Introduction to Pthreads

1. the POSIX threads programming interface
   - programming shared memory parallel computers
2. using Pthreads
   - our first program with Pthreads
   - attributes, Pthread creating and joining
3. The Work Crew Model
   - multiple threads cooperating on a task
   - sharing data between threads
   - implementing a critical section with `mutex`

MCS 572 Lecture 10
Introduction to Supercomputing
Jan Verschelde, 14 September 2016

Introduction to Pthreads

1. the POSIX threads programming interface
   - programming shared memory parallel computers
2. using Pthreads
   - our first program with Pthreads
   - attributes, Pthread creating and joining
3. The Work Crew Model
   - multiple threads cooperating on a task
   - sharing data between threads
   - implementing a critical section with `mutex`

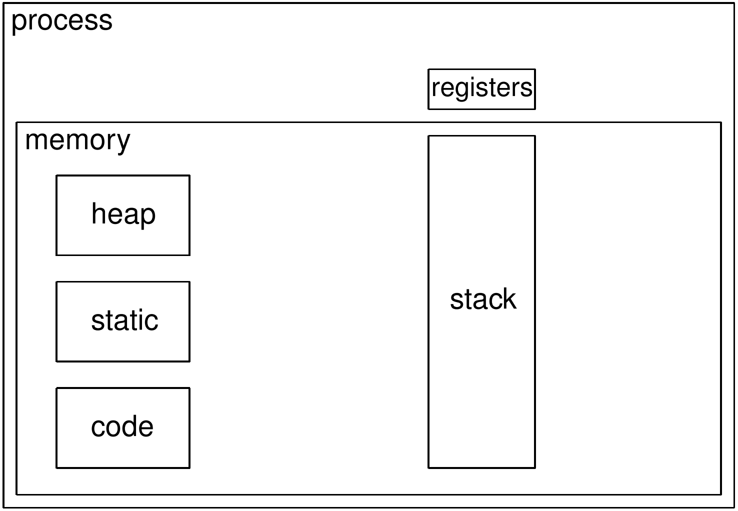
processes and threads

A thread is a single sequential flow within a process.

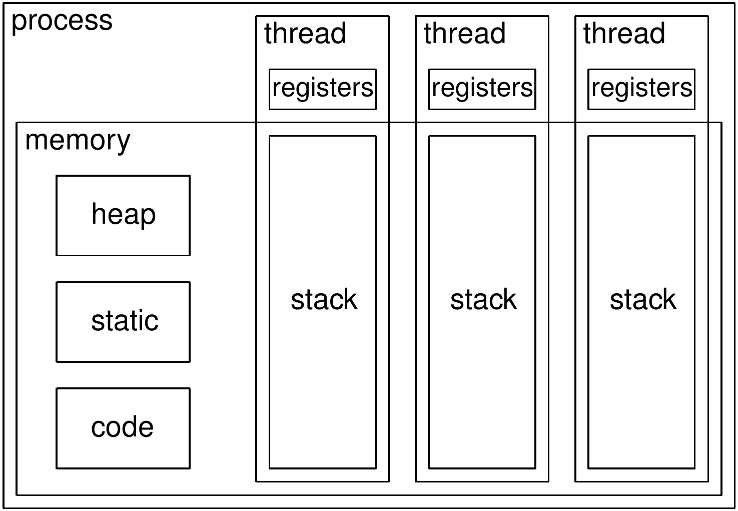
Multiple threads within one process share heap storage,
static storage, and code.
Each thread has its own registers and stack.

Threads share the same single address space and synchronization is needed when threads access same memory locations.

single threaded process



multithreaded process



processes and threads

A thread is a single sequential flow within a process.

Multiple threads within one process share

- heap storage, for dynamic allocation and deallocation,
- static storage, fixed space,
- code.

Each thread has its own registers and stack.

Difference between the stack and the heap:

- stack: Memory is allocated by reserving a block of fixed size on top of the stack. Deallocation is adjusting the pointer to the top.
- heap: Memory can be allocated at any time and of any size.

Threads share the same single address space and synchronization is needed when threads access same memory locations.

POSIX threads

For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard.

POSIX = Portable Operating System Interface

Implementations of this POSIX threads programming interface are referred to as POSIX threads, or Pthreads.

```
$ gcc -v
... output omitted ...
Thread model: posix
... output omitted ...
```

In a C program we just insert

```
#include <pthread.h>
```

and compilation may require the switch `-pthread`

```
$ gcc -pthread program.c
```

Introduction to Pthreads

the function each thread executes

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *say_hi ( void *args );
/*
 * Every thread executes say_hi.
 * The argument contains the thread id. */

int main ( int argc, char* argv[] ) { ... }

void *say_hi ( void *args )
{
   int *i = (int*) args;
   printf("hello world from thread %d!\n",*i);
   return NULL;
}
```

running `hello_pthreads`

```
$ make hello_pthreads
gcc -o /tmp/hello_pthreads hello_pthreads.c

$ /tmp/hello_pthreads
How many threads ? 5
creating 5 threads ...
waiting for threads to return ...
hello world from thread 0!
hello world from thread 2!
hello world from thread 3!
hello world from thread 1!
hello world from thread 4!
$
```

the main program

