Deadlock & SP Threads

Computer Operating Systems, Fall 2025

Instructors: Joel Ramirez

Head TAs: Maya Huizar, Akash Kaukuntla, Vedansh Goenka, Joy Liu

TAs:

Eric Zou, Joseph Dattilo, Aniket Ghorpade, Shriya Sane

Zihao Zhou, Eric Lee, Shruti Agarwal, Yemisi Jones

Connor Cummings, Shreya Mukunthan, Alexander Mehta, Raymond Feng

Bo Sun, Steven Chang, Rania Souissi, Rashi Agrawal

Sana Manesh

Administrivia

- PennOS:
  - 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 8th.
  - Your group (or at least the majority 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.
- No Recitation this week (y'all seem to be going through it.)
- OH Today is Virtual.

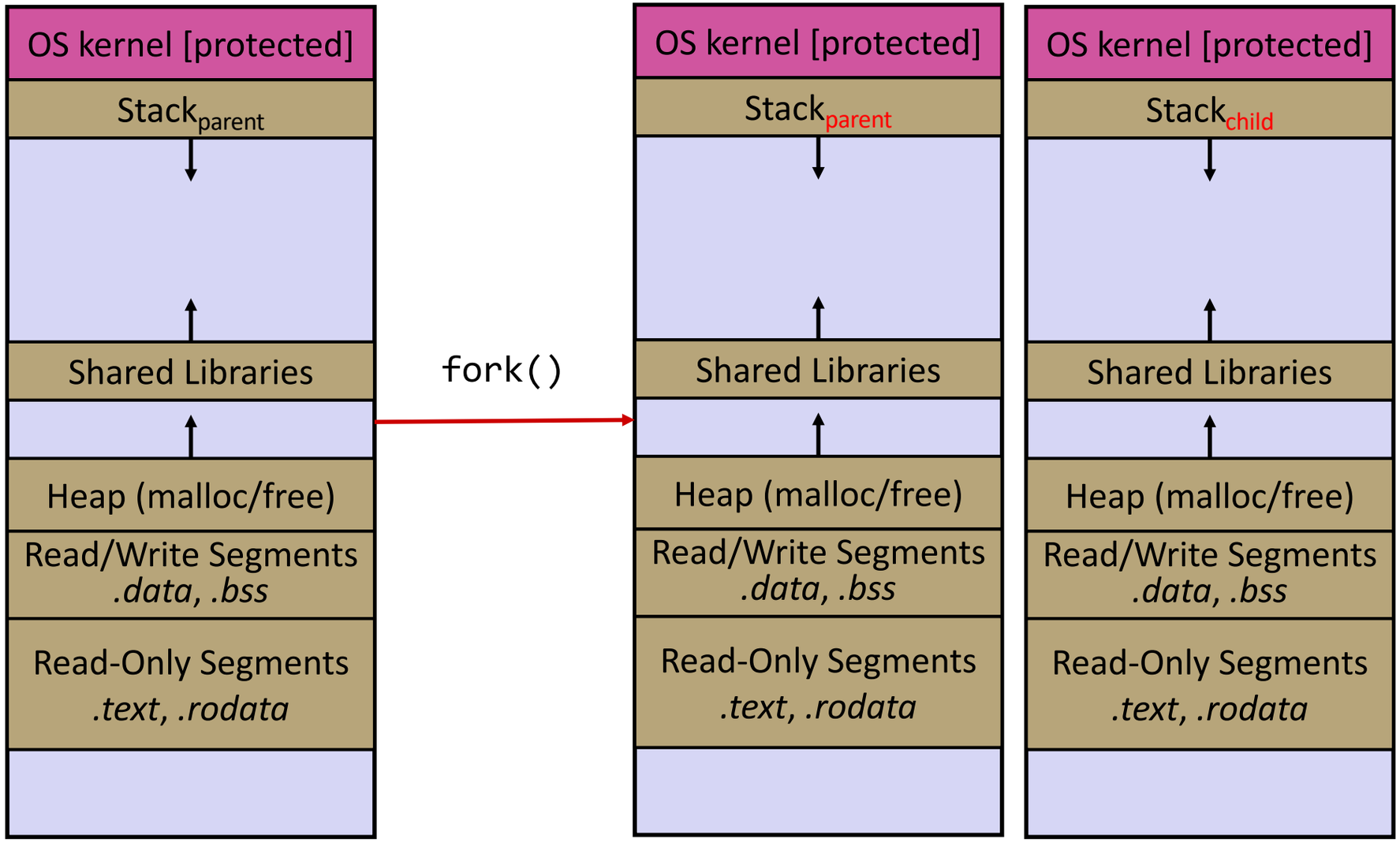
Lecture Outline

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

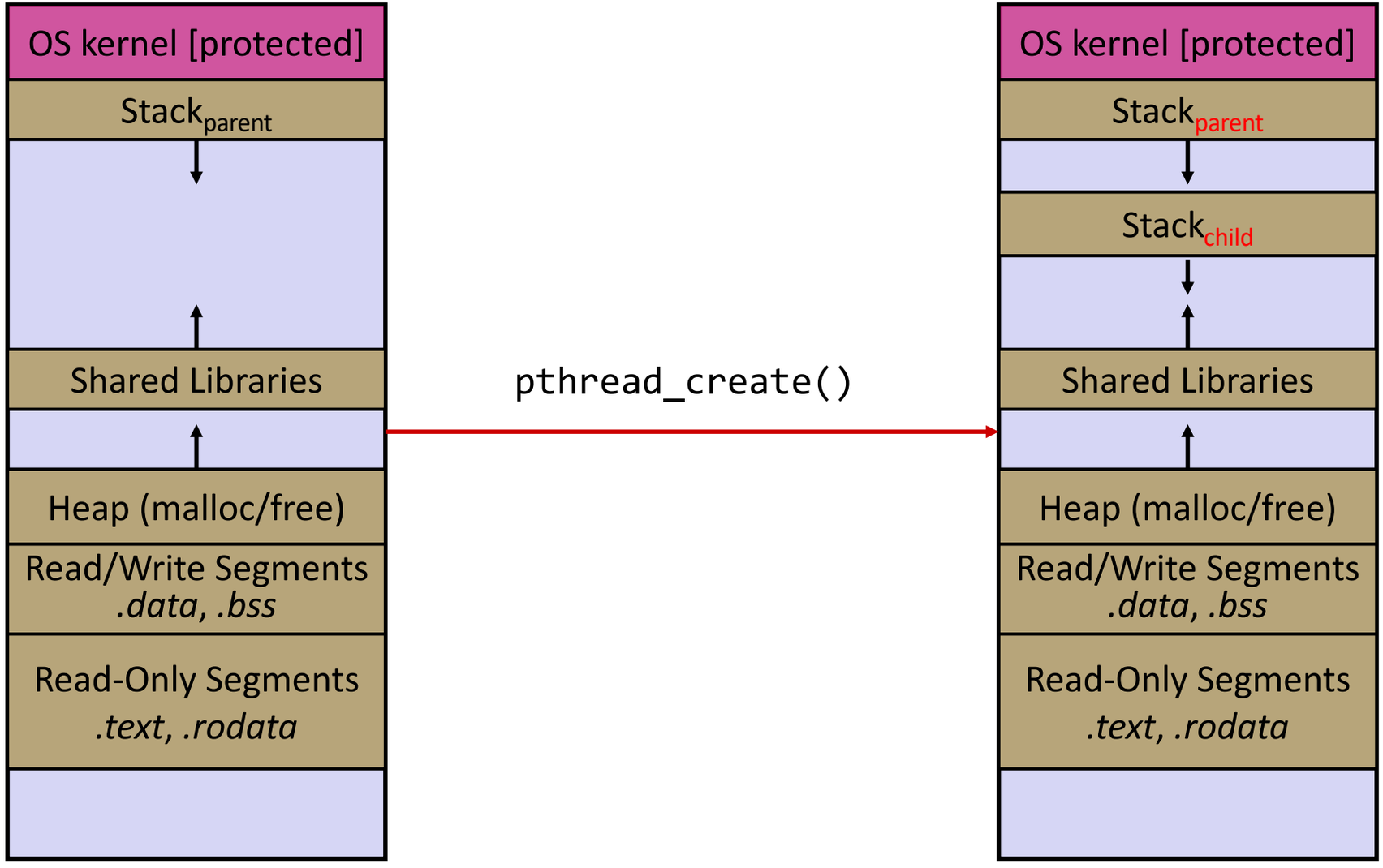
Threads vs. Processes

- 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

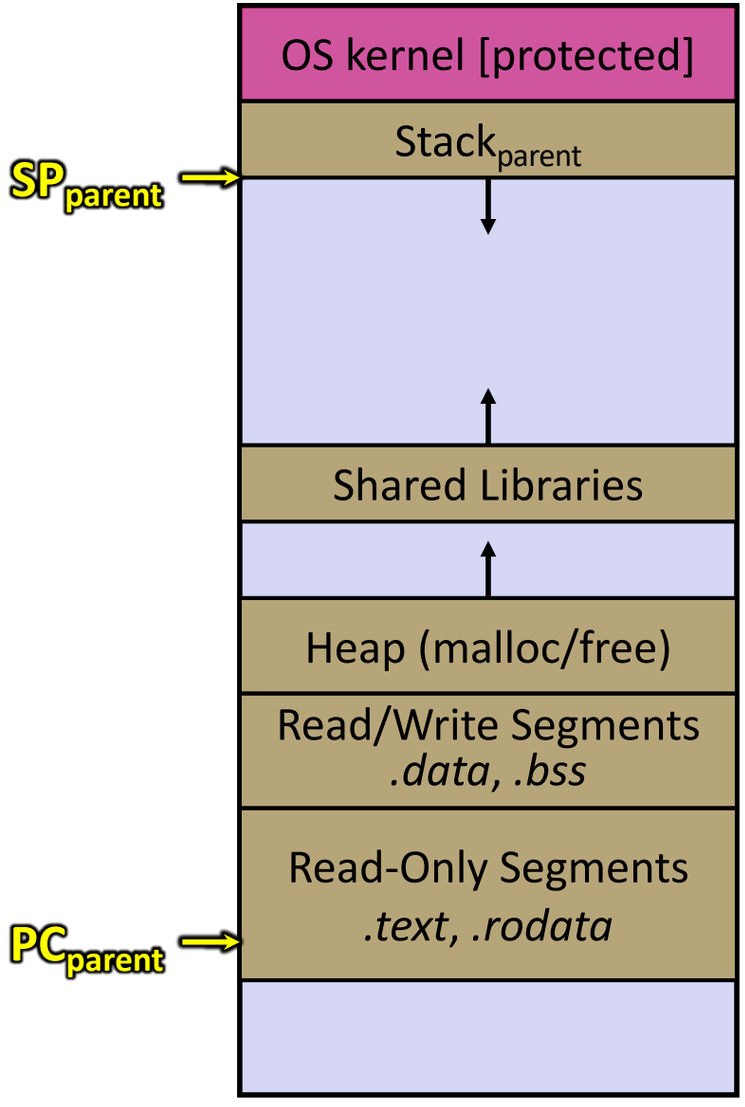
Threads vs. Processes



Threads vs. Processes

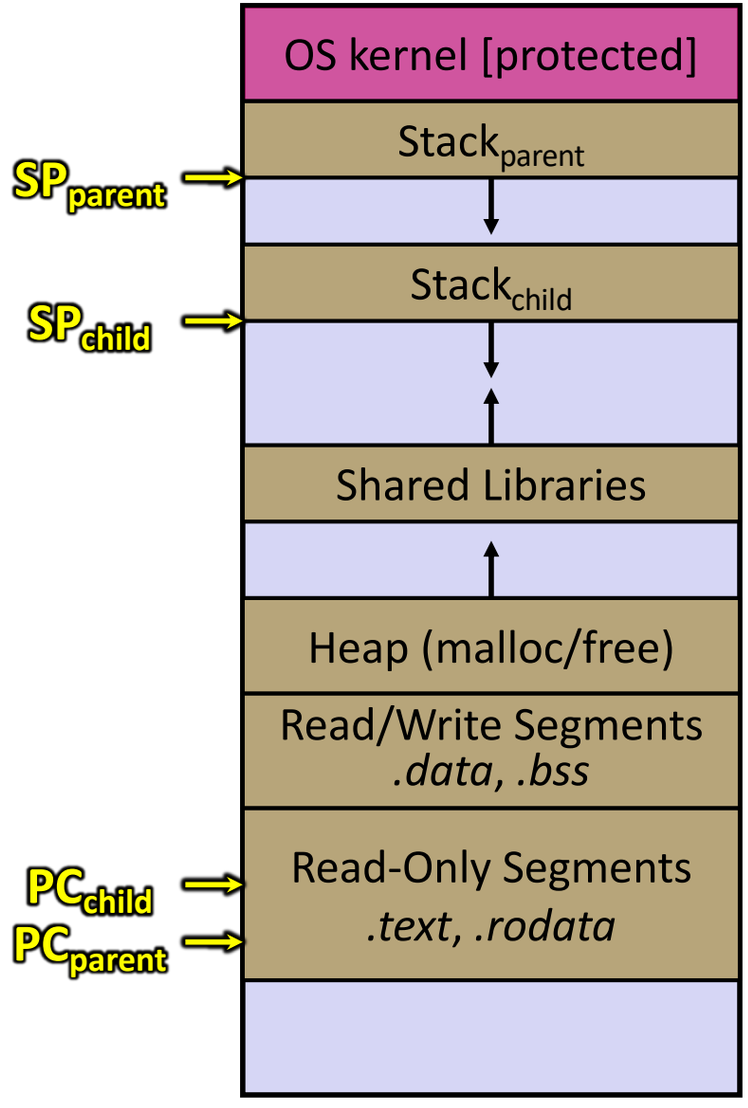


Single-Threaded Address Spaces



- 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/5480

- What are all the values that could be printed by this program?

