Section 4: Condition Variables

CS 162

September 24, 2021

Contents

1 Vocabulary

- **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 CS162 (Online) Office Hours

Suppose we want to use condition variables to control access to a CS162 (digital) 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 8 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.
- Students and TAs are polite to professors, and will let a waiting professor in first.

To summarize the constraints:

- Professor must wait if anyone else is in the room
- TA must wait if there are already 8 TA's in the room
- TA must wait if there is a professor in the room or waiting outside
- Students must wait if there is a professor in the room or waiting outside

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

#define TA_LIMIT    8
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--;
```

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

3 Coroutines (SP21 MT1)

Coroutines are functions whose execution can be suspended and resumed. You're likely familiar with them by a different name in the form of Python's generators. In this problem, we'll implement a simple version of coroutines in C. Our coroutines are *lazy*, meaning that they do not execute/continue executing until a value has been requested from them (via the next function). You'll implement coroutines according to the following API, defined in `coroutine.h`:

```
typedef struct coroutine coroutine_t;

/* Launches a new coroutine using `function`, passing it `aux` as its second
 * argument. Returns a new coroutine_t* which can be used to request values
 * from the coroutine using the `next` function. */
coroutine_t* launch_coroutine(void (*function)(coroutine_t*, void*), void* aux);

/* Used inside a coroutine to pass `value` to the corresponding caller of
 * `next`. As our coroutines are lazy, `yield` should block until another call
 * to `next` is made before continuing execution. */
void yield(coroutine_t* coroutine, void* value);

/* Obtains the next `yield`ed value from the coroutine, if there is one, and
 * stores it in `dest`. If and only if there is no value to be obtained
 * (because the coroutine has finished executing), the return value will be
 * `false`. */
bool next(coroutine_t* coroutine, void** dest);
```

An example of using this API is shown below.

```
/* Finite coroutine iterating over the characters of a string */
static void iterate(coroutine_t* coroutine, void* _string) {
char* string = (void*)_string;
  while (*string) {
    yield(coroutine, (void*)(*string++));
  }
}

/* Used to demonstrate laziness. */
static void say_hello(coroutine_t* coroutine, void* _) { printf("Hello world!\n"); }

int main() {
  char* alphabet = "ABCDEFG";
  coroutine_t* iterate_coroutine = launch_coroutine(iterate, alphabet);
  void* value;
  while (next(iterate_coroutine, &value)) {
    printf("%c ", (char)value);
  }
  printf("\n");
  coroutine_t* say_hello_coroutine = launch_coroutine(say_hello, NULL);
}
```

The output from the above program should be

```
A B C D E F G
```

Notice that `Hello world` is not part of the output. This is because as mentioned before, our coroutines are lazy, and there was no call to `next` made with `say_hello_coroutine`, therefore the function `say_hello` never began running.

Fill in the blanks below to complete the implementation of coroutines. You can assume that all syscalls with not error, and you can use as many lines as necessary. Your implementation must not be subject to race-conditions, must not busy-wait, and must not leak resources (namely memory). Furthermore, we emphasize that this is a coding question: all solutions must be given in the C programming language. No credit will be awarded for pseudocode, the contents of comments, code which calls "helper functions" which do not exist, etc.

```
struct coroutine {
  void* value;
  bool finished;
  void (*function)(coroutine_t*, void*);
  void* aux;
  sem_t has_yielded;
  sem_t next_requested;
  pthread_t thread;
};

static void* coroutine_stub(void* _coroutine) {
  coroutine_t* coroutine = (coroutine_t*)_coroutine;
  sem_wait(&coroutine->next_requested);
  coroutine->function(coroutine, coroutine->aux);
  coroutine->finished = true;
  sem_post(&coroutine->has_yielded);
  return NULL;
}

coroutine_t* launch_coroutine(void (*function)(coroutine_t*, void*), void* aux) {
  coroutine_t* coroutine = malloc(sizeof(coroutine_t));
  coroutine->value = NULL;
  coroutine->finished = false;
  coroutine->function = function;
  coroutine->aux = aux;
  sem_init(&coroutine->has_yielded, 0, 0);
  sem_init(&coroutine->next_requested, 0, 0);
  pthread_create(&coroutine->thread, NULL, coroutine_stub, coroutine);
  return coroutine;
}
void yield(coroutine_t* coroutine, void* value) {
  ------------------------------------------
  ------------------------------------------
  ------------------------------------------
}

