Threads & Synchronization

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Administrivia

- Full PennOS is due Mon Nov 27
- Will post some more info before next lecture about the demo & testing functionality

- Recitation after lecture will talk more about PennOS, maintaining the abstraction and integrating the two pieces of PennOS

Poll Everywhere

pollev.com/tqm

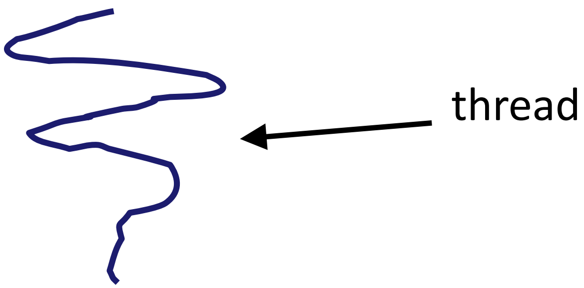
- Any questions, comments or concerns from last lecture?

Lecture Outline

- **Threads Quick Refresher**
- Shared Resources & Data Races
- Disable Interrupts
- Peterson's Algorithm
- Mutex
- TSL

Introducing Threads

- Separate the concept of a process from the "*thread of execution*"
  - Threads are contained within a process
  - Usually called a thread, this is a sequential execution stream within a process

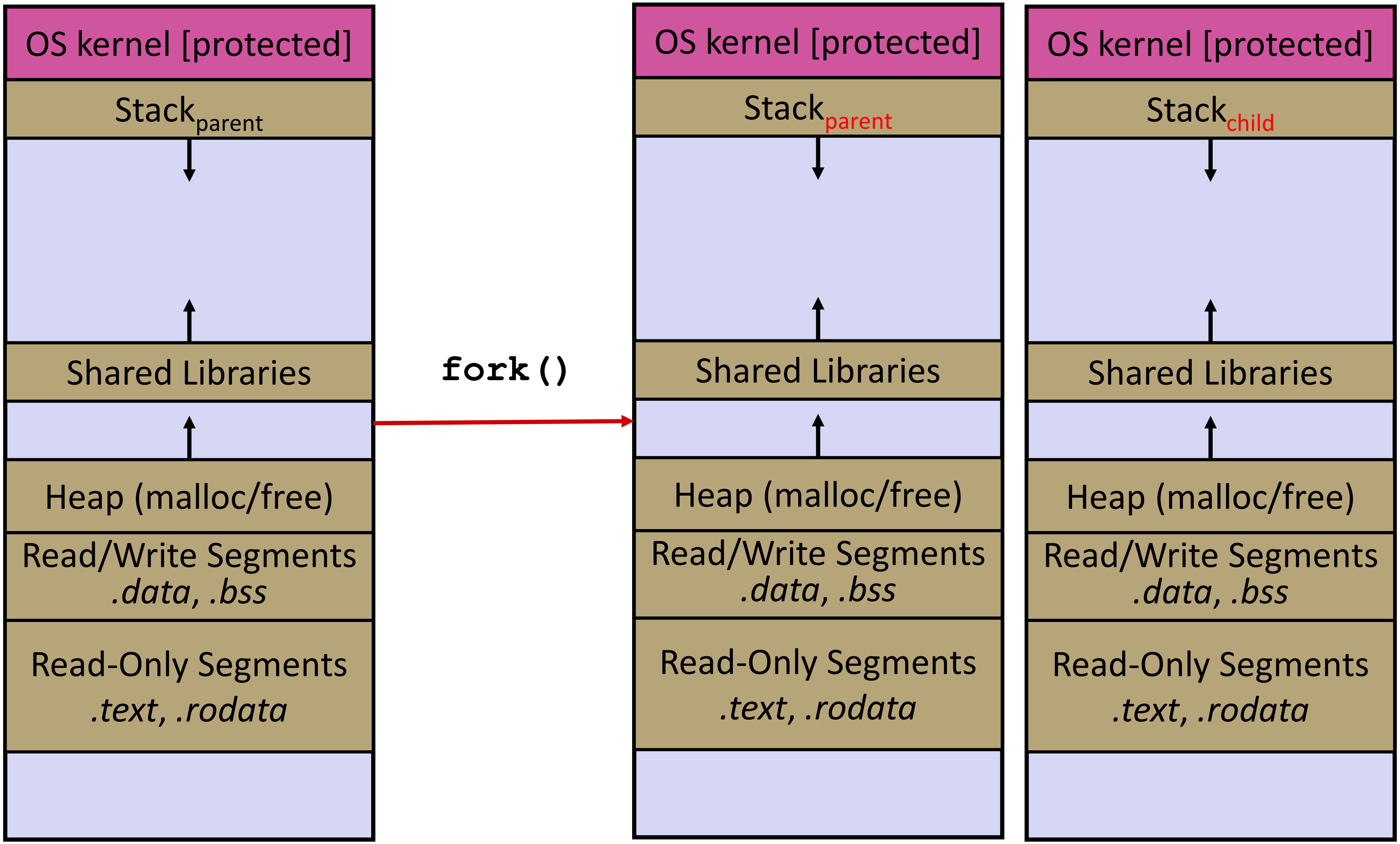


- In most modern OS's:
  - Threads are the *unit of scheduling.*

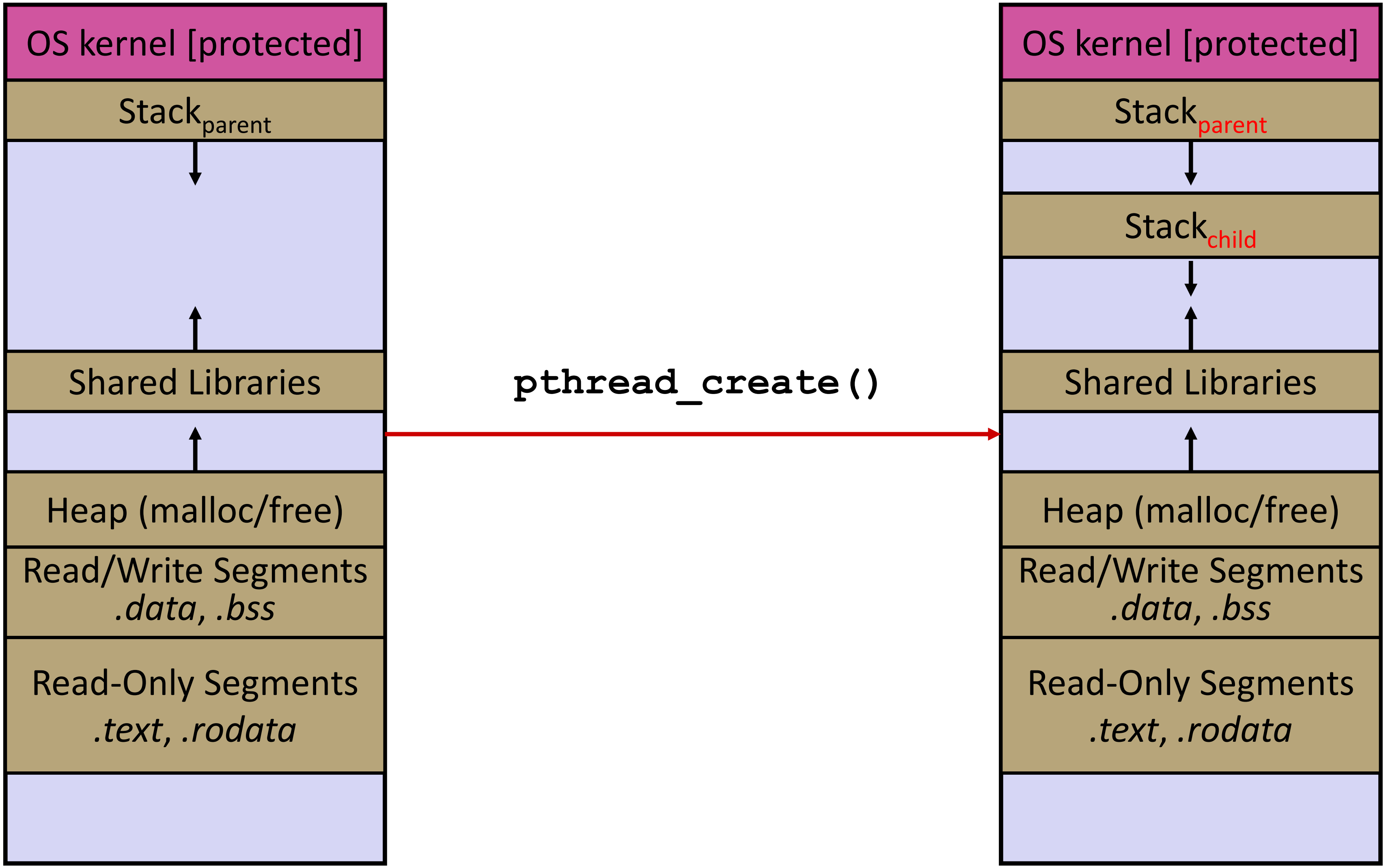
Threads vs. Processes

- 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



POSIX Threads (pthreads)

- The POSIX APIs for dealing with threads
  - Declared in `pthread.h`
    - Not part of the C/C++ language
  - To enable support for multithreading, must include `-pthread` flag when compiling and linking with `gcc` command
