CS2106: Introduction to Operating Systems
Lab Assignment 2 – Process Operations in UNIX

Important:

- **The deadline of submission through LumiNUS is Wed, 23 September, 2pm**
- The total weightage is 7% (+2% bonus):
    - Exercise 1: 2 % **[Lab Demo Exercise]**
    - Exercise 2: 2 %
    - Exercise 3: 1 %
    - Exercise 4: 1 %
    - Exercise 5: 1 %
    - Exercise 6: 2 % **[Bonus marks]**

You must ensure the exercises work properly on the SoC Compute Cluster, hostnames: xcne0 – xcne7, Ubuntu 20.04, x86_64, GCC 9.3.0.

Background

All computer systems typically need to run processes that are not directly attached to a terminal or a display i.e. "in the background". These processes can provide important functions to the system e.g. configuring the network, syncing the system time, and so on. They could also be servers like HTTP (web) or SSH servers.

Such background processes are typically called *services*, and they are usually managed by a *service manager**. Functions of a service manager can include

- setting up the environment for services to run
- launching services based on the system administrator's configuration
- logging of the services' output and other events
- restarting services when they fail

Examples of service managers include:

- Windows's Service Control Manager (Services.exe)
- systemd**
- OpenRC**
- monit
- s6

* There is no strict definition of a service manager, but in general they perform those functions.

** These also have other functions aside from being a service manager.

Your Task

Your task in this lab is to write a simple service manager.

At the end of the lab, your service manager will be able to:

- start and stop services
- wait for services to complete
- construct pipelines (as in shell pipelines)
- print the status of services
- redirect services' standard output and error to log files

The service manager accepts user commands that perform above operations.

The reading and parsing of user commands **has been handled for you**. You are nonetheless encouraged to read and understand the provided code.

Implementation Instructions

The lab archive contains the following files:

- **`Makefile`**: the Makefile (will be replaced when grading)
- **`main.c`**: the file containing **`main`**, as well as the code handling user commands (will be replaced when grading)
- **`sm.h`**: the file containing **`struct`** declarations and function declarations (will be replaced when grading)
- **`sm.c`**: the source file containing empty function definitions for you to implement

You should **NOT** change the function signatures of the function definitions that are provided as **`main.c`** expects those functions to have those signatures.

You will probably need some state to keep track of running services; please define that in **`sm.c`**.

The skeletons of two functions **`sm_init`** and **`sm_free`** are provided in **`sm.c`**. You may use them to initialise and clean up, respectively, any memory and/or other resources that your program uses.

To compile your program, either run **`make`** or invoke GCC directly:

```
$ gcc -std=c99 -Wall -Wextra -D_POSIX_C_SOURCE=200809L –D_GNU_SOURCE -o sm main.c sm.c
```

You may use **`-Werror`** to make all warnings errors. You will not be penalised for warnings, but you are strongly encouraged to resolve all warnings. Warnings are indications of potential undefined behaviour which may cause your program to behave in inconsistent and unexpected ways that may not always manifest (i.e. it may work when you test, but it may fail when grading). You are also advised to use **`valgrind`** to make sure your program is free of memory errors.

You can assume that all operations succeed and no invalid input is provided to your program. In any case, it is good practice to check the return values of all syscalls (and indeed, all functions) and handle any errors that occur.

Your program will be tested with at most 32 services during the entire lifetime of the program.

Command Format

To run the program, simply:

```
$ ./sm
```

You will then see an **`sm>`** prompt where you can input the following commands. These commands are accepted by the implementation in **`main.c`**:

Note: command parsing is handled for you. This is provided as a reference to aid in your testing.

- **`start <program1> [args...] [| <program2> [args...]]...`**: Start the service with the given process(es). (Exercises 1 and 2)
  - e.g. **`start /bin/sleep 0`**
  - e.g. **`start /bin/echo hello | /bin/cat | /bin/sha256sum`**
  - **`|`** must be surrounded by a space on both sides, otherwise it will not be treated as a pipe
  - Note that each **`start`** command specifies one service. See the specification of exercise 2 below for details on multi-process services.
- **`startlog ...`**: The same as above, but redirect standard output and error to a log file. (Exercise 4)
- **`status`**: Show the status of all services. (Exercise 1)
- **`showlog <service number>`**: Print the log file for the specified service to standard output. (Exercise 5)
- **`wait <service number>`**: Wait for the given service to terminate. (Exercise 3)
  - e.g. **`wait 1`**
- **`stop <service number>`**: Stop the given service. (Exercise 3)
  - e.g. **`stop 1`**
- **`shutdown`**: Terminate all services and exit. (Exercise 3)

The **`<service number>`**s are the numbers reported by **`status`**. Each command will be implemented using a corresponding C function (prefixed by **`sm_`**) in **`sm.c`**. A skeleton code has been provided for you. Follow the instructions in Exercises 1–5 to complete the implementation.

