Lab 3

Introduction to Programming Laboratory

Goals

- Pthread A+B
- Mutexes
- Job submission
- Counting CPUs
- Task: π approximation
- Further reading

Pthread A+B

Suppose we want to run a thread to caculate A+B and get the result from the thread...

pthread_create

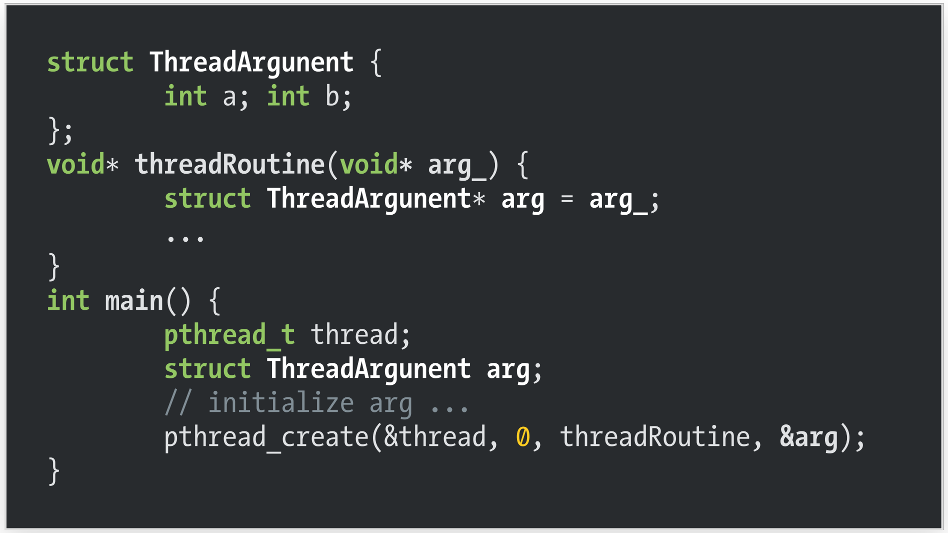
creates a thread

pthread_create

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`
- `thread` is the pointer to the pthread_t object
- `attr` is the options we want to set on the thread. We can pass `0` if we don't want any options
- `start_routine` is the function pointer for the thread to run
- `arg` is the argument to the `start_routine`

pthread_create

notice how the thread function gets the argument



```
struct ThreadArgunent {
        int a; int b;
};
void* threadRoutine(void* arg_) {
        struct ThreadArgunent* arg = arg_;
        ...
}
int main() {
        pthread_t thread;
        struct ThreadArgunent arg;
        // initialize arg ...
        pthread_create(&thread, 0, threadRoutine, &arg);
}
```

pthread_exit

exits a thread; we can also return data from the thread

pthread_join

waits for a thread to exit; can be used to retrieve the returned data from the thread

exit and join

```
void* threadRoutine(void* arg_) {
        ...
        int* c = malloc(sizeof(int));
        *c = 3;
        pthread_exit(c);
}
int main() {
        ...
        int* result;
        pthread_join(thread, (void**)&result);
}
```

Full example

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

struct ThreadArgunent {
    int a;
    int b;
};

void* threadRoutine(void* arg_) {
    struct ThreadArgunent* arg = (struct ThreadArgunent*)arg_;
    int* c = (int*)malloc(sizeof(int));
    *c = arg->a + arg->b;
    pthread_exit(c);
}

int main() {
    pthread_t thread;
    struct ThreadArgunent arg;
    arg.a = 1;
    arg.b = 2;
    pthread_create(&thread, 0, threadRoutine, &arg);
    int* result;
    pthread_join(thread, (void**)&result);
    printf("result: %d\n", *result);
```

Also at `/home/ipl19/x/lab3/aPlusB.c`

Compilation

Add `-pthread` to the flags of `gcc`/`g++`/`mpicc`/`mpicxx`.

Mutexes

Initialization

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

or...

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, 0);
```

Usage

```
pthread_mutex_lock(&mutex);
// critical section
pthread_mutex_unlock(&mutex);
```

Job submission

Add `-c#` to `srun`/`sbatch` flags, where `#` is the number of CPUs per process.

srun / sbatch flags review

- `-N`: number of nodes
- `-n`: number of processes
- `-c`: CPUs per process
- `-t`: time limit
- `-J`: name of job

Counting CPUs

Typically, we want to launch 1 thread for each CPU available.

