Threads, Data Races, Locks & spthreads

Operating Systems, Fall 2026

Instructors: Travis McGaha

Head TAs: Shriya Sane

Arnuv Batra
Jeffrey Oduman
Kevin Xue
Leo Zhang
Khoi Dinh
Abdullah Goher
Shreya Mukunthan
Sydney Fitzgerald
Monisha Krothapalli
Daniel Price
Rania Souissi
Veer Kakar
Wesley Liu

Poll Everywhere

pollev.com/tqm

- How are you doing? Anything you want to talk about?

Logistics

- Penn-FAT is out!
  - It is due October 29th in pairs!
  - It is a HUGE assignment. And make sure to put in a lot of elbow grease.
  - If you get started on it and make progress every week, then it won't be too much work
  - The hardest part is debugging when something goes wrong
  - Some questions on the midterm will ask about concepts enforced by this assignment.
  - Recitation last week was designed to be helpful for it!

- Penn-fat milestone due by end of the week!
  - Please try to meet with your TA as soon as possible!
  - If you fail, you can take it again one more time (you will get different questions)

Logistics

- Final exam schedule posted:
  - Tuesday, December 15, 2026: 3:00pm to 5:00pm in JMHH G06

- Midterm coming up in class on October 22nd – details posted today
  - One double sided sheet of notes
  - 90 minutes
  - In lecture, you are either in Towne 100 or Moore 216
    - unsure where that is? Follow me after class
  - **Covers everything up to AND including lecture this lecture* (no spthreads)**
  - **RECITATION THIS WEEK IS MIDTERM REVIEW**
  - Midterm review in class Tuesday next week
  - Clobber policy still exists ☺

Lecture Outline

- **Threads Refresher & Data Races**
- Synchronization Mechanisms
- Data Race vs Race Condition
- Is a mutex needed? (Peterson's)
- It only gets worse
- spthreads

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

The Pain Point: Shared Resources

- POSIX.1: Some resources are shared between threads and processes
- All Threads Share the Following
  - PID, Parent PID, PGID, Controlling Terminal, File Descriptors, Interval timers (sleep, alarm…)
  - nice value (niceness applies *per process*, not per thread according to POSIX)
  - ***global memory (data and heap segments)*** ← ***Our focus***
- The following attributes are distinct for each thread
  - Stack
  - thread ID (the *pthread_t* data type)
  - signal mask (pthread_sigmask(3))
  - the *errno* variable

Data Races

- Two memory accesses form a data race if different threads access the same location, ***at least one is a write, and the accesses can co-occur.***
  - The state of a program can vary depending on scheduling…
    - **Which thread ran first?**
    - **When did a thread get interrupted?**

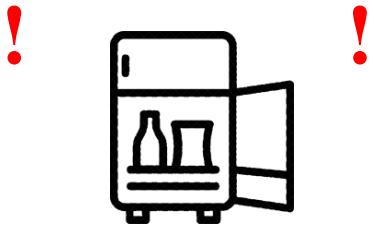
Data Race Example

- If your fridge has no milk, then go out and buy some more
  - What could go wrong?

```
if (!milk) {
  buy(milk);
}
```

- If you live alone:

- If you live with a roommate:



Poll Everywhere

pollev.com/tqm

- Idea: Leave a note!
  - Does this fix the problem?

```
if (!note) {
  if (!milk) {
    leave(note);
    buy(milk);
    remove(note);
  }
}
```

A. Yes, problem fixed
B. No, could end up with no milk
C. No, could still buy multiple milk
D. We're lost…

Threads and Data Races

- Data races might interfere in painful, non-obvious ways, depending on the specifics of the data structure
- Example: two threads try to read from and write to the same shared memory location
  - Could get "correct" answer
  - Could accidentally read old value
  - One thread's work could get "lost"
- Example: two threads try to push an item onto the head of the linked list at the same time
  - Could get "correct" answer
  - Could get different ordering of items
  - Could break the data structure! ☠

This should remind you of the signal interruption!

We were essentially interrupting one stream of execution with another!

Remember this?

- What does this print?

