Problem Set 2: Processes & Threads[1]

1. Short Answer Questions

Write brief answers to the following questions in a text file named `short_answer.txt`.

1. How does having kernel mode and user mode hardware enable the operating system to protect itself from users and users each others?
2. For each of the following, briefly explain why it should (or should not) be privileged.
   a) set value of a timer
   b) read the clock
   c) issue a trap instruction
   d) switch from user to kernel mode
   e) switch from kernel to user mode
3. What is the purpose of an interrupt? How is this different than a trap?
4. What is a system call? And how do we typically access system calls?
5. Including the original (parent) process, how many processes total are created by the following?

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{
      fork();
      fork();
      fork();
      return 0;
};
```

6. Consider a program (process A) that calls fork() and creates process B which does **not** call exec(). Explain the effect on process A for each of the following. (State the effect **and** explain why).
   a) B changes the value of a global variable.
   b) B changes the value of a local variable.
   c) B crashes due to dereferencing a NULL pointer.
7. Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be more than the number of processors in the system. Discuss the performance implications of the following scenarios.
   a) The number of kernel threads allocated to the program is less than the number of processors
   b) The number of kernel threads allocated to the program is equal to the number of processors.
   c) The number of kernel threads allocated to the program is greater than the number of processors but less than the number of user-level threads.

2. Coding Questions

For the following two sub-questions, submit your .c files, **plus** a capture of each program's output running with the **argument value as 23**.

[1]Created by Mohamed Hefeeda, modified by Brian Fraser. Some parts provided by Prof. Wei Tsang Ooi of National University of Singapore. Some questions are from Operating Systems Concepts (OSC) by Silberschatz, Galvin and Gagne.

2.1 Processes

Collatz conjecture defines the series:

$$n_{i+1} = \begin{cases} n_i \div 2 & \text{if } n_i \text{ is even} \\ (3 \times n_i) + 1 & \text{if } n_i \text{ is odd} \end{cases}$$

which is conjectured to always eventually reach the number 1.

- Example: $n_0 = 35$:
  35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1
- Write a C program named `collatz.c` which:
  - Accepts n_0 as a command line argument.
    - Add error checking to ensure parameter is a positive (>= 1) integer.
  - Use the `fork()` system call to spawn a child process which computes the sequence of numbers and prints the sequence to the screen and exits.
  - After spawning the new process with `fork()`, have the initial process wait until the second process completes.

2.2 Threads

Write a multi-threaded Linux program named `fibonacci.c` which accepts a single integer parameter from the command line (*N*), and then prints out the first *N* Fibonacci numbers to the screen.

- You must launch one thread which computes all *N* Fibonacci numbers, storing them into a data structure accessible by the main thread.
  - Hint: a dynamically allocated array is simple! Declare a global variable that is a pointer to `int`s. In main(), dynamically allocate enough space to hold the N values. Have your thread function then use the global variable (pointer to the `int`s) to store the values.
  - You could also have the thread function dynamically allocate the space and then return the pointer via `pthread_exit()`. See tutorial video posted to YouTube on dynamic memory allocation and threads for more.
- After launching the calculating thread, the main thread must wait until the calculating thread completes.
- Once the calculating thread completes, the main thread writes the results to the screen.
  - The calculating thread **must not** write any text to the screen.

3. Measuring System Call Overhead

In this question, you will run a little experiment to measure the overhead of system calls. We will focus on two particular calls: `fork()`, which is supposed to be expensive, and `getpid()`, which is one of the simplest system calls available.

- Use the following source code and create a file named `measureSystemCall.c`:

```c
#include <time.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
/* Your Answers Here

*/
double timespec_to_ms(struct timespec *ts)
{
        return ts->tv_sec*1000.0 + ts->tv_nsec/1000000.0;
}

int main()
{
        struct timespec start_time, end_time;

        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start_time);
        /* begin timing */

        /* end timing */
        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end_time);
        printf("%f ms\n", timespec_to_ms(&end_time)
                - timespec_to_ms(&start_time));

}
```

The code above measures the CPU time spent in the process executing the code marked between `/* start timing */` and `/* end timing */`.

To compile the code, use the following command:

```
gcc -O0 measureSystemCall.c  -o measureSystemCall  -lrt
```

The flag `-lrt` links the executable binary to the library `librt.a`, which provides the implementation of function `clock_gettime()`. The flag `-O0` (that's a big oh character, followed by a zero) tells the compiler not to optimize our code.

In the comment block near the top write your answers to the questions below. Record all time values to one decimal place, such as “10.3”.

1. First setup a loop that executes 1,000,000 times without calling any functions. This will time how long it takes to execute just the loop that many times.

   a) Write a brief description of the system you are running on (VM? Laptop? Server?...)
   b) Record the total time you measured.
   c) What is the average overhead per pass through the loop?

   - Note that 1ms = 1,000,000ns. Hint: Makes the math easy when looping 1,000,000 times.

2. Now measure the time it takes to execute the function `getpid()`, which returns the value of the current process ID.

   - Modify the program above to measure the total CPU time spent calling `getpid()` 1,000,000 times.
   - On Linux, the implementation of `getpid()` does not result in a system call. The function caches the process ID, so that repeated calls to `getpid()` do not result in expensive system calls. Thus, what you measured is basically the cost of calling a function.

a) Record the total time you measured.
b) What is the average time per pass through the loop (in ns) when calling a function?
(Just divide the total by the number of iterations)

3. Modify the program above to measure the total CPU time spent calling `syscall(SYS_getpid)` 1,000,000 times. Take note of the time (in ms) printed by the program. Note that your program will give you different answers every time your run your program, but they should be in the same order of magnitude.

   a) Record the total time you measured.
   b) What is the average time per pass through the loop (in ns) when making a system call?

4. Modify the program above to measure the total CPU time spent executing the following line of code **1,000 times** (not a million!). If when you run this you see the error message, ensure you reduced the number of times to 1,000. If it still fails, try running it 100 times.

```
int pid = fork();
if (pid == 0) {
      return 0;
}
if (pid < 0) {
      printf("ERROR: Fork failed.\n");
      errorCount++;
}
wait(0);
```

   a) Record the total time you measured.
   b) What is the average time per pass through the loop (in ns) when calling both `fork()` and `wait()`?
   c) Explain what happens in terms of the parent process (system calls executed) and child process each time the above code is run.

Note: You should never do the following:

```
for (i = 0; i < N; i++) {
      fork();
}
```

which will attempt to create 2^N processes.

4. Deliverables

Submit the following deliverabels in a `ZIP` or `tar.gz` file to CourSys:

1. `Ps2.txt: which contains all written question parts(short answer + coding).`
2. `collatz.c`
3. `fibonacci.c`
4. `measureSystemCall.c`

You may include a Makefile if you like. Please remember that all submissions will automatically be compared for unexplainable similarities.