

Important: Please do all projects on `opsys`

Adding functionality to our system

Purpose

In the previous project we created a utility that just launched user processes with various limits. In this project we will be adding functionality that we will use in later on projects.

Task

As in the previous project, you will have two executables, `oss` and `worker`. The `oss` executable will be launching workers, who stick around for a bit. However, we have a few more details, the most important of which is that our processes stick around based on a simulated clock that we create, NOT system time. **IMPORTANT:** Your code should not call `sleep` or `usleep` or any related function that waits based on some amount of system time. If you do so, your project will be penalized 50%.

Our main executable (`oss`) will now be maintaining a "simulated system clock" in shared memory. This system clock is not tied to the main clock of the system, but instead done separately. The clock consists of two separate integers (one storing seconds, the other nanoseconds) in shared memory, both of which are initialized to zero. This system clock must be accessible by the children, so it is required to be in shared memory. The children will not be modifying this clock for this assignment, but they will need to look at it.

In addition to this, `oss` will also maintain a process table (consisting of Process Control Blocks, one for each process). This process table does not need to be in shared memory. The first thing it should keep track of is the PID of the child process, as well as the time right before `oss` does a fork to launch that child process (based on our own simulated clock). It should also contain an entry for if this entry in the process table is empty (ie: not being used). I suggest making your process table an array of structs of PCBs, for example:

```
struct PCB {
    int occupied;           // either true or false
    pid_t pid;              // process id of this child
    int startSeconds;       // time when it was forked
    int startNano;          // time when it was forked
};

struct PCB processTable[20];
```

Note that your process table is limited in size and so should reuse old entries. That is, if the process occupying entry 0 finishes, then the next time you go to launch a process, it should go in that spot. When `oss` detects that a process has terminated, it can clear out that entry and set its `occupied` to 0, indicating that it is now a free slot.

worker, the children

The worker takes in two command line arguments, this time corresponding to the maximum time it should decide to stay around in the system. For example, if you were running it directly you might call it like:

```
./worker 5 500000
```

It should first output what it was called with as well as its PID and PPID.

```
Worker starting, PID:6577 PPID:6576
Called with:
Interval: 5 seconds, 500000 nanoseconds
```

The worker will then attach to shared memory and examine our simulated system clock. It will then figure out what time it should terminate by adding up the system clock time and the time passed to it (in our simulated system clock, not actual time). This is when the process should decide to leave the system and terminate.

For example, if the system clock was showing 6 seconds and 100 nanoseconds and the worker was passed 5 and 500000 as above, the target time to terminate in the system would be 11 seconds and 500100 nanoseconds. The worker will then go into a loop, constantly checking the system clock to see if this time has passed. If it ever looks at the system clock and sees values over the ones when it should terminate, it should output some information and then terminate.

So what output should the worker send? Upon starting up, it should output the PID, its PPID, the system clock and when it will terminate. This should be in the following format:

```
WORKER PID:6577 PPID:6576
SysClockS: 5 SysclockNano: 1000 TermTimeS: 11 TermTimeNano: 500100
--Just Starting
```

The worker should then go into a loop, checking for its time to expire (IT SHOULD NOT DO A SLEEP). It should also do some periodic output. Every time it notices that the seconds have changed, it should output a message like:

```
WORKER PID:6577 PPID:6576
SysClockS: 6 SysclockNano: 45000000 TermTimeS: 11 TermTimeNano: 500100
--1 seconds have passed since starting
```

and then one second later it would output:

```
WORKER PID:6577 PPID:6576
SysClockS: 7 SysclockNano: 500000 TermTimeS: 11 TermTimeNano: 500100
--2 seconds have passed since starting
```

Once its time has elapsed, it would send out one final message:

```
WORKER PID:6577 PPID:6576
SysClockS: 11 SysclockNano: 700000 TermTimeS: 11 TermTimeNano: 500100
--Terminating
```

oss, the parent

The task of oss is to launch a certain number of worker processes with particular parameters. These numbers are determined by its own command line arguments.

Your solution will be invoked using the following command:

```
oss [-h] [-n proc] [-s simul] [-t timelimitForChildren] [-i intervalInMsToLaunchChildren]
```

At the start of oss, print out the parameters passed to it.

```
OSS starting, PID:6577 PPID:6576
Called with:
-n 5
-s 3
-t 4.5
-i 0.2
```

While the first two parameters are similar to the previous project, the `-t` parameter is slightly different. It now stands for the amount of simulated time, not real time, that should pass before it terminates. It can also be a float value, for example it could be called with 1.5, so a process could run for 1.5 seconds.

The `-i` parameter is new. This specifies a minimum interval between when you launch child processes, which is specified in a fraction of a second. For example, if `-i 0.1` is passed, then your system would launch a child no more than once every 100 milliseconds. This is set to allow user processes to slowly go into the system, rather than flooding in initially. Make sure you read this value as a float.

When started, oss will initialize the system clock and then go into a loop and start doing a `fork()` and then an `exec()` call to launch worker processes. However, it should only do this up to simul number of times. So if called with a `-s` of 3, we would launch no more than 3 initially. oss should make sure to update the process table with information as it is launching user processes.

This seems close to what we did before, however, will not be doing `wait()` calls as before. Instead, `oss()` will be going into a loop, incrementing the clock and then constantly checking to see if a child has terminated. Rough pseudocode for this loop is below:

```
while (stillChildrenToLaunch || haveChildrenInSystem) {

    incrementClock();

    Every half a second of simulated clock time, output the process table to the screen

    checkIfChildHasTerminated();

    if (childHasTerminated) {
        updatePCBOfTerminatedChild;
    }

    possiblyLaunchNewChild(obeying process limits and time bound limits)

}
```

Output that OSS has terminated with information

The check to see if a child has terminated should be done with a nonblocking `wait()` call. This can be done with code along the lines of:

```
int pid = waitpid(-1, &status, WNOHANG);
if (pid > 0) {
    // process with PID of pid has terminated
}
```

`waitpid` will return 0 if no child processes have terminated and will return the pid of the child if one has terminated. This is nonblocking, so if no child has terminated it will go right by this call. Note that you will be looping over this many thousands of times in your project, so do not add normal output in this loop each time or you will be spammed.

