NORTHERN ARIZONA UNIVERSITY

School of Informatics, Computing, and Cyber Systems

Pthreads

CS450/CS550: Parallel Programming

Week 3

Software for macOS users

- By default, macOS has **clang** used as a C/C++ compiler
- Clang is consistent with **gcc** (GNU Compiler Collection), however, to ensure unequivocal results for your homework assignments, clang should be replaced with the Homebrew gcc:
  - install brew from the project web-site (https://brew.sh/)
  - install the Homebrew gcc (currently – its 13th version):
    `brew install gcc`
  - make an alias to the Homebrew gcc:
    `sudo ln -s $(which gcc-13) /usr/local/bin/gcc`

Basic definitions

- **Pthreads** (POSIX Threads) is a parallel execution model independent from a programming language
- **POSIX** (Portable Operating System Interface) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems
- **IEEE** (Institute of Electrical and Electronics Engineers) is a professional association for electronics and electrical engineering, and related disciplines
- Pthreads allows a program to control multiple flows of work that overlap in time. Each flow is referred to as a **thread**, and control over these flows is achieved by making calls to the Pthreads API
- **Pthreads API** was defined by the IEEE standard – POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)

Pthreads

- The purpose of using the POSIX thread library in computer programs is to execute software faster
- Pthreads API allows the spawn of a new concurrent process flow
- Threads are effective on multi-processor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing
- Threads require less overhead than forking or spawning a new process because the system does not initialize a new system virtual memory space and environment for the process
- All threads within a process share the same address space

NAU NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

Pthreads

- Pthread library contains more than 80 functions:
  - Thread management: create, exit, detach, join, …
  - Thread cancellation
  - Mutex locks: init, destroy, lock, unlock, …
  - Condition variables: init, destroy, wait, timed wait, …
  - Semaphores: init, post, wait, etc.
- Programs must include the file `pthread.h`, eventually – `semaphore.h` if semaphores are used
- Programs may need to be linked with the pthread library (`-lpthread`) in a makefile

Naming convention

- Types: `pthread[_object]_t`
- Functions: `pthread[_object]_action`
- Constants/Macros: `PTHREAD_CONST`
- Examples:
  - `pthread_t`: the type of a thread
  - `pthread_create()`: creates a thread
  - `pthread_mutex_t`: the type of a mutex lock
  - `pthread_mutex_lock()`: lock a mutex
  - `PTHREAD_CREATE_DETACHED`

Create a thread

- **`pthread_create()`** creates a new thread, it returns 0 if thread creation was successful, otherwise gives error code

```
int pthread_create(
pthread_t *thread,
pthread_attr_t *attr,
void * (start_routine) (void *),
void *arg);
```

- `thread`: outputs the Id of the new thread
- `attr`: input argument that specifies the attributes of the thread to be created (if NULL provided, default attributes are used)
- `start_routine`: a function to use as the start of the new thread; must have a prototype: `void * start_routine (void*)`
- `arg`: an argument to pass to the routine; if the routine requires multiple arguments, they must be passed in an array or a structure

Example 1: create a thread

pthread_join()

- causes the calling thread to wait for the thread `t` to terminate:

```
int pthread_join(pthread_t t, void **retval);
```

- `t`: input parameter, id of the thread to wait on
- `retval`: if not NULL, the `pthread_join` copies the exit status of the target thread into the location pointed to by that parameter
- on success, returns 0; otherwise, returns an error number
- multiple calls for the same thread are not allowed (the same thread cannot be joined again)

pthread_self() and pthread_equal()

- returns the thread Id for the calling thread:
  `pthread_t pthread_self(void)`
- this can be used to determine what thread is executing
- the Id of threads could also be used to assign different tasks to the created threads
- to test equality of threads, `pthread_equal()` is used:
  ```
  int pthread_equal(pthread_t id1,
                    pthread_t id2)
  ```
  the result is 0, if threads' ids are not equal

pthread_kill() and pthread_exit()

- `pthread_kill()` sends the signal `sig` to thread `t`, that is available in the same process as the caller:
  `int pthread_kill(pthread_t t, int sig);`
  - `sig`: signal number directed to thread
  - returns 0 to indicate success, otherwise – it returns an error code, and no signal is sent
- `pthread_exit()` function terminates the calling thread:
  `void pthread_exit(void *retval);`
  - returns a value via `retval` that is available to another thread in the same process that calls `pthread_join`
  - this function always succeeds (does not produce errors)

Example 2: calculating the sum

Comments on the code

- `pthread_create(&worker_thread,NULL,do_work,(void*)arg)`
  - the `do_work` function executes with the arguments given by `arg`
  - `(void*)arg` is a pointer, basically to anything
  - this syntaxis is used because the pthread library doesn't want to limit the type of data that can be accessed
- `void *do_work(void *arg)`
  - the `do_work()` return type is void and it's a pointer
  - it takes the argument passed to it by the `pthread_create`
  - it's a void pointer to make possible to point to any data type

Example 3: calculating the sum in parallel

Problems with multithreaded applications

- So far, we have seen that we can perform two tasks easily when they are independent
  - Summing an array with N threads can be done in parallel because the partial sums can be added together at the end
- However, what if the tasks are not independent?
  - For example, you want to sort an array in parallel?
  - How do you break up the array, give the work to the threads and recombine the result?
    - The position of each data element is related to the other elements

Example 4: race condition

Race conditions

- A **race condition** can arise in software when a computer program has multiple code paths that are executing at the same time
- There is a race condition if there is **at least one** execution path that leads to an incorrect outcome
- It doesn’t matter how unlikely that execution path is: if its probability is not zero, the program is **incorrect and needs to be fixed**
- In this course we’ll often look at code and then wonder: is there a race condition?

