Section 3: Synchronization and Pipes

CS 162

February 14, 2020

Contents

1 Vocabulary

- **int signal(int signum, void (*handler)(int))** - signal() is the primary system call for signal handling, which given a signal and function, will execute the function whenever the signal is delivered. This function is called the signal handler because it handles the signal.
- **pipe** - A system call that can be used for interprocess communication.

  More specifically, the `pipe()` syscall creates two file descriptors, which the process can `write()` to and `read()` from. Since these file descriptors are preserved across `fork()` calls, they can be used to implement inter-process communication.
- **test_and_set** - An atomic operation implemented in hardware. Often used to implement locks and other synchronization primitives. In this handout, assume the following implementation.

```
int test_and_set(int *value) {
    int result = *value;
    *value = 1;
    return result;
}
```

- **compare_and_swap (CAS)** - An atomic operation implemented in hardware. This operation compares the contents of a memory location, `ptr` with some value `old`. If the contents are equal, CAS will write `new` into `ptr` and return `true`; otherwise, it will return `false`.

```
bool compare_and_swap(int *ptr, int old, int new) {
    if(*ptr != old){
        return false;
    } else{
        *ptr = new;
        return true;
    }
}
```

- **Condition Variable** - A synchronization variable that provides serialization (ensuring that events occur in a certain order). A condition variable is defined by:
  - a lock (a condition variable + its lock are known together as a **monitor**)
  - some boolean condition (e.g. `hello < 1`)
  - a queue of threads waiting for the condition to be true

  In order to access any CV functions **OR** to change the truthfulness of the condition, a thread must/should hold the lock. Condition variables offer the following methods:
  - **cv_wait(cv, lock)** - Atomically unlocks the lock, adds the current thread to **cv**'s thread queue, and puts this thread to sleep.
  - **cv_notify(cv)** - Removes one thread from **cv**'s queue, and puts it in the ready state.
  - **cv_broadcast(cv)** - Removes all threads from **cv**'s queue, and puts them all in the ready state.

When a **wait()**ing thread is notified and put back in the ready state, it also re-acquires the lock before the **wait()** function returns.
When a thread runs code that may potentially make the condition true, it should acquire the lock, modify the condition however it needs to, call notify() or broadcast() on the condition's CV, so waiting threads can be notified, and finally release the lock.
Why do we need a lock anyway? Well, consider a race condition where thread 1 evaluates the condition C as false, then thread 2 makes condition C true and calls **cv.notify**, then 1 calls **cv.wait** and goes to sleep. Thread 1 might never wake up, since it went to sleep too late.

- **Hoare Semantics** - In a condition variable, wake a blocked thread when the condition is true and transfer control of the CPU and ownership of the lock to that thread immediately. This is difficult to implement in practice and generally not used despite being conceptually easier to deal with.
- **Mesa Semantics** - In a condition variable, wake a blocked thread when the condition is true with no guarantee on when that thread will actually execute. (The newly woken thread simply gets put on the ready queue and is subject to the same scheduling semantics as any other thread.) The implications of this mean that you must check the condition with a while loop instead of an if-statement because it is possible for the condition to change to false between the time the thread was unblocked and the time it takes over the CPU.

2 Warmup

2.1 Hello World

Will this code compile/run?
Why or why not?

```
pthread_mutex_t lock;
pthread_cond_t cv;
int hello = 0;

void print_hello() {
    hello += 1;
    printf("First line (hello=%d)\n", hello);
    pthread_cond_signal(&cv);
    pthread_exit(0);
}

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, (void *) &print_hello, NULL);
    while (hello < 1) {
      pthread_cond_wait(&cv, &lock);
    }
    printf("Second line (hello=%d)\n", hello);
}
```

3 Signals

The following is a list of standard Linux signals:

```
Signal     Value      Action       Comment
--------------------------------------------------------------------
SIGHUP        1       Terminate    Hangup detected on controlling terminal
                                   or death of controlling process
SIGINT        2       Terminate    Interrupt from keyboard (Ctrl - c)
SIGQUIT       3       Core Dump    Quit from keyboard (Ctrl - \)
SIGILL        4       Core Dump    Illegal Instruction
SIGABRT       6       Core Dump    Abort signal from abort(3)
SIGFPE        8       Core Dump    Floating point exception
SIGKILL       9       Terminate    Kill signal
SIGSEGV      11       Core Dump    Invalid memory reference
SIGPIPE      13       Terminate    Broken pipe: write to pipe with no
                                   readers
SIGALRM      14       Terminate    Timer signal from alarm(2)
SIGTERM      15       Terminate    Termination signal
SIGUSR1   30,10,16    Terminate    User-defined signal 1
SIGUSR2   31,12,17    Terminate    User-defined signal 2
SIGCHLD   20,17,18    Ignore       Child stopped or terminated
SIGCONT   19,18,25    Continue     Continue if stopped
SIGSTOP   17,19,23    Stop         Stop process
SIGTSTP   18,20,24    Stop         Stop typed at tty
SIGTTIN   21,21,26    Stop         tty input for background process
SIGTTOU   22,22,27    Stop         tty output for background process
```

