Section 4: Lab 2 (contd.)

Section 4: 1/30/2020
Please pick up section handout as you come in :)

Administrative

- Lab 2 design doc part 2 due tomorrow (1/31)
- Lab 2 due next Friday
- Feedbacks are pushed to your repo as a separate branch, lives in the feedback folder

A little note:

- Both spawn and fork create children!
    - If process A spawns process B and forks process C, A is the parent of both B and C

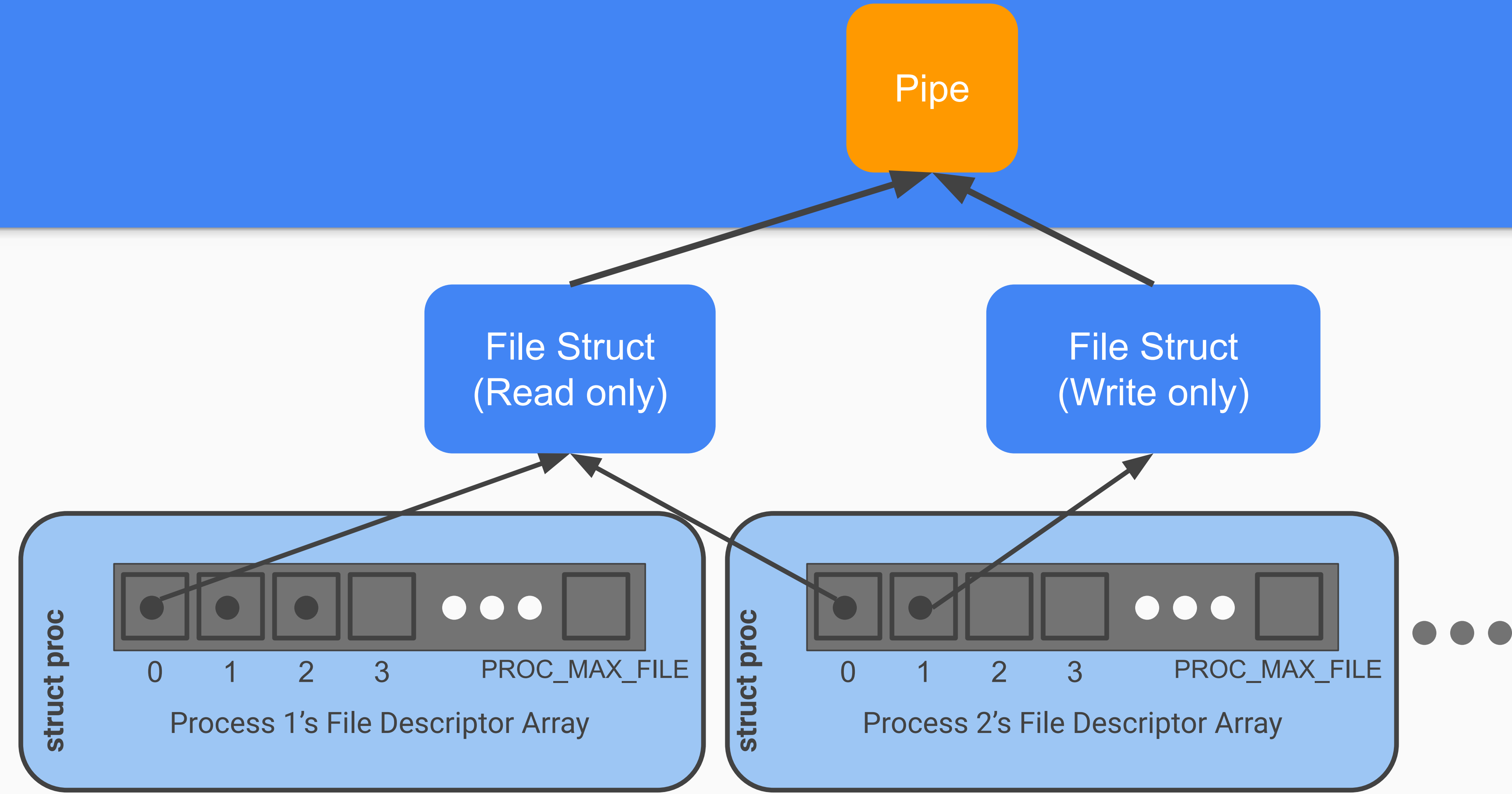
Pipes

What is a pipe

- A special “file” that is stored in memory
- Used for inter-process communication (IPC)

pipe(fds)

