Pthread Programming

Jinzhen Wang
jwang96@charlotte.edu

Department of Computer Science
UNC Charlotte

Learning Outcomes

At the end of this lecture, you will be able to

- Write a simple program that uses threads
- Give one example of data race
- Be able to achieve mutual exclusion
- Give one code example that deadlocks
- Name Coffman's four conditions for deadlocking
- Name one complex synchronization primitive

We run many programs on a computer concurrently

- What is running on your laptop?
- Many processes, many of which has spawned many logical threads
- Many more logical threads than cores

What is Thread?

- Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.
- The "procedure" that runs independently from its main program
  - Imagine a main program (a.out) that contains a number of procedures.
  - Imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system.
  - That would describe a "multi-threaded" program.

What is Thread in OS?

- First, UNIX Process
- A process is created by the
- operating system, and requires a fair amount of "resources"
- Processes contain information about program resources and program execution state, including
    - Process ID, process group ID, user ID, and group ID Environment
    - Working directory
    - Program instructions Registers
    - Stack, Heap
    - File descriptors Signal actions ...

What is Thread in OS?

- Threads is similar to Processes, but lighter and smaller
- They use and exist within these process resources;
- They are able to be scheduled by the operating system and run as independent entities
- The independent flow of control is accomplished because a thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data.

In short

- In UNIX environment, a thread:
  - Exists within a process and uses the process resources
  - Has its own independent flow of control
  - Duplicates only the essential resources it needs to be independently schedulable
  - May share the process resources with other threads that act equally independently
  - Dies if the parent process dies
  - Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

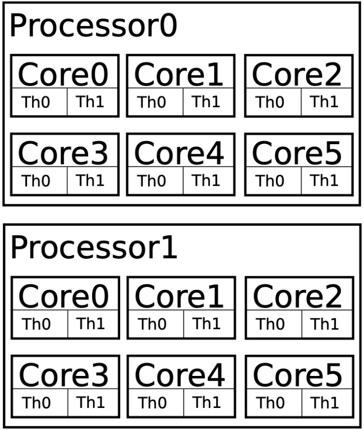
In short

- In UNIX environment, a thread:
  - Exists within a process and uses the process resources
  - Has its own independent flow of control
  - Duplicates only the essential resources it needs to be independently schedulable
  - May share the process resources with other threads that act equally independently
  - Dies if the parent process dies
  - Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.
- **Synchronization Issues**
- Because threads within the same process share resources:
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
  - Two pointers having the same value point to the same data.
  - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

The OS maps logical threads to execution contexts

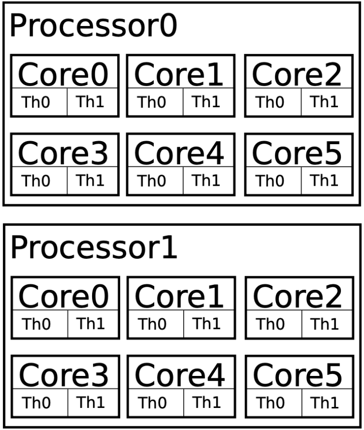
- Since there are more threads than execution contexts, the operating system must interleave execution of threads on the processor. Periodically... the OS will:
  - Interrupts the processor
  - Copies the register state of threads currently mapped execution contexts to OS data structures in memory
  - Copies the register state of other threads it now wants to run onto the processors execution context registers
  - Tell the processor to continue: now these logical threads are running on the processor

Interacting with OS



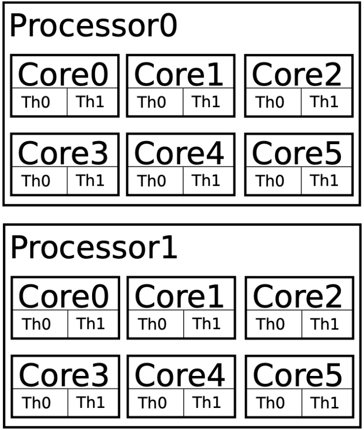
- Hardware
  - Threads in Operating System
  - Processors Cores
  - Physical threads

Interacting with OS



- Hardware
  - Threads in Operating System
  - Processors Cores
  - Physical threads
- OS mapping
  - The OS creates a kernel thread per physical thread
  - Posix threads are scheduled on kernel threads (with time sharing, context switching)

Interacting with OS



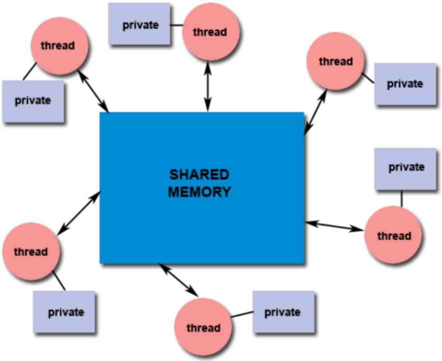
- Hardware
  - Threads in Operating System
  - Processors Cores
  - Physical threads
- OS mapping
  - The OS creates a kernel thread per physical thread
  - Posix threads are scheduled on kernel threads (with time sharing, context switching)
- Restricted mapping
  - pthread_setaffinity_np to restrict kernel threads mapping

How to program thread?