Example

```c
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#include <sched.h>
#include <stdio.h>
int main() {
        cpu_set_t cpuset;
        sched_getaffinity(0, sizeof(cpuset), &cpuset);
        printf("%d cpus available\n", CPU_COUNT(&cpuset));
}
```

Also available at `/home/ipl19/x/lab3/cpus.c`

Playing with srun

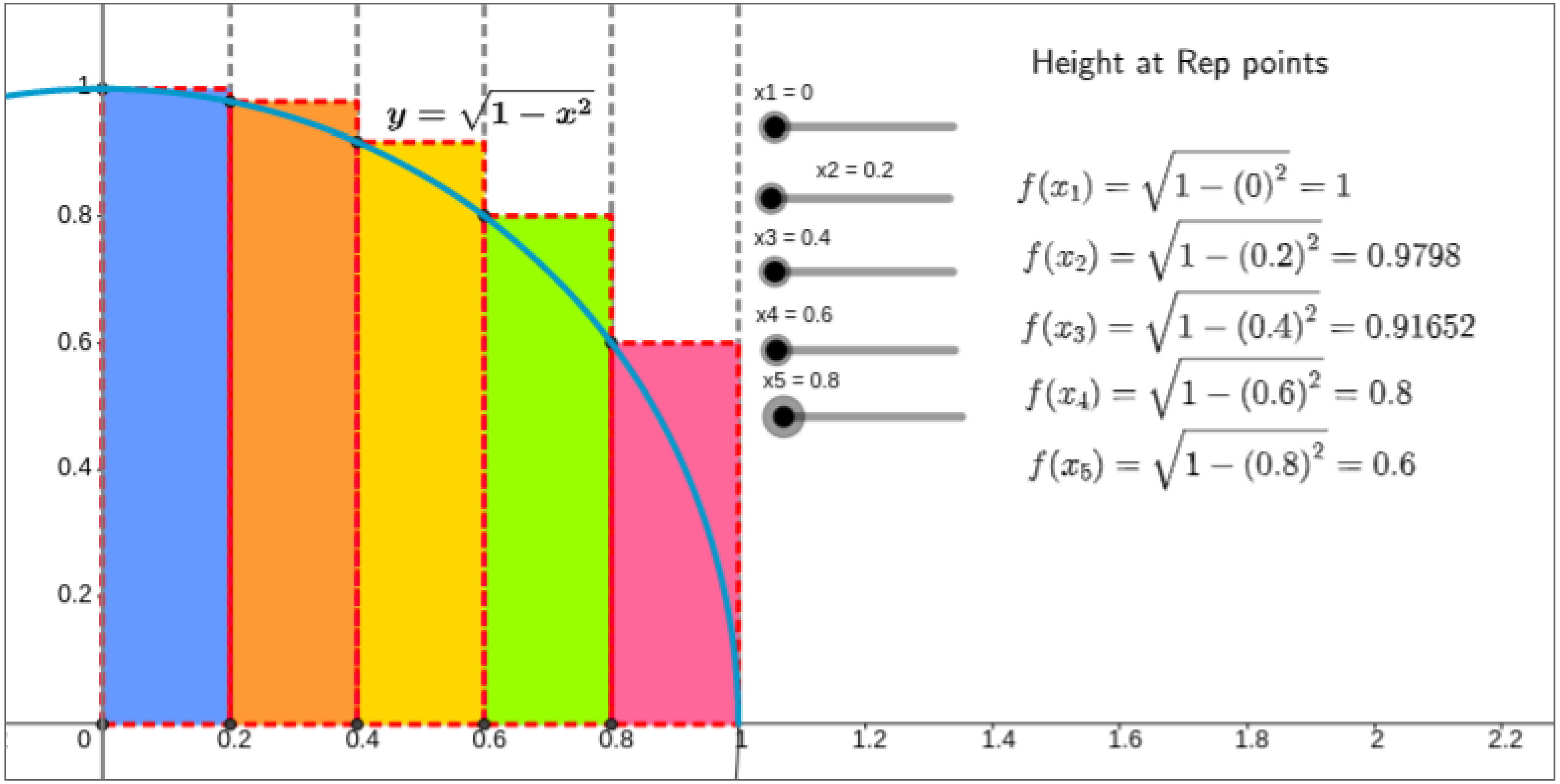
Task: π approximation

*You are required to demo to TA before leaving

Task description

Use the Left Riemann Sum to approximate the value of π.

$$4\sum_{i=0}^{k-1}\frac{\sqrt{1-(\frac{i}{k})^2}}{k}$$



Height at Rep points

$$f(x_1) = \sqrt{1 - (0)^2} = 1$$

$$f(x_2) = \sqrt{1 - (0.2)^2} = 0.9798$$

$$f(x_3) = \sqrt{1 - (0.4)^2} = 0.91652$$

$$f(x_4) = \sqrt{1 - (0.6)^2} = 0.8$$

$$f(x_5) = \sqrt{1 - (0.8)^2} = 0.6$$

Requirements

- `srun -c# ./lab3 SLICES`
- # = number of CPUs, SLICES = number of slices
- Launch # threads to parallelize computation
- Output your result with at least 6 digits in 1 line
- Name your source code `lab3.c` or `lab3.cc`
- Name your executable `lab3`
- Demo with TA

Further reading

std::thread equalviant of A+B

```cpp
#include <thread>
#include <iostream>
void threadRoutine(int a, int b, int* c) {
        *c = a + b;
}
int main() {
        int c;
        std::thread th(threadRoutine, 1, 2, &c);
        th.join();
        std::cout << "result: " << c << std::endl;
}
```

std::mutex equalviant of pthread_mutex_t

```
#include <mutex>

std::mutex mux;

void threadRoutine() {
        ...
        {
                std::lock_guard<std::mutex> g(mux);
                // critical section
        }
        ...
}
```