KIT104 ICT Architecture and Operating Systems
Tutorial Week 12 – UNIX Processes

1. Processes

We will start by using the **ps** command to look at the processes running on your UNIX server.

A	Login to your account on alacritas. Change into your `kit104` directory. On the shell command line, type in the following command (this command displays a lot of processes, so the `more` filter is used to break up the output): `$ ps -ef | more` Use the `man` to find out what the `-e` and `-f` switches do to the `ps` command. -e Select all processes. Identical to -A. -f does full format listing.
B	Type in the following command: `$ ps -eo pid,user,s,comm | more` Use `man` to find out how the `-o` option modifies the output of the `ps` command. -o format user-defined format.
C	Repeat the above `ps` command, this time taking note of the states of the processes. Process states: O Process is running on a processor. S Sleeping: process is waiting for an event to complete. R Runnable: process is on run queue. T Process is stopped.

D	To find out what processes you have running on your UNIX server, use the following command line:
	
	`$ ps -ef | grep your_username`
	
	For example:
	
	`$ ps -ef | grep jchen`
	
	This command causes `ps` to list all processes, and then uses the `grep` filter to restrict the output to just those lines that contain the username `jchen`.

E	An alternative is to use `-u` switch: `$ ps -u your_username`

	For example: `$ ps -u jchen`

2. Process Tree

When you execute most commands (eg `pwd`, `ls`, `who`, etc.) the shell spawns a new process. The shell is the **parent** and the created process is the **child**. A collection of parents and children is a **process tree**. We will investigate the process tree on your UNIX server using the `ps` command.

F	Execute the following command: `$ ps -eo ppid,pid,user,s,comm > ptree.txt` This saves the process listing to a file (`ptree.txt`). What does the `ppid` parameter represent?
G	Open the `ptree.txt` file using a text editor. Find the entry in the file for your current command interpreter (eg, `bash`). The following is an example of such entry: `24470  24471  jchen  S  bash` Use the parent process-id to identify the name of the process that created your current command interpreter process (ie, to identify the parent of `bash`).
H	**Repeat this process until you arrive at process 1 (`init`), thus creating the process tree for your current command interpreter. Show your tutor your process tree. (5 marks)** A process tree example is this: 0 1 root S init 1 2280 root S sshd 2280 14835 root S sshd 14835 14837 jchen S sshd 14837 14838 jchen S bash 14838 14916 jchen R ps 0 1 root S init 1 1627 root S sshd 1627 5942 root S sshd 5942 5961 chaofanz S sshd 5961 5962 chaofanz S bash 5962 6620 chaofanz R ps

3. The Dining Philosophers Problem

In this tutorial you are going to run some C/C++ programs which implement one of the most interesting classic problems of process synchronisation – the Dining Philosophers Problem.

You are not required to understand all aspects of the C/C++ programs provided in this tutorial. It should be sufficient to be aware of the nature of the programs and their functional descriptions provided here. However, you are recommended to pay attention to how the system calls `fork()` and `exec()` are used to create new processes in these programs: system call `fork()` creates a new process by cloning the caller, and system call `exec()` places a new program in the new process to run.

3.1 Task 1: `start.c` and `philo.c`

A	Under your `kit104` directory, make a new directory named `wk12`. Change into this new directory. Copy the three programs (**`start.c`**, **`philo.c`**, and **`philo2.cc`**) from the following directory on your UNIX server to your present working directory: `/units/kit104/tutorials` We are going to look at `start.c` and `philo.c` first. The program `philo2.cc` is for next section (this is a C++ program).
B	Read `start.c`. This program uses the system calls `fork()` and `execl()` (a variation of `exec()`) to create 5 new processes simulating 5 philosophers.
C	Read `philo.c`. This program simulates the life cycle of one philosopher. A philosopher gets the left chopstick and then the right chopstick before it starts eating. The program uses the system call `sleep` (to waste CPU time) to simulate both eating and thinking. When it finishes eating it returns both chopsticks so that its neighbours can use them.

D	Compile the two programs using the following command lines:
	
	`$ gcc -o start start.c`
	`$ gcc -o philo philo.c`
	
	Run the output file ./`start`. This executable file forks 5 child processes, each running a copy of `philo`. Please do not run `philo` independently (you might like to try to run it independently and you would see that the output does not make much sense).

E	Watch how a philosopher thinks and eats. If you think that the output is a little overwhelming you should focus on the activities of a particular philosopher (eg, philosopher 1).

