Lab 2

Part 2

Admin

- Lab 2 has 2 parts with separate design docs and due dates
  - part 1 design due today 1/25 (**no late days**) so we can give you timely feedback
  - part 2 design due 2/01 (**no late days**)
  - part 1 code due 2/02 (with late days)
  - part 2 code due 2/09 (with late days)
- Pset/ Quiz 1 due tomorrow 1/26
  - 11:59pm
  - No late days
  - 30 Questions on Gradescope on Week 1-3 content
  - Not timed

Monitors in xk

- Lock
  - xk condition variable API only supports spinlock (an impl. choice)
- Condition
  - the shared data that threads are synchronizing on
  - 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)

Sleep, Wakeup, and Chan

- 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

Sleep, Wakeup, and Chan

- 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(), pipe() for lab2
- sleep in synch.c is not the sleep system call

sleep = wait
wakeup = broadcast
no equivalent in xk = signal

```c
struct fridge {
  struct spinlock lk; // assume initialized
  int yogurt = 0;
  int strawberry = 0;
}

void make_breakfast(struct fridge* fridge) {
  acquire(&fridge->lk);
  while (fridge->yogurt == 0 && fridge->strawberry < 2) {
    // temporarily release the lk when we sleep
    // so that the fridge state may be accessed and modified
    // when sleep returns, lk is acquired again (implicitly)
    sleep(fridge, &fridge->lk);
  }
  // consume the yogurt and strawberry
  fridge->yogurt = 0;
  fridge->strawberry -= 2;
  release(&fridge->lk);
}

void fill_fridge(struct fridge* fridge) {
  acquire(&fridge->lk);
  fridge->yogurt += 1;
  fridge->strawberry += 2;
  wakeup(fridge);
  release(&fridge->lk);
}
```

Monitor Pattern Example

Process 1
Status: running

Process 2
Status: runnable

Process 1 needs to wait for some condition which depends on proces 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). 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: running

Process 2
Status: runnable

Process 1 wakes up and can continue 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 3
Status: sleeping
on condvar

Process 2
Status: running

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

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

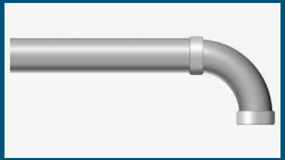
What if Process 1 does something that causes the condition to become false again?

Lab 2 - Pipe

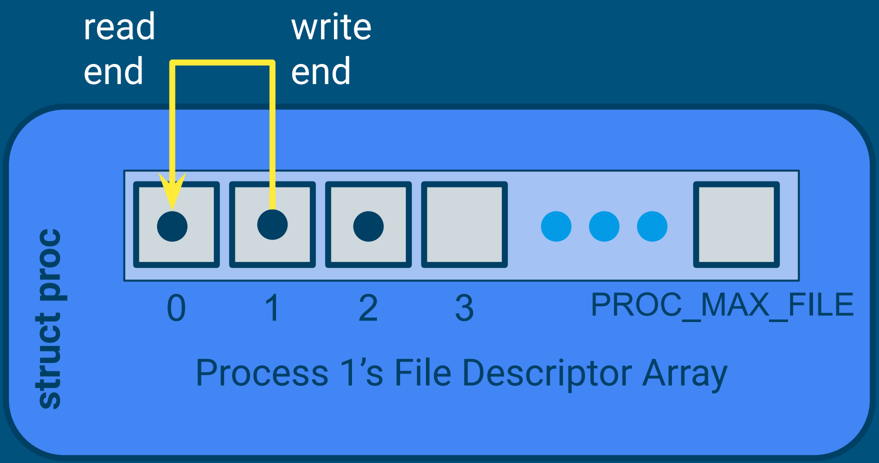
pipe(fds)

- Creates a pipe (kernel buffer) for process to read and write
- From the user perspective: returns two new file descriptors
  - fds[0] = “read end”, not writable
  - fds[1] = “write end”, is not readable
- You’ll want to make this compatible with existing file syscall interface
- Pipe allows processes to communicate with each other
  - parent opens a pipe, forks a child, and now they both have access to the pipe ends
  - typically one process only leaves one end open (closes the read end or the write end)

