Computer Science 335
Parallel Processing and HPC
Siena College
Fall 2024

Lab 3: Processes and MPI Introduction

Due: 9:00 AM, Friday, September 27, 2024

This brief lab exercise will introduce you to computing with multiple processes. First, we will see an example using the Unix `fork` system call. Then we will run our first message passing programs.

You must work individually on this lab.

Learning goals:

1. To see the basics of how processes are created and can communicate with each other in a Unix environment.
2. To run an MPI application.

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `processes-lab-yourgitname`, for this lab.

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.

Introduction

The first mechanism that will will study in detail for introducing parallelism into our programs is to have multiple processes in execution that cooperate to solve a problem. Those processes will not share any memory – in fact, they will often be running on different physical pieces of hardware. When those processes need to communicate with each other (which they'll almost always need to do to perform a meaningful parallel computation), they will send messages to each other.

This approach is called the *message passing paradigm*. It is very flexible in that message passing programs can be executed by creating multiple processes on the same physical system (usually one with multiple processors/cores), or by creating them on different systems that can communicate across some network medium.

Important characteristics of the message passing paradigm include:

- **Locality** - each process accesses *only* its local memory.

- **Explicit parallelism** - messages are sent and received explicitly, so the programmer controls all parallelism. The compiler does not introduce any parallelism.
- **Cooperation** - for every message, the sending process and the receiving process must both participate for the communication to take place.

Creating Unix Processes

Programs running within a Unix-based operating system (OS) can use a *system call* named `fork()` to create new processes.

A system call is a C function provided by the OS that user programs can call to perform an operation that requires OS-level functionality. Creating and launching new processes is an example of such functionality.

The `fork` system call accomplishes this by duplicating the process that calls it. That is, the child process is a copy of the parent, and on return of the `fork` call, is in execution at the same point. which would be the statement immediately following the call to `fork`.

Its return value indicates whether the process is the child (if `fork` returned 0) or parent (if `fork` returned a positive integer). A -1 return indicates that the `fork` call failed, perhaps because a system limit was reached or because the calling process does not have permission to create new processes.

Typical usage follows a pattern such as:

```
pid=fork();
if (pid) {
  parent stuff;
}
else {
  child stuff;
}
```

A complete program that uses `fork` along with three other system calls (`wait`, `getpid`, and `getppid`) is in the `forking` example in your repository for this lab.

Compile and run the program on `noreaster.teresco.org`.

Question 1: How many times did the program print the output from line 20? Why? (2 points)

Question 2: How many times did the program print the output from line 28? Why? Which line was printed by the parent and which by the child? (3 points)

Question 3: Where did the number printed by line 42 come from? (1 point)

Processes created using `fork` do not share *context*, so the parent and child each have their own global space, call stack, and heap space. The OS provides mechanisms where special blocks of memory can be allocated and shared among processes running on the same system, but we will not use that approach here. Instead, we will rely on a form of message passing to communicate. Unlike programs that achieve parallelism with threads (such as the Java threads we looked at briefly and the C threads we will look at later this semester), cooperating processes that communicate with message passing could potentially be running on different systems.

There are two very low-level interprocess communication mechanisms provided by Unix systems.

- Two processes running on the same system can communicate through a named or an unnamed *pipe*.
- Two processes that might be running on different systems must communicate across a network, which is most commonly done using *sockets*.

Sockets and pipes provide only a very rudimentary interprocess communication (IPS). Each "message" sent through a pipe or across a socket has a unique sender and unique receiver and is really nothing more than a stream of bytes. The sender and receiver must add any structure to these communcations.

The program in the `sockets` directory of your repository is a very simplistic example of two processes that perform IPC over raw sockets. It is included mainly to show you that you don't want to be doing this if you can help it.

To run this program, use the `make` command to build the two executables: `client` and `server`. They'll both have to be run, so start two terminals. Pick a port number (something over 1000) that will be used so the sockets in each program can connect up. Suppose you pick 5423. In one terminal, start the server:

```
./server -p 5423
```

It will then be ready to accept connections through the socket on port 5423 from clients that connect on that same port.

In the other terminal, start the client:

```
./client -p 5423 -h localhost
```

You will see messages in both terminals. The client will prompt for what you want to do next. That will be specified by a single character. The client will send that character to the server over the socket using the `write` system call, and it will be read from the socket by the server using the `read` system call. Certain characters will cause additional information to be sent through the socket. 'a' sends an array of integer values. 'i' resends the identification string that was sent to the server when the client started. 'q' will cause the client to terminate.

These are processed by the `switch/case` constructs near the bottom of the code for the client and the server.

Practice Program: Add a new command to the client and have it handled by the server. The command should be triggered by the command 'r' and should send a number and a string, and in response the server should print the message that number of times. (10 points)

No one wants to program with raw sockets for anything beyond the simplest IPC tasks. Abstractions exist to support various kinds of tasks at a higher level. We will use one such abstraction: a message passing library.

Message Passing Libraries

Message passing is supported through a set of library routines. This allows programmers to avoid dealing with the hardware directly. Programmers want to concentrate on the problem they're trying to solve, not worrying about writing to special memory buffers or making TCP/IP calls or even creating sockets.

Many such libraries have existed such as P4, PVM, MPL, and Chameleon. The Message Passing Interface (MPI) has become an industry standard, under the guidance of the MPI Forum.

We will be looking at MPI in detail for the next few weeks. For today, you will be considering an MPI-based "Hello, World" program, provided in your repository in the `mpihello` directory.

On `noreaster.teresco.org`, compile `mpihello` by running the `make` command.

Question 4: What compiler command is used when you run `make` for the `mpihello` program? (1 point)

Notice that `make` uses this compiler command because MPI programs need to know about additional libraries, so most systems provide a different command that is able to find the extra MPI libraries.

You should now have an executable `mpihello`. Run it.

Question 5: What is the output? (1 point)

The standard command line, where you type the name of the program you wish to run, results in an MPI program that has a single process.

The mechanism to run an MPI program and launch multiple processes is somewhat system-dependent, but often involves a command such as `mpirun` or `mpiexec`. On noreaster, we will use `mpirun`. To run two processes:

```
mpirun -np 2 ./mpihello
```

Question 6: What is the output? If you run repeatedly, do you always get the same output? Why or why not? (2 points)

Now run with increasing powers of two for the number of processes, up to 128.

Question 7: About how long did it take to run the 128 process case? (2 points)

Question 8: What is the range of rank values for a run using a given number of processes? (1 point)

Notice that the first executable statement in an MPI program's `main` function is a call to the `MPI_Init` function, and the last is a call to the `MPI_Finalize` function. MPI programs should not have any executable code outside that block.

Question 9: Which symbols (functions, global variables, type definitions, etc.) that are used by this program are likely to be provided by `mpi.h`? (2 points)

Question 10: Briefly describe the function of each of the other MPI functions called in this example. (The `man` pages for these functions should be helpful here.) (6 points)

Practice Program: Add code to the `mpihello` example that prints a message "Hey, I'm the rank 0 process!" only on the process with rank 0, and a message "Wow, I'm the highest-ranked process!" on the process with the highest rank. (4 points)

Submission

Commit and push your code, and make sure your answers to the lab questions are in your `README.md`, in a document linked from there, or in a PDF document mentioned there.

Grading

This assignment will be graded out of 35 points.

Feature	Value	Score
Question 1	2	
Question 2	3	
Question 3	1	
sockets enhancements	10	
Question 4	1	
Question 5	1	
Question 6	2	
Question 7	2	
Question 8	1	
Question 9	2	
Question 10	6	
`mpihello` additions	4	
Total	35	