Week 10 – Friday

COMP 3400

Last time

- What did we talk about last time?
- Exam 2 post mortem
- Language approaches to threading
- Practice with threads
    - Prime counting

Questions?

Assignment 6

Thread Practice

Concurrent prime number search

- Let's write a threaded program that counts the number of primes less than 100,000,000
- We'll spawn a number of threads and divide up the range of values from 0 to 100,000,000 evenly
- To send data to each thread and get the result, we'll use dynamically allocated versions of the following struct:

```
struct range {
  unsigned long start;
  unsigned long end;
  unsigned long count;
};
```

Algorithm

- Divide the total number by the number of threads to determine how many numbers to give each thread
- Loop through all threads:
  - Allocate a **`range`** struct to hold the lower and upper value for each thread
  - Create each thread
- Loop through all threads:
  - Join them
- Inside each thread:
  - Loop from the lower to the upper value and increment a counter if the value is prime
  - Store the count into the **`range`** struct
  - Call **`pthread_exit()`** when done

POSIX thread functions

- As a reminder, here are the POSIX functions we need

```
int pthread_create (pthread_t *thread, const pthread_attr_t *attr,
      void *(*start_routine)(void*), void *arg);
```

  - Create a new thread (not as bad as it looks)

```
void pthread_exit (void *value_ptr);
```

  - Exit from the current thread (giving a pointer to the result, if any)

```
void pthread_join (pthread_t thread, void *value_ptr);
```

  - Join a thread (getting a pointer to its result, if any)

Synchronization

Synchronization

- Now you have all the tools needed to create, run, and join threads
- But you don't have any tools to avoid the problem of race conditions
- **Synchronization** is used to coordinate between threads, often by enforcing critical sections, sections of code that only one thread can be executing at a time
- Common synchronization tools:
  - Locks (mutexes)
  - Semaphores
  - Barriers
  - Condition variables
- If used incorrectly, however, synchronization tools can lead to other problems such as deadlock and livelock

Examples of synchronization

- The following are common examples of synchronization:
  - Multiple threads share a data structure, but only one can write to it at a time
  - Only so many threads can access a shared resource to avoid slowdowns
  - Certain events need to happen in a certain order
  - Some calculations must be done before an action can be taken
- Performing synchronization so that the result is correct while avoiding performance penalties is challenging

Critical sections

- Recall that a critical section is a section of code that it's safe for only a single thread to be executing
- Often this is because non-atomic memory accesses (such as reading a value, doing calculations, and then writing back to memory) can get inconsistent results if more than one thread is executing them concurrently
- A common use of synchronization tools is to block threads trying to access a critical section if a thread is already executing it

