onto the stack
- `%rsp`: points to the top of the stack (lowest address)

Quick Review: X86_64 Calling Conventions

From 351:

- Local variables are stored on the stack
- If an array is an argument, the array contents are stored on the stack and the register contains a pointer to the array's beginning

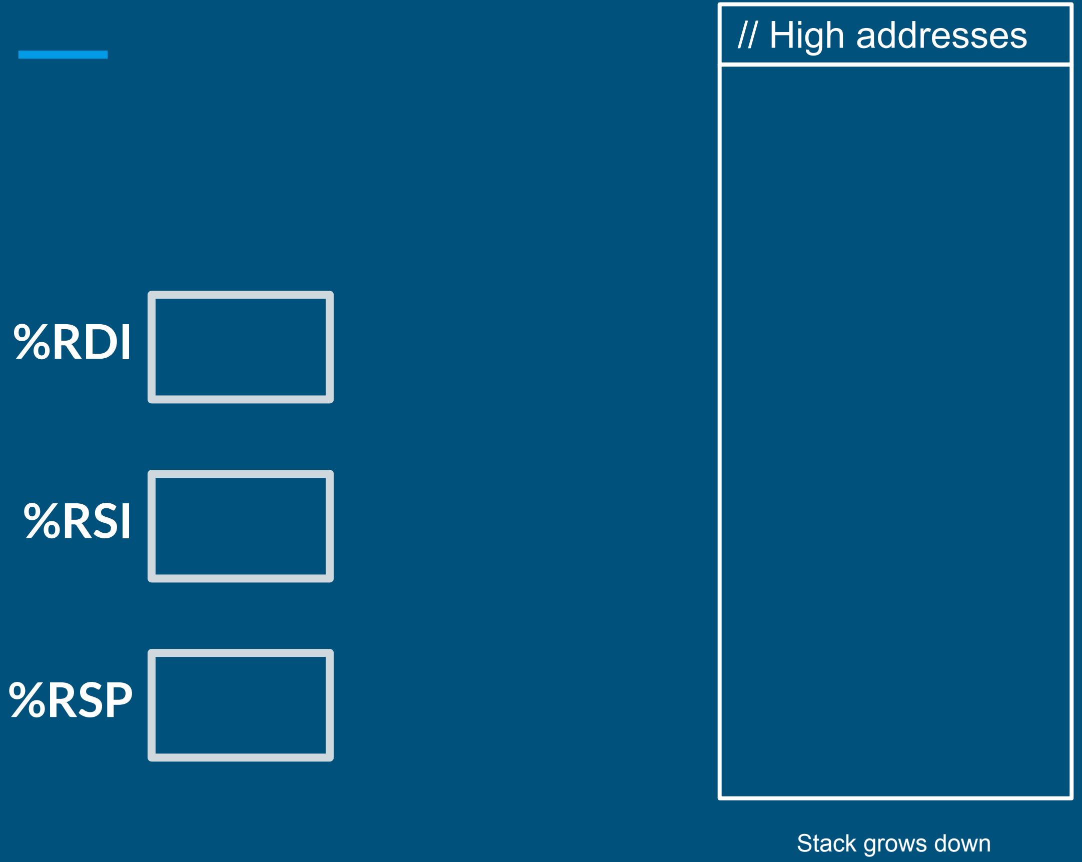
Stack For User Process



- 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
- Why? Alignment
- Why NULL pointer? Convention

Let's Practice!

Practice Exercise 1

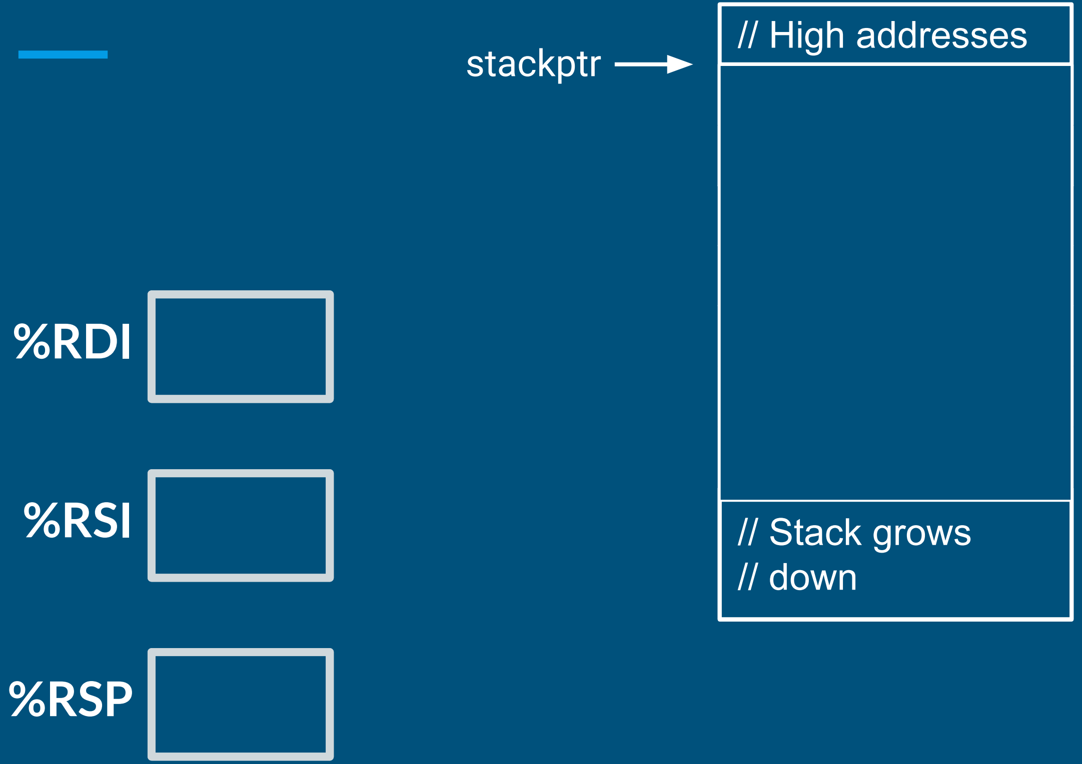


Now it's your turn!