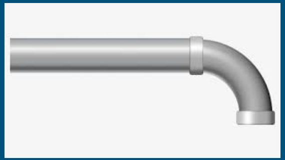
Pipes



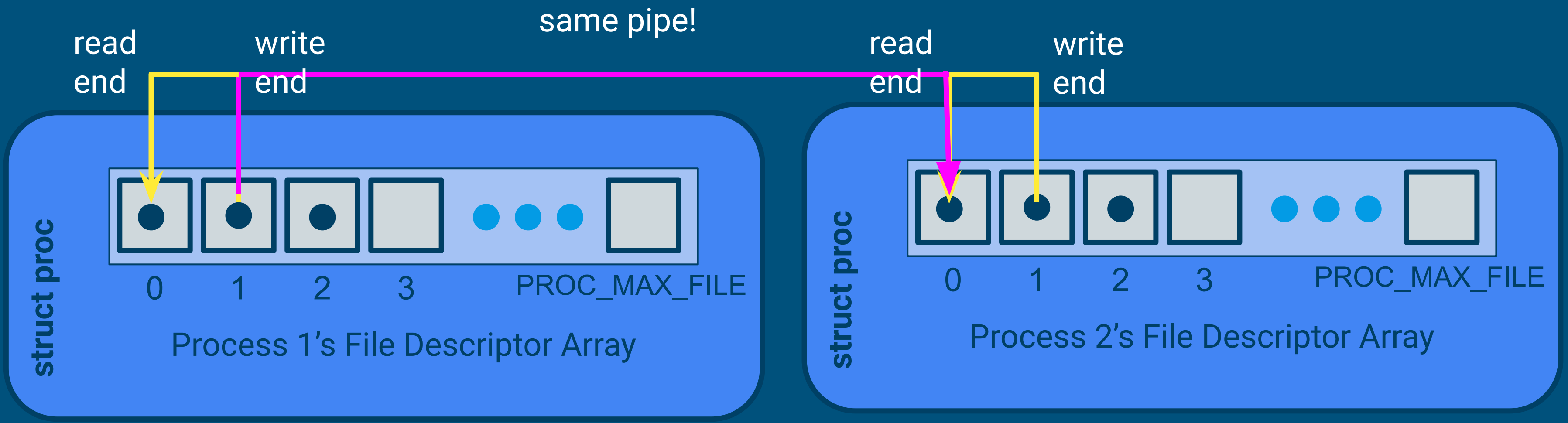
- A mechanism for process communication
- By calling sys_pipe, a process sets up a writing and reading end to a "holding area" where data can be passed between processes



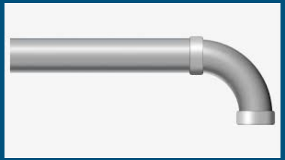
Pipes



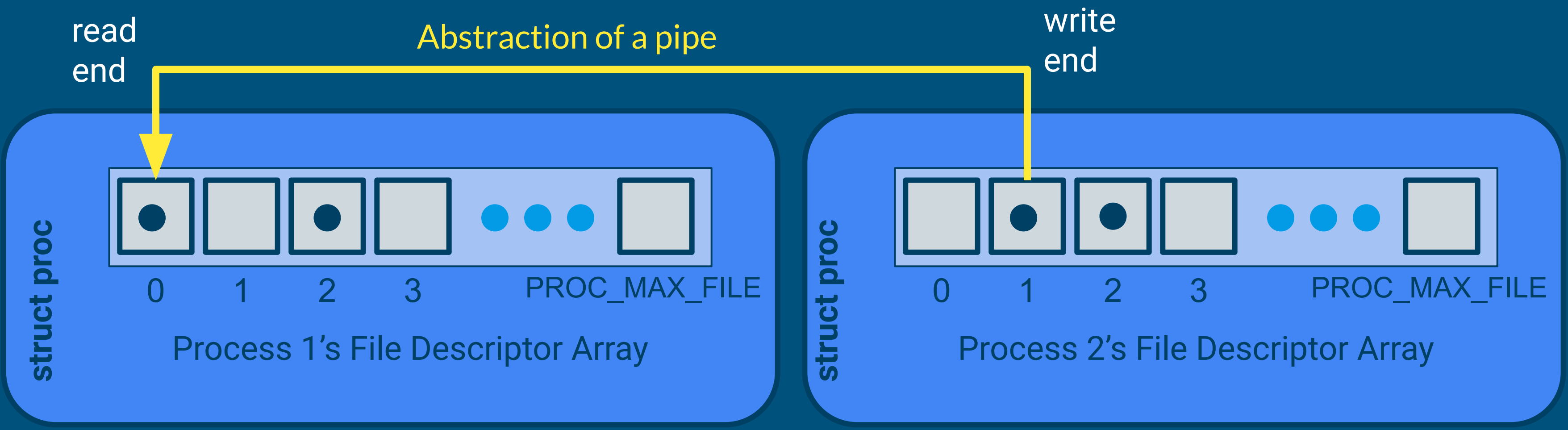
- Process 1 calls fork(), fd table is duplicated