Always prints 0, the global counter is not shared across processes, so the parent's global never changes

```c
#define NUM_PROCESSES 50
#define LOOP_NUM 100

int sum_total = 0;

void loop_incr() {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  printf("Process ID: %d with sum total of %d.\n", getpid(), sum_total);
}

int main(int argc, char **argv) {
  pid_t pids[NUM_PROCESSES];  // array of process IDs

  // create processes to run loop_incr()
  for (int i = 0; i < NUM_PROCESSES; i++) {
    pids[i] = fork();
    if (pids[i] == 0) {
      loop_incr();
      exit(EXIT_SUCCESS);
    }
  }

  // wait for all child processes to finish
  for (int i = 0; i < NUM_PROCESSES; i++) {
    waitpid(pids[i], NULL, 0);
  }

  printf("The ultimate sum total is %d\n", sum_total);
  return EXIT_SUCCESS;
}
```

Remember this?

- What does this print?

Usually 5000

```c
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

void *loop_incr(void *arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  printf("Thread ID: %d with sum total of %d.\n", gettid(), sum_total);
  return NULL;
}

int main(int argc, char **argv) {
  pthread_t thds[NUM_THREADS];  // array of thread IDs

  // create threads to run loop_incr()
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_create(&thds[i], NULL, &loop_incr, NULL) != 0) {
      fprintf(stderr, "pthread_create failed\n");
    }
  }

  // wait for all non-main threads to finish
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(thds[i], NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
    }
  }

  printf("The ultimate sum total is %d\n", sum_total);
  return EXIT_SUCCESS;
}
```

Previous Demos:

- See `total.c` and `total_processes.c`
  - Threads share an address space, if one thread increments a global, it is seen by other threads
  - Processes have separate address spaces, incrementing a global in one process does not increment it for other processes

- NOTE: sharing data between threads is unsafe if done wrong (we are doing it wrong in this example); let's expand on that now ☺

Increment Data Race

- What can be written in one line in C `sum_total++` is multiple assembly instructions in one. The increment looks something like this in assembly:

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

Equivalent pseudo →

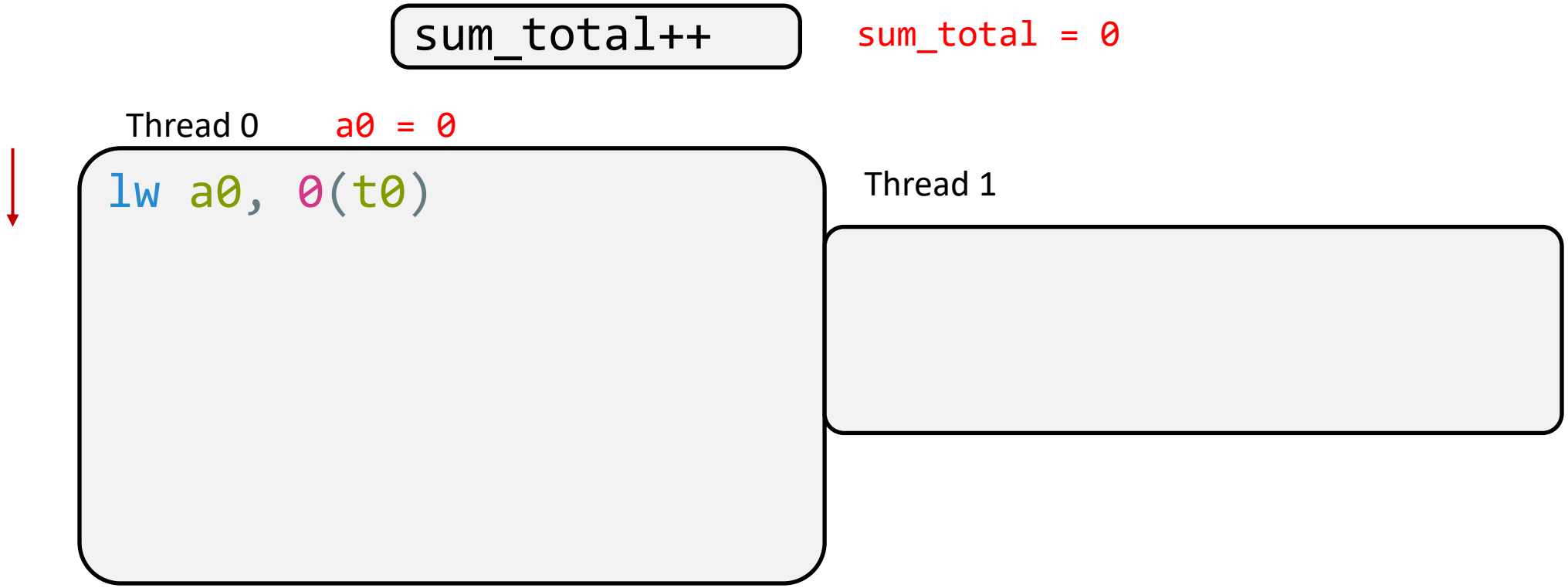
```
a0   = *t0
a0   = a0 + 1
*t0 = a0
```

- What happens if we context switch to a different thread while executing these three instructions?

