

Computer Science 335
Parallel Processing and HPC
Siena University
Fall 2026

Lab 9: Introduction to pthreads

Due: 10:00 AM, Friday, October 30, 2026

In this lab, we will return to the paradigm for parallelism you have all used with Java: multithreading. So far with our work in C, we have been working with message passing. In that case, each process has its own private address space. That is, each process has its own variables and any information to be exchanged among processes needs to be done with explicit message passing. Multithreading usually allows for the use of shared memory. Many operating systems provide support for threads, and a standard interface has been developed: *POSIX Threads* or *pthreads*.

You may work alone or with a partner on this lab.

Learning goals:

1. To learn the basics of POSIX thread creation and destruction in C

Getting Set Up

In Canvas, you will find a link to the starter GitHub repository for this lab, and you should fork that repository into your own GitHub account and make sure it is shared with your instructor. Only one member of the group should fork the repository on GitHub, then others should request a link to be granted write access.

You may answer the lab questions right in the `README.md` file of your repository, or use the `README.md` to provide a link to a Google document that has been shared with your instructor or the name of a PDF of your responses that you would upload to your repository.

Readings

Before starting, read through Pacheco Sections 4.1 and 4.2.

---

pthreads Basics

The basic idea is that we can create and destroy threads of execution in a program, on the fly, during its execution. These threads can then be executed in parallel by the operating system scheduler. If we have multiple processors, we should be able to achieve a speedup over the single-threaded equivalent.

We start with a look at a pthreads "Hello, world" program, which is in the `pthreadhello` directory of your repository for this lab.

The most basic functionality involves the creation and destruction of threads:

- `pthread_create(3THR)` – This creates a new thread. It takes 4 arguments. The first is a pointer to a variable of type `pthread_t`. Upon return, this contains a thread identifier that may be used later in a call to `pthread_join()`. The second is a pointer to a `pthread_attr_t` structure that specifies thread creation attributes. In the `pthreadhello` program, we pass in `NULL`, which will request the system default attributes. The third argument is a pointer to a function that will be called when the thread is started. This function must take a single parameter of type `void *` and return `void *`. The fourth parameter is the pointer that will be passed as the argument to the thread function.
- `pthread_exit(3THR)` – This causes the calling thread to exit. This is called implicitly if the thread function called during the thread creation returns. Its argument is a return status value, which can be retrieved by `pthread_join()`.
- `pthread_join(3THR)` – This causes the calling thread to block (wait) until the thread with the identifier passed as the first argument to `pthread_join()` has exited. The second argument is a pointer to a location where the return status passed to `pthread_exit()` can be stored. In the `pthreadhello` program, we pass in `NULL`, and hence ignore the value.

Prototypes for pthread functions are in `pthread.h` and programs need to link with `libpthread.a` (use `-lpthread` at link time).

Any global variables in your program are accessible to all threads. Local variables are directly accessible only to the thread in which they were created, though the memory can be shared by passing a pointer as part of the last argument to `pthread_create()`.

Practice Program: Create a new version of the `pthreadhello.c` program in a file `pthreadhello-more.c` in the `pthreadmore` directory of your repository that takes a command-line parameter that specifies the number of threads to create. This is very similar to what you see in Pacheco Program 4.1, except we will pass pointers to entries of an array of numbers 1 through n as the thread parameter as is done in the 2-thread version. (10 points)

A more interesting example is in `proctree_threads`.

This example builds a "tree" of threads to a depth given on the command line. It includes calls to `pthread_self()`, which returns the thread identifier of the calling thread.

Try it out and study the code to make sure you understand how it works.

Question 1: What is the output when you run the program for a tree of depth 3 on noreaster? (1 points)

Question 2: How many calls are made to the `split_proc` function for a tree of depth *n*? (3 points)

Question 3: What is the largest tree you can create on noreaster before you start to get thread creation errors? (1 point)

Multithreaded Pi

Practice Program: Write a pthread version of the Monte Carlo approximation of π. Call your program `pthread_pi.c` in the `pi` directory of your repository. See below for some tips. (20 points)

- Your program should take two command-line parameters: the number of threads and the number of random points to be generated by each of those threads.
- Store the number of points to generate in each thread in a global variable so your `main` function can set it, and your thread function can access it.
- Pass a pointer to an element of your array of per-thread "in circle" counts to each `pthread_create` call. That thread will be responsible for accumulating its own count in that variable. Then the main thread will have all of those values in its array when the threads have completed.

Submission

Commit and push!

Grading

This assignment will be graded out of 35 points.

Feature	Value	Score
`pthreadhello-more.c`	10	
Lab Question 1	1	
Lab Question 2	3	
Lab Question 3	1	
`pthread-pi.c`	20	
Total	35	