Threading in UNIX

Erik Saule
esaule@uncc.edu

Parallel Computing

09/12/2018

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

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

The assignment will ask you to

- Write a static loop based scheduler
- Write a dynamic loop based scheduler
- Show overhead associated with thread management and synchronization

Outline

How to make threads in UNIX?

In the olden times

- Threads are nothing else than processes that share memory
- So you could create a segment of shared memory with `shm_open`
- Then different processes can collaboratively work
- Mostly used to synchronize different programs nowadays

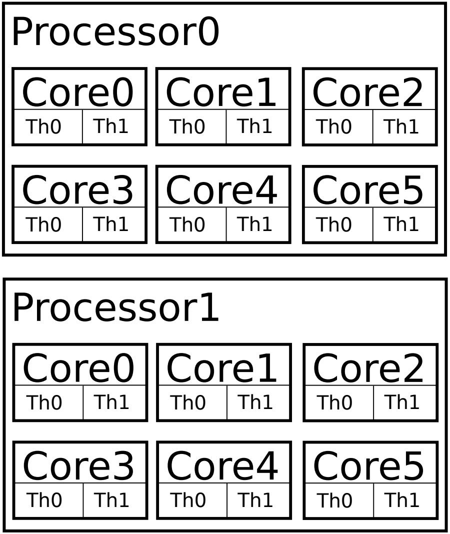
Threading libraries

- Most typical one is pthreads in UNIX
- Gives you different execution contexts within the same process
- Pretty much what you expect from threads
- (In Linux, they are implemented as different linked processes so they show up in `top` and `ps`)

Hello World!

```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 Erik.";
  char sm[] = "Hello Erik!";

  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;
}
```

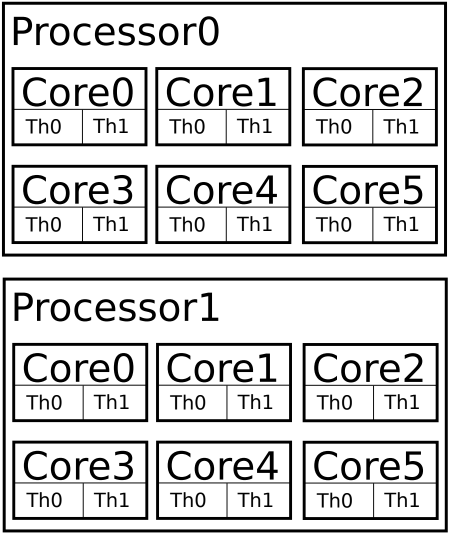
Interaction with the OS



Hardware

- Processors
- Cores
- Physical threads

Interaction with the OS



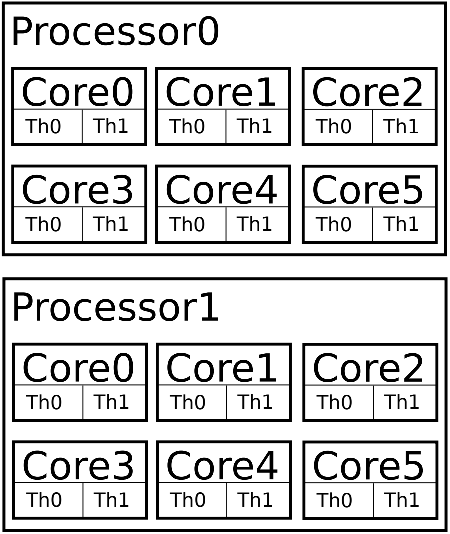
Hardware

- 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)

Interaction with the OS



Hardware

- 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

Outline

(Data) Race conditions

Race conditions

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.

Typical data race exemple

```c
#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;
}
```

Mutual exclusion

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);

- 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)

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);

  //this will print 5 000 000
  printf ("%d\n", val);
  return 0;
}
```

Mutexes can cause Deadlocks

```c
#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: Ressources 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 $(i+1)\%n$

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.