- With the `sys_pipe`, a process sets up a writing and reading end to a “holding area” (buffer) where data can be passed from process to process
    - From user’s perspective: Two new files will be allocated, one will be the “read end” (not writable), and one will be the “write end” (not readable).
- Pipe is blocking: when data is not available, reader blocks until new data is written, when buffer is full, writer blocks until reader reads data out of the buffer
    - Implementation: conditional variable



Pipe
File Struct
(Read only)
File Struct
(Write only)
struct proc
0
1
2
3
PROC_MAX_FILE
Process 1's File Descriptor Array
struct proc
0
1
2
3
PROC_MAX_FILE
Process 2's File Descriptor Array

Pipe allocation

- Pipes should be allocated at runtime, when sys_pipe is called
- A pipe is associated with two files: read_end and write_end
- file structure can be allocated through fs_alloc_file,

Changing the behavior for reading/writing

- Function pointers: a pointer that points to codes
- Example: stdin_read, stdout_write
- Set file operation handlers to point to pipe operations, bypassing the inode layer (the disk)

```
static struct file_operations pipe_file_operations = {
    .read = pipe_read,
    .write = pipe_write,
    .close = pipe_close
};

file->f_ops = &pipe_file_operations;
```

pipe(fds)

- Need a pipe struct to track information
  - A way to avoid race conditions - there can be many readers and writers
  - A way to notify the other end when the state changes
    - mechanism for reader to block and wakeup
    - mechanism for writer to block and wakeup
  - A buffer to store data
    - data itself
    - number of bytes written
  - need to know if read end or write end is closed
    - affects the other end
    - tell us when the pipe can be freed

Pipe Scenarios (exercise)

Section handout, page 1:

If a pipe no longer has a reader, a write call should return ERR_END.
If a pipe no longer has a writer and the buffer is empty, a read should return a 0.

- What should read return if the pipe no longer has a writer, but the buffer has data?
- What should happen if write end closes while the reader is sleeping?
    - does the reader sleep forever?
    - if not, what should it return when it wakes up?
- When can you clean up a pipe (buffer, allocated struct)?

Spawn with Args

Set up stack (spawn with args)

- So that we can start a process with arguments
- The arguments are stored at the bottom of the stack, before the stackframe of main function
    - because stack grows downwards, the bottom of the stack has higher address
- User perspective: *int main (int argc, char **argv)*
    - First argument will always be **argc** (number of arguments)
    - Second argument will always be **argv**, an array of strings (first string is always the name of the program)