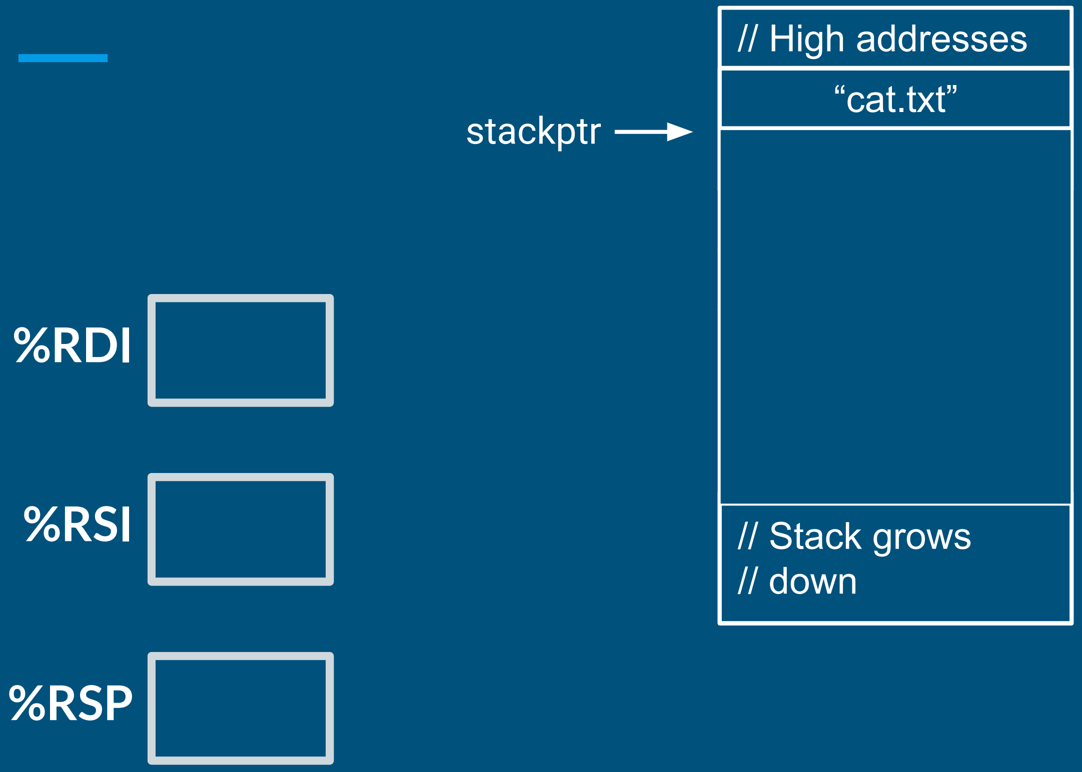
Draw stack layout and determine register values for exec() called with:

"cat cat.txt"

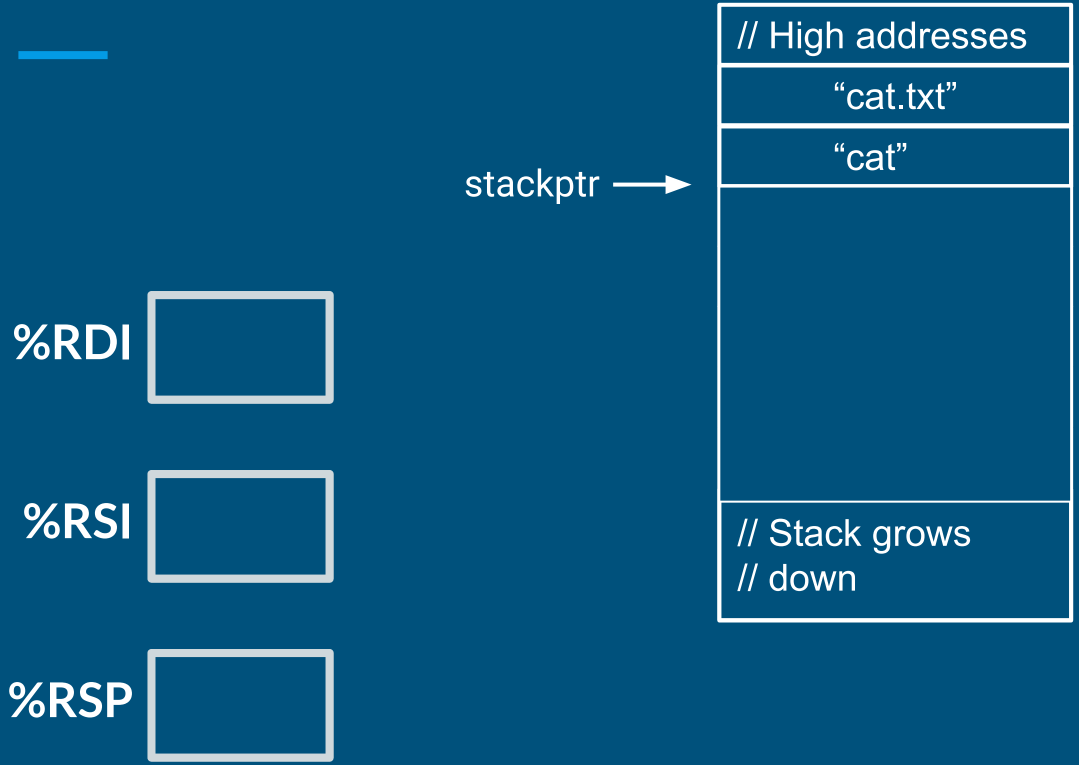
Practice Exercise 1: “cat cat.txt” Solution