- **Reminder: Each thread has its own registers to work with. Each thread would have its own** `a0 & t0`
  - But they would share the memory location stored within `t0`

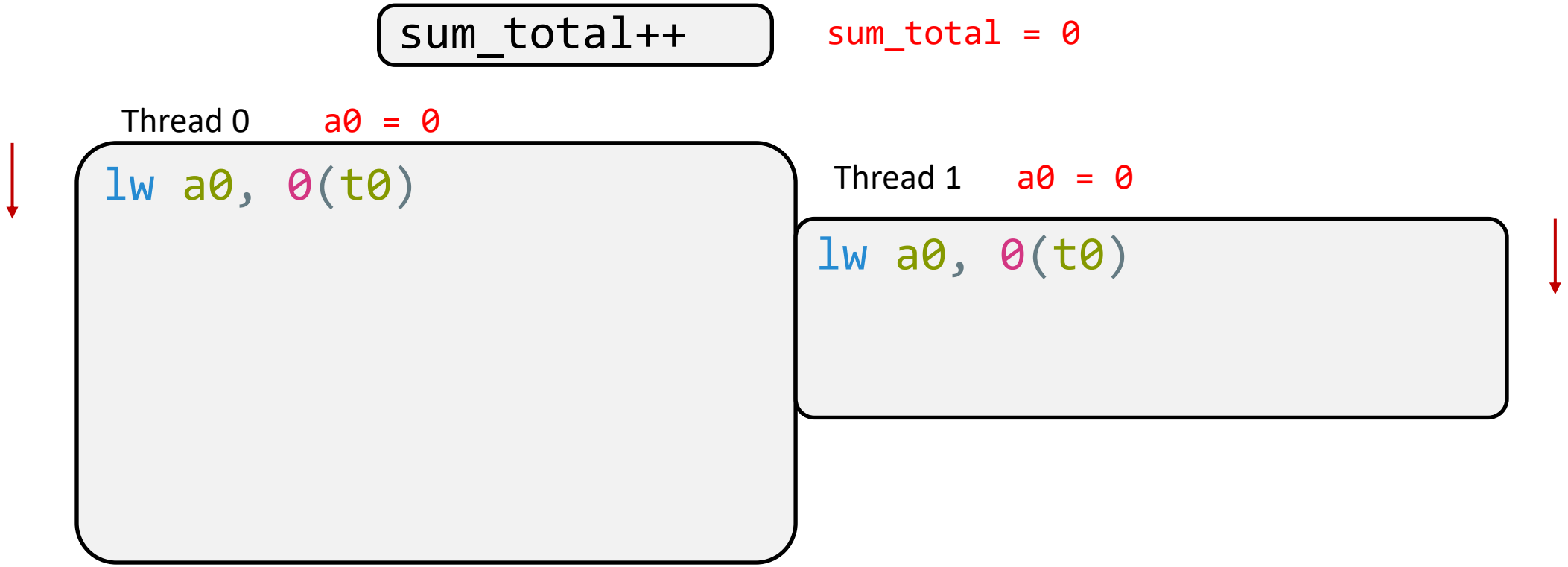
Increment Data Race

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



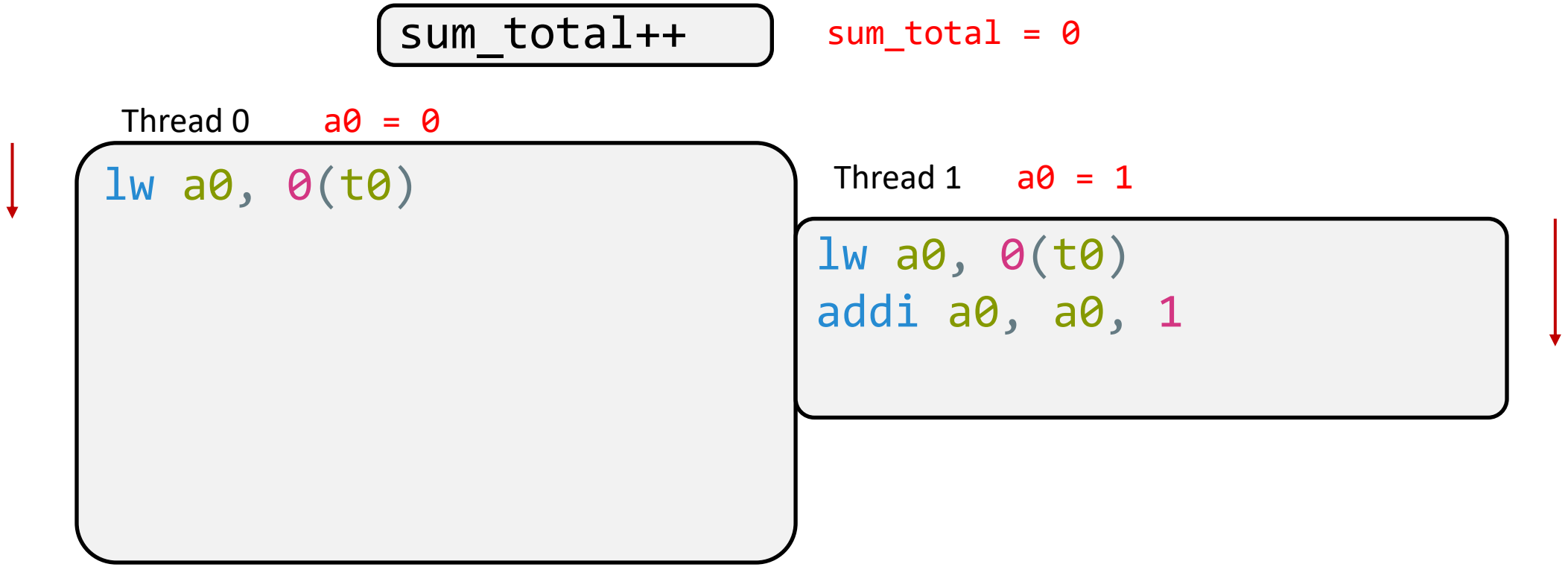
Increment Data Race