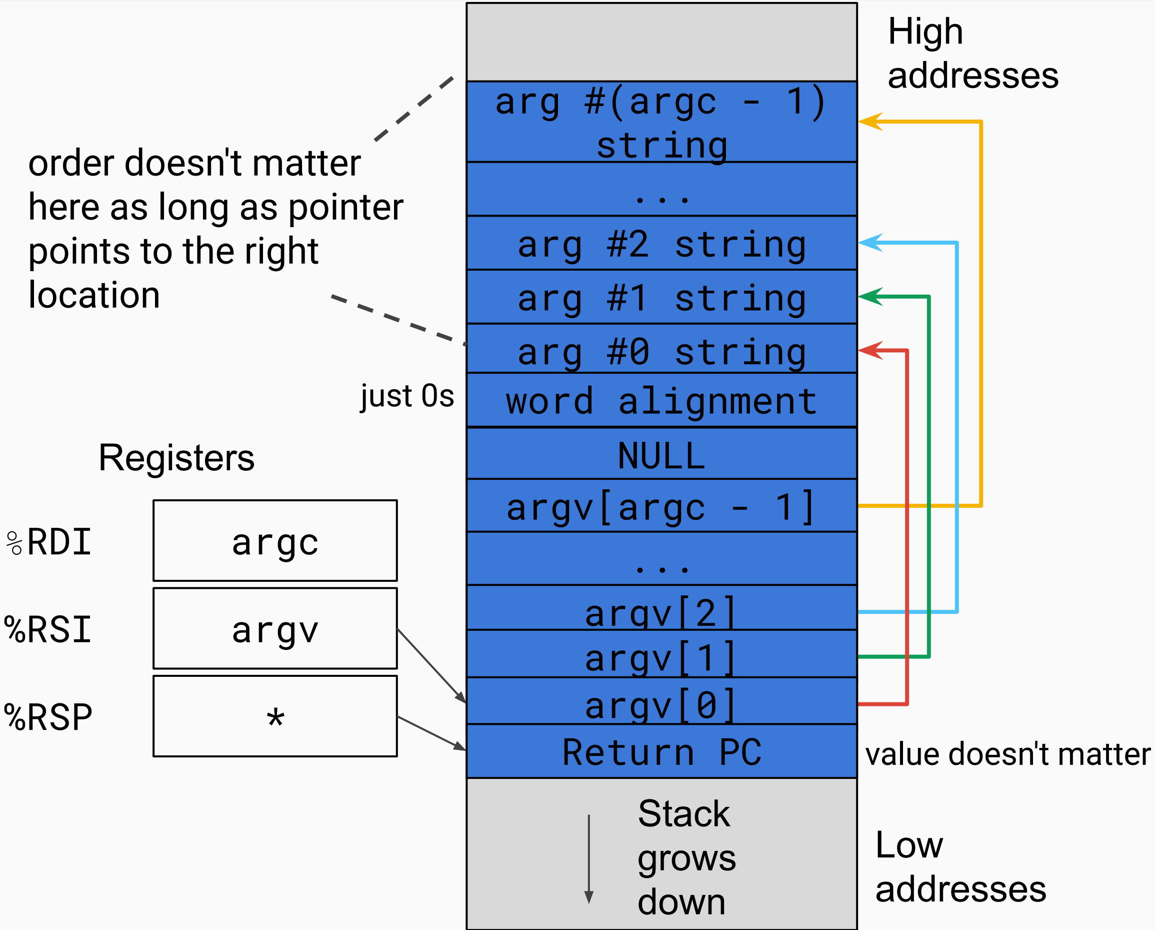
x86-64 Calling Conventions

- `%rdi`
  - Holds the first argument
- `%rsi`
  - Holds the second argument
- `%rsp`
  - Points to the top of the stack/lowest address (stack grows down)
- If arguments are arrays, store them on the stack and store a pointer to the array in the register

A little note

- The provides code sets up a pointer to the stack in kernel address space
- However, addresses pushed onto the stack must be the address in user's address space
- **USTACK_ADDR()** helps transform a kernel virtual address to user address
  - It only works for first page of stack, which is fine. we will only set up one page of stack on start up

Stack Layout



- argv is an array of pointers, therefore %RSI points to an array on the stack
- Since each element of the argv array is a char *, each element points to a string stored elsewhere on the stack.
- You can think of all variables stored above the return PC on the stack as local variables of the caller.
- **word alignment:** push 0 to stack until current stackptr is word aligned (multiple of 8s)

