Threads Cont. & Deadlock

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Maya Huizar Sid Sannapareddy

Administrivia

- PennOS:
  - PennOS Due Dates milestones updated
  - Groups have been assigned
  - TA’s have been assigned to groups
  - You have the first milestone, which needs to be done before end of day Tuesday the 8^{th}.
  - Your group (or at least most of your group) needs to meet with your assigned TA and display the expectations laid out in the PennOS Specification

- Videos containing some demos of a functioning PennOS posted on the schedule.
- Recitation Tomorrow will help with PennOS and Milestone 0
- No OH for Travis on Friday: it is July 4^{th}

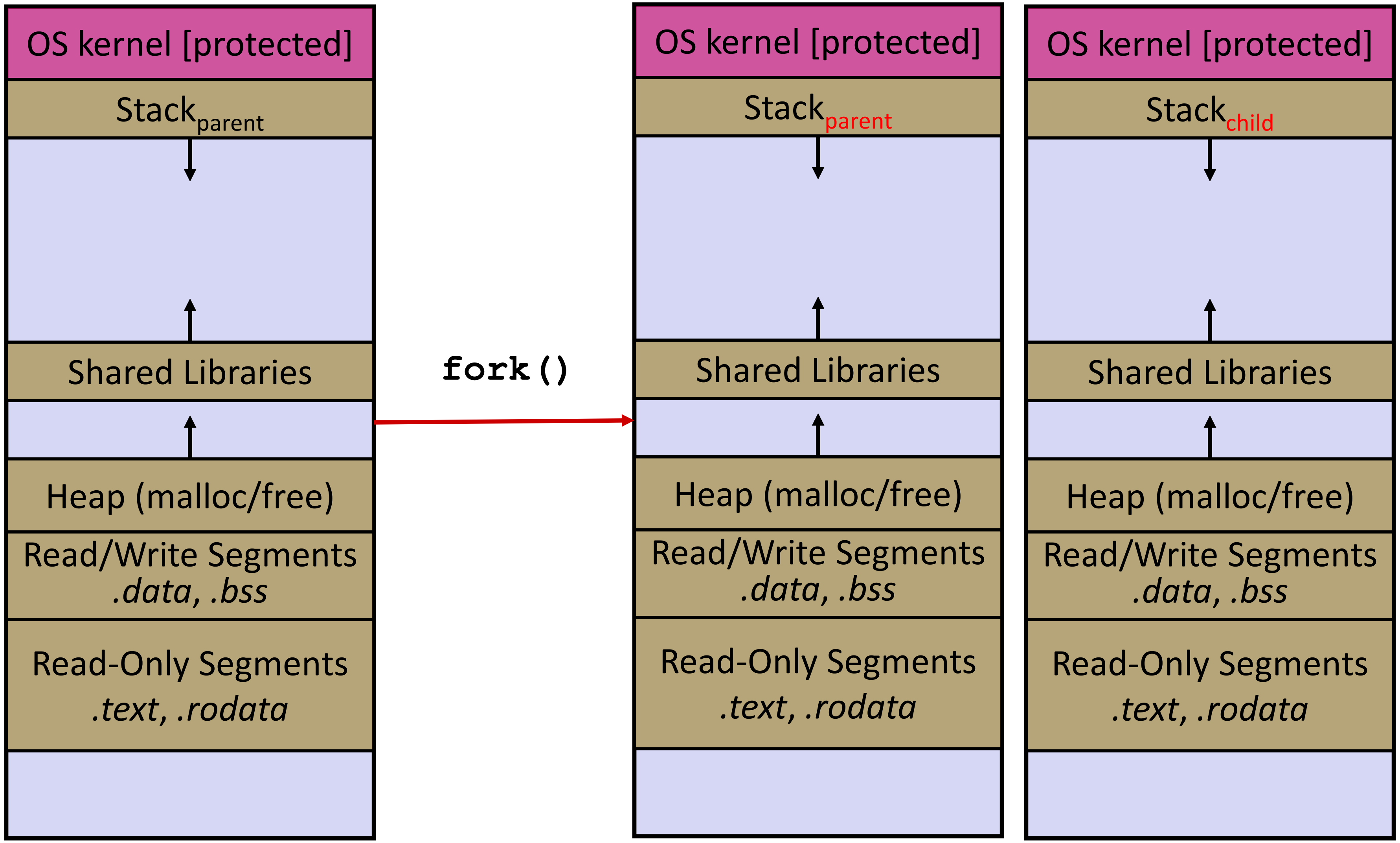
Lecture Outline

- **Threads & Lock refresher**
- Spthreads
- tsl
- Disable interrupts
- Deadlock & Preventing Deadlock