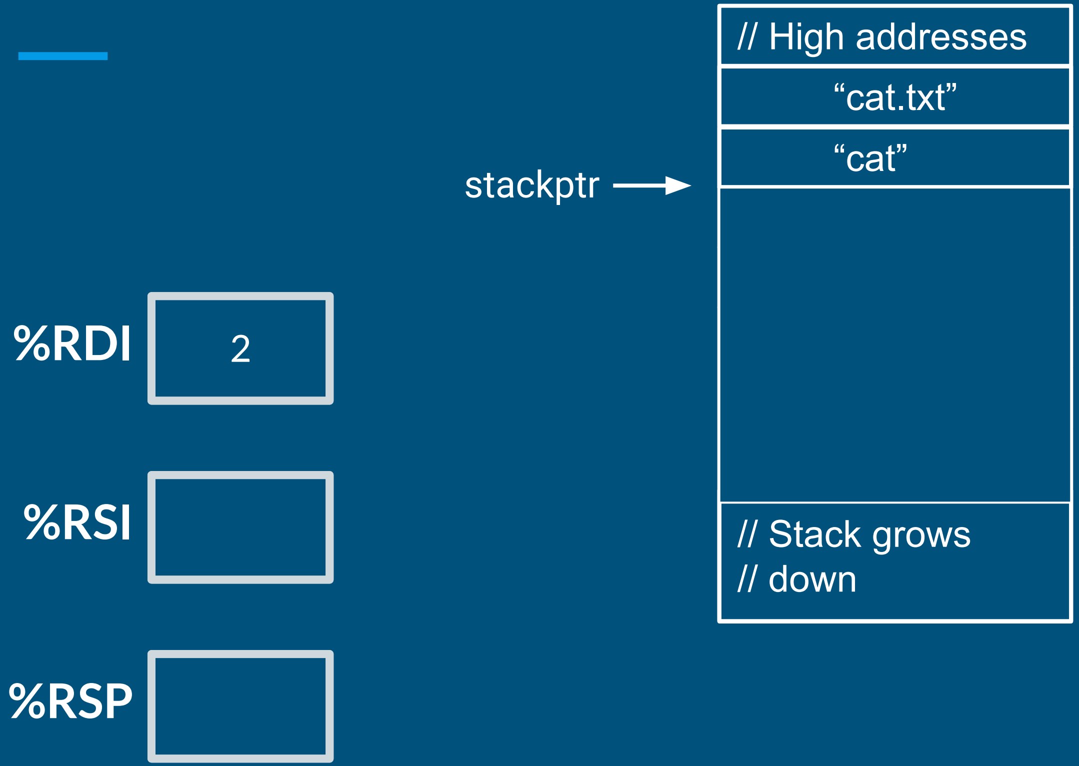
Practice Exercise 1: “cat cat.txt” Solution