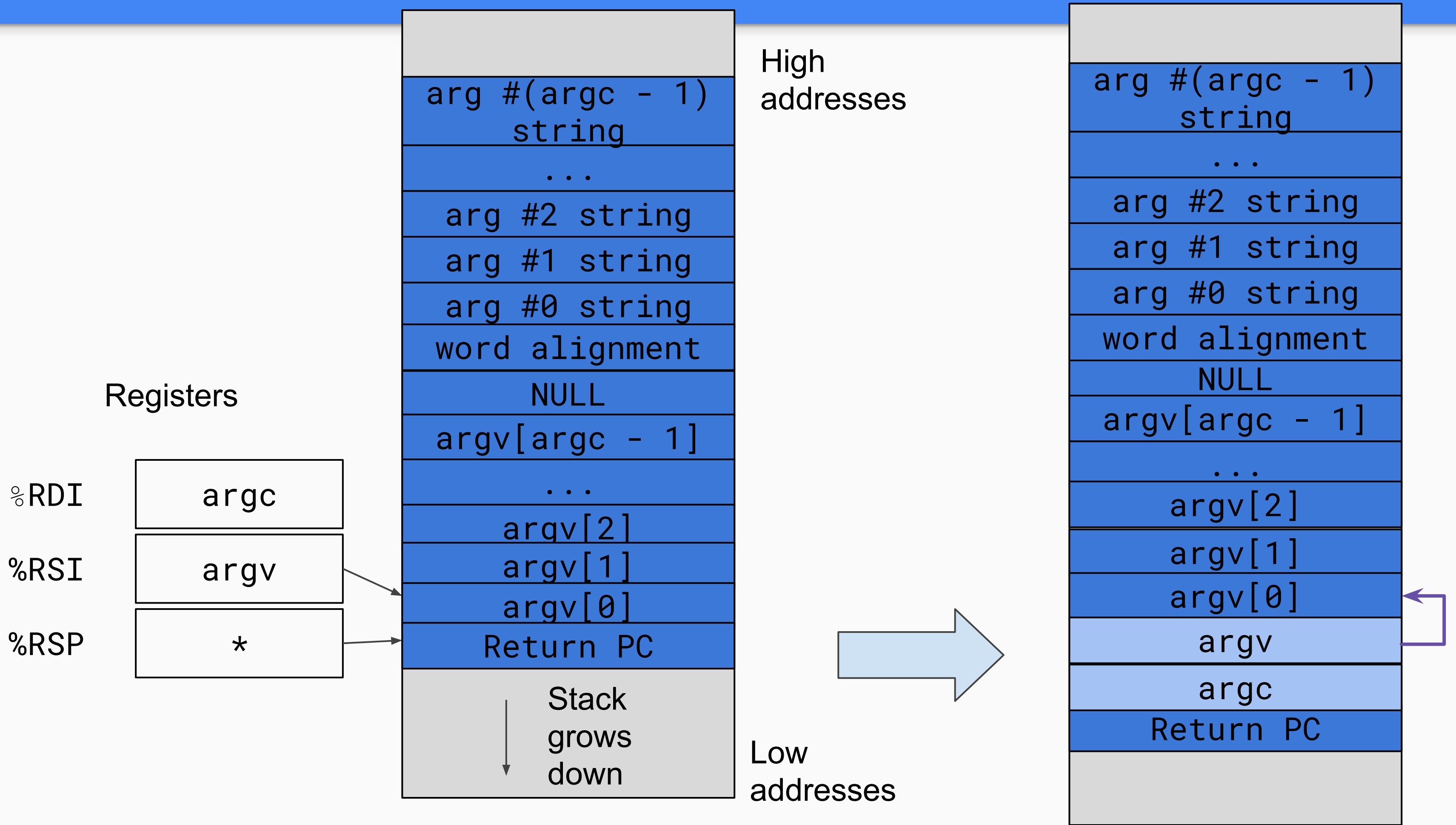
osv specific stack convention

- To accommodate different calling convention for various architectures, osv always pushes argv, argc, and return address on the stack in stack_setup
- Machine dependent tf_proc() will actually set up the proper registers

```
    tf->cs = (SEG_UCODE << 3) | DPL_USER;
    tf->ss = (SEG_UDATA << 3) | DPL_USER;
    tf->rflags = FL_IF;
    tf->rsp = stack_ptr;
    tf->rip = entry_point;
    // also need to set up arguments for new process
    tf->rdi = sp[1];
    tf->rsi = sp[2];
}
```

