HPS

Vanessa End

POSIX Threads

Table of contents

1 Reminders on Shared Memory

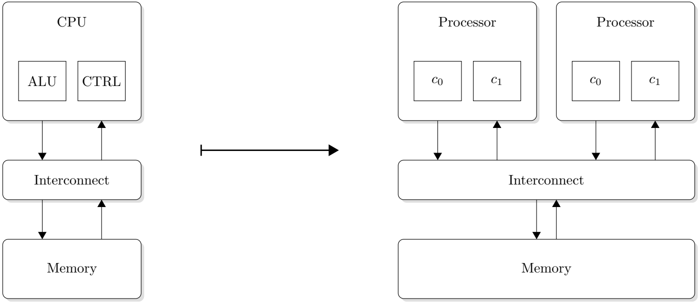
2 POSIX Threads

Learning Objectives

After this session, the participants should be able to

- compile and run a pthread program
- know how to spawn and join threads with pthreads
- know what a critical section is and how to handle it with mutexes and semaphores

Reminder



Many processors share the same memory $\rightarrow$ communication and coordination through memory.

Breakout 1: Shared Memory - 5 minutes

What needs to be kept in mind in shared memory programming?

What are POSIX threads

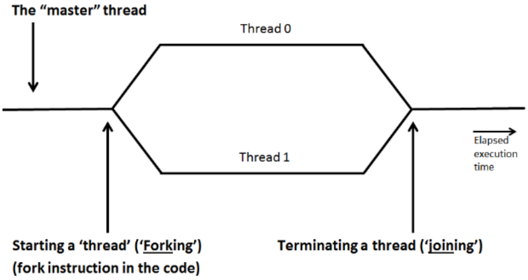
- standard for Unix-like operating systems
  - i.e.: Linux, MacOS, Solaris, ...
- library to be linked with C programs
- very low level programming
  - low overhead
  - not very user-friendly
- let’s you explicitely control threads with additional functions

Compiling and Running

- Include in source file: `#include <pthread.h>`
- Compile and link: `gcc -g -Wall -o pth_hello pth_hello.c -lpthread`
- Run: `./pth_hello <number of threads>`

Spawning/Forking and Joining threads

- explicitely spawn thread(s) with a given function func
  `pthread_create(&thread_handle, NULL, func, (void *)thread);`
- explicitely join threads once done
  `pthread_join(thread_handle, NULL);`



Breakout 2: Hello World - 10 minutes

1. Take a look at `pth_hello.c`.
   1. Identify the thread function. Where does the the function get the value of `thread_count` from?
   2. What is special about the variable `thread_count`?
2. Compile and run the program multiple times with different thread counts. What do you see?

Estimation of π

$$\pi = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots + (-1)^n \frac{1}{2n+1} + \ldots\right)$$

Single thread code

```
int n = 100;
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

Breakout 3: Estimation of π - 10 minutes

1. What steps do you need to take to parallelize the above code snippet with pthreads?

Breakout 4: Estimation of π 2/2 - 20 minutes

1. Try to parallelize the code snippet above yourself. Use `pth_pi_skeleton.c` for some guidelines if you do not want to try it all by yourself. If you do create the source code from scratch, please consider the following:
   - It should take the number of threads and n as input.
   - Add print statements in the thread function, which print the thread rank, the current value of the thread private sum (`my_sum`) and the current value of the global sum (`sum`).
   - Add a print statement for the global sum after joining the threads.
   - Add a print statement for the estimation of π.
2. Compile and run `pth_pi.c` or your program. Run it with: `./pth_pi <num threads> <n>`
   1. What changes, when you change the number of threads? E.g., try 1, 2, 4, 8.
   2. What changes, when you change n? E.g., try 8, 100, 200, 1000
   3. Run the program multiple times with 4 threads and $n = 100$. Is the output always the same?

Critical Sections.

In the last example you saw, that threaded programs have so called **critical sections**. Theses are sections, where multiple threads want to access the same variable.

```
void *Thread_sum(void *rank) {
  [...]
  sum += my_sum;

  printf("[%ld] my_sum: %f\n",my_rank, my_sum);
  printf("[%ld] sum: %f\n",my_rank, sum);
  [...]
```

This means we want to **sequentialize** the access to these variables.

Mutexes in pthreads

- Mutexes ensure **mut**ually **ex**clusive access to critical sections and are natively supported by Pthreads.
- Mutexes need to be initialized, can then lock and unlock a section and should be destroyed, once they are not needed anymore:

```
int pthread_mutex_init(pthread_mutex_t *mutex_p,
                      const pthread_mutexattr_t *attr_p);
int pthread_mutex_destroy(pthread_mutex_t *mutex_p);
int pthread_mutex_lock(pthread_mutex_t *mutex_p);
int pthread_mutex_unlock(pthread_mutex_t *mutex_p );
```

Steps to mutexify the estimation of *pi*.

What steps need to be taken to protect the critical section?

1. mutex as a global variable
2. initialize mutex in main function
3. identify code lines which need to be protected
4. lock mutex in the thread function before accessing critical code
5. unlock mutex in the thread function after accessing critical code
6. destroy mutex in main function

Breakout 5: Mutexify the estimation of π - 15 minutes

1. In the lecture, we discussed what steps need to be taken to protect the access to `sum` with mutexes. Now take the threaded estimation of π and add the mutex yourself. You can use `pth_pi_mutex_skeleton.c` for guidelines or use your code from above.
2. Compile and run `pth_pi_mutex.c` with different numbers of threads and different values of n. Run it with: `./pth_pi_mutex <num threads> <n>`
    1. Run the program multiple times with 4 threads and $n = 100$. Is the output always the same? What differences do you see compared to the version without mutexes?

Mutex Wrapup

So, what can mutexes do and what can't they do?

- they can serialize access to a critical section
- there is no way of ordering threads with one mutex
- you can run into a deadlock, if you do not unlock the mutex properly
  - this is also critical when using multiple mutexes!

Further Means of Access Restriction

- read/write-locks
  - part of the pthread interface
  - access control depending on whether variable is read or written to
- Semaphores
  - not part of pthreads $\rightarrow$ more details following
- Condition Variables and mutexes
  - used to save resources instead of blocking on a mutex
- Barriers
  - need to be implemented by the programmer
  - e.g., with busy-wait, condition variables or semaphores

Semaphores

A semaphore is a means for signalling, and not part of the Pthreads standard.

```c
#include <semaphore.h>

sem_t semaphore;
int initial_value;

int sem_init(&semaphore, 0, initial_value);
int sem_destroy(&semaphore);
int sem_post(&semaphore); //increments semaphore value
int sem_wait(&semaphore); //decrements semaphore value
int sem_getvalue(&semaphore, &value); //does not alter value
```

Especially useful in producer-consumer scenarios.

Producer-Consumer with Mutex vs Semaphore

- imagine the producer filling an array and the consumers wanting to do something with the contents
- the consumers need to know, when they can read from the array
- with a mutex, I can lock the complete array or design a node structure with one mutex per array entry
- with a semaphore, the consumers can take a value as long as the semaphore value is positive $\rightarrow$ more flexible and dynamic

Ordering access with semaphores

In some scenarios it might make sense to order access to a shared variable (i.e., non-commutative functions like matrix multiplication)

- using the value of the semaphore together with, e.g., the rank of a thread, access can be ordered
- see the optional exercise for more details on that

Questions and Further Reading

- `https://man7.org/linux/man-pages/man7/pthreads.7.html`
- `https://man7.org/linux/man-pages/man0/semaphore.h.0p.html`