3.1 Signal Handlers

Fill in the blanks for the following function using syscalls such that when we type Ctrl-C, the user is prompted with a message: "Do you really want to quit [y/n]? ", and if "y" is typed, the program quits. Otherwise, it continues along.

```
void sigint_handler(int sig)
{
    char c;
    printf(\Ouch, you just hit Ctrl-C?. Do you really want to quit [y/n]?");
    c = getchar();
    if (c == 'y' || c == 'Y')
        ____________________________;
}

int main() {
    ____________________________;
    ...

}
```

4 Pipes

4.1 Basic Pipes

In the following code we use a pipe to communicate data between 2 file descriptors.

```
int main() {
     int fds[2];
     pipe(fds);
     int rfd = fds[0];
     int wfd = fds[1];
     char *str = "hello world";
     size_t bytes_written = 0;
     size_t total = 0;
     while ((bytes_written = write(wfd, &str[total], strlen(&str[total]) + 1) {
         total += bytes_written;
     }
     char *read = malloc(strlen(str) + 1);
     total = 0;
     size_t bytes_read;
     while ((bytes_read = read(rfd, &read[total], 50)) {
         total += bytes_read;
     }
     printf("%s", read);
     return 0;
}
```

What would the code above print out?

4.2 Pipe and Fork

Now, we use pipes in order for 2 processes to share data between each other.

```
int main() {
    int fds[2];
    pipe(fds);
    pid_t child_pid = fork();
    if (child_pid) {
         int rfd = fds[0];
         int wfd = fds[1];
         char *str = "hello world";
         size_t bytes_written = 0;
         size_t total = 0;
         while ((bytes_written = write(wfd, &str[total], strlen(&str[total]) + 1) {
             total += bytes_written;
            }
    } else {
         char *read = malloc(strlen(str) + 1);
         total = 0;
```

```
         size_t bytes_read;
         while ((bytes_read = read(rfd, &read[total], 50)) {
             total += bytes_read;
         }
             printf("%s", read);
        }
        return 0;
}
```

What would the above program print?

4.3 Pipes and Forks Again

```
int main() {
    int fds[2];
    pipe(fds);
    int rfd = fds[0];
    int wfd = fds[1];
    pid_t child = fork();
    if (child) {
         close(rfd);
         close(wfd);
         // write(wfd, "hello world", 12);
         return 0;
    }
    char *str = "hello world";
    size_t bytes_written = 0;
    size_t total = 0;
    while ((bytes_written = write(wfd, &str[total], strlen(&str[total]) + 1) {
             total += bytes_written;
    }
    char *read = malloc(strlen(str) + 1);
    total = 0;
    size_t bytes_read;
    while ((bytes_read = read(rfd, &read[total], 50)) {
         total += bytes_read;
    }
    printf("%s", read);
    return 0;
}
```

What would the above program print?

We rerun the program except now the write syscall is uncommented. What happens during execution to the child and parent process?

5 Synchronization

5.1 test_and_set

In the following code, we use test_and_set to emulate locks.

```
int value = 0;
int hello = 0;

void print_hello() {
    while (test_and_set(&value));
    hello += 1;
    printf("Child thread: %d\n", hello);
    value = 0;
    pthread_exit(0);
}

void main() {
    pthread_t thread1;
    pthread_t thread2;
    pthread_create(&thread1, NULL, (void *) &print_hello, NULL);
    pthread_create(&thread2, NULL, (void *) &print_hello, NULL);
    while (test_and_set(&value));
    printf("Parent thread: %d\n", hello);
    value = 0;
}
```

Assume the following sequence of events:
1. Main starts running and creates both threads and is then context switched right after
2. Thread2 is scheduled and run until after it increments hello and is context switched
3. Thread1 runs until it is context switched
4. The thread running main resumes and runs until it get context switched
5. Thread2 runs to completion
6. The thread running main runs to completion (but doesn't exit yet)
7. Thread1 runs to completion

Is this sequence of events possible? Why or why not?

At each step where `test_and_set(&value)` is called, what value(s) does it return?

Given this sequence of events, what will C print?

Is this implementation better than using locks? Explain your rationale.