- POSIX Threads
  - is an execution model that exists independently from a language
  - It allows a program to control multiple different flows of work that overlap in time.
  - Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API.
  - Usually referred to as pthreads.
- pthreads defines a set of C programming language types, functions and constants.

Shared Memory Model

- pthreads follows a Shared Memory Model
  - All threads have access to the same global, shared memory
  - Threads also have their own private data
  - Programmers are responsible for synchronizing access (protecting) globally shared data.



Example

```c
#include <stdio.h>
#include <pthread.h>
void* f(void* p) {
  printf ("%s\n", p);
  return NULL;
}
int main () {
  pthread_t teach, student[50];
  char pm[] = "Hello, my name is Jon Wang.";
  char sm[] = "Hello Jon!";
  pthread_create(&teach, NULL, f, pm); //create a new thread
  pthread_join (teach, NULL); //wait for completion
  //create 50 threads
  for (int i=0; i < 50; ++i)
    pthread_create(&student[i], NULL, f, sm);
  //wait for the 50 threads to complete
  for (int i=0; i < 50; ++i)
    pthread_join(student[i], NULL);
return 0; }
```

The pthreads API

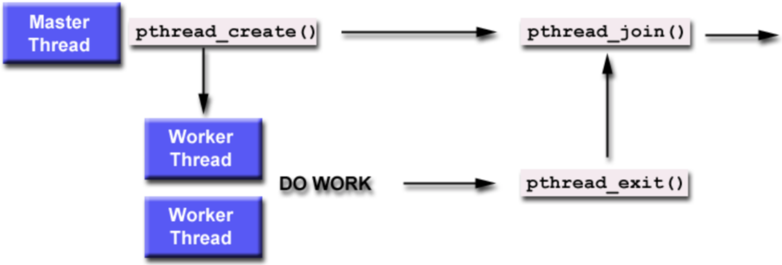
- pthreads API can be informally grouped into four major groups:
  - **Thread management:** Routines that work directly on threads - creating, detaching, joining, etc. They also include functions to set/query thread attributes (joinable, scheduling etc.)
  - **Mutexes:** Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provide for creating, destroying, locking and unlocking mutexes.
  - **Condition variables:** Routines that address communications between threads that share a mutex. Based upon programmer specified conditions. This group includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.
  - **Synchronization:** Routines that manage read/write locks and barriers.

Some Key APIs

- `pthread_create(thread, attr, func, args)`
  - Create a new thread to run func(args)
  - `pthread_exit` or `exit` for terminate the thread
  - On success, `pthread_create()` returns 0; on error, it returns an error number, and the contents of *thread are undefined.
- `pthread_join(thread, retval)`
  - The `pthread_join()` function waits for the thread specified by thread to terminate.
  - If that thread has already terminated, then it returns immediately.
  - On success, `pthread_join()` returns 0; on error, it returns an error number.

Fork/Join Model

- The `pthread_create()` forks new threads to run tasks
- The `pthread_join()` is one way to wait the tasks to finish



(Data) Race conditions

Race condition

- They happen when the timing of concurrent operations can make the program incorrect.
- Not only in shared memory programming, but also in distributed memory, or electronics.

Data race

- Race condition that happens in shared memory programming when two threads access the same variable with reads and write without being synchronized.