Practice Exercise 1: “cat cat.txt” Solution

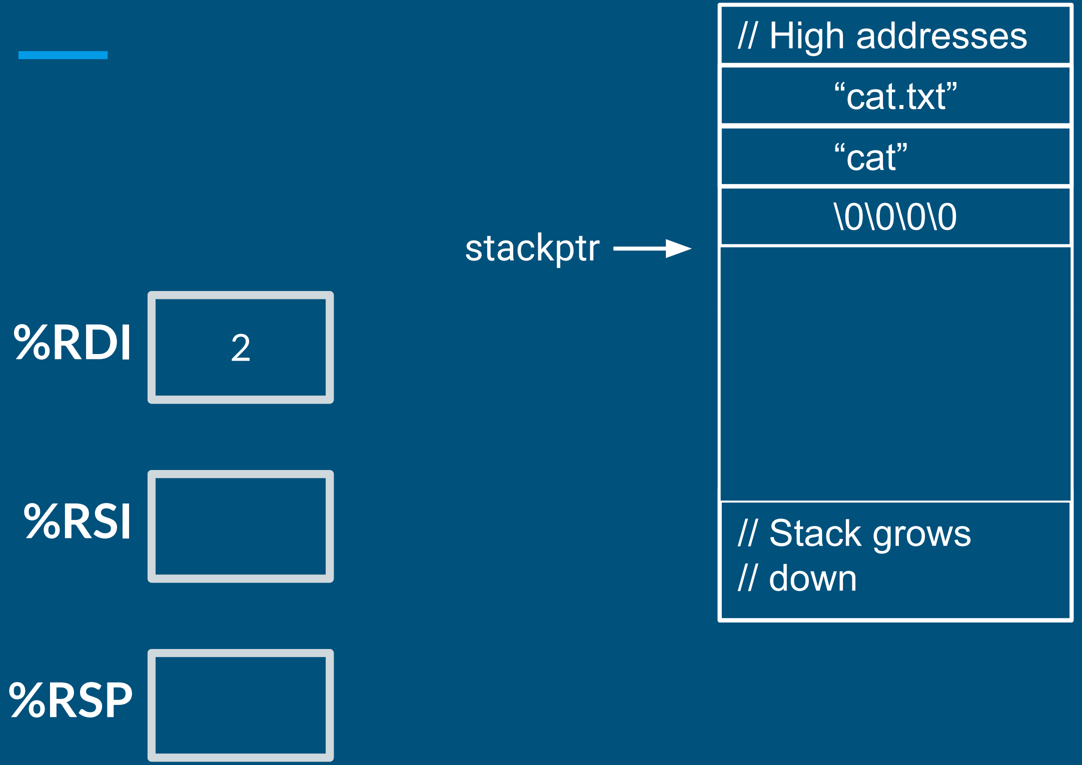


Practice Exercise 1: “cat cat.txt” Solution



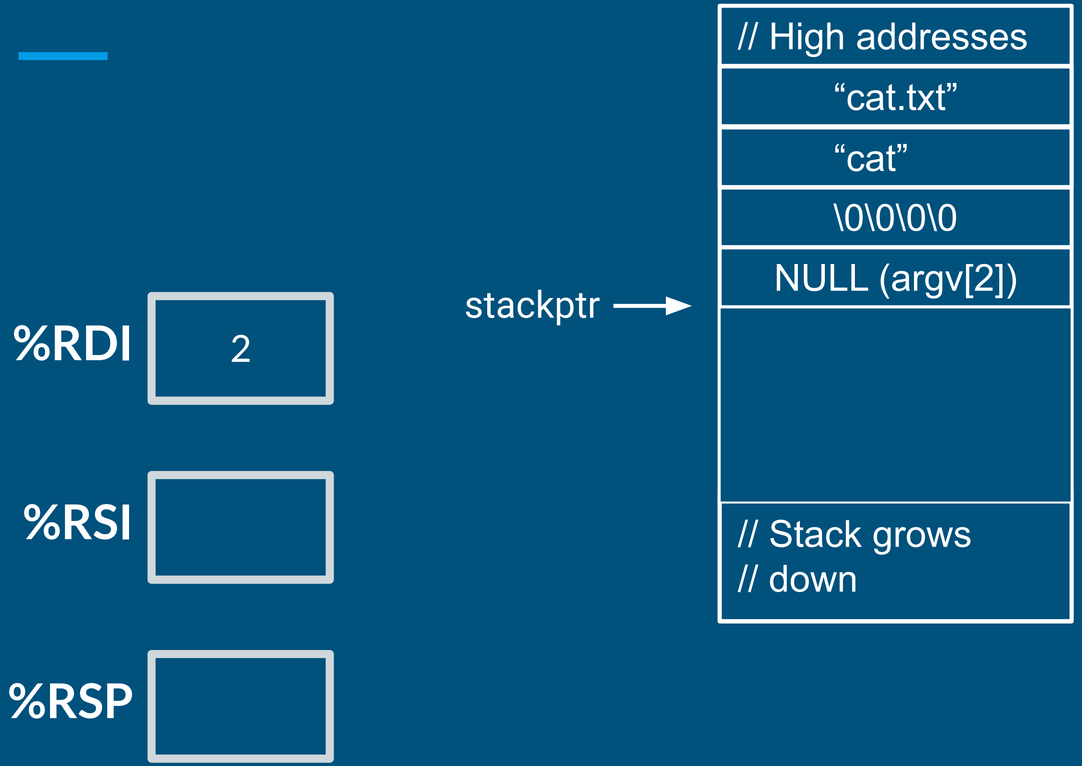
- RDI holds argc, which is 2

Practice Exercise 1: "cat cat.txt" Solution



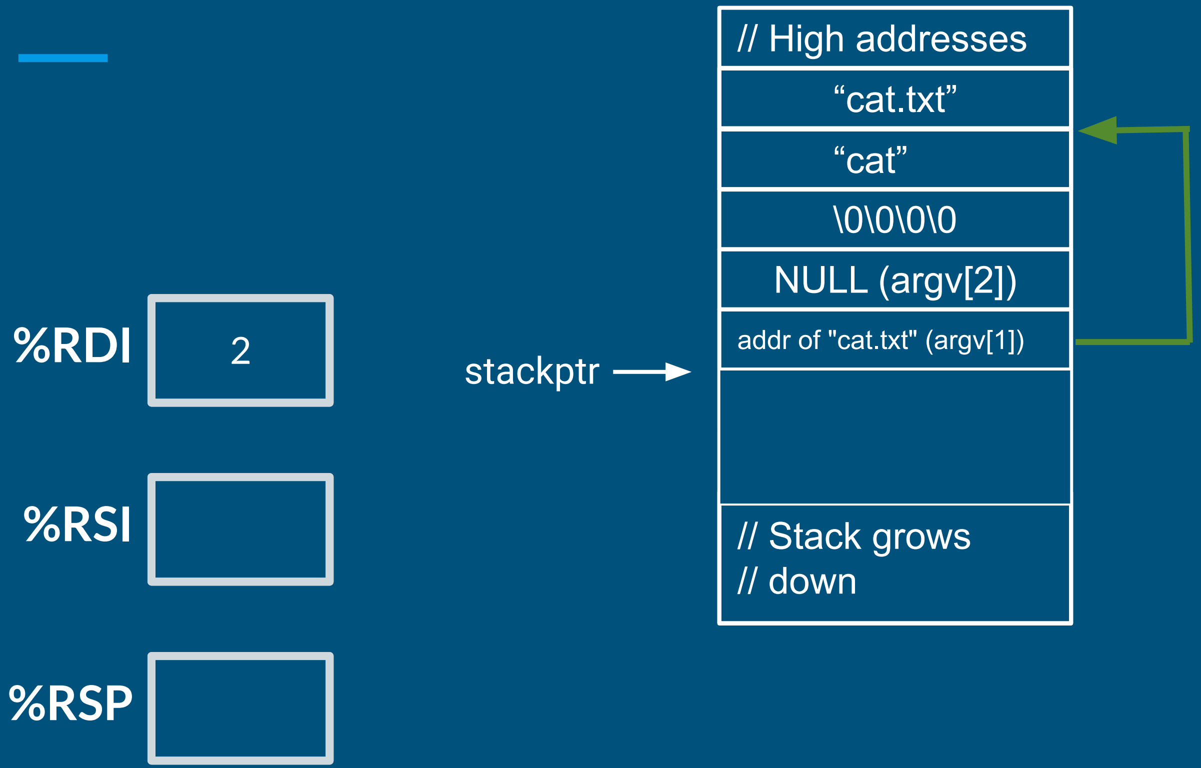