```
int main ( int argc, char* argv[] ) {
   printf("How many threads ? ");
   int n; scanf("%d",&n);
   {
      pthread_t t[n];
      pthread_attr_t a;
      int i,id[n];
      printf("creating %d threads ...\n",n);
      for(i=0; i<n; i++)
      {
         id[i] = i;
         pthread_attr_init(&a);
         pthread_create(&t[i],&a,say_hi,(void*)&id[i]);
      }
      printf("waiting for threads to return ...\n");
      for(i=0; i<n; i++) pthread_join(t[i],NULL);
   }
   return 0;
}
```

avoiding sharing of data between threads

To each thread we pass its unique identification label.
To `say_hi` we pass the address of the label.

With the array `id[n]` we have `n` distinct addresses:

```
pthread_t t[n];
pthread_attr_t a;
int i,id[n];
for(i=0; i<n; i++)
{
   id[i] = i;
   pthread_attr_init(&a);
   pthread_create(&t[i],&a,say_hi,(void*)&id[i]);
}
```

Passing `&i` instead of `&id[i]` gives to every thread the same address, and thus the same identification label.

Introduction to Pthreads

1. the POSIX threads programming interface
   - programming shared memory parallel computers
2. using Pthreads
   - our first program with Pthreads
   - attributes, Pthread creating and joining
3. The Work Crew Model
   - multiple threads cooperating on a task
   - sharing data between threads
   - implementing a critical section with `mutex`

using Pthreads in 3 steps

1. Declare threads of type `pthread_t` and attribute(s) of type `pthread_attri_t`.
2. Initialize the attribute `a` as `pthread_attr_init(&a);` and create the threads with `pthreads_create` providing
   1. the address of each thread,
   2. the address of an attribute,
   3. the function each thread executes, and
   4. an address with arguments for the function.

   Variables are shared between threads if the same address is passed as argument to the function the thread executes.
3. The creating thread waits for all threads to finish using `pthread_join`.

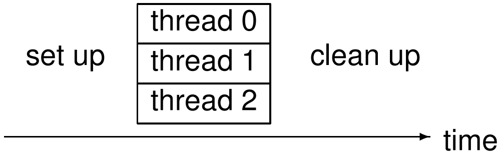
Introduction to Pthreads

1. the POSIX threads programming interface
   - programming shared memory parallel computers
2. using Pthreads
   - our first program with Pthreads
   - attributes, Pthread creating and joining
3. The Work Crew Model
   - multiple threads cooperating on a task
   - sharing data between threads
   - implementing a critical section with `mutex`

the work crew model

Instead of the manager/worker model where one node is responsible for the distribution of the jobs and the other nodes are workers, with threads we can apply a more collaborative model.

A task performed by three threads in a work crew model:



If the task is divided into many jobs stored in a queue, then the threads grab the next job, compute the job, and push the result onto another queue or data structure.

processing a queue of jobs

We will simulate a work crew model with Pthreads:

- Suppose we have a queue with n jobs.
- Each job has a certain work load (computational cost).
- There are t threads working on the n jobs.

A variable `nextjob` is an index to the next job.

In a critical section, each thread

- reads the current value of `nextjob`,
- increments the value of `nextjob` with one.

Introduction to Pthreads

1. the POSIX threads programming interface
   - programming shared memory parallel computers
2. using Pthreads
   - our first program with Pthreads
   - attributes, Pthread creating and joining
3. The Work Crew Model
   - multiple threads cooperating on a task
   - sharing data between threads
   - implementing a critical section with `mutex`

constant values and pointers

```
typedef struct
{
   int id;        /* identification label */
   int nb;        /* number of jobs */
   int *nextjob; /* index of next job */
   int *work;     /* array of nb jobs */
} jobqueue;
```

Every thread gets a job queue with two constants and two adrresses.
The constants are the identification number and the number of jobs.
The identification number labels the thread and is different for each thread, whereas the number of jobs is the same for each thread.

The two addresses are the index of the next job and the work array.
Because we pass the addresses to each thread,
each thread can change the data the addresses refer to.

generating n jobs

```
jobqueue *make_jobqueue ( int n )
{
   jobqueue *jobs;

   jobs = (jobqueue*) calloc(1,sizeof(jobqueue));
   jobs->nb = n;
   jobs->nextjob = (int*)calloc(1,sizeof(int));
   *(jobs->nextjob) = 0;
   jobs->work = (int*) calloc(n,sizeof(int));

   int i;
   for(i=0; i<n; i++)
      jobs->work[i] = 1 + rand() % 5;

   return jobs;
}
```

processing the jobs by n threads

```
int process_jobqueue ( jobqueue *jobs, int n )
{
   pthread_t t[n];
   pthread_attr_t a;
   jobqueue q[n];
   int i;
   printf("creating %d threads ...\n",n);
   for(i=0; i<n; i++)
   {
      q[i].nb = jobs->nb; q[i].id = i;
      q[i].nextjob = jobs->nextjob;
      q[i].work = jobs->work;
      pthread_attr_init(&a);
      pthread_create(&t[i],&a,do_job,(void*)&q[i]);
   }
   printf("waiting for threads to return ...\n");
   for(i=0; i<n; i++) pthread_join(t[i],NULL);
   return *(jobs->nextjob);
}
```

Introduction to Pthreads

running the processing of the job queue