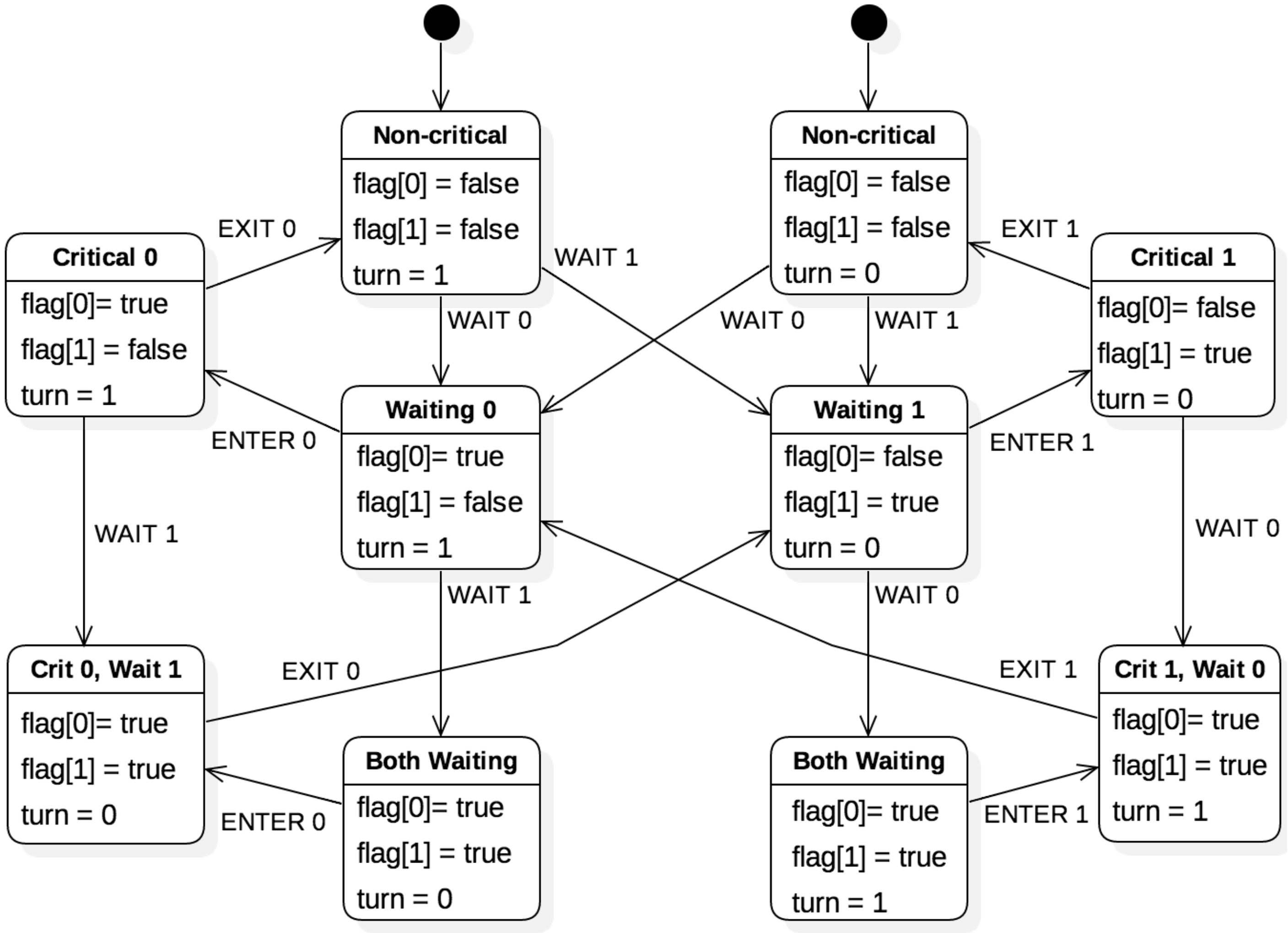
Peterson's solution

- Peterson's solution demonstrates a way to enforce a critical section for two threads
- Here's the idea, where the **`flag`** array and **`turn`** are shared variables

```
flag[self] = true;
turn = other; // Politely assume it's the other person's turn

// Execute loop until it's safe to enter
while (flag[other] == true && turn == other) ;

// Here's where the code for the critical section goes

flag[self] = false; // Mark yourself as finished
```

Why Peterson's solution works

- This state diagram shows all the possible states the system can be in
- There's no state where both 0 and 1 are in the critical section
- The only changes to memory that matter are atomic writes