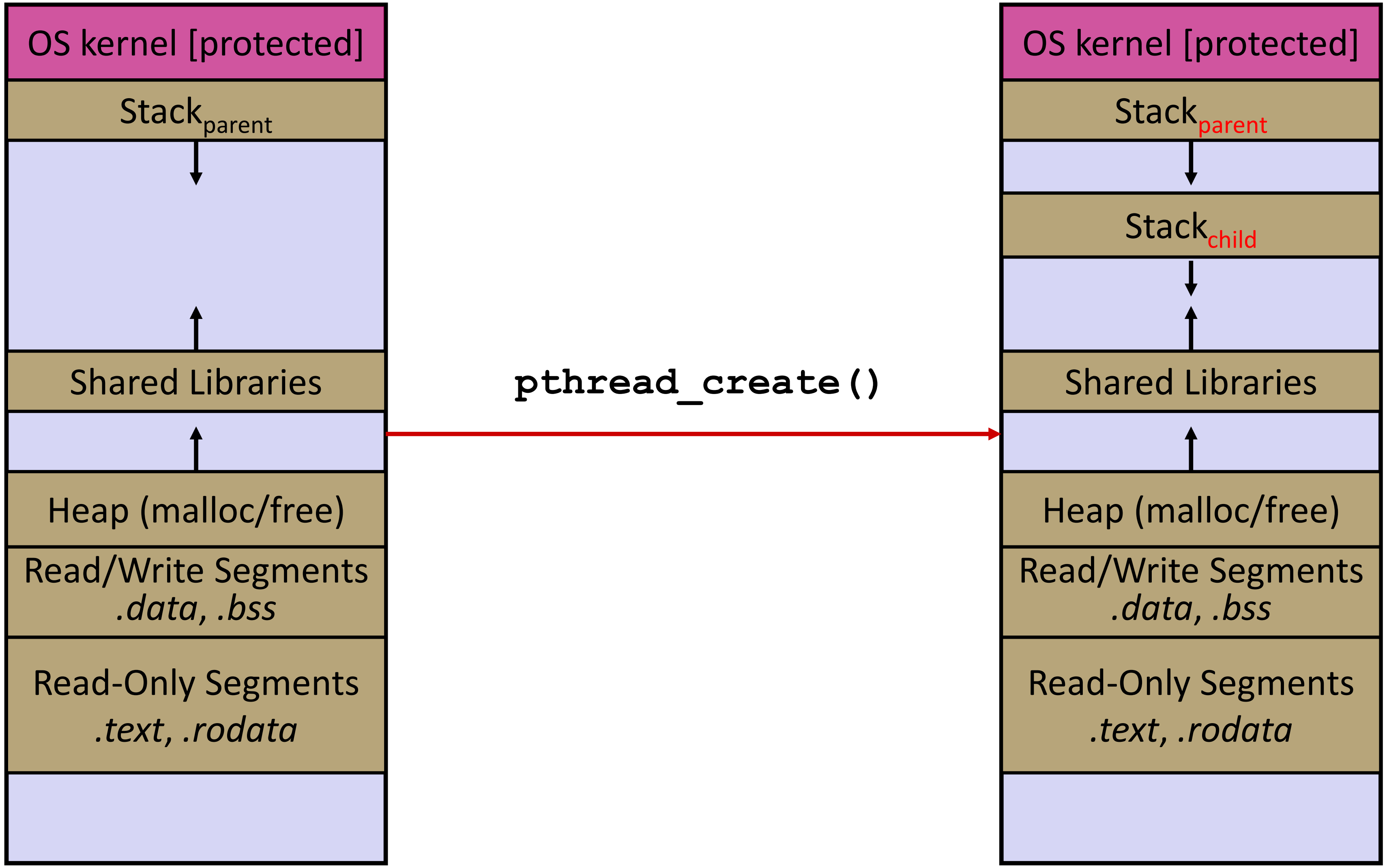
Threads vs. Processes

- In most modern OS's:
  - A Process has a unique: address space, OS resources, & security attributes
  - A Thread has a unique: stack, stack pointer, program counter, & registers
  - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