```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 + 1;
    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;
}
```

Poll Everywhere

pollev.com/5480

- What are all the values that could be printed by this program?

```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;
}
```

This function could print 6, 7 or 8 depending on how these threads are pre-empted.

1. 6 is if 5 + 1 happens first
2. 7 is if 5 + 2 happens first
3. 8 is if (2) and (1) happen in either order.

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

- 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

- Pseudocode:

```
// non-critical code

lock.acquire();   <-- block if locked
// critical section
lock.release();

// non-critical code
```

Lock API

- Locks are constructs that are provided by the operating system to help ensure synchronization
  - There are many types of locks (e.g. Mutex Lock, Spin Lock…)

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

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
  - Lock *all* steps that must run uninterrupted; only lock the milk.
  - (*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()
```

pthread mutex locks

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

- initializes the mutex object pointed to by mutex according to the mutex attributes specified in mutexattr.
  - You could even make locks shareable across processes...

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

- If the mutex is currently unlocked, it becomes locked and owned by the calling thread.
- If the mutex is already locked by another thread, **pthread_mutex_lock**() ***suspends*** the calling thread until the mutex is unlocked.

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

- unlocks the given mutex. The mutex is ***assumed*** to be locked and owned by the calling thread on entrance to **pthread_mutex_unlock**(). Linux allows any thread to unlock a mutex, even if it isn't its owner. But, this isn't true across OS's.

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

Check the man page..

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`
- How about with optimizations?
  - Let's see `total_locking_opt.c` with compiler optimizations.

Poll Everywhere

pollev.com/5480

- 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

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

- B) Thread-1 executes line 15 while Thread-2 executes line 15.
  Choose one:
  1. Could lead to a race condition.
  2. There is no possible race condition.
  3. 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/5480

- 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

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

- B) Thread-1 executes line 14 while Thread-2 executes line 16.
  Choose one:
  1. Could lead to a race condition.
  2. There is no possible race condition.
  3. 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 (the goat) wrote about a year ago.
  - It does not exist anywhere else
  - you likely won't find any documentation on it outside of this course
    - Until y'all feed it into GPT so much that it becomes global memory of the model.
- Main difference:
  - When you create a thread, it starts "suspended"
  - Threads can be explicitly continued and suspended
    - (functionality that you'd use for preempting threads for your schedular in PennOS)
    - They were made to make PennOS easier to debug and write.
  - When there is a corresponding spthread function, call that instead of the pthread function

spthread functions