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



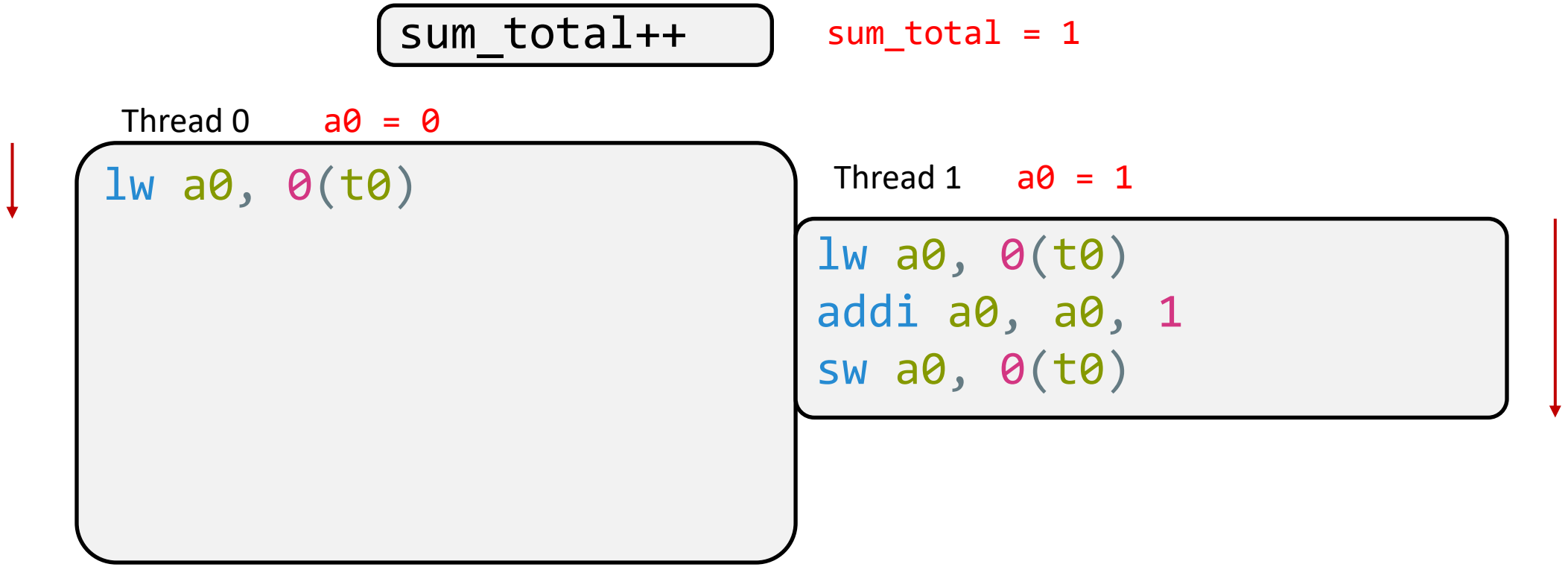
Increment Data Race

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



Increment Data Race

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



Increment Data Race

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

```
sum_total++
```

`sum_total = 1`