Exercise 1: Start single-process services and show their status (1% submission + 1% demo)

This exercise has two sub-parts.

Let us first define a service. A service is simply a group of processes as defined by the user. We will only deal with single-process services in this exercise.

Part 1a: Implement the single-process **`start`** command in your service manager. To achieve this, implement the function **`void sm_start(const char *processes[])`**. A skeleton has been provided for you in **`sm.c`**.

This function accepts a single argument which is simply an array of strings. The array contains the path and arguments of the process, followed by a **`NULL`**. The function should simply launch the given process and return immediately.

For example, the command line **`/bin/sleep 0`** will be an array

```
{"/bin/sleep", "0", NULL}
```

The strings may be freed after your function returns. If you need any of the arguments, make a copy.

All paths provided will be relative or absolute paths to an executable (i.e. you do not need to do anything to the paths; passing them to the relevant syscall will be sufficient).

Note that the launched processes will run in the background, as children of the service manager, but will inherit the standard output and error of the service manager. You can close the standard input (FD 0) of the launched processes. Assume that the launched processes of single-process services will not read from standard input.

You may want to record the details of the launched process for each service, as needed, so you can implement **`sm_status`** later.

These syscall families may be useful:

- `fork`
- `exec`

Test your implementation by running **`sm`** and entering a start command, e.g.

```
$ ./sm

sm> start /bin/sleep 10
```

You can use a process monitor like **`htop`** or **`top`** to check the processes spawned.

Part 1b: Implement the **`status`** command in your service manager. For this, implement the function **`size_t sm_status(sm_status_t statuses[])`**. A skeleton has been provided for you in **`sm.c`**.

The function will be passed an already-allocated array of **`sm_status_t`**, which is a struct declared in **`sm.h`**. It should fill each entry in the array with the details and status of each service **in the same order the services were launched**. It should return the number of entries filled in.

You will need to declare your own state variables to keep track of the services launched.

These are the fields in **`sm_status_t`**:

- **`pid`**: PID, in field **`pid`**
- **`path`**: Path to the service as given when the service was launched
- **`running`**: **`true`** if the service is still running

You may assume that enough space in the array is provided for all the entries.

This syscall family may be useful:

- **`wait`**

Test your implementation by running **`sm`** and entering a start command, e.g.

```
$ ./sm
sm> start /bin/sleep 10
sm> status
0. /bin/sleep (PID 1234): Running
```

Demo: Demo both parts 1a and 1b.

Exercise 2: Start multi-process services (2%)

Extend the **`start`** command to work for multi-process services.

The array of strings passed to the **`sm_start`** function actually consists of multiple sub-arrays, each terminated with a **`NULL`**. After the last process, there is an additional **`NULL`**.

For example, the command line **`/bin/sleep 0`** will be an array (note the additional **`NULL`**, not shown earlier)

```
{"/bin/sleep", "0", NULL, NULL}
```

and **`/bin/echo hello | /bin/cat | /bin/sha256sum`** will be an array

```
{"/bin/echo", "hello", NULL, "/bin/cat", NULL, "/bin/sha256sum", NULL, NULL}
```

If a service contains multiple processes, you should launch all processes and join the standard outputs and inputs of consecutive processes, akin to a shell pipeline. That is, if the service definition has three processes A, B and C in that order, then:

- the standard input of A can be closed (assume that A will not read from **`stdin`**)
- the standard output of A is connected to the standard input of B
- the standard output of B is connected to the standard input of C
- the standard output of C is left alone

You will also have to modify **`sm_status`** so that if the service consists of multiple processes, the details, and status of the *last* process in the pipeline is reported. It is not needed for this exercise, but you should keep track of all the PIDs in a service, as you will need them in exercise 3.

These syscall families may be useful, in addition to those mentioned earlier:

- **`pipe`**
- **`dup`**

Test your implementation by running **`sm`** and entering a start command, e.g.

```
$ ./sm
sm> start /bin/echo hello | /bin/cat | /bin/sha256sum
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03  -
sm> status
0. /bin/sha256sum (PID 1234): Exited
```

Note that depending on the order in which `sm` and `sha256sum` print, your output may look slightly different.

Again, you can use a process monitor like **`htop`** or **`top`** to check the processes spawned. Note that for the example given, the processes will terminate nearly instantaneously.

You should probably also check that starting a single process still works as expected.

Exercise 3: Stop service, wait on service, and shutdown (1%)

Implement the **`stop`**, **`wait`** and **`shutdown`** commands in your service manager. For this, implement the functions **`void sm_stop(size_t index)`**, **`void sm_wait(size_t index)`** and **`void sm_shutdown(void)`**.