sp is stackptr in kernel virtual address

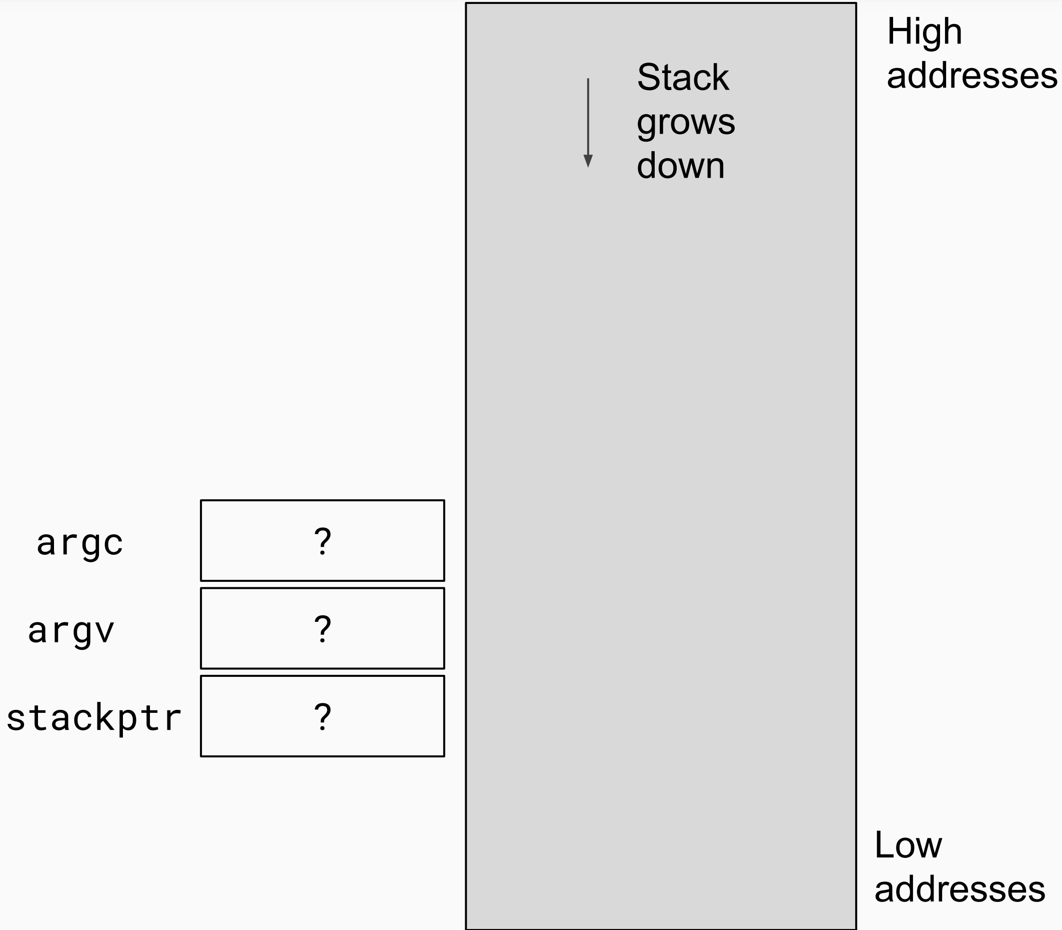
osv version


Registers
%RDI
argc
%RSI
argv
%RSP
*
arg #(argc - 1) string
...
arg #2 string
arg #1 string
arg #0 string
word alignment
NULL
argv[argc - 1]
...
argv[2]
argv[1]
argv[0]
Return PC
Stack grows down
High addresses
Low addresses
arg #(argc - 1) string
...
arg #2 string
arg #1 string
arg #0 string
word alignment
NULL
argv[argc - 1]
...
argv[2]
argv[1]
argv[0]
argv
argc
Return PC

Let’s Practice!

(Get out some paper and pens!)

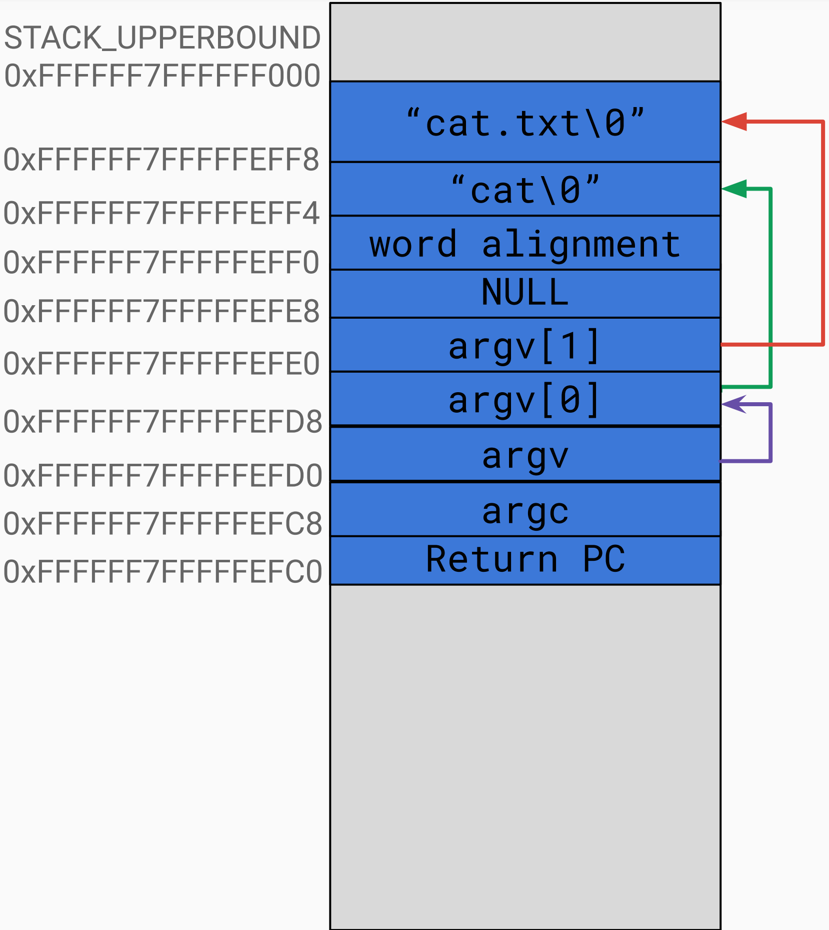
Practice Exercise 1 - spawn(“cat cat.txt”)



TODO:

Draw out the stack layout for process spawned with “cat cat.txt”.