Thread 0 `a0 = 1`

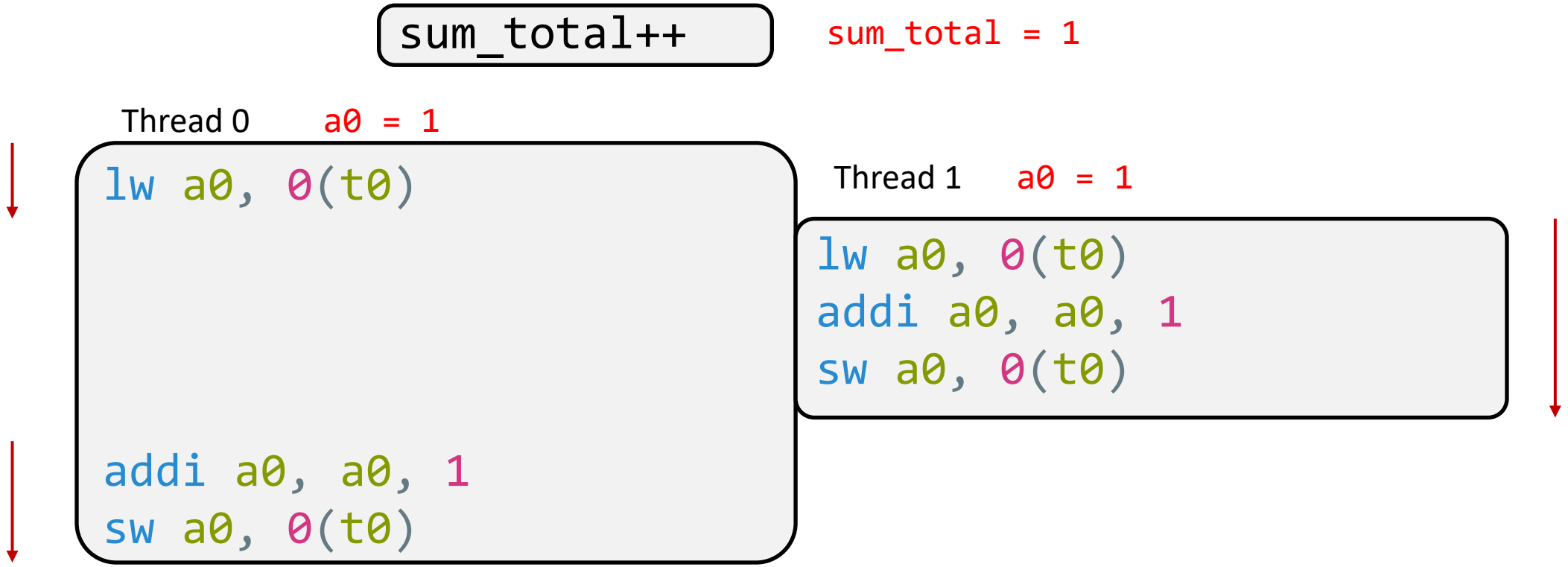
```
lw a0, 0(t0)

addi a0, a0, 1
```

Thread 1 `a0 = 1`

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

Increment Data Race

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



- With this example, we could get 1 as an output instead of 2, even though we executed `addi` twice.