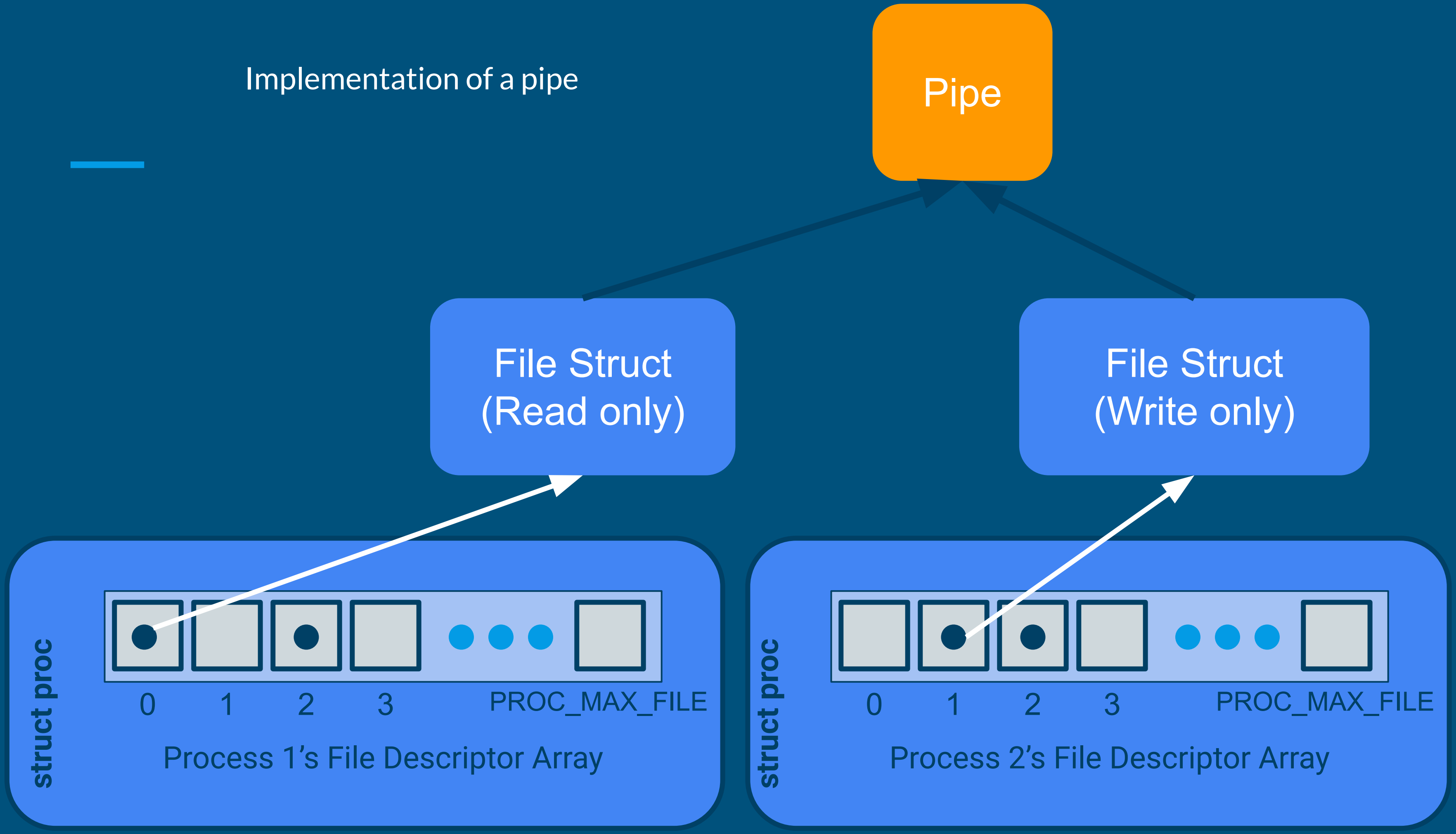
Pipes



- Process 1 close(1), process 2 close(0)
- And now we have a pipe across processes



Implementation of a pipe



Pipes

- Where should pipe be allocated?
  - pipes should be allocated at runtime, as requested
  - how does xk do dynamic memory allocation?
    - (hint: kstack is also dynamically allocated)
- When can you free the pipe and its buffer?
  - remember there may be multiple read ends and write ends
- 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?
- Pipe operations go through file syscall
  - Need a way to determine if a struct file is an inode or a pipe

Pipes Impl. Tips

- What metadata/information do you need for pipe?
  - offset to read from
  - offset to write to
  - whether the read end is still open
  - whether the write end is still open
  - # of bytes available in the buffer
  - lock and condition variables
  - PID of waiting writer
- Similar to the bounded buffer problem

Lab 2 - Exec

exec(program, args)

- Fully replaces the current program; it does not create a new process
- How to 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(program, args)