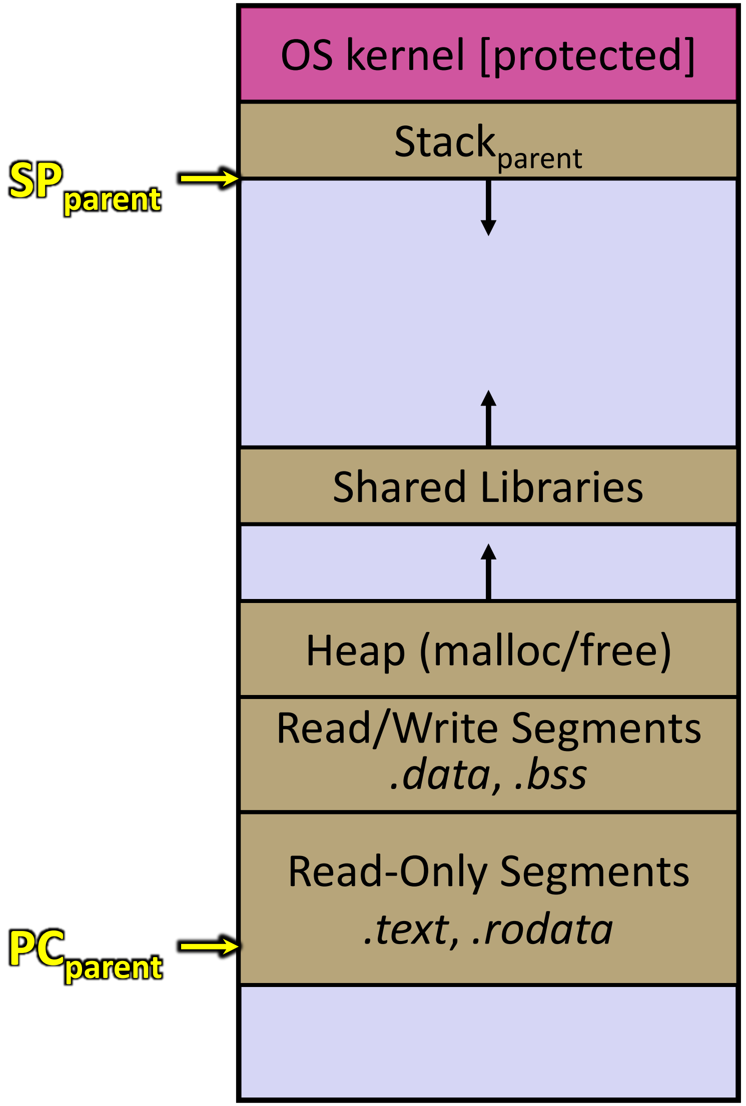
Threads vs. Processes



Threads vs. Processes

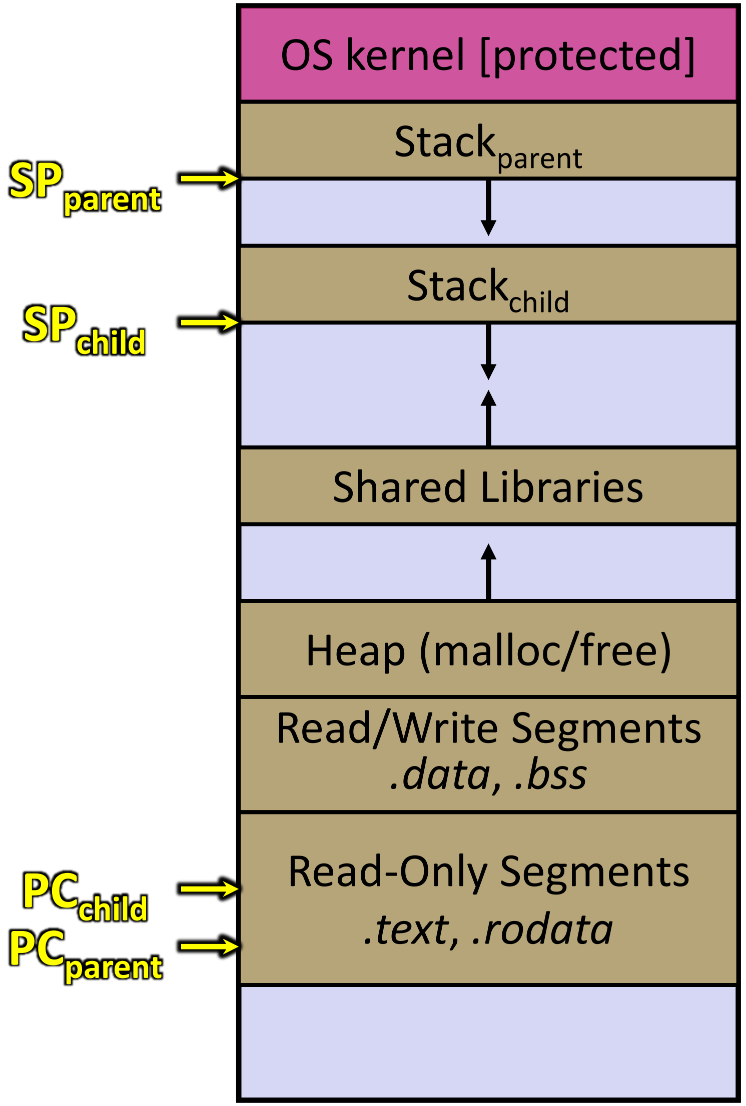


Single-Threaded Address Spaces



- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`

Multi-threaded Address Spaces



- After creating a thread
  - *Two* threads of execution running in the address space
    - Original thread (parent) and new thread (child)
    - New stack created for child thread
    - Child thread has its own *values* of the PC and SP
  - Both threads share the other segments (code, heap, globals)
    - They can cooperatively modify shared data

Poll Everywhere

pollev.com/tqm

- What are the possible outputs of this code?

```c
int global_counter = 5;

void* t_fn(void* arg) {
  int num = * (int*) arg;

  global_counter += num;

  printf("%d\n", global_counter);

  free(num);
  return NULL;
}
```

```c
int main() {
  pthread_t thds[2];

  for (int i = 0; i < 2; i++) {
    pthread_t temp;
    int* arg = malloc(sizeof(int));
    *arg = i;
    pthread_create(&temp, NULL, t_fn, arg);
    thds[i] = temp;
  }

  for (int i = 0; i < 2; i++) {
    pthread_join(thds[i], NULL);
  }

  return EXIT_SUCCESS;
}
```

Lock Synchronization

- Use a "Lock" to grant access to a *critical section* so that only one thread can operate there at a time
  - Executed in an uninterruptible (*i.e.* atomic) manner

- Pseudocode:

```
// non-critical code

lock.acquire();
// critical section
lock.release();

// non-critical code
```

block if locked

- Lock Acquire
  - Wait until the lock is free, then take it