Poll Everywhere

pollev.com/tqm

- What is the minimum value that could be printed by main thread?
  - Single Core
  - Concurrent Threading Only

- Important: all three instructions execute on each loop

```
lw a0, 0(t0)
addi a0, a0, 1
sw a0, 0(t0)
```

```c
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

void *loop_incr(void *arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  printf("Thread ID: %d with sum total of %d.\n", gettid(), sum_total);
  return NULL;
}

int main(int argc, char **argv) {
  pthread_t thds[NUM_THREADS];  // array of thread IDs

  // create threads to run loop_incr()
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_create(&thds[i], NULL, &loop_incr, NULL) != 0) {
      fprintf(stderr, "pthread_create failed\n");
    }
  }

  // wait for all non-main threads to finish
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(thds[i], NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
    }
  }

  printf("The ultimate sum total is %d\n", sum_total);
  return EXIT_SUCCESS;
}
```

Lecture Outline

- Threads Refresher & Data Races
- **Synchronization Mechanisms**
- Data Race vs Race Condition
- Is a mutex needed? (Peterson's)
- It only gets worse
- spthreads

Synchronization

- Synchronization is the act of preventing two (or more) concurrently running threads from interfering with each other when operating on shared data
  - Need some mechanism to coordinate the threads
    - "Let me go first, then you can go"
  - Many different coordination mechanisms have been invented

- Goals of synchronization:
  - Liveness – ability to execute in a timely manner (informally, "something good eventually happens")
  - Safety – avoid unintended interactions with shared data structures (informally, "nothing bad happens")

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

- Pseudocode:

```
// non-critical code

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

// non-critical code
```

block if locked

- 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
  - 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**
  - What if a thread tries to acquire a lock it already has?

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* mutexattr);
```

- Initializes the mutex object pointed to by `mutex` according to the mutex attributes specified in `mutexattr`
- Use NULL for default attributes

```
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

pthread Mutex Locks

```
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
- Linux allows ***any*** thread to unlock a mutex, even if it isn't its owner!
  - This is ***not*** true across operating systems

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

- Destroys the given mutex, freeing any resources it holds

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.

Lecture Outline

- Threads Refresher & Data Races
- Synchronization Mechanisms
- **Data Race vs Race Condition**
- Is a mutex needed? (Peterson's)
- It only gets worse
- spthreads

Discuss

```c
pthread_mutex_t lock;
bool print_ok = false;

void* thrd_fn1(void* arg) {
  pthread_mutex_lock(&lock);
  print_ok = true;
  pthread_mutex_unlock(&lock);
  return NULL;
}

void* thrd_fn2(void* arg) {
  pthread_mutex_lock(&lock);
  if (print_ok) {
    printf("print ok is true\n");
  } else {
    printf("print ok is false\n");
  }
  pthread_mutex_unlock(&lock);
  return NULL;
}

int main() {
  // assume main sets ups the threads & locks, etc.
```

- Does this code have a data race?
  - Can this program enter an “invalid” (unexpected or error) state from having concurrent memory accesses?
- Follow up: Does this code feel good?

Race Condition vs Data Race

- **Data Race**: When there are concurrent accesses to a shared resource, with at least one write, that can cause the shared resource to enter an invalid or "unexpected" state.

- **Race Condition**: Where the program has different behavior depending on the ordering of concurrent threads. This can happen even if all accesses to shared resources are "atomic" or "locked"
  - THINK SCHEDULER! SCHEDULER SCHEDULER SCHEDULER!

- The previous example has no data race, but it does have a race condition
  - Data races are a subset of race conditions

Thread Communication

- Sometimes threads may need to communicate with each other to know when they can perform operations

- Example: Producer and consumer threads
  - One thread creates tasks/data
  - One thread consumes the produced tasks/data to perform some operation
  - The consumer thread can only produce things once the creator has created them

- Need to make sure this communication has no data race or race condition

Lecture Outline

- Threads Refresher & Data Races
- Synchronization Mechanisms
- Data Race vs Race Condition
- **Is a mutex needed? (Peterson's)**
- It only gets worse
- spthreads

Discuss

- Let's try a more complicated software approach…
- We create two threads running `thread_code`, one with `me = 0`, other thread has `me = 1`
- Each thread tries to increment `sum_total`. Does this work?