Practice Exercise 1 - spawn("cat cat.txt") Solution

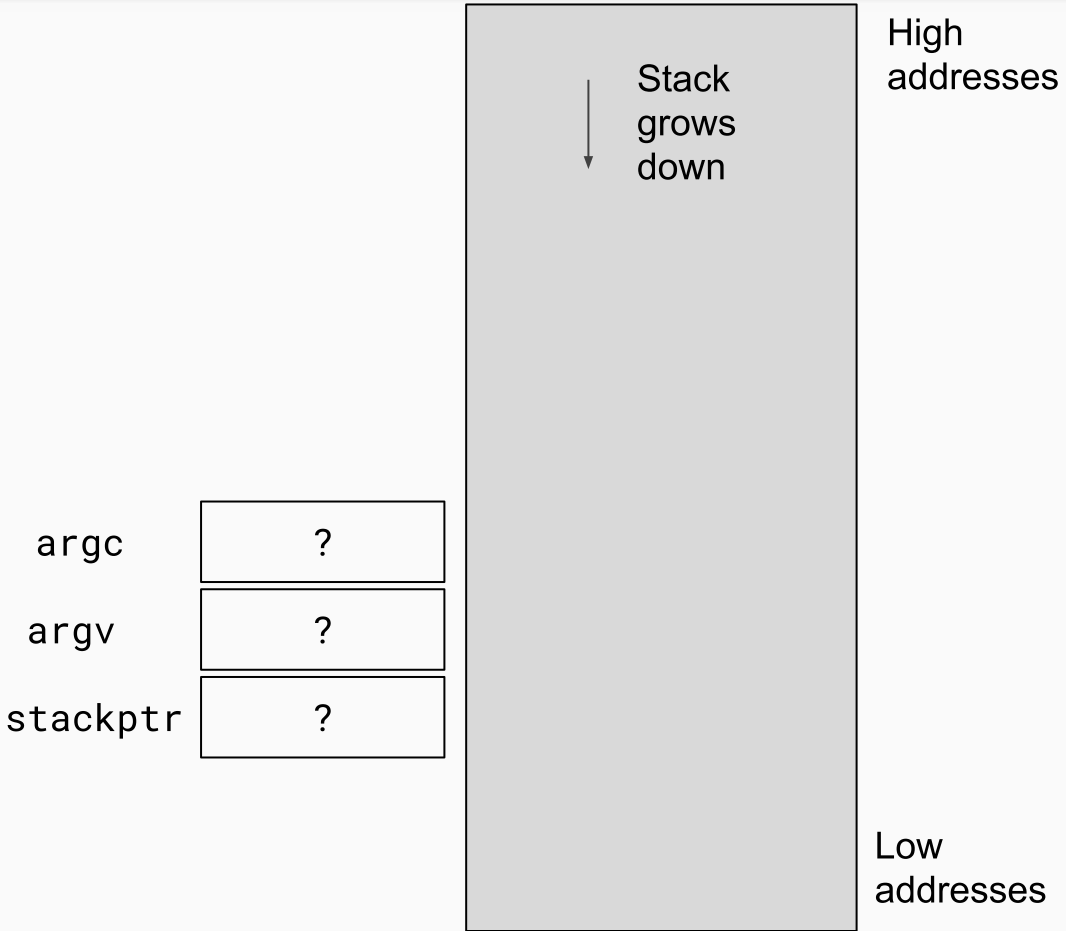


argc: 2

argv: 0xFFFFFF7FFFFFEFD8

stackptr: 0xFFFFFF7FFFFFEFC0

Practice Exercise 2 - spawn("echo hello world”)



TODO:

Draw out the stack layout for process spawned with “echo hello world”.

Practice Exercise 2 - spawn("echo hello world”)