- Lock Release
  - Release the lock
  - If other threads are waiting, wake exactly one up to pass lock to

Lock API

- Locks are constructs that are provided by the operating system to help ensure synchronization
  - Often called a mutex or a semaphore

- Only one thread can acquire a lock at a time,
  No thread can acquire that lock until it has been released

- Has memory barriers built into it and usually uses TSL to ensure that acquiring the lock is atomic (more on TSL and memory barriers in a little bit)

Milk Example – What is the Critical Section?

- What if we use a lock on the refrigerator?
  - Probably overkill – what if roommate wanted to get eggs?
- For performance reasons, only put what is necessary in the critical section
  - Only lock the milk
  - But lock *all* steps that must run uninterrupted (*i.e.* must run as an atomic unit)

```
fridge.lock()
if (!milk) {
  buy milk
}
fridge.unlock()
```

↓

```
milk_lock.lock()
if (!milk) {
  buy milk
}
milk_lock.unlock()
```

pthreads and Locks

- Another term for a lock is a **mutex** ("mutual exclusion")
  - `pthread.h` defines datatype `pthread_mutex_t`

-
```
int pthread_mutex_init(pthread_mutex_t* mutex,
                       const pthread_mutexattr_t* attr);
```
  - Initializes a mutex with specified attributes

-
```
int pthread_mutex_lock(pthread_mutex_t* mutex);
```
  - Acquire the lock – blocks if already locked Un-blocks when lock is acquired

-
```
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```
  - Releases the lock