The output of oss instead should consist of, every half a second in our simulated system, outputting the entire process table in a nice format. For example:

```
OSS PID:6576 SysClockS: 7 SysclockNano: 500000
Process Table:
Entry Occupied PID StartS StartN
0      1      6577 5      500000
1      0      0      0      0
2      0      0      0      0
...
19     0      0      0      0
```

OSS should only terminate after all workers have finished. When OSS terminates, it should output a report on system information, reporting how many processes ran in the system and how long in total the processes ran for.

```
OSS PID:6576 Terminating
5 workers were launched and terminated
Workers ran for a combined time of 20 seconds 500000 nanoseconds.
```

Incrementing the clock

Each iteration in oss you need to increment the clock. So how much should you increment it? You should attempt to very loosely have your internal clock be similar to the real clock. This does not have to be precise and does not need to be checked, just use it as a crude guideline. So if you notice that your internal clock is much slower than real time, increase your increment. If it is moving much faster, decrease your increment. Keep in mind that this will change based on server load possibly, so do not worry about if it is off sometimes and on other times.

Clock race conditions

We are not doing explicit guarding against race conditions. As your child processes are examining the clock, it is possible that the values they see will be slightly off sometimes. This should not be a problem, as the only result would be a clock value that was incorrect very briefly. This could cause the child process to possibly end early, but I will be able to see that as long as your child processes are outputting the clock.

Signal Handling

In addition, I expect your program to terminate after no more than 60 REAL LIFE seconds. This can be done using a timeout signal, at which point it should kill all currently running child processes and terminate. It should also catch the `ctrl-c` signal, free up shared memory and then terminate all children.

This can be implemented by having your code send a termination signal after 60 seconds. You can find an example of how to do this in our textbook in a source file called `periodicasterik.c`

Implementation details

It is required for this project that you use version control (git), a `Makefile`, and a `README`. Your `README` file should consist, at a minimum, of a description of how to compile and run your project, any outstanding problems that it still has, and any problems you encountered.

Your makefile should also compile BOTH executables every time. This requires the use of the `all` prefix.

As we are using shared memory, make sure to check and clear out shared memory if you have had errors. Please check the `ipcrm` and `ipcs` commands for this.

Suggested implementation steps

1. Set up the source files and Makefiles [Day 1]
2. Have oss create shared memory with a clock, then fork off one child and have that child access the shared memory and ensure it works. [Day 2]
3. Write code to parse options and receive the command parameters. [Day 3]
4. Implement the worker taking in an option and doing its loop and output while oss increments. Do not do this in a loop. [Day 4]
5. Implement oss to fork() and then exec() off one worker to do its task and have oss figure out when it is done. [Day 5]
6. Get oss to fork off workers up until the -n parameter and do their tasks [Day 6-7]
7. Implement the simultaneous restriction and interval launching parameter, as well as implement the process table and store data in it. [Day 8-9]
8. Implement the auto shutdown after 60 seconds. Can test this with terminating in shorter intervals if you want [Day 10]
9. Testing and make sure your README file indicates how to run your project. Give a one-line example that would let me know how to run it. DO NOT SIMPLY COPY/PASTE THIS DOCUMENT INTO THE README [Day 10+]

Criteria for success

Please follow the guidelines. Start small, implement one functionality, test. Do not wait until the last minute and contact me if you are having issues.

README

It is required that you submit a README file. This file should start with the following:

```
Name: Yourname
Date: When project started
Environment: What environment you used (linux, vi)
How to compile the project:
    Type 'make'
Example of how to run the project:
    ./oss -n 3 -s 2 -t 4 -i 0.6
```

In addition, if you use generative AI, you must include a section describing your use of generative AI in detail. This should include a bare minimum of what generative AI was used and some list of prompts that you used. I also want at least a few sentence description of in general how useful you found it.

For example, at a bare minimum something like:

```
Generative AI used : chatgpt
Prompts:
    Please give me a C++ command line program to play tic-tac-toe between two people
    Change the previous code to add the capability to play multiple games
    Add the capability to keep track of the previous record of players.
    Given the previous code, please add a command line option to allow one player to cheat
    etc
```

Summary: The initial game generated worked well, but it had one bug that I had to fix. It did

Submission

Handin an electronic copy of all the sources, `README`, `Makefile(s)`, and results. Create your programs in a directory called `username.1` where `username` is your login name on opsys. Once you are done with everything, *remove the executables and object files*, and issue the following commands:

```
chmod 700 username.2
```

```
cp -p -r username.2 /home/hauschildm/cs4760/assignment2
```

If you have to resubmit, add a `.2` to the end of your directory name and copy that over.

Do not forget `Makefile` (with suffix or pattern rules), your versioning files (`.git` subdirectory), and `README` for the assignment. If you do not use version control, you will lose 10 points. I want to see the log of how the program files are modified. Therefore, you should use some logging mechanism, such as `git`, and let me know about it in your `README`. You must check in the files at least once a day while you are working on them. I do not like to see any extensions on `Makefile` and `README` files.

Before the final submission, perform a `make clean` and keep the latest source checked out in your directory.

You do not have to hand in a hard copy of the project. Assignment is due by 11:59pm on the due date.