```
******************************************************************************
* Announcements

1. TAs: 4 TA/graders assigned, office hours TBA
2. VMs: being prepared
3. HW1 draft almost done

******************************************************************************
* What's a OSs role?

Traditional:
- manages h/w resources, sits b/t users and h/w
- ensure safe access to h/w, security, fairness, efficiency

Linus sez:
- OS has to regulate access to shared data structures

******************************************************************************
* quiz C program solution

/* use C comments */

// Here's a sample kernel C code snippet.  It could be used by parts of the
// kernel which create new processes, and need to assign a process ID (PID) to
// a new process/task.  There are several serious bugs and some smaller
// problems in this code.  Identify them as best you can.  You may sketch out
// new code below.

#define MAX 1024
int process_list[MAX];  // assume initialized at OS start time
// need to define "locking semantics": what lock(s) are used on what data
// structures, who needs to take/release them and when.
lock_t pid_l; // ensure it's initialized to proper value

// describe function purpose, arguments (if any), return values (success and
// failure),  any side effects (does it malloc/free or un/lock anything)
// that the caller needs to know about.  Arguments: const vs. non-const.
// calling conventions (does caller expect to pass valid/initialized
// pointers, are objects supposed to be locked already?)

int get_new_process_id(void) // need a type
{
  int i;
  lock(pid_l); // this'll protect reading+writing to shared d-s
  for (i = 0; i < MAX; i++) {
    //    lock(pid_l); // ?
    if (process_list[i] == 0) {
      //      lock(pid_l); // not enough to protect writing to shared d-s
      process_list[i] = 1;
      unlock(pid_l);
      return(i);
    } // end of "if" statement
    //    unlock(pid_l);
  } // end of "for" loop
  // no free slot, need to return some error
  unlock(pid_l); // don't leave fxn w/ lock held! (leads to deadlocks)
  return -ENOSPC; // ENOMEM, anything else that's helpful
}

int release_process_id(pid_t p)
{
  // grab same lock
  // assign 0 to index p of array
  // return success/failure
```

```
    // may want to check that the slot value is '1' indeed. any other value
    // would indicate a bug/error SOMEWHERE in the code.
  }

  // SERIOUS ISSUES:

  // initialization

  // when array is full, what value do we return? need to return some
  // non-valid value, error path through function. (May or may not be caught
  // by the compiler).

  // exceeding process_list array bounds by one.  Buffer overflow, array index
  // out of bounds, off-by-one bug.

  // assignment inside 'if' instead of '==' comparison

  // locking!!!


  // EFFICIENCY

  // we're always iterating from 0..1023, most low numbered slots are full, so
  // we're wasting time looking at them.  Instead, we could keep some sort of
  // an index of the "last used slot" and start from there.  If we add another
  // variable, do we need to protect it?  Using same lock or another?  Maybe
  // we don't need to lock it -- what will be the impact (maybe just
  // inefficiency).  But Now we'll also have to cycle back to start of array,
  // if we can't find an empty slot b/t current point and 1023.

  // we're using an int (32 bits) to store just a Boolean 0/1.  Instead, use a
  // boolean array (if supported by compiler), else use a bitmap.
  // int process_list[MAX / sizeof(int)];
  // but now finding the slot number requires bitwise ops on the actual int
  // inside this more compact array.


  // MINOR ISSUES:

  // misleading indentation on L22


  ******************************************************************************
  * System calls

  a function running inside kernel, on behalf of a user process
  - running in a different addr space, so how do communicate?
  - user processes have NO access to the syscall code or d-s
  - kernel cannot assume valid inputs

  Communicate b/t user and kernel:

  - user need to communicate: args, what syscall to run

  - kernel needs to communicate back: return value, errno, fill in buffers
    provided by user process.

  - 1. user process can write to its own virtual memory, and kernel can find the
    actual phys. memory to read info.  But mem is slower than registers.  Use
    mem only on architectures where you don't have enough registers, and you
    need to pass "many" arguments.

  - 2. CPU registers to communicate args and retval.  Registers are faster to
    use, but there's only a handful of them available.
```

```
- use an interrupt (exception?) to tell kernel that there's a syscall you
  want to execute.

The process to call a system call:

(A) In Userland

1. decide on method to communicate with kernel (assume registers)
2. put args for syscall into registers R2, R3, R4, ...
- registers are CPU word wide (32 or 64 bits) [int is 4B, or 32 bits long)
- this means you cannot pass wholesale buffers/strings, only start mem addr
  of a buf in userland.  Meaning you pass pointers or INTs only.
3. store syscall number into a register, say R1
- can't pass syscall name -- strings not efficient as numbers
- meaning: syscall numbers MUST match b/t user and kernel!
4. call a system call specific interrupt (Int 80h)
- invoking the CPU inside the OS.

(B) In kernel mode

1. got an interrupt 80h
2. CPU will preserve current state (whatever's running on CPU now)
3. call service routine for this interrupt (syscall handler)
4. service routine runs
- which process invoked the interrupt -- put process into WAIT/SLEEP state
- get syscall num from R1
- prepare stack frame from R2, R3, R4, etc. (manually done using assembly
  code)
- invoke appropriate sys_XXX function (from dispatch table syscall_table[R1])
5. now we're actually running syscall code, functions often called sys_read,
   sys_write, etc.
- assume sys_XXX is done, put retval into yet another register, say R5
6. wake up process (move from WAIT to READY)
- some time later, scheduler picks process and runs it...

(C) back in userland
- process wakes up
- looks at shared R6 for retval
- based on retval, can return value to caller inside process
- or handle error numbers.

Syscalls (man page) return on error -1
- specific errors come from errno (global var in libc)
Kernel return -errno (ENOENT, ENOMEM)
- in libc, there's syscall wrappers, that check "R6", and based on that they
assign the error to global errno var, and then force a return of -1 to
caller.
```