```
int spthread_create(...);
```

- The created pthread will be suspended before it executes the specified routine. It must first be continued with `spthread_continue` before it will start executing.

```
int spthread_suspend(spthread_t thread);
```

- spthread_suspend function will signal to the specified thread to suspend execution

```
int spthread_suspend_self();
```

- spthread_suspend function will cause calling thread to suspend itself

```
int spthread_continue(spthread_t thread);
```

- Will signal to the specified thread to resume execution if suspended.

There are many similar functions (one to one) with pthread library. Checkout the spthread .h and .c for more…

Poll Everywhere

pollev.com/5480

- 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/5480

- 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() {// all thread stuff up here
  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?

```
int spthread_disable_interrupts_self();
```

  - Calling this function from an spthread prevents it from being suspended/prempeted until re-enabled by the sibling function right below this text.

```
int spthread_enable_interrupts_self();
```

  - Calling this function from an spthread re-enables it to being suspendable. Should be called after it's sibling function "spthread_disable_interrupts_self".

Lecture Outline

- Threads & Lock refresher
- Spthreads
- **Test-Set-Lock**
- 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
  - You then check the value of R to see if it changed…

TSL to implement Mutex

- A mutex is *pretty* much this:

```
mutex_lock(int lock) {
   int prev_value = TSL(&lock); // imagine R is returned.

   // if prev_value = 1, then it was already locked
   while (prev_value == 1) {
      block(); //block itself…another thread could pre-empt here.
      prev_value = TSL(lock); //atomic
   }
}

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

`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 Spin Lock…

- A spinlock is pretty much this:

```
spin_lock(lock) {
   while (TSL(&lock) == 1);
}
spin_unlock(lock) {
  lock = 0;
}
```

```
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

- No blocking, no suspending, etc. (Why is that useful…?)
- Used to synchronize across multiple cores when you *can not block or suspend yourself* & *when you know you will release the lock quickly. (i.e. in an interrupt)*
  - *You would first disable interrupts (on your cpu) and aquire the spin lock. That way, the thread that acquired the lock can finish, while other threads (on other CPUs) spin waiting for it to come back, since they can't be pre-empted.*
  - *Used in kernal land to synchronize things the user will never see.*

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

Note: This is said in the context of spthread. Don’t go in your OS interviews and say let’s disable interrupts as a go to answer unless you know exactly why you’d want to do that.

Disabling Interrupts

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