- Setting up a new virtual address space
  - `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!
  - Look at what vspacefree does

exec(program, args): args setup

int main(int argc, char** argv)

argc: The number of elements in argv

argv: An array of strings representing program arguments

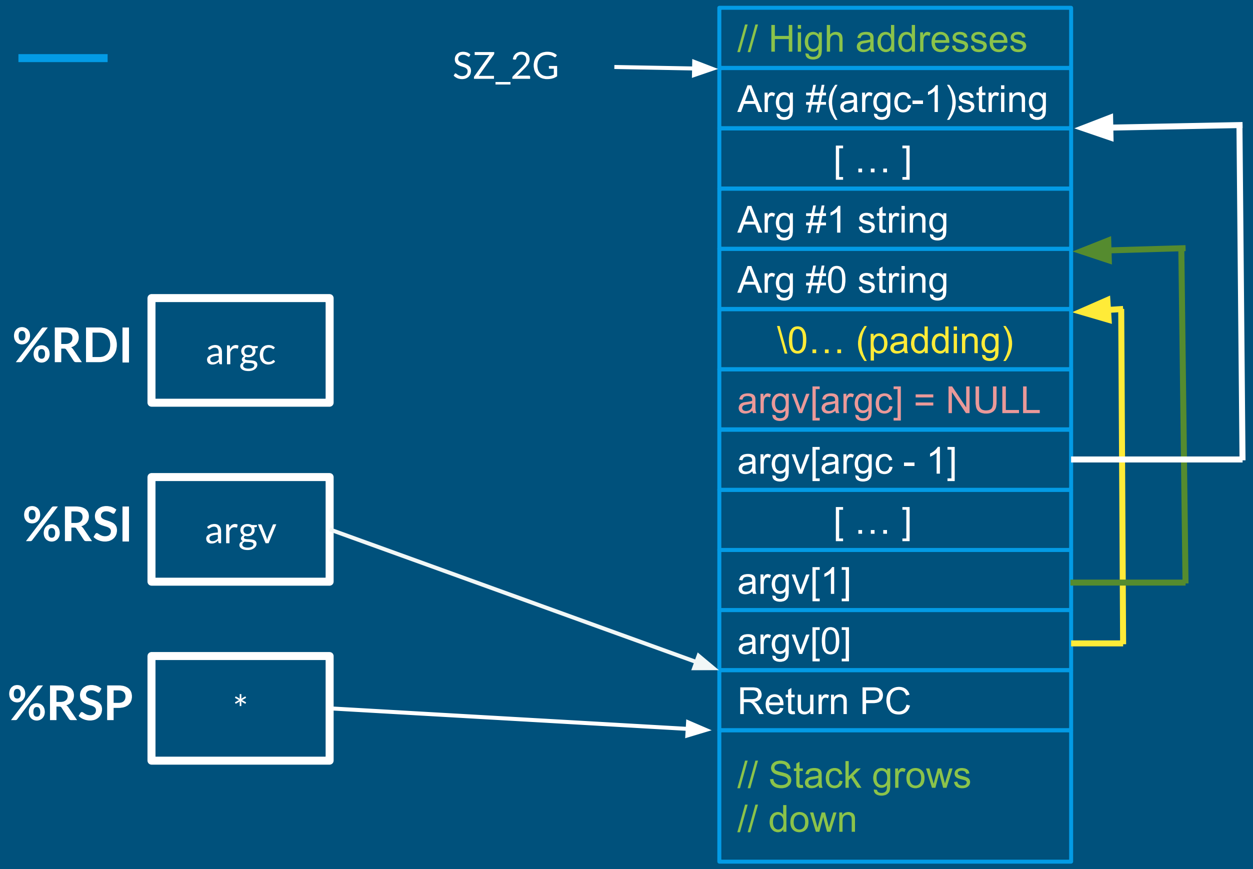
- First is always the name of the program
- Argv[argc] = 0

X86_64 Calling Conventions

- %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)

- 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

Questions?

Autograder Tips

- Autograder runs each test individually and then all part1/part2 tests
- part1 and part2 tests are run with make ICOUNT=2/4/6/8/10
  - ICOUNT is an argument to the Makefile
    - should make your bug show up more consistently (per configuration)
    - vary the amount of instruction interleaving (with different icount values)
    - ICOUNT is default to 10 when you run make qemu
  - If your kernel fails on certain ICOUNT config, you can reproduce it locally with make qemu ICOUNT=2/4/6/8/10 to debug

Debugging Tips: Trap Errors

- Trap Errors
  - `unexpected trap 14 from cpu 0 rip ffffffff80102f27 (cr2=0x0)`
  - `trap 14`: page fault, invalid memory access (most of the time)
  - `rip ffffffff80102f27`: line of code caused the page fault
  - `cr2=0x0`: the memory address that caused the page fault

```
(gdb) info line *0xffffffff80102f27
Line 41 of "kernel/sysfile.c"
   starts at address 0xffffffff80102f23 <sys_write+85>
   and ends at 0xffffffff80102f2d <sys_write+95>.
```

```
 40    int *a = NULL;
 41    *a = 4;
```

For more details, check out debugging.md

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)