- RDI holds argc, which is 2

Practice Exercise 1: “cat cat.txt” Solution



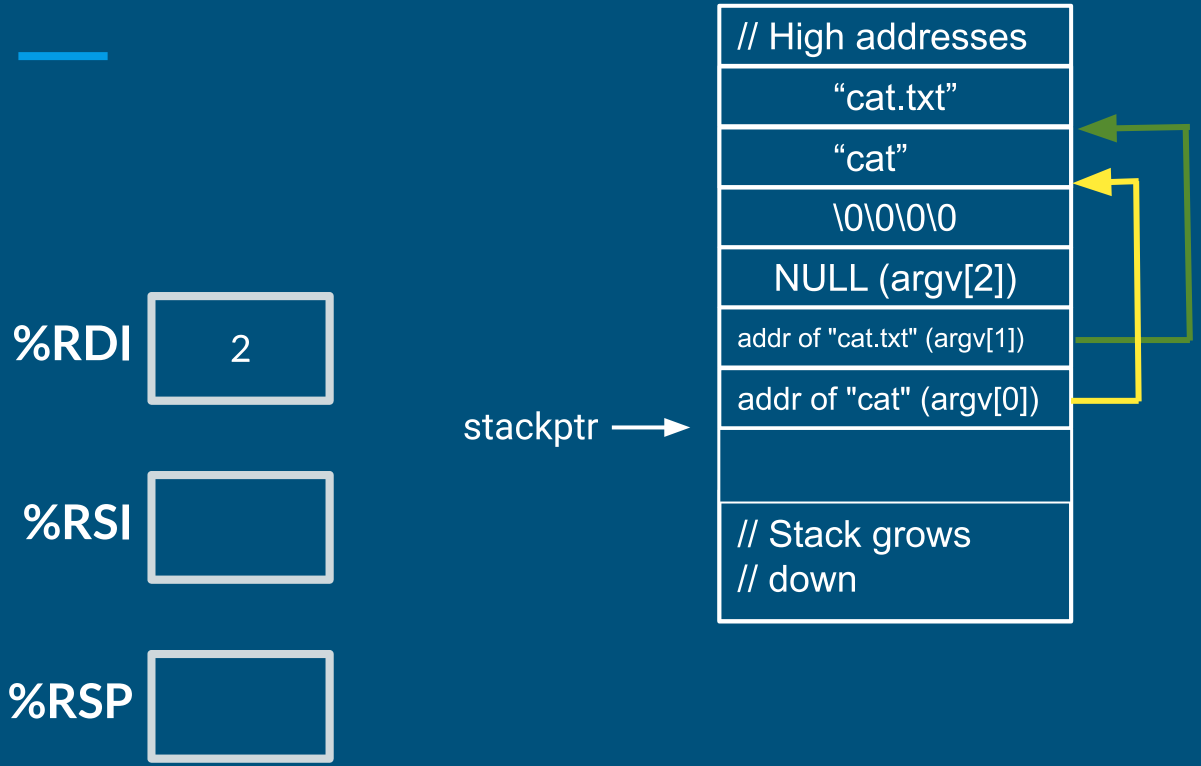
- RDI holds argc, which is 2

Practice Exercise 1: “cat cat.txt” Solution



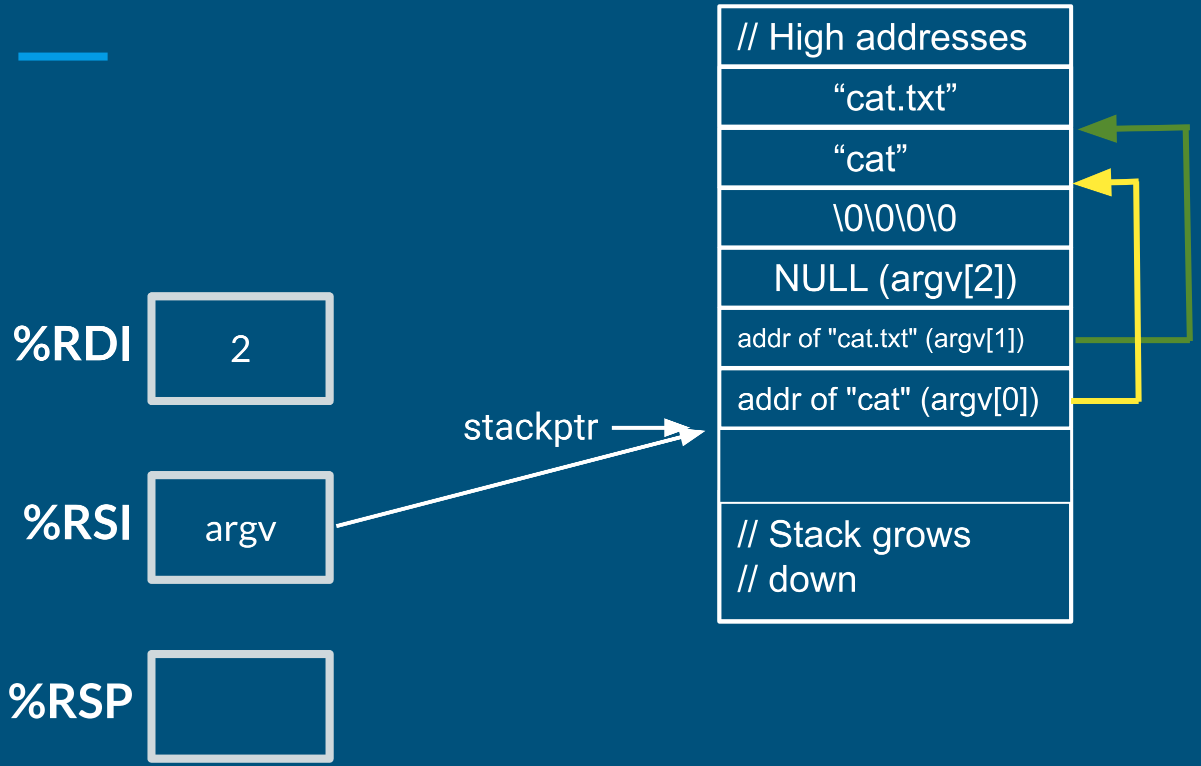
- RDI holds argc, which is 2