-
```
int pthread_mutex_destroy(pthread_mutex_t* mutex);
```
  - "Uninitializes" a mutex – clean up when done

pthread Mutex Examples

- See `total.c`
  - Data race between threads
- See `total_locking.c`
  - Adding a mutex fixes our data race
- How does `total_locking` compare to sequential code and to `total`?
  - Likely *slower* than both– only 1 thread can increment at a time, and must deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
    - See `total_locking_better.c`

pthread Mutex Examples

- See `total.c`
  - Data race between threads
- See `total_locking.c`
  - Adding a mutex fixes our data race
- How does `total_locking` compare to sequential code and to `total`?
  - Likely *slower* than both– only 1 thread can increment at a time, and must deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
    - See `total_locking_better.c`

Poll Everywhere

pollev.com/tqm

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

- Thread-1 executes line 8 while Thread-2 executes line 21.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 15 while Thread-2 executes line 15.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

```
1   // global variables
2   pthread_mutex_t lock;
3   int g =    0;
4   int k = 0;
5
6   void fun1() {
7     pthread_mutex_lock(&lock);
8     g += 3;
9     pthread_mutex_unlock(&lock);
10    k++;
11  }
12
13  void fun2(int a, int b) {
14    g += a;
15    a += b;
16    k = a;
17  }
18
19  void fun3() {
20    pthread_mutex_lock(&lock);
21    g = k + 2;
22    pthread_mutex_unlock(&lock);
23  }
```

Poll Everywhere

pollev.com/tqm

- The code below has three functions that could be executed in separate threads. Note that these are not thread entry points, just functions used by threads:
  - Assume that "lock" has been initialized

- Thread-1 executes line 8 while Thread-2 executes line 14
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

- Thread-1 executes line 14 while Thread-2 executes line 16.
  Choose one:
  - Could lead to a race condition.
  - There is no possible race condition.
  - The situation cannot occur.

```
1  // global variables
2  pthread_mutex_t lock;
3  int g =    0;
4  int k = 0;
5
6  void fun1() {
7    pthread_mutex_lock(&lock);
8    g += 3;
9    pthread_mutex_unlock(&lock);
10   k++;
11 }
12
13 void fun2(int a, int b) {
14   g += a;
15   a += b;
16   k = a;
17 }
18
19 void fun3() {
20   pthread_mutex_lock(&lock);
21   g = k + 2;
22   pthread_mutex_unlock(&lock);
23 }
```

Lecture Outline

- Threads & Lock refresher
- **Spthreads**
- tsl
- Disable interrupts
- Deadlock & Preventing Deadlock

Key Differences of spthread vs pthread

- spthread is something Travis wrote about a year ago.
  - It does not exist anywhere else
  - You likely won’t find any documentation on it outside of this course

- Main difference:
  - When you create a thread, it starts “suspended”
  - Threads can be explicitly continued and suspended
  - When there is a corresponding spthread function, call that instead of the pthread function

Poll Everywhere

pollev.com/tqm

- There are issues here. What are they?

```
vector(int) vec;

void* s_fn(void* arg) {
  while(true) {
    int num = rand();
    // generate a random number
    vector_push(&vec, num);
  }
  return NULL;
}
```

```
int main() {
  vec = vector_new(int, 10, NULL);
  // initialize a length 10 vector of ints

  spthread_t thds[2];
  spthread_create(&(thds[0]), NULL, s_fn, NULL);
  spthread_create(&(thds[1]), NULL, s_fn, NULL);

  int curr_thread = 0;
  while(vector_len(&vec) < 200) {
    spthread_continue(thds[curr_thread]);
    sleep(1);  // sleep for 1 seconds
    spthread_suspend(thds[curr_thread]);

    curr_thread = 1 - curr_thread;
  }
  printf("%d\n", vector_len(&vec));
}
```

Poll Everywhere

pollev.com/tqm

- Adding a lock causes another issue, what issue is it?