    - `gcc -g -Wall -pthread -o main main.c`
  - Implemented in C
    - Must deal with C programming practices and style

Creating and Terminating Threads

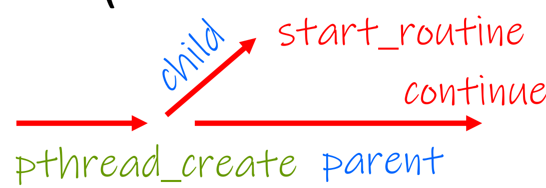
Output parameter.
Gives us a "thread_descriptor"

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

Argument for the thread function

Function pointer!
Takes & returns void*
to allow "generics" in C

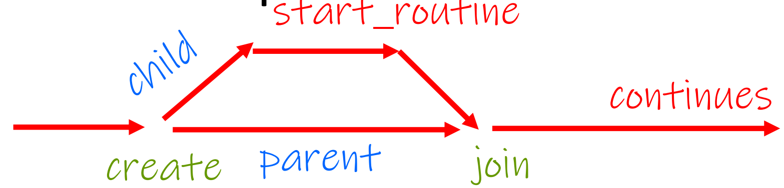
- Creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
- Returns **0** on success and an error number on error (can check against error constants)
- The new thread runs **`start_routine`**`(arg)`



What To Do After Forking Threads?

- `int pthread_join(pthread_t thread, void** retval);`
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of **`waitpid`**`()`
  - The exit status of the terminated thread is placed in `**retval`

Parent thread waits for child thread to exit, gets the child's return value, and child thread is cleaned up



Why Threads?

- Advantages:
  - You (mostly) write sequential-looking code
  - Threads can run in parallel if you have multiple CPUs/cores
    - Takes advantage of the multiple cores
    - Can make progress on multiple tasks at once, even if only 1 core
- 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 Quick Refresher
- **Shared Resources & Data Races**
- Disable Interrupts
- Peterson's Algorithm
- Mutex
- TSL

Shared Resources

- Some resources are shared between threads and processes

- Thread Level:
  - Memory
  - Things shared by processes

- Process level
  - I/O devices
    - Files
    - terminal input/output
    - The network

Issues arise when we try to shared things

Data Races

- Two memory accesses form a data race if different threads access the same location, and at least one is a write, and they occur one after another
  - Means that the result of a program can vary depending on chance **(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…

Poll Everywhere

pollev.com/tqm

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

We can be interrupted between checking note and leaving note ☹

```
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…

*There are other possible scenarios that result in multiple milks

you	roommate
Check note	
	Check note
Check milk	
Leave note	
	Check milk
	Leave note
	Buy milk
Buy milk	

time

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! ☠

Poll Everywhere

pollev.com/tqm

- What does this print?