Practice Exercise 1: “cat cat.txt” Solution



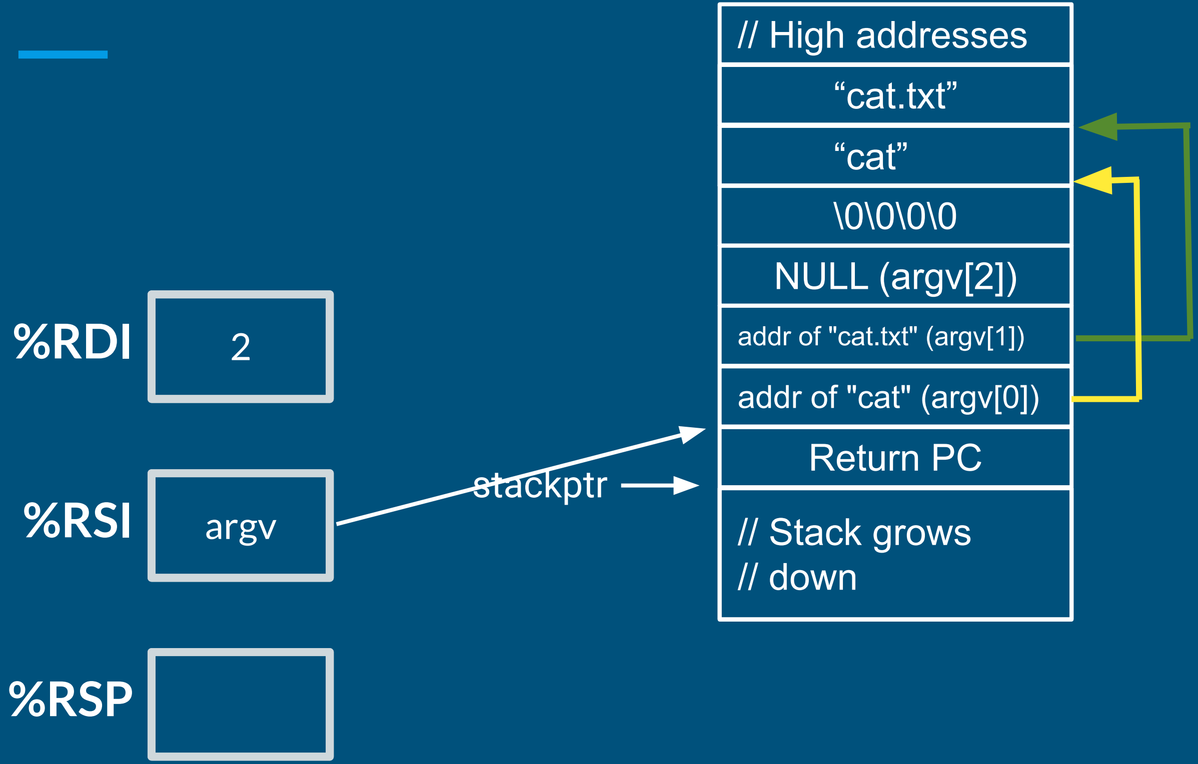
- RDI holds argc, which is 2

Practice Exercise 1: “cat cat.txt” Solution



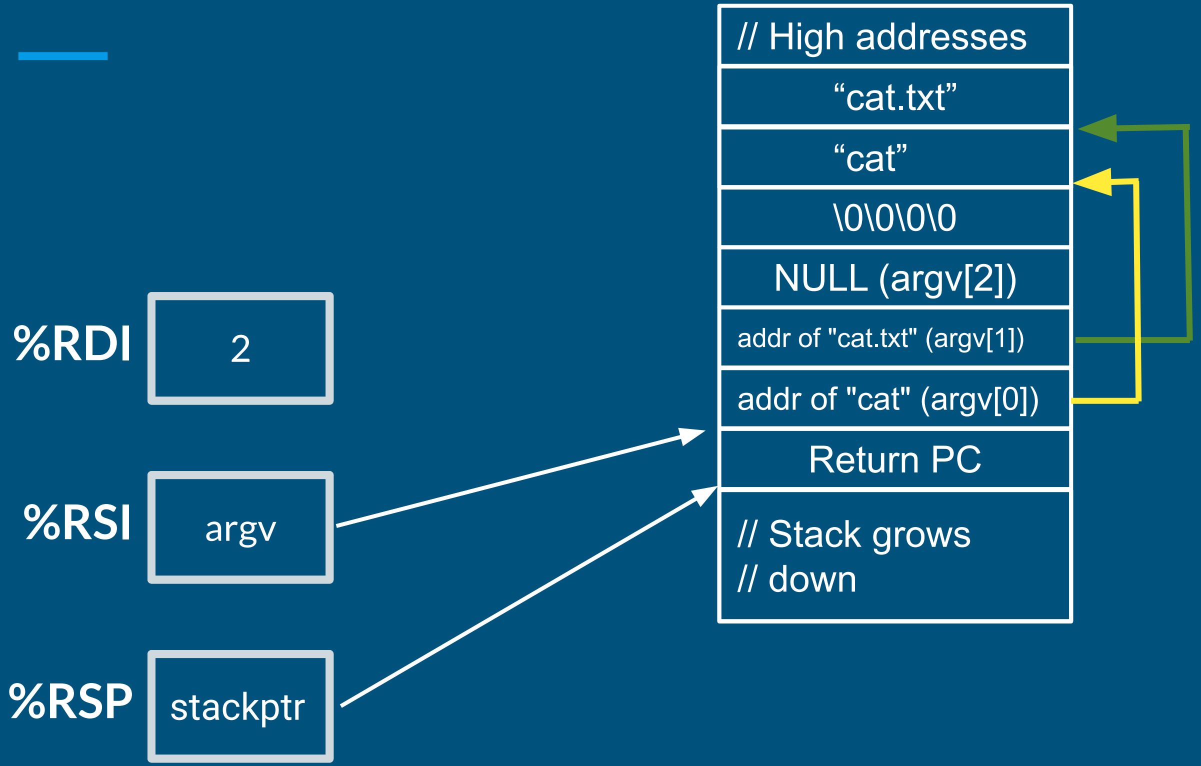
- RDI holds argc, which is 2
- RSI holds argv: the beginning of the argv array