```
vector(int) vec;
pthread_mutex_t lock;

void* s_fn(void* arg) {
  while(true) {
    int num = rand();
    pthread_mutex_lock(&lock);
    vector_push(&vec, num);
    pthread_mutex_unlock(&lock);
  }
  return NULL;
}
```

```
int main() {
  pthread_mutex_init(&lock, NULL);
  int curr_thread = 0;
  while(true) {
    pthread_mutex_lock(&lock);
    if (vector_len(&vec) < 200) {
      pthread_mutex_unlock(&lock);
      break;
    }
    pthread_mutex_unlock(&lock);
    spthread_continue(thds[curr_thread]);
    sleep(1);  // sleep for 1 seconds
    spthread_suspend(thds[curr_thread]);

    curr_thread = 1 - curr_thread;
  }
  printf("%d\n", vector_len(&vec));
}
```

Shared Data & spthread

- The calls to `spthread_suspend` and `spthread_continue` will not return until that thread actually continues/suspends

- This can cause an issue when we use locks to maintain shared memory

- What do we do instead?
  - spthread_disable_interrupts_self
  - spthread_enable_interrupts_self

Lecture Outline

- Threads & Lock refresher
- Spthreads
- **tsl**
- Disable interrupts
- Deadlock & Preventing Deadlock

TSL

- TSL stands for **T**est and **S**et **L**ock, sometimes just called **test-and-set**.

- TSL is an atomic instruction that is guaranteed to be atomic at the hardware level

- `TSL R, M`
  - Pass in a register and a memory location
  - R gets the value of M
  - M is set to 1 AFTER setting R

TSL to implement Mutex

- A mutex is pretty much this:

```
pthread_mutex_lock(lock) {
   prev_value = TSL(lock);

   // if prev_value = 1, then it was already locked
   while (prev_value == 1) {
      block();
      prev_value = TSL(lock);
   }
}

pthread_mutex_unlock(lock) {
  lock = 0;
  wakeup_blocked_threads(lock);
}
```

Lecture Outline

- Threads & Lock refresher
- Spthreads
- tsl
- **Disable interrupts**
- Deadlock & Preventing Deadlock

Disabling Interrupts

- If data races occur when one thread is interrupted while it is accessing some shared code….

  What is we don’t switch to other threads while executing that code?

- This can be done by disabling interrupts: no interrupts means that the clock interrupt won’t go off and interrupt the currently running thread

Disabling Interrupts

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

`sum_total = 1`

Thread 0

