CIS 4480: Recitation 06

Joel Ramirez

October 2025

Name: _____ **PennID:** _____

1 Threading and Synchronization Worksheet

1.1 Pthreads

Thread Interleaving and Shared State

The following program creates two threads that each update a shared global variable.

```c
int shared = 0;

void* worker(void* arg) {
    int id = *(int*)arg;
    for (int i = id; i < id + 2; i++) {
        int temp = shared;
        temp++;
        shared = temp;
    }
    return NULL;
}
int main() {
    pthread_t t1, t2;
    int a = 1, b = 2;

    pthread_create(&t1, NULL, worker, &a);
    pthread_create(&t2, NULL, worker, &b);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    printf("Final value of shared = %d\n", shared);
    return 0;
}
```

Questions:

- Instead of incrementing the shared value directly, we first load it into a local variable, increment it, and then write it back. Vedansh claims this protects the shared variable. Why doesn't this actually work?

- What range of possible final values might `shared` take? Justify your reasoning.

Using spthreads...

```c
int main(void) {
    spthread_t t1, t2;
    int a = 1, b = 2;

    spthread_create(&t1, NULL, worker, &a);
    spthread_create(&t2, NULL, worker, &b);

    spthread_join(t1, NULL);
    spthread_join(t2, NULL);

    printf("Final value of shared = %d\n", shared);
    return 0;
}
```

Question: A student tries to write this haphazardly before watching lecture and notices that the program hangs – what gives? Explain in the context of spthreads.

2 Spthreads

```c
// declare scheduler() and globals from sched-demo.c
// see the schedular implementation at the end of the questions.
static int shared = 0;

static void* worker(void* arg) {
    int id = *(int*)arg;
    for (int i = id; i < id + 2; i++) {
        int temp = shared;
        temp++;
        shared = temp;
    }

    counter += 1;

    return NULL;
}
void cancel_and_join(spthread_t thread) {
  spthread_cancel(thread);
  spthread_continue(thread);
  spthread_suspend(thread); // forces the spthread to hit a cancellation point
  fputs("waiting on a thread to exit.\n", stderr);
  spthread_join(thread, NULL);
}

int main(void) {
    spthread_t temp;
    int a = 1, b = 2;

    spthread_create(&temp, NULL, worker, &a);
    threads[0] = temp;
    spthread_create(&temp, NULL, worker, &b);
    threads[1] = temp;

    scheduler();

    for (int i = 0; i < NUM_THREADS; i++) {
        cancel_and_join(threads[i]);
    }
    printf("Final value of shared = %d\n", shared);
    return 0;
}
```

2.1 Using a Scheduler

Question: Even if we use a scheduler, we might still run into issues. What are we missing in the previous implementation?

Follow-up: With what you mentioned missing, where exactly are those mechanisms missing in the code? Identify the critical regions where they should be placed and write what is missing.

```
static void* worker(void* arg) {

    int id = *(int*)arg;
    for (int i = id; i < id + 2; i++) {


        int temp = shared;
        temp++;
        shared = temp;


    }

    counter += 1;

    return NULL;
}
```

Question: If we have a large enough quantum or time slice, do we still have to worry about these issues? Why or why not? Which "algorithm" would support us in not worrying about these issues?

Question: What does it mean to cancel a thread? Check out the man page! This will help your understanding of why we have `cancel_and_join()` for our spthreads. *How does* ***sp_suspend*** *force a thread to be cancelled*?

2.2 Test-and-Set

Implementing a Spinlock

```
int lock = 0;

void acquire() {
    while (test_and_set(&lock))
        ; // spin
}

void release() {
    lock = 0;
}
```

Integration with Mutexes In a real threading library, `spthread_mutex_lock()` might be implemented using `test_and_set` internally. Sketch how `test_and_set` could be used to implement blocking rather than spinning when the lock is unavailable. You can imagine that `test_and_set` is called atomically.

```
    int test_and_set(int *lock){









    }
```

```c
static spthread_t threads[NUM_THREADS];
static const int centisecond = 10000;  // 10 milliseconds

static int counter = 0;

static void alarm_handler(int signum) {}

static void scheduler(void) {
  int curr_thread_num = 0;

  // mask for while scheduler is waiting for
  // alarm to go off
  sigset_t suspend_set;
  sigfillset(&suspend_set);
  sigdelset(&suspend_set, SIGALRM);

  // just to make sure that
  // sigalrm doesn't terminate the process
  struct sigaction act = (struct sigaction){
      .sa_handler = alarm_handler,
      .sa_mask = suspend_set,
      .sa_flags = SA_RESTART,
  };
  sigaction(SIGALRM, &act, NULL);

  // make sure SIGALRM is unblocked
  sigset_t alarm_set;
  sigemptyset(&alarm_set);
  sigaddset(&alarm_set, SIGALRM);
  pthread_sigmask(SIG_UNBLOCK, &alarm_set, NULL);

  struct itimerval it;
  it.it_interval = (struct timeval){.tv_usec = centisecond * 10};
  it.it_value = it.it_interval;
  setitimer(ITIMER_REAL, &it, NULL);

  // locks to check the global value done
  while (counter != 2) {
    curr_thread_num = (curr_thread_num + 1) % NUM_THREADS;
    spthread_t curr_thread = threads[curr_thread_num];

    spthread_continue(curr_thread);
    sigsuspend(&suspend_set);
    spthread_suspend(curr_thread);
  }
}
```