Lab 2

Part 2
Admin

- Lab 2 has 2 parts with separate design docs and due dates
  - Part 1 Code due 4/22 (grace period and late days)
  - Part 2 Design due 4/24 (no grace period or late days)
  - Part 2 Code due 5/01 (grace period and late days)

- Pset 3 due tomorrow 4/19
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

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

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

argv / &argv[0]

must be validated for an 8 byte pointer before we can access argv[0] and validate string0
exec(path, argv) arguments validation

&argv[1] must be validated for an 8 byte pointer before we can access argv[1] and validate string1
exec(path, argv) arguments validation

- &argv[2] 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
  - vspacealoadcode 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**!

```c
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

// Stack grows down

(argc-1)string
[ ... ]
Arg #1 string
Arg #0 string
\0… (padding)
argv[argc] = NULL
argv[argc - 1]
[ ... ]
argv[1]
argv[0]

Return PC

// High addresses

argv

// Stack grows down

argv

argv[argc]

argv[argc - 1]

\0… (padding)

argv[1]

argv[0]

argv[argc]

argv[argc - 1]

\0… (padding)

argv[1]

argv[0]
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

// High addresses

// Stack grows
// down

%RDI

%RSI

%RSP

stackptr
Practice Exercise 1: “cat cat.txt” Solution

// High addresses

“cat.txt”

// Stack grows
// down

stackptr

%RDI

%RSI

%RSP
Practice Exercise 1: “cat cat.txt” Solution

// High addresses

“cat.txt”

“cat”

// Stack grows
// down

stackptr
Practice Exercise 1: “cat cat.txt” Solution

- RDI holds argc, which is 2

```
stackptr
```

<table>
<thead>
<tr>
<th>// High addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>“cat.txt”</td>
</tr>
<tr>
<td>“cat”</td>
</tr>
</tbody>
</table>

// Stack grows
// down
Practice Exercise 1: “cat cat.txt” Solution

// Stack grows
// down

stackptr

<table>
<thead>
<tr>
<th>// High addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>“cat.txt”</td>
</tr>
<tr>
<td>“cat”</td>
</tr>
<tr>
<td>\0\0\0\0</td>
</tr>
</tbody>
</table>

- RDI holds argc, which is 2
Practice Exercise 1: “cat cat.txt” Solution

- RDI holds argc, which is 2

<table>
<thead>
<tr>
<th>Stack Pointer</th>
<th>High Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL (argv[2])</td>
<td>“cat”</td>
</tr>
<tr>
<td>\0\0\0\0</td>
<td>“cat.txt”</td>
</tr>
</tbody>
</table>

%RDI: 2

%RSI: 

%RSP: 

stackptr
Practice Exercise 1: “cat cat.txt” Solution

%RDI 2

%RSI

%RSP

// High addresses
“cat.txt”
“cat”
\0\0\0\0
NULL (argv[2])
addr of "cat.txt" (argv[1])

// Stack grows
// down

● RDI holds argc, which is 2
Practice Exercise 1: “cat cat.txt” Solution

%RDI 2

%RSI

%RSP

// High addresses

“cat.txt”

“cat”

\0\0\0\0

NULL (argv[2])

addr of "cat.txt" (argv[1])

addr of "cat" (argv[0])

// Stack grows
// down

● 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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>%RDI</td>
<td>2</td>
</tr>
<tr>
<td>%RSI</td>
<td>argv</td>
</tr>
<tr>
<td>%RSP</td>
<td></td>
</tr>
</tbody>
</table>

// High addresses
- “cat.txt”
- “cat”
- \0\0\0\0
- NULL (argv[2])
- addr of "cat.txt" (argv[1])
- addr of "cat" (argv[0])

// Stack grows // down
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.

<table>
<thead>
<tr>
<th>%RDI</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>%RSI</td>
<td>argv</td>
</tr>
<tr>
<td>%RSP</td>
<td>stackptr</td>
</tr>
</tbody>
</table>

// High addresses
- “cat.txt”
- “cat”
- \0\0\0\0
- NULL (argv[2])
- addr of "cat.txt" (argv[1])
- addr of "cat" (argv[0])

Return PC

// Stack grows // down
Practice Exercise 2

// High addresses

%RDI
%RSI
%RSP

Stack grows down

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)

```
argv[3] = NULL
argv[2]
argv[1]
argv[0]
Return PC

// Stack grows // down
```

```
// High addresses
“500”
“-9”
“kill”
\0\0\0\0
```
Questions?