Non-critical
flag[0] = false
flag[1] = false
turn = 1

Non-critical
flag[0] = false
flag[1] = false
turn = 0

EXIT 0
WAIT 1
WAIT 0
WAIT 0
WAIT 1
EXIT 1

Critical 0
flag[0]= true
flag[1] = false
turn = 1

Waiting 0
flag[0]= true
flag[1] = false
turn = 1

Waiting 1
flag[0]= false
flag[1] = true
turn = 0

Critical 1
flag[0]= false
flag[1] = true
turn = 0

ENTER 0
ENTER 1
WAIT 1
WAIT 1
WAIT 0
WAIT 0
EXIT 0
EXIT 1

Crit 0, Wait 1
flag[0]= true
flag[1] = true
turn = 0

Both Waiting
flag[0]= true
flag[1] = true
turn = 0

Both Waiting
flag[0]= true
flag[1] = true
turn = 1

Crit 1, Wait 0
flag[0]= true
flag[1] = true
turn = 1

ENTER 0
ENTER 1

Synchronization properties

- We often want three synchronization properties:
    - **Safety:** There's never more than one thread in the critical section
        - Also called **mutual exclusion**
    - **Liveness:** If no thread is in the critical section and one or more threads try to enter, one thread will be able to
        - Also called **progress**
    - **Fairness:** Assuming that no thread will stay in the critical section forever, a thread trying to get into the critical section will eventually get in
        - Also called **bounded waiting**
- Peterson's solution provides all three

Why Peterson's solution doesn't work in general

- It's only described for two threads and gets messy for more
- It requires thinking about which variables to set rather than providing more general tools (like locks)
- It requires busy waiting (repeatedly executing a loop)
- It's not guaranteed to work on modern hardware that sometimes switches the order of instructions for better pipelining
    - These changes are guaranteed to work in a single-threaded context but can't take into account what other threads are doing

Locks

Locks

- A key synchronization tool is called a **lock** (or a **mutex**, short for *mutual exclusion*)
- Critical sections can be protected by a lock
  - First code acquires the lock
  - Then it performs the code in the critical section
  - Then it releases the lock
- For POSIX threads, lock functionality is provided by several mutex functions that operate on **`pthread_mutex_t`** objects

Lock features

- Mutual exclusion
  - Locks start unlocked
  - Only one thread can acquire a lock at a time
  - No other thread can acquire a lock until it's been released
- Non-preemption
  - A lock must be voluntarily released by the thread that acquired it
- Atomic operations
  - Acquire and release are atomic operations
- Blocking acquires
  - If a thread tries to acquire a lock, it's blocked and added to the queue
  - When the thread holding the lock releases it, only one thread acquires it

POSIX mutex functions

```
int pthread_mutex_init (pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

- Create a mutex with the specified attributes

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

- Destroy an existing mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

- Acquire a mutex, blocking until you succeed

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

- Try to acquire a mutex, returning non-zero if another thread has the mutex

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

- Release the mutex

Mutex example

- Here's a thread that uses a mutex when incrementing a global variable

```
int global = 5;

// Each increment thread gets a pointer to the mutex
void *
increment (void *args)
{
  pthread_mutex_t *mutex = (pthread_mutex_t *) args;

  // Lock for the critical section, then release
  pthread_mutex_lock (mutex);
  global++;
  pthread_mutex_unlock (mutex);

  pthread_exit (NULL);
}
```

Main program

- The following program creates the mutex and passes it to two threads
- Note that the mutex lives on the stack, but that's okay since this function won't return until after the other threads are done

```
pthread_t threads[2];
pthread_mutex_t mutex;

// Initialize the mutex
pthread_mutex_init (&mutex, NULL);

// Create the child threads, passing pointers to the mutex
assert (pthread_create (&threads[0], NULL, increment, &mutex) == 0);
assert (pthread_create (&threads[1], NULL, increment, &mutex) == 0);

// Join the threads
pthread_join (threads[0], NULL);
pthread_join (threads[1], NULL);

// Confirm the result
assert (global == 7);
```

Length of critical sections

- The first example on the previous slide will take much longer, since it has to lock and unlock 1,000,000 times
- On the other hand, the second example will block all other threads from running code that depends on the lock until it's finished
- Neither is very realistic, since incrementing a variable 1,000,000 times in a loop is ridiculous
- There's no simple solution: depends on the problem
- Always getting the right answer is the first goal and then tuning for better performance comes second

Ticket Out the Door

Upcoming

Next time…

- Finish locks
- Semaphores

Reminders

- **Finish Assignment 6**
  - **Due tonight by midnight!**
- Work on Project 3
- Read section 7.4