Practice Exercise 1: “cat cat.txt” Solution



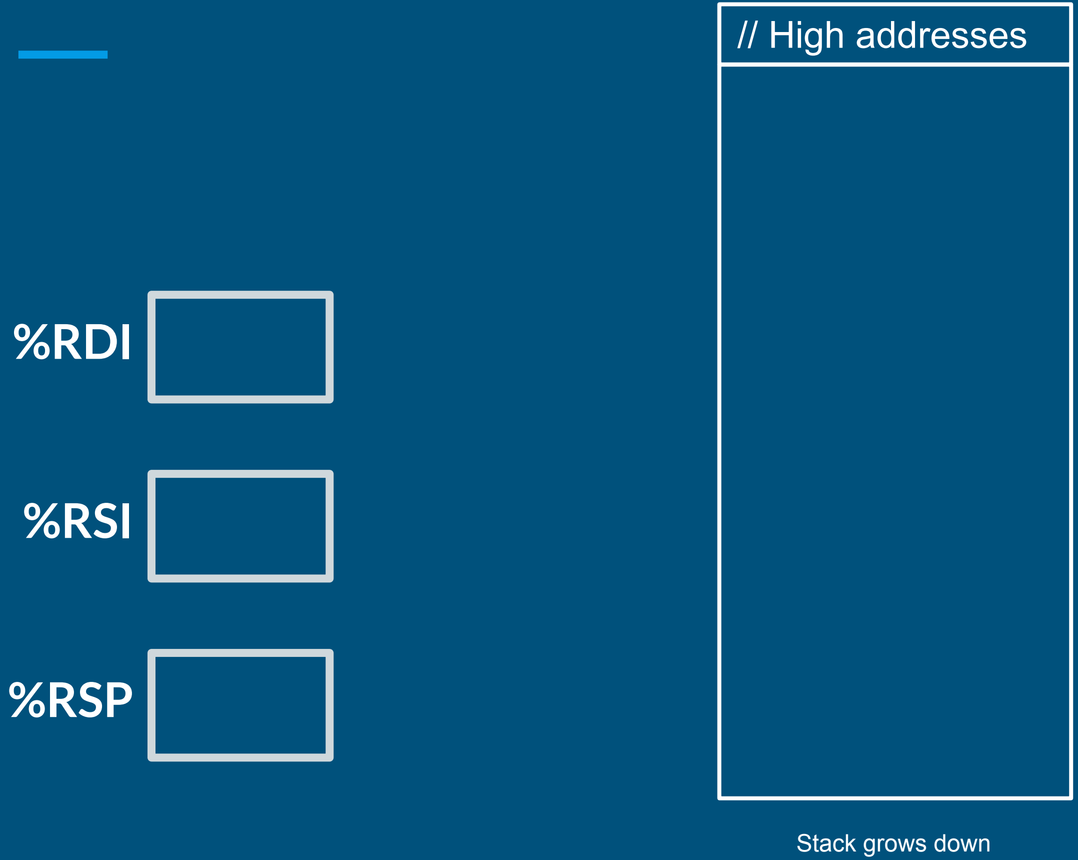
- RDI holds argc, which is 2
- RSI holds argv: the beginning of the argv array
- The specific value of the return PC doesn’t matter (program exits from main without returning)

Practice Exercise 1: "cat cat.txt" Solution



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

Practice Exercise 2

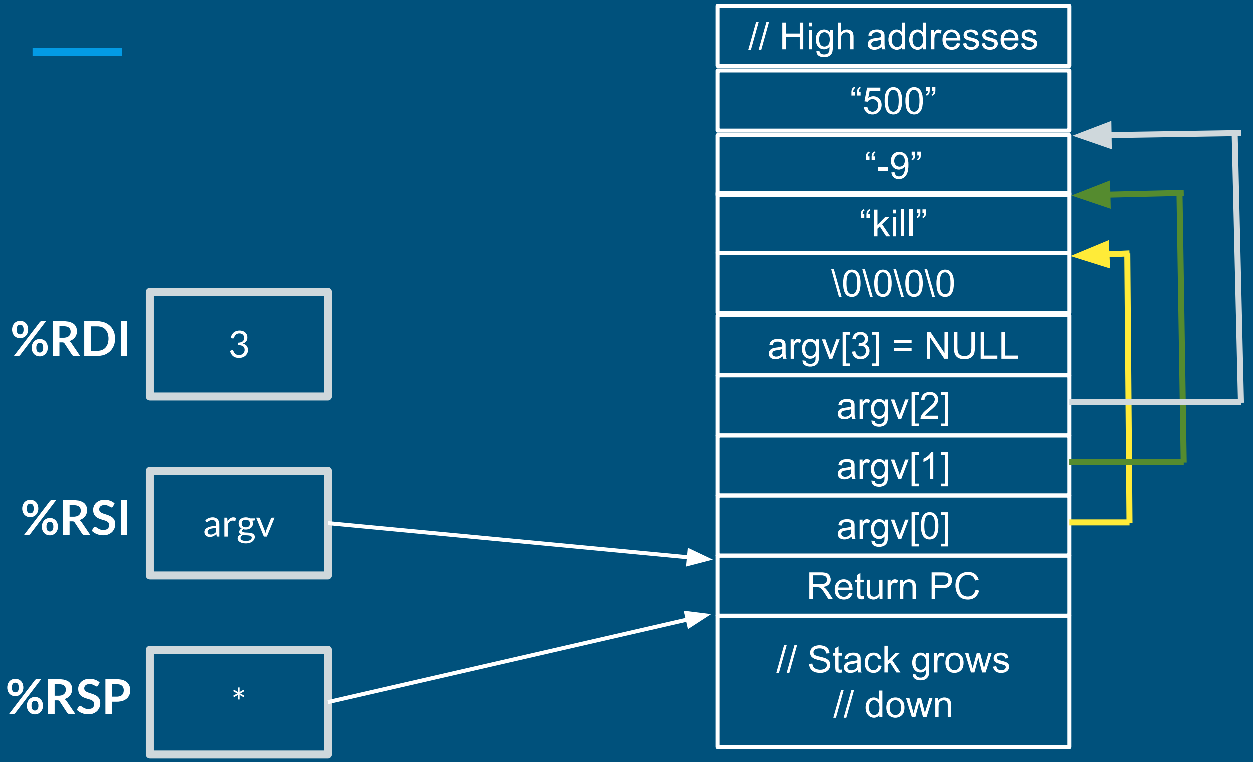


Now it's your turn!