```
++sum_total
```

Thread 0

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

Thread 1

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

`sum_total = 1`

`sum_total = 2`

- If one core, then once one is put on, then it will not relinquish its ownership until it re-enabled 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 as you'd need to use a spin lock in tandem.

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.

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*).
  - `pthread_mutex_t recmutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP; //legit code.`
- 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

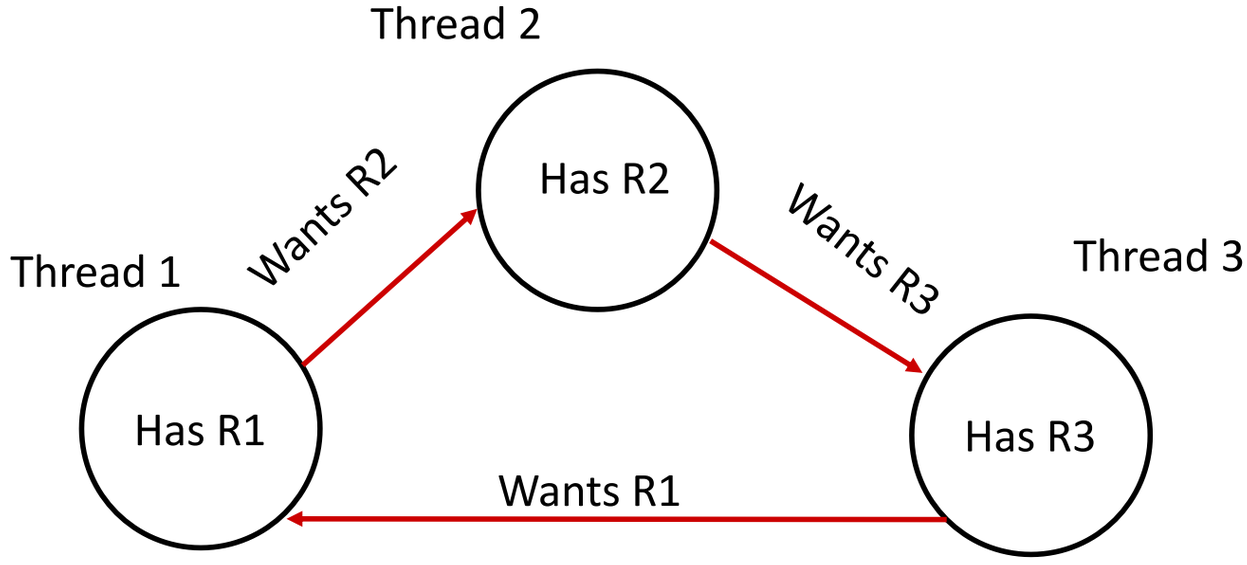
- A computer has multiple threads, finite resources, and the threads want to acquire those resources
  - Some of these resources require exclusive access
- A thread can acquire resources:
  - All at once
  - Accumulate them over time
  - If it fails to acquire a resource, it will (by default) wait until 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: Mutual Exclusion

- **Mutual Exclusion**: at least one resource must be held exclusively by one thread

- You usually need mutual exclusion or you don't, so it is hard to avoid.
- Some resources require exclusive access

- A lot of work done related to this
  - called: Lock-free programming, Lock-less programming, or Non-blocking algorithms
  - General idea is to take advantage of operations that are atomic at the hardware level when sharing is needed

Deadlock Prevention: Hold and Wait

- **Hold and Wait**: a thread must be holding a resource, requesting a resource that is held by a thread, and then waiting for it.

- What if we had each thread acquire all resources it needs in the beginning "at once"
  - Not always practical, a thread may not know ahead of time all the resources it will need

Deadlock Prevention: No Preemption

- **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

- If we force a thread to release a resource, how do we ensure it is in a valid state?
  - Undoing actions and recovering valid state is complex

Deadlock Prevention: Circular Wait

- **Circular Wait**: 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

- Break cycles in resource acquisition.
- We could enforce an ordering to resource acquisition.

- Challenge: Still we may not know all resources we need ahead of time

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!

Poll Everywhere

pollev.com/5480

- 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

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

- B) Thread-1 executes line 15 while Thread-2 executes line 15.
  Choose one:
  1. Could lead to a race condition.
  2. There is no possible race condition.
  3. 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/5480

- 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

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

- B) Thread-1 executes line 14 while Thread-2 executes line 16.
  Choose one:
  1. Could lead to a race condition.
  2. There is no possible race condition.
  3. 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/5480

- 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;
}
```

Could have a data race. A thread is suspended mid pushing onto the vector and another thread tries to push onto the same vector.

Poll Everywhere

pollev.com/5480

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

- Possible deadlock occurs. If we suspend a thread that has a lock, then main tries to acquire the lock, main will get stuck. Since main gets stuck the suspended thread holding the locking will not resume.

Discuss

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

- Yes, if it tries to acquire a lock that it already has.