```
$ /tmp/process_jobqueue
How many jobs ? 4
4 jobs :  3 5 4 4
How many threads ? 2
creating 2 threads ...
waiting for threads to return ...
thread 0 requests lock ...
thread 0 releases lock
thread 1 requests lock ...
thread 1 releases lock
*** thread 1 does job 1 ***
thread 1 sleeps 5 seconds
*** thread 0 does job 0 ***
thread 0 sleeps 3 seconds
thread 0 requests lock ...
```

session continued ...

```
thread 0 releases lock
*** thread 0 does job 2 ***
thread 0 sleeps 4 seconds
thread 1 requests lock ...
thread 1 releases lock
*** thread 1 does job 3 ***
thread 1 sleeps 4 seconds
thread 0 requests lock ...
thread 0 releases lock
thread 0 is finished
thread 1 requests lock ...
thread 1 releases lock
thread 1 is finished
done 4 jobs
4 jobs :  0 1 0 1
getafix:Lec10 jan$
```

using `mutex`

Three steps to use a `mutex` (mutual exclusion):

1. initialization

   ```
   pthread_mutex_t L = PTHREAD_MUTEX_INITIALIZER;
   ```

2. request a lock

   ```
   pthread_mutex_lock(&L);
   ```

3. release the lock

   ```
   pthread_mutex_unlock(&L);
   ```

the main function

```
pthread_mutex_t read_lock = PTHREAD_MUTEX_INITIALIZER;

int main ( int argc, char* argv[] )
{
   printf("How many jobs ? ");
   int njobs; scanf("%d",&njobs);
   jobqueue *jobs = make_jobqueue(njobs);
   if(v > 0) write_jobqueue(jobs);

   printf("How many threads ? ");
   int nthreads; scanf("%d",&nthreads);
   int done = process_jobqueue(jobs,nthreads);
   printf("done %d jobs\n",done);
   if(v>0) write_jobqueue(jobs);

   return 0;
}
```

the function `do_job`

```
void *do_job ( void *args )
{
   jobqueue *q = (jobqueue*) args;
   int dojob;
   do
   {
      dojob = -1;
      /* code omitted */
   } while (dojob != -1);

   if(v>0) printf("thread %d is finished\n",q->id);

   return NULL;
}
```

the do while loop

```
do
{
   dojob = -1;
   if(v > 0) printf("thread %d requests lock ...\n",q->id);
   pthread_mutex_lock(&read_lock);
   int *j = q->nextjob;
   if(*j < q->nb) dojob = (*j)++;
   if(v>0) printf("thread %d releases lock\n",q->id);
   pthread_mutex_unlock(&read_lock);
   if(dojob == -1) break;
   if(v>0) printf("*** thread %d does job %d ***\n",
                  q->id,dojob);
   int w = q->work[dojob];
   if(v>0) printf("thread %d sleeps %d seconds\n",q->id,w);
   q->work[dojob] = q->id; /* mark job with thread label */
   sleep(w);
} while (dojob != -1);
```

fine granularity

Pthreads allow for the finest granularity.

Applied to the computation of the Mandelbrot set:

- One job = the computation of the grayscale of one pixel, in a 5,000-by-5,000 matrix.
- The next job has number $n = 5,000 * i + j$, where $i = n/5,000$ and $j = n \bmod 5,000$.

The Dining Philosophers Problem

an example problem to illustrate synchronization

The problem setup, rules of the game:

1. Five philosophers are seated at a round table.
2. Each philosopher sits in front of a plate of food.
3. Between each plate is exactly one chop stick.
4. A philosopher thinks, eats, thinks, eats, ...
5. To start eating, every philosopher
    1. first picks up the left chop stick, and
    2. then picks up the right chop stick.

Why is there a problem?

The problem of the starving philosophers:

- every philosoper picks up the left chop stick, at the same time,
- there is no right chop stick left, every philosopher waits, ...

Bibliography

Online documents on Pthreads:

- Guide to the POSIX Threads Library, April 2001,
  by Compaq Computer Corporation, Houston Texas.
- Threading Programming Guide,
  Mac OS X Developer Library, 2010.

Summary + Exercises

In the book by Wilkinson and Allen, Pthread examples are in §8.7.2.

Exercises:

1. Modify the `hello world!` program with so that the master thread prompts the user for a name which is used in the greeting displayed by thread 5. Note that only one thread, the one with number 5, greets the user.
2. Consider the Monte Carlo simulations we have developed with MPI for the estimation of π. Write a version with Pthreads and examine the speedup.

two extra exercises

3. Consider the computation of the Mandelbrot set as implemented in the program `mandelbrot.c` of lecture 7. Write code for a work crew model of threads to compute the grayscales pixel by pixel. Compare the running time of your program using Pthreads with your MPI implementation.
4. Write a simulation for the dining philosophers problem. Could you observe starvation? Explain.