bool next(coroutine_t* coroutine, void** dest) {
  ------------------------------------------
  ------------------------------------------
  ------------------------------------------
  ------------------------------------------
```

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

4 Pintos Pipes

In this question you will be implementing support for a highly-simplified version of pipes within PintOS. Specifically, your pipes implementation will be simplified in the following ways:

- Each process has exactly one pipe.
- A process may only write to a pipe belonging to one of its direct children, using the `write_pipe` syscall.
- A process may only read from its own pipe, using the `read_pipe` syscall.
- Attempting to read from an empty pipe will result in the process being blocked until data is available (this is also true of normal Linux pipes). However, writing to a full pipe should not block - for the sake of keeping this question short, we are assuming a process will never write more data to a pipe than the pipe has capacity for.
- Pipes are not given file descriptors (hence the dedicated syscalls), and as such none of the conditions surrounding what happens if a write/read descriptor for a pipe is still open apply to this problem.

Maintaining pipe state

Fill in the blanks below with whatever additional state you need to implement support for pipes. (Hint: some of this should be quite similar to what you did in project 1 in order to implement the wait syscall)

```
#define PIPE_BUFFER_SIZE 256
/*
 * Assume the pipe struct is properly reference counted
 * and therefore properly freed, in a manner similar to project 1
 * and the example shown with "Shared Data" in Section 2.
 *
 * Also assume that adding a child's pipe to its parent's
 * child_pipes list is taken care of for you.
 */

struct pipe {
  /* Reference count handling fields (not shown) */
  tid_t tid;
  uint8_t buffer[PIPE_BUFFER_SIZE];
  struct list_elem elem; /* List element for child_pipes */
  ----------
  ----------
  ----------
  ----------
};

struct thread {
  tid_t tid;
  struct list child_pipes;
  /* Fields irrelevant to this problem (not shown) */
  ----------
};

static void start_process(/* Omitted */) {
```

```
  struct thread* cur = thread_current();
  /* Initialize interrupt frame, load executable, initialize
  * other state in thread struct (not shown).
  */
  ----------
  ----------
  ----------
  ----------
}
```

Reading and Writing

Assume that below functions are called when the `read_pipe` and `write_pipe` syscalls are made, respectively. The below functions should work even if the `read_pipe` and `write_pipe` syscalls are made multiple times during a thread's lifetime.

```
/*
 * Kills the current thread if pointer is not a valid pointer
 * within the user address space
 */
static void verify(void* pointer) {
  /* Code not shown */
}

/*
 * Returns the pipe belonging to child_tid if child_tid the tid
 * of a child of the current process, and returns NULL otherwise
 */
static struct pipe* helper(tid_t child_tid) {
  /* Code not shown */
}

/*
 * If the pipe is empty, this should block until data is available.
 * Otherwise it should read up to `length` bytes from the current thread's
 * pipe. If there are fewer than `length` bytes available, but more than zero,
 * read them all, but do not wait for more data to become available
 *
 * For example if there are 20 bytes of data currently in the pipe, and
 * length is given as 50, this function should read in the 20 bytes from the pipe
 * and then return (i.e. do not wait around for 50 bytes to become available)
 *
 * Should return the number of bytes read
 */
static int sys_read_pipe(uint8_t* buffer, unsigned length) {
  verify(buffer);
  verify(buffer + length - 1);
  struct thread* cur = thread_current();
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
}
```

```
/*
 * Writes ‘length‘ bytes from ‘buffer‘ to thread ‘child_tid‘’s pipe
 * Assume that there is always enough space in the pipe’s buffer
 * to write ‘length‘ bytes without overflowing it.
 */
static int sys_write_pipe(tid_t child_tid, uint8_t* buffer, unsigned length) {
  verify(buffer);
  verify(buffer + length - 1);
  struct thread* cur = thread_current();
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
  ----------
  return 0;
}
```