C++ Threads, std::atomic, Condition Variables

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

Administrivia

- HW07: Due end of day Tuesdeay

- Midterm Grading
  - Grades posted sometime on Thursday
  - Just waiting on a makeup exam
  - No talking about it yet please!

- HW08: Posted tomorrow.
  - Can finish half of it after this lecture.
    Other half will benefit from Wednesday's Lecture (Parallel Algorithms)

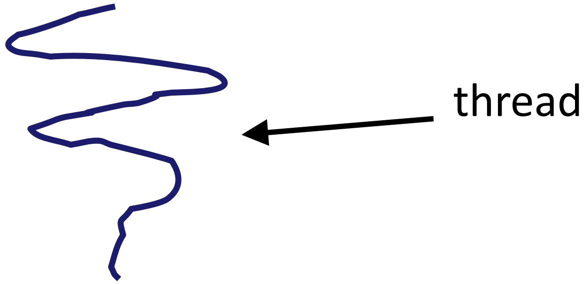
- Mid Semester Survey
  - Due End-of-day Nov 1st (Saturday)

Lecture Outline

- **C++ Threads**
  - **std::jthread**
  - **std::mutex**
  - **std::scoped_lock**
- std::atomic
- Condition variable

Remember 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

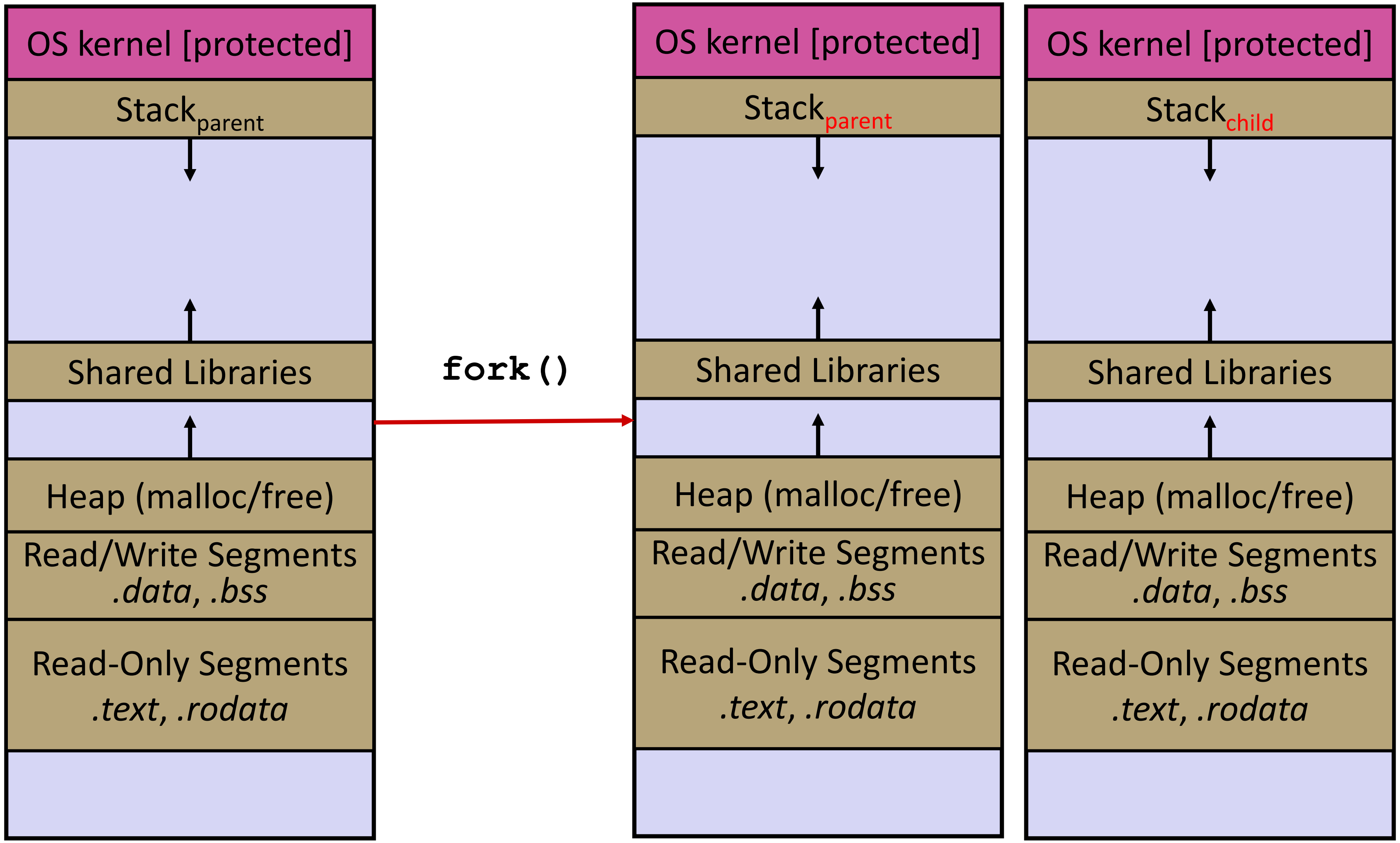


- In most modern OS's:
  - Threads are the *unit of scheduling.*

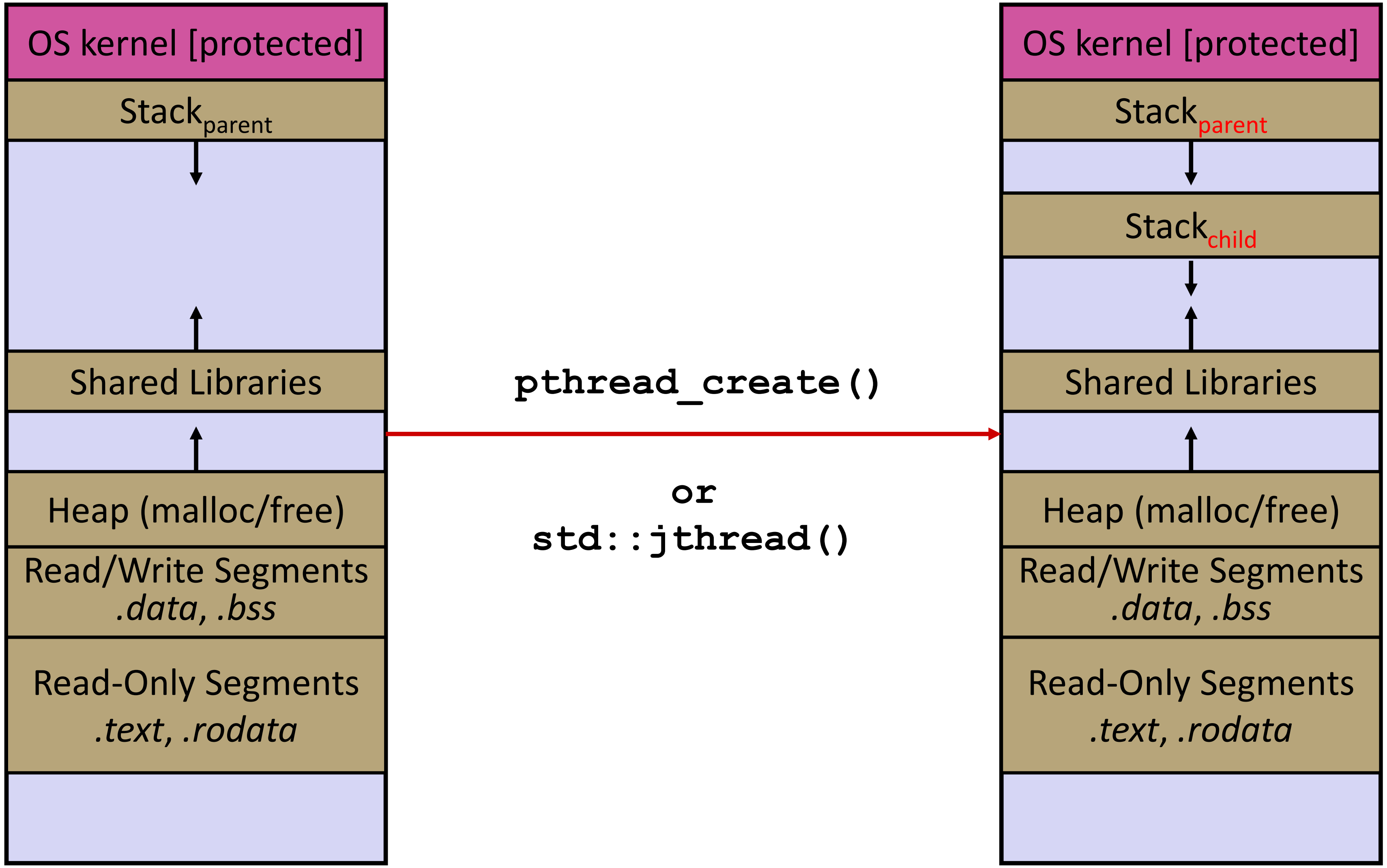
Threads vs. Processes

- 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



Introducing C++ Threads

simple_jthread.cpp

```
void thread_func1(std::string to_print) {
  std::cout << to_print;
}

void thread_func2(int x, int y) {
  std::cout << x + y << std::endl;
}

int main() {
  std::string hello("Hello!\n");
  std::jthread thd1(thread_func1, hello);

  thd1.join();

  std::jthread thd2(thread_func2, 3, 4);
}
```