Example of data race

```
#include <stdio.h>
#include <pthread.h>
void* f(void* p) {
  int* val = (int*) p;
  for (int i=0; i< 100000; ++i)
    *val += 1;
  return NULL;
}
int main () {
  pthread_t th[50];
  int val = 0;
  for (int i=0; i < 50; ++i)
    pthread_create(&th[i], NULL, f, &val);
  for (int i=0; i < 50; ++i)
    pthread_join(th[i], NULL);
  //this usually does not print 5 000 000
  printf ("%d\n", val);
  return 0;
}
```

Mutexes can help prevent data race

Mutexes are used to protect shared resources

- Only one thread can hold the mutex at a time
- Trying to lock a mutex that is already locked pauses the thread
- If multiple threads wait on a mutex, any of them could be the next in line
- (Check variants in manual)

Mutex

```
//To initialize
pthread_mutex_t mut;
pthread_mutex_init (&mut, NULL);
std::stack<int> s;
//To access the stack
pthread_mutex_lock (&mut);
s.push(2);
pthread_mutex_unlock (&mut);
//To free the mutex
pthread_mutex_destroy (&mut);
```

Mutexes can help prevent data race

```
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t mut; //the software engineer in me cries
void* f(void* p) {
  int* val = (int*) p;
  for (int i=0; i< 100000; ++i) {
    pthread_mutex_lock(&mut);
    *val += 1;
    pthread_mutex_unlock(&mut);
  }
  return NULL;
}
int main () {
  pthread_t th[50]; int val = 0;
  pthread_mutex_init(&mut, NULL);
  for (int i=0; i < 50; ++i) pthread_create(&th[i], NULL, f, &val);
  for (int i=0; i < 50; ++i) pthread_join(th[i], NULL);
  pthread_mutex_destroy(&mut);
  printf ("%d\n", val);//this will print 5 000 000
  return 0;
}
```

Mutexes can cause Deadlocks

```
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t mut1, mut2;
void* f1(void* p) {
  int* val = (int*) p;
  for (int i=0; i< 100000; ++i) {
        pthread_mutex_lock (&mut1);
        pthread_mutex_lock (&mut2);
        *val += 1;
        pthread_mutex_unlock (&mut2);
        pthread_mutex_unlock (&mut1);
  }
  return NULL;
}
void* f2(void* p) {
  int* val = (int*) p;
  for (int i=0; i< 100000; ++i) {
        pthread_mutex_lock (&mut2);
        pthread_mutex_lock (&mut1);
        *val += 1;
        pthread_mutex_unlock (&mut1);
        pthread_mutex_unlock (&mut2);
  }
  return NULL;
}
```

- When in bad luck, it is possible that thread 1 takes mut1 and thread 2 takes mut2.
- Both threads are stuck waiting on the mutex held by the other thread.

Deadlock happens when all Coffman conditions are true

- In a 1971 paper, Coffman *et al.* showed that four conditions are necessary and sufficient for entering a deadlock:
  - **Mutual Exclusion:** Resources are held exclusively by a thread
  - **Hold and Wait:** Threads hold a resource and wait on another one
  - **No Preemption:** Resources can only be released by the thread that hold them
  - **Circular wait:** Threads are in a cycle where thread i waits on a resource held by $(i+1)\%n$

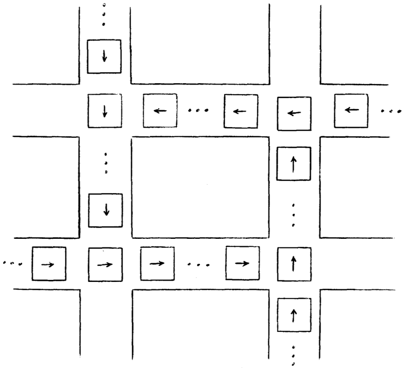


Fig. 1. Traffic deadlock.

E. G. Coffman, M. Elphick, and A. Shoshani. 1971. System Deadlocks. ACM Comput. Surv. 3, 2 (June 1971), 67–78. https://doi.org/10.1145/356586.356588

Common strategies to avoid deadlocks

Ordering locks

If locks are always taken in the same order, then the *Circular Wait* condition can not be true.

Backing off

If threads eventually back off after failing to hold a lock for some time, then the *Hold and Wait* condition can not be true.

Canceling Transactions

In relational databases, if two transaction write tables in different orders, one of the transaction might be canceled, reverting the changes caused by one. This makes the *No Preemption* condition false.

Two assignments

- Module 2 - HW1: Sep. 16th
- Module 2 - HW2: Sep. 23rd

Locking variants

Mutex

- The thread is unscheduled if the lock is not available. $\rightarrow$ Not continue to run until the mutex becomes available. Avoid busy-waiting (the thread repeatedly checks if the mutex is available)

Spinlock

- The thread enters a busy loop if the lock is not available. $\rightarrow$ Busy-waiting.

Mutex

- The thread sleeps if the lock is not available (no busy-waiting).
- **Used when the critical section might take time to execute.**
- Higher overhead due to context switching but no CPU wastage.