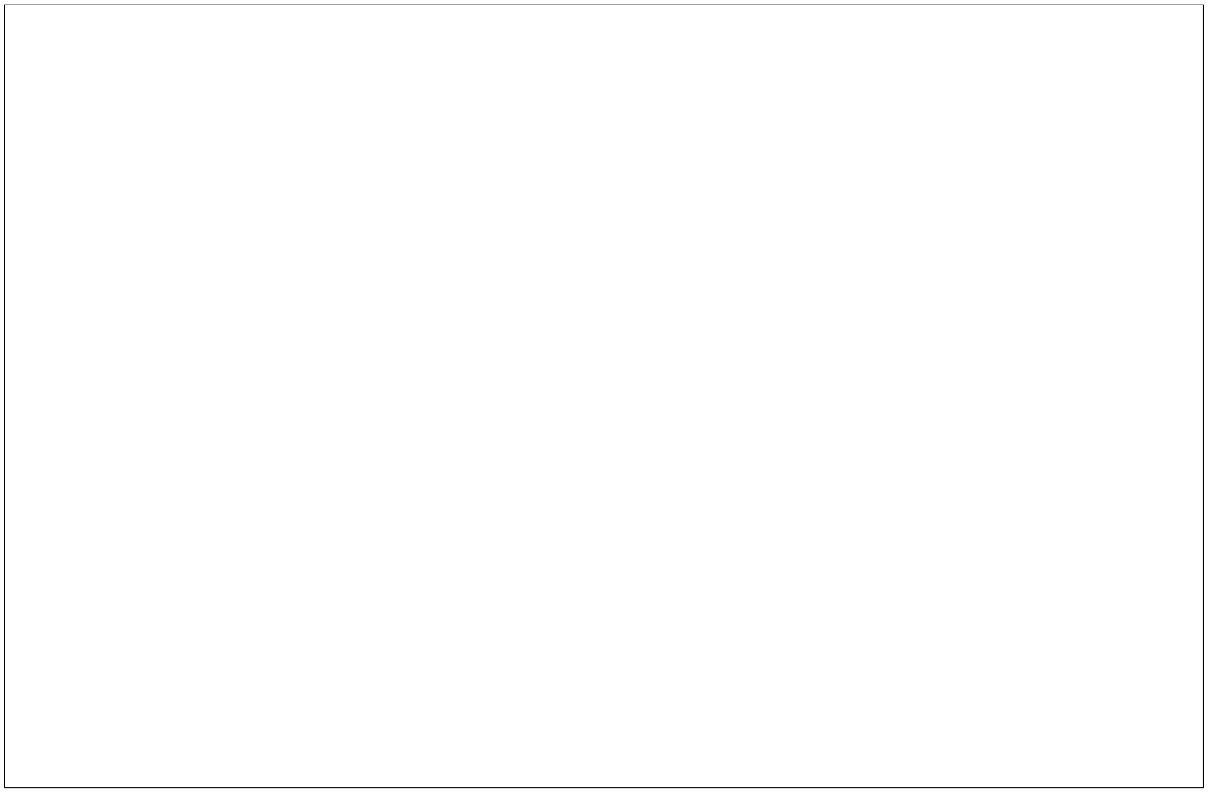
5.2 Condition Variables

Consider the following block of code. How do you ensure that you always print out "Yeet Haw"? Assume the scheduler behaves with Mesa semantics. (Pseudocode is OK) You may only add lines, so the trivial answer of not checking the value of ben before printing is not correct.

```
int ben = 0;

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (ben == 1) {
        printf("Yeet Haw\n");
    } else {
        printf("Yee Howdy\n");
    }
    exit(0);
}

void *helper(void *arg) {
    ben += 1;
    pthread_exit(0);
}
```



5.3 CS162 Office Hours

Suppose we want to use condition variables to control access to a CS162 office hours room for three types of people: students, TA's, and professors. A person can attempt to enter the room (or will wait outside until their condition is met), and after entering the room they can then exit the room. The follow are each type's conditions:

- Suppose professors get easily distracted and so they need solitude, with no other students, TA's, or professors in the room, in order to enter the room.
- TA's don't care about students inside and will wait if there is a professor inside, but there can only be up to 7 TA's inside (any more would clearly be imposters from CS161 or CS186).
- Students don't care about other students of TA's in the room, but will wait if there is a professor inside.

To summarize the constraints:

- Professor must wait if anyone else is in the room
- TA must wait if there are already 7 TA's in the room
- TA must wait if there is a professor in the room
- student must wait if there is a professor in the room

```
typedef struct lock { . . . } lock        // lock.acquire(),lock.release()
typedef struct cv { . . . } cv            // cv.wait(&lock),cv.signal(), cv.broadcast()

#define TA_LIMIT 7
typedef struct {
    lock lock;
    cv student_cv;
    int waitingStudents, activeStudents;
    cv ta_cv, prof_cv;
    int waitingTas, waitingProfs;
    int activeTas, activeProfs;
} room_lock;
```

```
/* mode = 0 for student, 1 for TA, 2 for professor */
enter_room(room_lock *rlock, int mode) {
    ------------------------------------------------------------
    ------------------------------------------------------------
    if (mode == 0) {
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
        rlock->activeStudents++;
    } else if (mode == 1) {
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
```

```
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        rlock->activeTas++;
    } else {
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        rlock->activeProfs++;
    }
    ------------------------------------------------------------------
    ----------------------------------------------------------------
}

exit_room(room_lock *rlock, int mode) {
    ------------------------------------------------------------------
    ----------------------------------------------------------------
    if (mode == 0) {
        rlock->activeStudents--;
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
    } else if (mode == 1) {
        rlock->activeTas--;
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------
        ----------------------------------------------------------------

    } else {
        rlock->activeProfs--;
        ----------------------------------------------------------------
```

```
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
        ------------------------------------------------------------
    }
    ------------------------------------------------------------
    ------------------------------------------------------------
}
```