Draw stack layout and determine register values for exec() called with:

"kill -9 500"

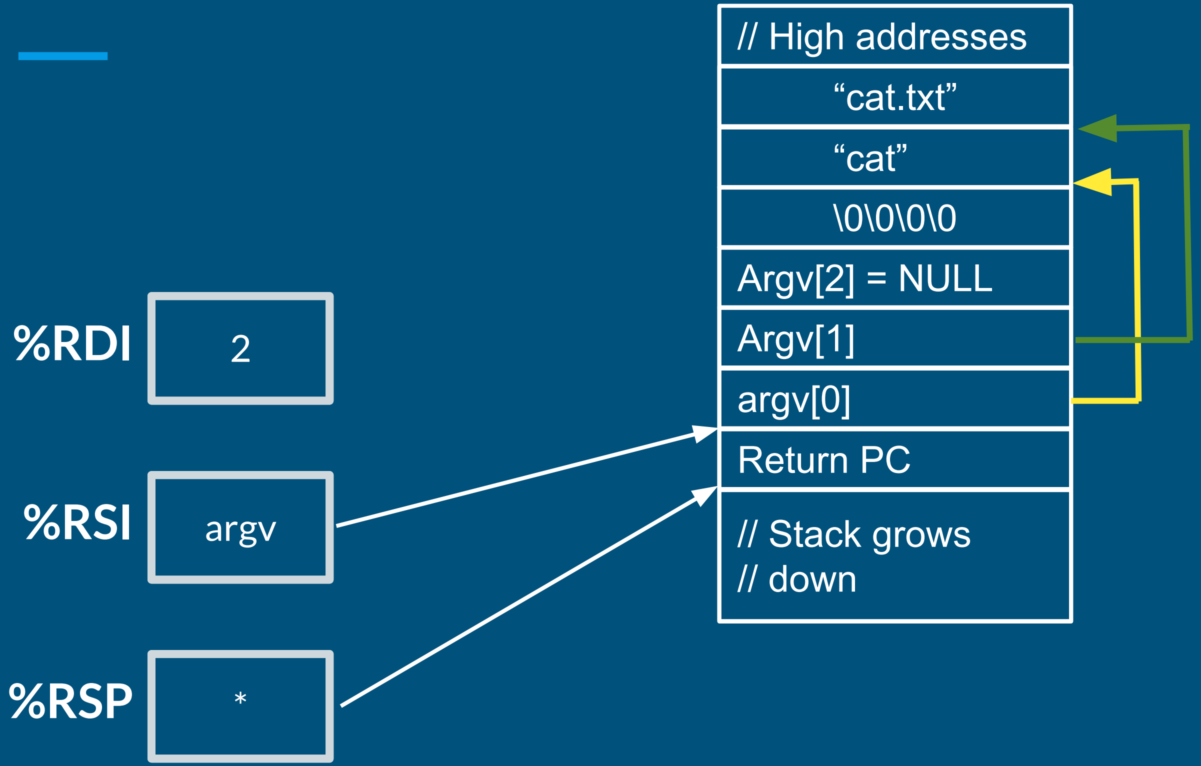
Practice Exercise 2: “kill -9 500” Solution



- RDI holds argc, which is 3
- 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)

Questions?

Practice Exercise 1: Solution



- 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)

Debugging Tips: Record & Replay

Starting with lab2, there are multiple processes, meaning more concurrent accesses to the kernel code, which might make bugs harder to reproduce.

`make qemu-record`

record all external events to a log file

helpful if you can record the race condition

`make qemu-gdb-replay`   `(pair with make gdb)`

replay according to the log file, but with gdb (similar to make qemu-gdb)

Monitor Pattern Example

Process 1
Status: running

Process 2
Status: runnable

Process 1 needs to wait for some condition which depends on process 2.

Monitor Pattern Example

Process 1
Status: asleep
on condvar

Process 2
Status: running

Process 1 goes to sleep on some channel related to this condition (doesn't matter what chan is, as long as both processes agree). At some point, Process 2 gets scheduled to run.

Monitor Pattern Example

Wake up all processes sleeping on condvar!

Process 1
Status: asleep on condvar

Process 2 did work that Process 1 was waiting for

Process 2
Status: running

When process 2 finishes its task, it wakes up all processes sleeping on the appropriate channel.

Monitor Pattern Example

Process 1
Status: runnable

Process 2
Status: runnable

Process 1 is set to runnable because of the wake up call.

Monitor Pattern Example

Process 1
Status: running

Process 2
Status: runnable

Process 1 is eventually scheduled to run and can continue its work.

Monitor Pattern Example

When the process wakes up, it should check the condition and go back to sleep if it's false.

Why?

Monitor Pattern Example 2

Process 1
Status: sleeping
on condvar

Process 2
Status: running

Process 3
Status: sleeping
on condvar

Now, there are 2 processes sleeping on the same channel.

Monitor Pattern Example 2

Process 1
Status: sleeping on condvar

Process 3
Status: sleeping on condvar

Wake up all processes sleeping on condvar!

Process 2
Status: running

Process 2 wakes up all processes sleeping on the channel.

Monitor Pattern Example 2

Process 1
Status: running

Process 2
Status: runnable

Process 3
Status: runnable

Both processes are woken up, and the scheduler decides to run Process 1.

Monitor Pattern Example 2

Process 1
Status: running

Process 2
Status: runnable

Process 3
Status: runnable

What if Process 1 does something that causes the condition to become false again?