F	Launch another UNIX shell to check to see that 6 new processes (one `start` and five `philo`'s) have been added to your process listing. If you are using the SSH Secure Shell client on a Windows PC, you can easily launch another UNIX shell by clicking on "Window", then on "New Terminal in Current Directory". You should also use the command `ps` with appropriate options to check to see that `start` is the parent of the five `philo`'s.
G	Which process ID represents which process? If the process ID's for the five `philo` processes are 1111, 1112, 1113, 1114, and1115, then 1111 should represent philosopher 1, 1112 should represent philosopher 2, …, 1115 should represent philosopher 5.
H	Keep watching the output of `start` until you see that philosopher 1 is at least 5 years old. Next, we are going to do something unusual.

I	Terminate philosopher 1 while it is **thinking**. You need to watch carefully when you do this (of course you do it on the other shell, the one you just used to display your process listing).
	
	Would this affect the activities of the other philosophers? Watch the output and think about why or why not.
	
	(You might like to check on the other shell to verify that philosopher 1 has actually been killed.)
	
	To terminate a process you can use this command line:
	
	`$ kill -9 Process-ID`
	
	For example, if 1111 is the process ID for philosopher 1, you can type in:
	
	`$ kill -9 1111`

J	Press `CTRL-C` to terminate all philosophers (as well as `start`). Run `start` again. Terminate philosopher 1 while it is **eating,** or while it is **holding at least one chopstick**. Would this affect the activities of the other philosophers? Watch the output and think about why or why not. (Again you might like to check on the other shell to verify that philosopher 1 has actually been killed.)

PID TTY TIM
5961 ? 00:00:
5962 pts/20 00:0
8323 pts/20 00:0
8324 pts/20 00:0
8325 pts/20 00:0
8326 pts/20 00:0
8327 pts/20 00:0
8328 pts/20 00:0
8383 ? 00:00:
8384 pts/24 00:0
8493 pts/24 00:0

ps -u chaofanz

stuck
5: Died with a chopstick - waited too long.
4: Died with a chopstick - waited too long.
3: Died with a chopstick - waited too long.
2: Died with a chopstick - waited too long.
Done

kill while eating
1: I now have my right chopst
1: I am going to eat now. I wa
3: I now have my left ch
3: Looking for my right
2: I am hungry; very hungr
2: Looking for my left chop
5: Died with a ch
3: Died with a chopstick
4: Died with a chops
2: Died without a chopstic
Done

	In the program `philo.c`, if a philosopher has waited for 80 weeks without successfully obtaining a chopstick (one second simulates one week), it dies (of hunger). You might wish to modify this number later if you have interest. Check on the other shell to verify that a philosopher has actually died.
K	If you are tired of watching the output, press `CTRL-C` to terminate all philosophers (as well as `start`). Check on the other shell again to ensure that this has happened.
L	Now, if you run `ls` under the directory where you executed `start`, you might see five file names that you did not expect: `chopstick1, chopstick2, …, chopstick5`. These are the results of `kill` or `CTRL-C` you used. They are empty files (with no actual contents saved on disk, only file names stored under the directory) which you can manually remove. But before you remove them you are recommended to run `chmod 700 *` so as to avoid messages such as "remove write-protected regular empty file …?"
M	For further investigation (in your own time): Let the five philosophers run without any interruptions to see how old these philosophers can live up to before they die. Depending on the system load, my best experimental result was that philosophers lived 64 years before they died – although the philosophers are programmed to live 100 years, they can not always make it.

3.2 Task 2: The C++ Implementation of the Dining Philosophers Problem

This time we are going to look at the C++ implementation of the Dining Philosophers Problem. Read the program `philo2.cc`. You should at least read the comment lines to get to know what each function does (a function is called a method in Java).

This C++ implementation uses the `fork()` system call to create five philosophers, without using the `exec()` system call. Additionally, a chopstick has been implemented as a **semaphore**: A philosopher tries to grab a chopstick by executing the `wait` operation on that semaphore, and a philosopher releases her chopsticks by executing the `signal` operation on the appropriate semaphores.

N	Compile the C++ program using the following command line: `$ g++  -o  philo2  philo2.cc` The command `g++` activates the C++ compiler on your UNIX server. Run `./philo2`. What does the output indicate? Press `CTL-C` to exit.
O	(Continued next page)

	Several possible remedies to the deadlock problem are available, eg. • **Allow at most four philosophers to be sitting simultaneously at the table (while making sure that five chopsticks have been created)** **Implement this remedy and show your solution to your tutor (5 marks).**

ps -u chaofanz

If you are wondering how it is possible to put the five philosophers in a deadlocked state, read lines 158 and 159 of `philo2.cc`.

(The End)