Spinlock

- The thread enters a busy-wait loop, consuming CPU cycles.
- **Used when the lock is expected to be held for a short time.**
- Low overhead for short critical sections; can waste CPU time if held too long.

Locking variants

Futex (Fast Userspace muTEXes)

- Spin lock for some time and then enter a kernel space wait. (This is what you actually get in Linux when using a mutex.)

FIFO locks

- Locks where the earliest thread to enter the lock is the first to be granted access to the resource.

Under the Hood

Spinlock

- Utilize the hardware-level atomic instructions: `test-and-set`.
- The threads that are trying to acquire the lock still take the CPU.

```
class SpinLock {
  private:
    int value=0; //0=Free;1=BUSY
  public:
    void acquire(){
      while (test_and_set(&value))
      ; //spin
    }
    void release(){
        value = 0;
        memory_barrier(); //synchronization
    }
}
```

Under the Hood

Mutex

- Often implemented as Multiple Processor Queueing Lock.
- Thread will try to acquire the lock, if fails, OS will move it to WAITING list.
- If the lock is released, OS will signal all threads in the waiting list.

```
class Lock{
  private:
  int value = FREE; SpinLock spinLock; Queue waiting;
  public:
    void acquire();
    void release();
  }
Lock::acquire(){
  spinLock.acquire();
  if (value != FREE){
    waiting.add(runningThread)
    scheduler.suspend(&spinLock)
  } else {
    value = BUSY;
    spinLock.release(); }
}
...
```

RW lock

Principle

- Consider the case where most of the threads will ever only read a shared array
- There is no reason to prevent them from reading concurrently.
- For writing, mutual exclusion is necessary.

API

- pthread_rwlock_init()
- pthread_rwlock_destroy()
- pthread_rwlock_rdlock()
- pthread_rwlock_wrlock()
- pthread_rwlock_unlock()

- The reader function uses `pthread_rwlock_rdlock()` to acquire the lock in read mode, meaning other readers can also hold the lock concurrently, but writers must wait.
- The writer function uses `pthread_rwlock_wrlock()` to acquire the lock in write mode, meaning no other thread (reader or writer) can access the shared data until this writer has finished.

Conditions

pthread_cond

Allows a thread to wait for a particular event to happen

- a queue to not be empty
- a queue to not be full
- ...

Usage

- Paired with a mutex
- pthread_cond_wait (cond, mutex);
  - waits on the condition to be signaled
  - and releases the mutex
  - takes the mutex back when the condition is signaled
- pthread_cond_signal (cond);
  - wakes one (any) of the waiting thread
- pthread_cond_broadcast (cond);
  - wakes all of the waiting thread

Playing ping-pong

```
void f1(std::mutex& mu,
        std::condition_variable_any& cond,
        bool& ping, bool& score) {
  unsigned int seed = 1;

  mu.lock();
  while (!score) {
    cond.wait(mu, [&]() {return ping;});

    if (!score){
      printf("ping\n");
      ping = !ping;

      if (rand_r(&seed) % 17 == 0) {
        std::cout<<"score 1"<<std::endl;
        score = true;
      }

      cond.notify_one();
    }
  }
  mu.unlock();
}
```

```
void f2(std::mutex& mu,
        std::condition_variable_any& cond,
        bool& ping, bool& score) {
  unsigned int seed = 2;

  mu.lock ();
  while (!score) {
    cond.wait(mu, [&](){return !ping;});

    if (!score){
      printf("pong\n");
      ping = !ping;

      if (rand_r(&seed) % 17 == 0) {
        std::cout<<"score 2"<<std::endl;
        score = true;
      }

      cond.notify_one ();
    }
  }
  mu.unlock();
}
```

External

pthreads:

- man -k pthread_
- D. Buttlar, J. Farrell, B. Nichols. Pthreads programming. O'Reilly. 1996
- POSIX.1-2001.
- A popular tutorial: `https://computing.llnl.gov/tutorials/pthreads/`

Deadlocks:

- E. G. Coffman Jr., M. J. Elphick, A. Shoshani. System Deadlocks. Computing Surveys 1971.

Relevant Wikipedia articles:

- `https://en.wikipedia.org/wiki/Race_condition`
- `https://en.wikipedia.org/wiki/Deadlock`
- `https://en.wikipedia.org/wiki/Synchronization_%28computer_science%29`
- `https://en.wikipedia.org/wiki/Reentrancy_%28computing%29`

Threading in C++:

- Since C++11: `http://www.cplusplus.com/reference/multithreading/`

Some other threading model:

- user-threading in Marcel `https://runtime.bordeaux.inria.fr/marcel/`