- C++ has the **`jthread`** class to represent a thread.
- Constructing an instead of `jthread` creates a thread
- No `void*`, just pass in the args you want to pass in
- When a `jthread` destructs, it will join the thread that it represents. Remember that join() blocks until the thread is joined in.
- How many possible outputs are there? (demo)

Passing refs to std::jthread

- To pass a ref to a jthread, you must wrap the argument with a call to `std::ref()`.

```
void thread_func(std::string& input) {
  input += " residue";
}

int main() {
  std::string str("system");
  std::jthread thd1(thread_func, std::ref(str));

  thd1.join();

  std::cout << str << endl;
}
```

- Why? short version:
  jthread is a template object and `std::ref` helps it deduce that the parameter is a ref and not a value.

std::mutex

- Similar to mutex, from pthread.h but cleaner to use:

```
int total = 0;
std::mutex total_mutex;


void thrd_fn() {
  total_mutex.lock();
  total += 1;
  total_mutex.unlock();
}
```

- Has methods
  - lock()
  - try_lock()
  - unlock()

- Rarely do we use the above functions, in C++ (and rust) we do (next slide)

std::scoped_lock

- `std::scoped_lock` is an object
  - Constuctor: acquires locks
  - Destructor: releases locks
- Takes advantage of RAII to manage "locks" as a resource like how vector's manage memory

```
int total = 0;
std::mutex total_mutex;

void thrd_fn() {
  // constructor: acquire the lock
  std::scoped_lock total_lock(total_mutex);

  total += 1;

  // scoped lock destructor
  // automatically releases the lock
}
```

- Forgetting to release a lock would prevent any other thread from acquiring said lock. Could halt your program ☹
- Since it is a destructor, it will also automatically unlock if an exception is thrown. (whereas using lock() and unlock() would not)
- Constructor can take multiple locks, and will acquire all of them.

Ed Practice!

Lecture outline

Instructions Instructions

- How many instructions does it take to set a bool to true?

```
bool x = false;

void thrd_fn() {
  x = true;
}
```

Do we need a lock for this?
What if multiple threads run thrd_fn?

- Is still "undefined behaviour" by the standard.
- Multiple actions can be re-ordered still ☹

- Just one! (sorta, any setup is not really important)

x86

```
mov      BYTE PTR x[rip], 1
```

arm

Calculate address

```
adrp     x8, x
mov      w9, #1
```

Set bool

```
strb     w9, [x8, :lo12:x]
```

risc-v

Calculate address

```
auipc   a1, %pcrel_hi(x)
```

Load "true" into a2

```
li      a2, 1
```

set x

```
sb      a2, %pcrel_lo(.Lpcrel_hi0)(a1)
```

Discuss & Raise Hands

- Remember this?
- Can this code ever print "AAAAAAA"?
- Assume that thd_main1 and thd_main2 are run by two separate threads.
- Assume no other threads use x or y

```
int x = 0;
int y = 0;

void thd_main1() {
  x = 3;
  y = 4;
}

void thd_main2() {
  if (y == 4) {
    if (x == 3) {
      cout << "We good :)\n";
    } else {
      cout << "AAAAAAAAAA\n";
    }
  }
}
```

Sequential Consistency

- Do we know that **`t`** is set before **`g`** is set?

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

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

Instruction & Memory Ordering

- Do we know that `t` is set before `g` is set?

NO

```
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

std::atomic

- Enforces memory order
- Guaranteed to not be a data race / undefined behavior
- Can be used for any "trivially copyable" type
  - Copy constructor does nothing other than copy bytes.

```
std::atomic<bool> x = false;
std::atomic<int> counter = 0;

void thrd_fn() {
  x = true;
  ++counter;
}
```

- May not use a lock!
  - Use function `is_always_lock_free()` or `is_lock_free()` to determine if it uses a lock

std::atomic

- How does it work?
  - Takes advantages of architecture specific instructions that are known to be atomic when available
  - If no such instruction is available, then a lock is used.

```
std::atomic<bool> x = false;
std::atomic<int> counter = 0;

void thrd_fn() {
  x = true;
  ++counter;
}
```

Lecture Outline

Aside: std::unique_lock

- std::unique_lock
  - Does things very similar to std::scoped_lock
  - Has more functionality! Including ability to release lock manually
  - Is easier to mess up in some cases
  - Prefer std::scoped_lock when possible

```
int total = 0;
std::mutex total_mutex;

void thrd_fn() {
  std::unique_lock total_lock(total_mutex);
  // constructor: acquire the lock

  total_lock.unlock();
  total_lock.lock();

  total += 1;

  // destructor automatically
  // releases the lock
}
```

- Why are we introducing?
  Must be used with
  std::condition_varaible
  - (introduce condition variable later)

Discuss