```
int sum_total = 0;
atomic bool flag[2] = {false, false};
atomic int turn = 0
void thread_code(void *arg) {
  int me = (int) arg;

  flag[me] = true;
  turn = 1 - me;
  while(flag[1 - me] && (turn != me)) { }
  ++sum_total;
  flag[me] = false;
}
```

Note: atomic <type> is a real thing just not with the syntax used here. Reading and writing are guaranteed atomic.

Peterson's Algorithm

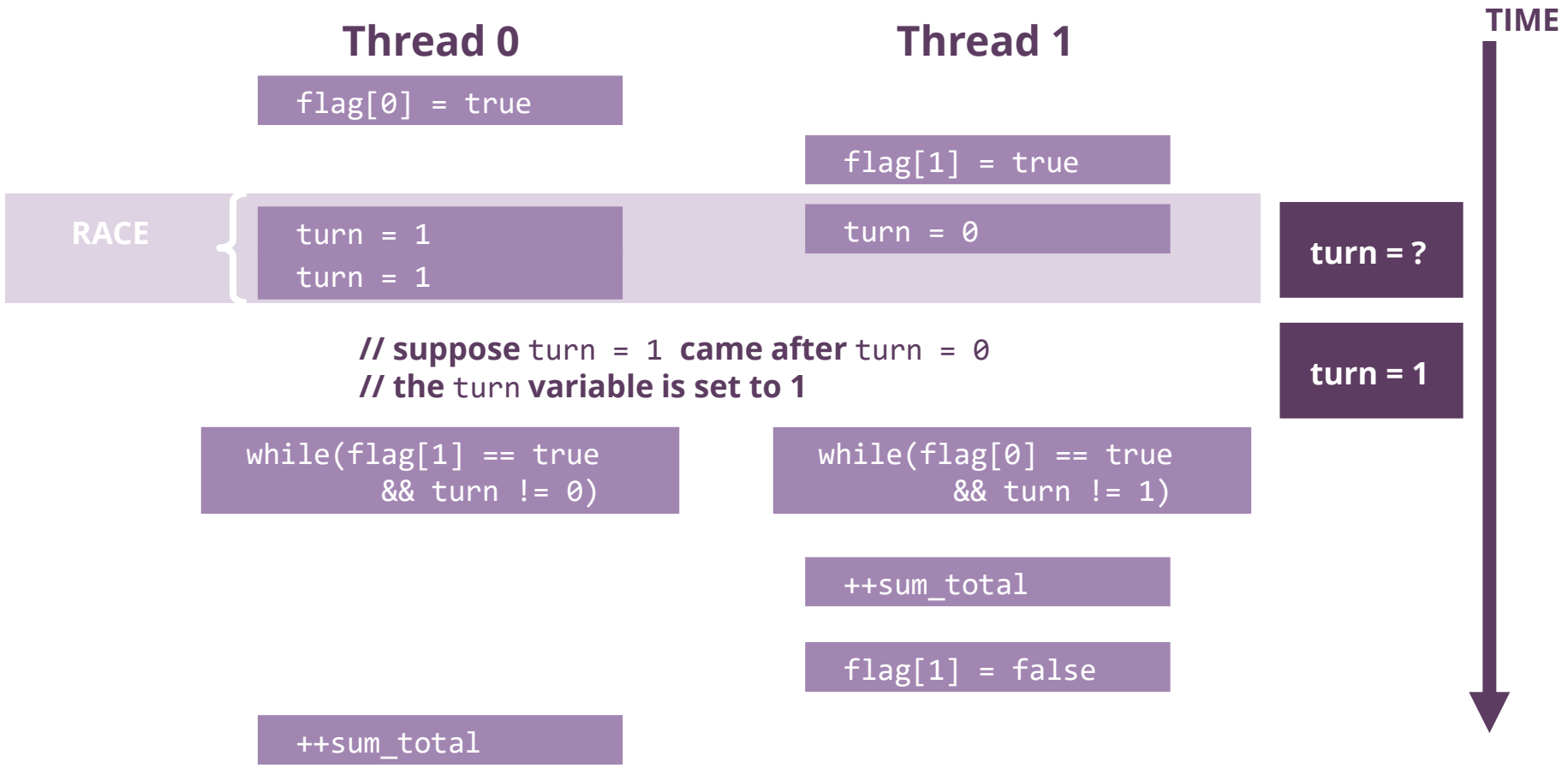
- What we just did was Peterson's algorithm
- Why does it work? (using an analogy)
  - Each thread first declares that they want to enter the critical section by setting their flag
  - Each thread then states (once) that the other should "go first"
    - This is done by setting the turn variable to `1 - me`
    - One of these assignments to the turn variable will happen last, that is the one that decides who goes first
  - One of the thread goes first (decided by the value of turn) and accesses the critical section, before saying it is done (by changing their flag to false)

Peterson's Algorithm

- What we just did was Peterson's algorithm
- Why does it work?
  - Case1:
    If P0 enters critical section, flag[0] = true, turn = 0. It enters the critical section successfully.
  - Case2:
    If P0 and P1 enter critical section, flag[0] and flag[1] = true

    Race condition on turn. Suppose P0 sets turn = 0 first. Final value is turn = 1. P0 will get to run first.

Explanation



Peterson's Assumptions

- Some operations are atomic:
  - Reading from the flag and turn variables cannot be interrupted
  - Writing to the flag and turn variables cannot be interrupted
  - E.g., setting turn = 1 or 0 will set turn to 0 or 1, you can be interrupted before or after, but not "during" when turn may have some intermediate value that is not 0 or 1

- That the instructions are executed in the specific order laid out in the code

Atomicity

- Atomicity: An operation or set of operations on some data are *atomic* if the operation(s) are indivisible, that no other operation(s) on that same data can interrupt/interfere.

- Aside on terminology:
  - Often interchangeable with the term “Linearizability”
  - Atomic has a different (but similar-ish) meaning in the context of data bases and ACID.

Lecture Outline

- Threads Refresher & Data Races
- Synchronization Mechanisms
- Data Race vs Race Condition
- Is a mutex needed? (Peterson's)
- **It only gets worse**
- spthreads

Discuss

- Can the assert fail here?