The functions **`sm_stop`** and **`sm_wait`** will be called with the index of the service to stop. The index corresponds to the same order as in exercise 1b (i.e. service launch order).

`sm_stop` should send **`SIGTERM`** to **all** the processes in a service, and then wait on them until they exit.

`sm_wait` should just wait on **all** the processes in a service.

`sm_shutdown` should do **`sm_stop`** on all services.

You will need to keep track of all the PIDs in a service.

If a process has already terminated, no signals should be sent to that process—the PIDs may have been reused! Note that in a multi-process service, some processes may terminate before others. Before sending a signal to a process, check the status of that child process.

If all the processes in a service have already exited, then attempting to stop or wait on the service should not cause any signals to be sent. You can assume that the index passed to your function (for **`sm_stop`** and **`sm_wait`**) is valid i.e. **`index < num_services`**, but it could refer to a service that has already completely exited.

You may assume that the processes in a service do not launch their own children.

These syscall families may be useful:

- **`kill`**
- **`wait`**

Exercise 4: Start with output redirection (1%)

Implement the **`startlog`** command in your service manager. For this, implement the function **`void sm_startlog(const char *processes[])`**. This function does the same as **`sm_start`**, except that the last process (or the only process) has its standard output *and* error redirected to a file **`serviceN.log`** in the current working directory of the service manager. **`N`** is the 0-based launch order of the current process i.e. the same order as in exercise 1b.

The log file should be opened in append mode (corresponding to `fopen` `"a"`).

You may extract the common functionality between **`sm_start`** and **`sm_startlog`** out into a separate function. This syscall family may be useful, in addition to those already mentioned:

- **`open`**

Remember to close the files in the parent process once they are no longer needed by the parent process!

Exercise 5: Show log file (1%)

Implement the **`showlog`** command in your service manager. For this, implement the function **`void sm_show_log(size_t index)`**. This function should print the log file for the specified service (again, specified with the 0-based launch order) to standard output. If the log file for the service does not exist for some reason e.g. that service was not launched with **`startlog`**, or some other program deleted the log file, it should instead print the following to standard output:

```
service has no log file
```

You do not need to track whether a service was started by `startlog` or not; your implementation can simply print the log file if it exists.

You may launch a helper program to print the log file or print it directly. If you launch a helper program by fork/exec, remember to wait on the child process.

Exercise 6 (bonus): Make it realistic (2%)

A real service manager does not run attached to a terminal, but instead itself forks into the background (i.e. daemonising), detaching from the terminal, and receives control messages sent by a control command (e.g. systemd's **`systemctl`**).

Make the service manager **`sm`** run in the background. Specifically, it should fork into the background so that the user is returned to the shell prompt immediately after launching **`sm`**.

Then, implement a command **`smc`** that launches into a prompt like the **`sm`** from the previous exercises and accepts the same commands. Any output should, of course, be printed from **`smc`**, and not the standard output of **`sm`**.

Add an additional command **`exit`** that simply exits **`smc`** (without shutting down **`sm`**). **`smc`** should also exit upon end-of-file from standard input (e.g. from Ctrl+D).

`shutdown` should cause **`smc`** to exit as well (in addition to shutting down **`sm`**).

You are free to use any means of IPC that you wish. Your program will be tested based on whether **`sm`** performs the correct operations in response to commands sent to **`smc`**.

If you need to create any files (such as Unix domain sockets), they must be created in the current working directory. In other words, simply use a relative path without specifying any directory. You can assume that `sm` and `smc` are run in the same working directory.

For this bonus exercise, you may modify **`main.c`** and **`sm.h`**. Your **`smc`** implementation should be done in **`smc.c`**. **`sm`** will be compiled from **`main.c`** and **`sm.c`** (as usual), and **`smc`** will be compiled from **`smc.c`**.

You may include an additional header file **`ipc.h`**, if required.

Please also include an additional **`README`** to briefly explain your IPC mechanism. Include details such as the IPC mechanism and any files or ports used.

Submission and Grading

Zip the following files as E0123456.zip (use your NUSNET ID, **NOT** your student number). Use capital E as prefix.

Do **not** add any additional folder structure. Your zip file should have this structure:

- **`E0123456.zip`**
  - **`sm.c`** (Ex 1–5)
  - **`ex6/`** (only if attempted)
    - **`main.c`**
    - **`sm.h`**
    - **`sm.c`**
    - **`smc.c`**
    - **`README`**
    - **`ipc.h`** (only if needed)

You may use the **`check_archive.sh`** script to check your zip archive.

We will test each exercise independently, but we will deduct marks if we observe that functionalities work only partially when other exercises are tested. For example, you will receive partial marks for the **`stop`** command if it passes only the testcases with single-process services and fails for multi-process services.