Lab 4: Multi-processing in Linux

Prof. Zichen Xu

Get Ready

- Find a zip file in our course page, called sources.tar
- Download and unzip it for this lab
- The purpose of the lab
  - Understanding the mechanism of multi-processing
  - Understanding the idea of process scheduling
  - Understanding multi-thread programming

The Fork Question

- Compile the c code called fork_problem.c
- Compile the c code called multi-fork.c
- The compilation shall be done by your OWN makefile
- Determine what will be output at the commented lines LINE X and LINE Y for fork_problem.c
- How many unique processes are created using the multi-fork?

Fork and fork more

- Complete the code for function fork1 in fork1.c
- Complete the code for function fork2 in fork2.c
- Compile both and run them
- Explain why they perform, differently.

Processes

- The Collatz conjecture is as follows

  Take any natural number n. If n is even, divide it by 2 producing n = n/2. If n is odd n = 3n + 1. If we repeat this process indefinitely, the conjecture is we ultimately arrive at 1. (Whether this is true for all numbers is unknown, but no counterexamples have been found.)

  For example, if n = 13, we get

  13, 40, 20, 10, 5, 16, 8, 4, 2, 1

- Write a C program that generates the Collatz sequence in the process. The initial number will be provided in the command line.

Processes (Cont.)

- You will write two versions of this program:

  (1) The first version outputs the sequence directly in process.

  (2) The second version has a child process overlay its address space (using the exec() system call) with the first program that generates the output. This will require the starting number to be passed on the command line. Please name this second program collatz.c Let's assume the first program is called lab1.c and is compiled to lab1 .

- Invoking exec(): There are many variations of the exec() command, with most differences related to arguments being passed to the command that is to run in the new process.
  - Imagine you had a program out.c and it was compiled to the executable out. This program is executed by entering

  ./out <integer value>
  - The program example.c (compiled to example) invokes out using the execlp() system call and is invoked as follows

  ./example 55

Threads

- In this part, we involve creating a child thread that generates the Collatz sequence, writing it to global data it shares with the parent. When the child thread terminates, the parent will output the sequence.

- Your program will run similarly to the previous labs where you will pass the original number on the command line, such as

  ./a.out 13

Threads (Cont.)

- You have some interesting choices for a data structure to store the sequence. One approach is to simply use an array of integer values to store the numbers in the sequence.

  For example, you may create a global array of the following capacity:

```
const int SIZE = 25;
int sequence[SIZE];
```

- The obvious issue with this approach is that static allocation may be either too large (i.e. you have created an array much larger than necessary) or too small (i.e. you have created an array that is too small to store the sequence.)

- A better strategy is to instead use a linked list.

Threads (Cont.)

- Using list.c for an example to see how to use linked list in C
- Using POSIX program thrd-posix.c to see how to use POSIX thread

- construct a multithreaded program, named collatz-thrd.c, that has the child thread construct the linked list, and populate the list using the Collatz sequence. When the child thread has terminated, the parent will output the sequence by traversing the list.

Lab submission

- A PDF file which is your lab4 report

- A zip file contains the following codes:
  - collatz.c, for the direct outputting the collatz sequence
  - collatz2.c, for using execlp() to invoke certain binaries to print the collatz sequence
  - and collatz-thrd.c, which is the thread version