Thread safety and re-entrance

Thread safe

A function is thread safe if it can safely be called from multiple threads.

Re-entrance

A function is re-entrant if its execution can be interrupted, a different thread can execute the same function, and the original can be resumed safely.

Basically, if a function does not hold a global state, it is re-entrant.

Clearly a re-entrant function is thread-safe.

Thread safety and re-entrance

Thread safe

A function is thread safe if it can safely be called from multiple threads.

Re-entrance

A function is re-entrant if its execution can be interrupted, a different thread can execute the same function, and the original can be resumed safely.
Basically, if a function does not hold a global state, it is re-entrant.
Clearly a re-entrant function is thread-safe.

Not all library functions are thread safe. For instance, `rand` is not, but `rand_r` is re-entrant.

Outline

Assignment overview

Preliminary

Just a pthread hello world.
nbthreads threads which print "I am 1 of nbthreads".

Static loop scheduler

- Numerical integration (lock always vs lock once)
- Static loop scheduler. Each thread does $\frac{1}{nbthreads}$ of the iterations.
- Performance on cluster.
- Study granularity and overhead.

Dynamic loop scheduler

- Dynamic loop scheduler. Threads pick iterations in an FCFS way.
- Performance on cluster.
- Study granularity and overhead.

Static loop scheduler for numerical integration

Idea

The N iterations of the numerical integration can be done independently. Need to be careful about the reduction variable.

In practice, with $n = 100$ and $nbthreads = 10$

- Thread 0 takes loop iterations $[0; 10[$.
- Thread 1 takes loop iterations $[10; 20[$.
- ...
- be careful of $nbthreads > n$ or $n\%nbthreads! = 0$

Race condition

- `iteration` sync: the global sum variable is updated every iteration
- `thread` sync: each thread maintain its own sum variable and update the global sum only once.

Dynamic loop scheduler for numerical integration

Idea

Do not pre-split the work. Each thread does what it can.
When a thread needs work, it checks whether the computation has ended.
If there work left, it takes a chunk of `granularity` iterations and do them.

In practice, with $n = 80$ and $nbthreads = 3$, $granularity = 10$

- Thread 0 comes first and take loop iterations [0; 10[.
- Thread 2 comes next takes loop iterations [10; 20[.
- Thread 1 comes next takes loop iterations [20; 30[.
- Thread 2 comes next takes loop iterations [30; 40[.
- Thread 1 comes next takes loop iterations [50; 60[.
- Thread 0 comes next takes loop iterations [70; 80[.
- Thread 2 comes next and quits
- Thread 1 comes next and quits
- Thread 0 comes next and quits

Race condition

- add `chunk` sync. which commits to the global sum after each chunk

Outline

Locking variants

Mutex

Mutex are kernel space. The thread is unscheduled if the lock is not available.

Spinlock

Spinlock are userspace. The thread enters a busy loop if the lock is not available.

Futex

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.

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 ()

Check the man pages for details

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
- Note that there is no "counter", signal does nothing if no threads are waiting

Playing ping-pong

```c
pthread_mutex_t mut;
pthread_cond_t cond;
bool score, ping;

void* f1(void* p) {
  unsigned int seed = 1;

  pthread_mutex_lock (&mut);
  while (!score) {

    while (!ping) {
      pthread_cond_wait(&cond, &mut);
    }

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

      if (rand_r(&seed) % 17 == 0) {
        printf ("score 1\n");
        score = true;
      }

      pthread_cond_signal (&cond);
    }
  }
  pthread_mutex_unlock (&mut);
  return NULL;
}
```

```c
void* f2(void* p) {
  unsigned int seed = 2;

  pthread_mutex_lock (&mut);
  while (!score) {
    while (ping) {
      pthread_cond_wait(&cond, &mut);
    }

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

      if (rand_r(&seed) % 17 == 0) {
        printf ("score 2\n");
        score = true;
      }

      pthread_cond_signal (&cond);
    }
  }
  pthread_mutex_unlock (&mut);
  return NULL;
}
```

Outline

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/