```
bool flag = false;
int x = 0;

void* thrd_fn1(void* arg) {
  x = 5;
  flag = true;
  return NULL;
}

void* thrd_fn2(void* arg) {
  if (flag) {
    assert(x == 5);  // crashes if x does not equal 5
  }
  return NULL;
}

int main() {
  // assume main creates two threads, one to run thrd_fn1 and another for thrd_fn2
```

Instruction & Memory Ordering

- Do we know that x is set before g is set?

```
bool g = false;
int x = 0

void some_func(int arg) {
  x = 5;
  g = true;
}
```

Instruction & Memory Ordering

- Do we know that x is set before g is set?

```
bool g = false;
int x = 0

void some_func(int arg) {
  x = 5;
  g = true;
}
```

NO

- The compiler may generate instructions that set g before x!
  - Why? Optimizations on program performance

Aside: Instruction & Memory Ordering

- The compiler may generate instructions with different ordering if it does not appear that it will affect the semantics of the function
  - Since `g = true;` is not affected by `x = 5;` then either one could execute first
- The Processor may also execute these in a different order than what the compiler says
- Why? Optimizations on program performance
  - If you want to know more, look into “Out-of-Order Execution” and “Memory Order”

Aside: Memory Barriers

- How do we fix this?

- We can emit special instructions to the CPU and/or compiler to create a "memory barrier"
  - "all memory accesses before the barrier are guaranteed to happen before the memory accesses that come after the barrier"
  - A way to enforce an order in which memory accesses are ordered by the compiler and the CPU

- The further you go the discrepancy becomes more nuanced
  - Do we want to force mem access to occur in the order of the written program?
  - Do we want to force that all memory access/modifications *to complete* before the next instruction executes?

Why Threads?

- Advantages:
  - You (mostly) write sequential-looking code
  - Threads can run in parallel if you have multiple CPUs/cores

- Disadvantages:
  - If threads share data, you need locks or other synchronization
    - Very bug-prone and difficult to debug
  - Threads can introduce overhead
    - Lock contention, context switch overhead, and other issues
  - Need language support for threads

Lecture Outline

- Threads Refresher & Data Races
- Synchronization Mechanisms
- Data Race vs Race Condition
- Is a mutex needed? (Peterson's)
- It only gets worse
- **spthreads**

Key Differences of spthread vs pthread

- spthread is something Travis (me) wrote in Spring 2024.
  - 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 will not show up on the midterm, it is only for PennOS**

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/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() {// 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".

That's all!

- See you next time!