Mutex

- Pthreads have a **blocking lock** that provides **mut**ual **ex**clusion (mutex), it's used to prevent race conditions
- Lock creation:
```
int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutexattr_t *attr);
```
  - returns 0 on success, an error code otherwise
  - `mutex`: output parameter, lock
  - `attr`: lock attributes, If `attr` is `NULL`, the default mutex attributes are used

Locking and unlocking with mutex

- The mutex object referenced by mutex can be locked by calling the following function:
  `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- To unlock the mutex, the following function is used:
  `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- both functions return 0 on success, an error code otherwise
- `mutex`: input parameter (a lock)

Cleaning up memory

- Releasing memory for a mutex:

```
int pthread_mutex_destroy
(pthread_mutex_t *mutex);
```

- Releasing memory for a mutex attribute:

```
int pthread_mutexattr_destroy
(pthread_mutexattr_t *mutex);
```

Example 5: mutex

Condition variables

- Condition variables should be used as a place to wait and be notified
- They are not the condition itself and they are not events. The condition is contained in the surrounding programming logic
- The typical usage pattern of condition variables is following:

```
pthread_mutex_lock (&lock);
while (SOME-CONDITION is false)
    pthread_cond_wait(&cond, &lock);
do_stuff();
pthread_mutex_unlock (&lock);
```

Create and destroy conditions

- Create a condition variable:
  `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);`
- Destroy a condition variable:
  `int pthread_cond_destroy(pthread_cond_t *cond);`
  - returns 0 on success, an error code otherwise
  - `cond`: output parameter, condition
  - `attr`: input parameter, attributes (default is `NULL`)

Waiting and waking up for conditions

- Waiting on a condition:

```
int pthread_cond_wait(pthread_cond_t *cond,
pthread_mutex_t *mutex);
```

- Unblock at least one of the blocked threads:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Unblock all threads currently blocked by the condition variable:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

  - functions return 0 on success, an error code otherwise
  - `cond`: input parameter, condition
  - `mutex`: input parameter, an associated mutex

School of Informatics, Computing, and Cyber Systems

Example 6: condition variables

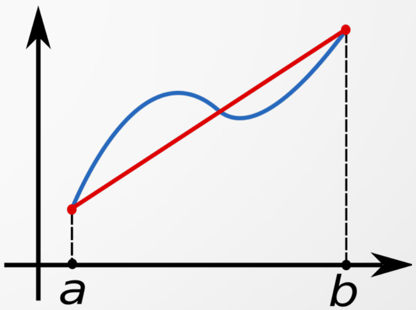
NAU NORTHERN ARIZONA UNIVERSITY
School of Informatics, Computing, and Cyber Systems

Assignment #1: trapezoidal rule

- **Trapezoidal rule** is a technique for approximating the definite integral:

  $\int_a^b f(x)dx \approx (b-a) \cdot \frac{1}{2} \cdot \big(f(a) + f(b)\big)$

  $\int_a^b f(x)dx \approx \frac{p}{2} \cdot \sum_{k=1}^{N-1}\big(f(x_k) + f(x_{k+1})\big)$

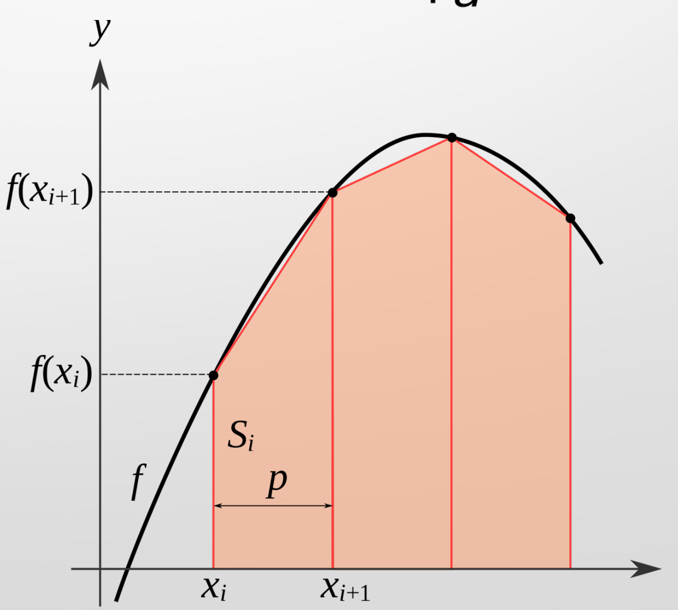


- **Trapezoid area**:

  $S = \frac{1}{2} \cdot h \cdot (b_1 + b_2)$

  b_1, b_2 – bases of the trapezoid

  h – altitude of the trapezoid



Assignment #1: standard normal distribution

Cumulative distribution function:

$$F(x) = \frac{1}{\sigma \cdot \sqrt{2\pi}} \cdot \int_{-\infty}^{x} e^{-\frac{1}{2}\cdot\left(\frac{x-\sigma}{\mu}\right)^2} dx$$

Probability density function:

$$f(x) = \frac{1}{\sigma \cdot \sqrt{2\pi}} \cdot exp\left[-\frac{1}{2} \cdot \left(\frac{x-\mu}{\sigma}\right)^2\right]$$

Standard distribution: $\mu = 0, \sigma = 1$