```
disable_interrupts();
++sum_total;
enable_interrupts();
```

Thread 1

```
disable_interrupts();
++sum_total;
enable_interrupts();
```

Disabling Interrupts

- Advantages:
  - This is one way to fix this issue

- Disadvantages
  - This is usually overkill
  - This can stop threads that aren't trying to access the shared resources in the critical section. May stop threads that are executing other processes entirely
  - If interrupts disabled for a long time, then other threads will starve
  - In a multi-core environment, this gets complicated

Lecture Outline

- Threads & Lock refresher
- Spthreads
- tsl
- Disable interrupts
- **Deadlock & Preventing Deadlock**

Liveness

- Liveness: A set of properties that ensure that threads execute in a timely manner, despite any contention on shared resources.

- When `pthread_mutex_lock();` is called, the calling thread blocks (stops executing) until it can acquire the lock.
  - What happens if the thread can never acquire the lock?

Liveness Failure: Releasing locks

- If locks are not released by a thread, then other threads cannot acquire that lock

- See `release_locks.c`
  - Example where locks are not released once critical section is completed.

Liveness Failure: Deadlocks

- Consider the case where there are two threads and two locks
  - Thread 1 acquires lock1
  - Thread 2 acquires lock2
  - Thread 1 attempts to acquire lock2 and blocks
  - Thread 2 attempts to acquire lock1 and blocks

Neither thread can make progress ☹

- See `milk_deadlock.c`

- Note: there are many algorithms for detecting/preventing deadlocks

Liveness Failure: Mutex Recursion

- What happens if a thread tries to re-acquire a lock that it has already acquired?

- See `recursive_deadlock.c`

- By default, a mutex is not re-entrant.
  - The thread won't recognize it already has the lock, and block until the lock is released

Aside: Recursive Locks

- Mutex's can be configured so that you it can be re-locked if the thread already has locked it. These locks are called *recursive locks* (sometimes called *re-entrant locks*).

- Acquiring a lock that is already held will succeed
- To release a lock, it must be released the same number of times it was acquired

- Has its uses, but generally discouraged.

Deadlock Definition

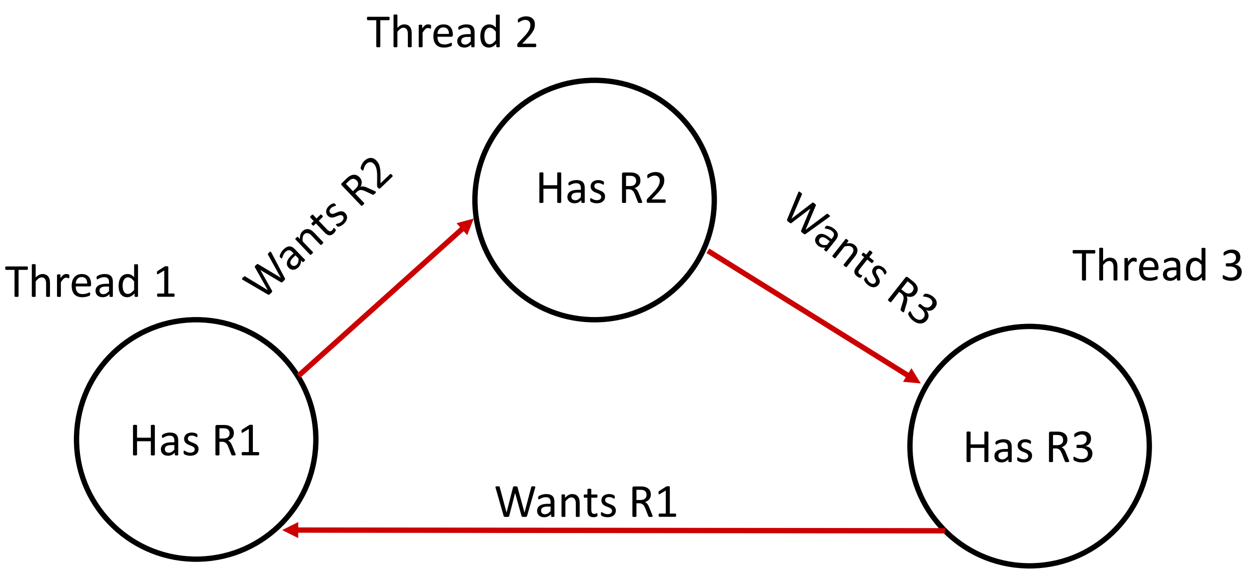
- A computer has multiple threads, finite resources, and the threads want to acquire those resources
  - Some of these resources require exclusive access
- A thread can acquire resources:
  - All at once
  - Accumulate them over time
  - If it fails to acquire a resource, it will (by default) wait until it is available before doing anything
- Deadlock: Cyclical dependency on resource acquisition so that none of them can proceed
  - Even if all unblocked threads release, deadlock will continue

Preconditions for Deadlock

- Deadlock can only happen if these occur simultaneously:
  - **Mutual Exclusion**: at least one resource must be held exclusively by one thread
  - **Hold and Wait**: a thread must be holding a resource, requesting a resource that is held by a thread, and then waiting for it.
  - **No preemption**: A resource is held by a thread until it explicitly releases it. It cannot be preempted by the OS or something else to force it to release the resource
  - **Circular Wait**:
    Can be a chain of more than 2 threads
    Each thread must be waiting for a resource that is held by another thread. That other thread must waiting on a resource that forms a chain of dependency

Circular Wait Example

- A cycle can exist of more than just two threads:



Discuss

- Can a thread deadlock if there is only one thread?

Deadlock Prevention

- If we can remove the conditions for deadlock, we could avoid prevent deadlock from every happening

Discuss

- We are running some code that uses threads, locks, and sometimes deadlocks. Which of these are most likely to be removed so that we can stop deadlocks.

- Deadlock can only happen if these occur simultaneously:
  - **Mutual Exclusion**: at least one resource must be held exclusively by one thread
  - **Hold and Wait**: a thread must be holding a resource, requesting a resource that is held by a thread, and then waiting for it.
  - **No preemption**: A resource is held by a thread until it explicitly releases it. It cannot be preempted by the OS or something else to force it to release the resource
  - **Circular Wait**:
    Can be a chain of more than 2 threads
    Each thread must be waiting for a resource that is held by another thread. That other thread must waiting on a resource that forms a chain of dependency

Deadlock Prevention Summary

- Prevent deadlocks by removing any one of the four deadlock preconditions
- But eliminating even one of the preconditions is often hard/impossible
  - Mutual Exclusion is necessary in a lot of situations
  - Forcing a lower priority process to release resources early requires rollback of execution
  - Not always possible to know all resources that an operating system or process will use upfront

That’s all!

- See you next time!