```
std::mutex m;
string data;

void produce() {
  std::ifstream infile("hello.txt");
  string line;
  getline(infile, data);

  std::scoped_lock(m);
  data = line;
}
```

```
void consume() {
  std::unique_lock(m);
  if (!data.empty()) {
    std::cout << data << std::endl;
  }
}
```

- Does this code have a data race?
  - Assume that there is one thread running produce() and another thread running consume()
  - Can this program enter an “invalid” (unexpected or error) state from having concurrent memory accesses?
  - Assume funcs don’t fail
- Any issues with this code?

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 behaviour depending on the ordering of concurrent threads. This can happen even if all accesses to shared resources are "atomic" or "locked"

- The previous example has no data-race, but it does have a race condition

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 consume things once the producer has produced them

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

Producer & Consumer Problem

- Common design pattern in concurrent programming.
  - There are at least two threads, at least one producer and at least one consumer.
  - The producer threads create some data that is then added to a shared data structure
  - Consumers will process and remove data from the shared data structure

- We need to make sure that the threads play nice

Aside: C++ deque

- I am using a c++ **`deque`** for this example so that we don't have to write our own data structure.

- Deque is a double ended queue, you can push to the front or back and pop from the front or back

```
// global deque of integers
// will be initialized to be empty
deque<int> dq {};

int main() {
  dq.push_back(3);        // adds 3
  int val = dq.at(0);     // access index 0
  dq.pop_front()          // delete first element
  cout << val << endl;    // should print 3
}
```

Producer Consumer Example

- Does this work?

- Assume that two threads are created, one assigned to each function

```
deque<int> dq {};

void producer_thread() {
  while (true) {
    dq.push_back(long_computation());
  }
}

void consumer_thread() {
  while (true) {
    while (dq.size() == 0) {
      // do nothing
    }
    int val = dq.at(0);
    dq.pop_front();
    do_something(val);
  }
}
```

Ed Discussion

- Std::atomic question

- Followed by producer consumer

Ed Discussion

- How do we use mutex to fix this? To make sure that the threads access dq safely.
  - You are only allowed to add calls to the following:
    `mutex::lock`
    `mutex::unlock`
    `scoped_lock`
    `unique_lock`
    `unique_lock::unlock()`
    `unique_lock::lock()`
  - Can add other mutexes if needed
- Similar code: `no_sync.cpp`

```
deque<int> dq {};
std::mutex dq_lock;

void producer_thread() {
  while (true) {
    dq.push_back(long_computation());
  }
}

void consumer_thread() {
  while (true) {
    while (dq.size() == 0) {
      // do nothing
    }
    int val = dq.at(0);
    dq.pop_front();
    do_something(val);
  }
}
```

Any issue?

- The code is correct, but do we notice anything wrong with this code?

- The consumer code “busy waits” when there is nothing for it to consume.
  - It is particularly bad if we have multiple consumers, the locks make the busy waiting of the consumers sequential and use more CPU resources.

Thread Communication: Naïve Solution

- Consider the example where a thread must wait to be notified before it can print something out and terminate

- Possible solution: "Spinning"
  - Infinitely loop until the producer thread notifies that the consumer thread can print

- See `spinning.cpp`
  - The thread in the loop uses A LOT of cpu just checking until the value is safe
  - Use **`top`** to see CPU util

- Alternative: Condition variables

Condition Variables

- Variables that allow for a thread to wait until they are notified to resume

- Avoids waiting clock cycles “spinning”

- Done in the context of mutual exclusion
  - a thread must already have a lock, which it will temporarily release while waiting
  - Once notified, the thread will re-acquire a lock and resume execution

std::condition_variable

- A type defined in <condition_variable>

- ```
  void condition_variable::wait(unique_lock& ul);
  ```
  - Atomically releases the mutex and blocks on the condition variable. Once unblocked (by one of the functions below), function will return and calling thread will have the mutex locked

- ```
  void condition_variable::notify_one();
  ```
  - Unblock one of the threads waiting on this

- ```
  void condition_variable::notify_all();
  ```
  - Unblock all threads waiting on this
  ```

condition_variable Internal Pseudo-Code

- Here is some pseudo code to help understand condition variables

```
void condition_variable::wait(unique_lock& ul) {
  ul.unlock();
  sleep_on_cond(calling_thd); // sleep till woken up
  ul.lock();
}
```

```
void condition_variable::notify_one() {
  wakeup(this->asleep.front());
  this->asleep.pop_front();
}
```

```
void condition_variable::notify_all() {
  for (thd : this->asleep) {
    wakeup(thd);
  }
  this->asleep.clear();
}
```

Demo: `cond.cpp`

- See `cond.cpp`
  - Changes our spinning code to use a condition variable properly
  - No issues with cpu utilization!

That’s all for now!

- Next time:
  - Deadlock && Parallel Algorithms

- Next Week:
  - Intro to the Network & Socket programming

- Hopefully you are doing well ☺