```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++;
  }
}


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) {
      // child
      loop_incr();
      exit(EXIT_SUCCESS);
    }
    // parent loops and forks more children
  }

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

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

  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?

```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++;
  }
}


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) {
      // child
      loop_incr();
      exit(EXIT_SUCCESS);
    }
    // parent loops and forks more children
  }

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

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

  return EXIT_SUCCESS;
}
```

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

Poll Everywhere

pollev.com/tqm

- What does this print?

```
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

// print out a sequence of LOOP_NUM numbers with thread identifying number
void* thread_main(void* arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  return NULL;  // return type is a pointer
}


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

  // create threads to run thread_main()
  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_create(&thds[i], NULL, &thread_main, NULL);
  }

  // wait for all child threads to finish
  // (children may terminate out of order, but cleans up in order)
  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(thds[i], NULL);
  }

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

  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?

```
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

// print out a sequence of LOOP_NUM numbers with thread identifying number
void* thread_main(void* arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  return NULL;  // return type is a pointer
}


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

  // create threads to run thread_main()
  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_create(&thds[i], NULL, &thread_main, NULL);
  }

  // wait for all child threads to finish
  // (children may terminate out of order, but cleans up in order)
  for (int i = 0; i < NUM_THREADS; i++) {
    pthread_join(thds[i], NULL);
  }

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

  return EXIT_SUCCESS;
}
```

Usually 5000

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 actually kinda unsafe if done wrong (we are doing it wrong in this example), **more on this NOW**

Increment Data Race

- What seems like a single operation `++sum_total` is actually multiple operations in one. The increment looks something like this in assembly:

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

- 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 R0**

Increment Data Race

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

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD   sum_total into R0
```

Thread 1

Increment Data Race

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

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD  sum_total into R0
```

Thread 1 **R0 = 0**

```
LOAD  sum_total into R0
```

Increment Data Race

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

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD   sum_total into R0
```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
```

Increment Data Race

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

sum_total = 1

Thread 0 **R0 = 0**

```
LOAD  sum_total into R0
```

Thread 1 **R0 = 1**

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

Increment Data Race

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

sum_total = 1

Thread 0 **R0 = 1**

```
LOAD   sum_total into R0


ADD    R0 R0 #1
```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
STORE  R0 into sum_total
```

Increment Data Race

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

sum_total = 1

Thread 0 **R0 = 1**

```
LOAD  sum_total into R0



ADD   R0 R0 #1
STORE R0 into sum_total
```

Thread 1 **R0 = 1**

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

- With this example, we could get 1 as an output instead of 2, even though we executed ++sum_total twice

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")

Lecture Outline

- Threads Quick Refresher
- Shared Resources & Data Races
- **Disable Interrupts**
- Peterson's Algorithm
- Mutex
- TSL

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 invoke the scheduler

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 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 Quick Refresher
- Shared Resources & Data Races
- Disable Intertupts
- **Peterson's Algorithm**
- Mutex
- TSL

Poll Everywhere

pollev.com/tqm

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

```
int sum_total = 0;
bool flag[2] = {false, false};
int turn = 0

void thread_code(int arg) {
  int me = arg;

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

Check the index of the other thread

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

TIME ↓

Thread 0

`flag[0] = true`

Thread 1

`flag[1] = true`

RACE

Thread 0:

```
turn = 0
turn = 0
```

Thread 1:

```
turn = 1
```

turn = ?

// suppose `turn = 0` came after `turn = 1`
// the `turn` variable is set to 0

turn = 0

Thread 0:

```
while(flag[1] == true
      && turn == 0)
```

Thread 1:

```
while(flag[0] == true
      && turn == 1)
```

Thread 1:

```
++sum+total
```

```
flag[1] = false
```

Thread 0:

```
++sum_total
```

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.

Aside: Instruction & Memory Ordering

- Do we know what t is set before g is set?

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

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

Aside: Instruction & Memory Ordering

NO

- Do we know what t is set before g is set?

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

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

The compiler may generate instructions that sets g first and then t
The Processor may execute these out of order or at the same time

Why? Optimizations on program performance

You can be guaranteed that `t` and `g` are set before `some_func` returns

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 `t = arg;` 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

Lecture Outline

- Threads Quick Refresher
- Shared Resources & Data Races
- Disable Interrupts
- Peterson's Algorithm
- **Mutex**
- TSL

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

// non-critical code
```

- 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 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?

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

- 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)

```
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`

Lecture Outline

- Threads Quick Refresher
- Shared Resources & Data Races
- Disable Interrupts
- Peterson's Algorithm
- Mutex
- **TSL**

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);
}
```