OPERATING SYSTEM
TD 1: Process scheduling

Global instructions

Keep in safe all versions of the programs and the outputs you will write and produce: they will be usefull for the next TDs.

1 Processes and processor observations

In this part, you will learn some basics about `/proc` folder and how to use the `top` command.

(1.1) Start by looking at the `/proc/cpuinfo` file to see the number of processors (hoping to find several). What kind of files are found in `/proc` in general? Explain `/proc/<pid>/stat` and `/proc/self`.

Run the `top` command and leave its terminal open somewhere: it will be used during all the TD and even more.

(1.2) What are the columns of the `top` command? And the lines above?

(1.3) Press `u` and enter your login to filter the displayed processes.
Press `f`, then select a sort column and press `s`, then `Esc` to exit.
Press `1` to display the use of different processors.
Press `f`, scroll down to the `P` line, and then press `Space` to add a column showing the processor running your command.
On which processors are the processes currently running? Why?

(1.4) Change the time granularity of the display refreshing by pressing `s` and then entering a (floating) number of seconds. What happens if we reduce this granularity a lot? Why?

2 Processor and processor time allocation

Remark: For all the rest of the TDs, compile with `-Wall`. It is better to see a warning and ignore it than not see it at all.

(2.1) Write a program that makes an infinite active wait. What do we see in `top` when we launch this program? If we launch several at the same time, how are they distributed on different processors? Why does CPU usage decrease if a message is displayed at each iteration? Why does a redirect in `/dev/null` fix the problem?

(2.2) Use the `taskset` program (or `numactl` or `hwloc-bind` if available) to lock a process on a single processor. For example:

```
taskset -c 0.2 <program>     # run a command on processors 0 or 2
taskset -p -c 0.2 7456       # move current process of pid 7456
taskset -p 03 7456           # list processors by a hexadecimal mask
```

Launch your program several times on the same processor at the same time and check in `top` that they are no longer distributed on the different available processors. How does the target processor manage these different processes that it must run?

(2.3) Run your process twice on the same processor with different priorities using the `nice` or `renice` commands. How does the allocation of processor time vary with the priority?

(2.4) Write a program that sleeps a very long time in the `sleep` system call. Expand your `top` window (and reduce the font if necessary) to see all tasks, including the sleeping programs. Check if the system also distributes these sleeping tasks between different processors, and explain.

3 Measurements of context changes

You will use the C function `gettimeofday` to measure durations. To avoid overflows, you can calculate a duration between two invocations with the following code:

```
unsigned long microseconds;
struct timeval tv1, tv2;
gettimeofday(&tv1, NULL);
    /* thing to measure */
gettimeofday(&tv2, NULL);
microseconds = (tv2.tv_sec-tv1.tv_sec)*1000000 + (tv2.tv_usec-tv1.tv_usec);
```

(3.1) Confirm that your code is working correctly by measuring the duration of a `sleep` system call.

(3.2) Measure the cost of the `gettimeofday()` call itself. Measure the cost of a `getpid()` (probably the simplest Linux system call). Compare the result with the use of the following code to force a real system call:

```
#include <unistd.h>
#include <asm/unistd.h>
Syscall (__NR_getpid);
```

(3.3) Write a program that make a loop of 100000 `sched_yield` system calls in order to constantly pass to another process, displays the total time of the loop and then restarts. Run two instances of this program at the same time on the same processor. What does the displayed time mean?

(3.4) Now launch only one instance of this program at a time. What does the output mean?

(3.5) Now run three (and then many more) instances of the program *on the same processor*. Quickly plot the displayed time curve according to the number of processes, and explain it. What do you think about the cost of changing contexts and how it will affect the performances according to the duration of the timeslices?

4 Timeslices measurements

(4.1) Write a program calling `gettimeofday` and displaying a message when the duration between the last two calls is *unusually* long. The message should contain this duration and the pid. Adjust the threshold to detect significant distortions. Run this program twice on the same processor and check that the alternation of the two programs is well displayed, but not the noise.

(4.2) Write a program that loops on writing how long time it was executed, then how long time it was unscheduled. We will use variations of the duration of a `gettimeofday` to do this. Explain the observed behavior when a single instance of this program is launched on the same processor, then two instances.

(4.3) Draw the graph showing the evolution of the timeslice according to the priority.

5 User, system and real time

(5.1) Use the `/usr/bin/time` tool (slightly better than the `time` command of your shell) to find out where your `sched_yield` program is running. Explain the disaplyed times (you can compare with the `getpid()` system call, real or virtual). Restart the experiment with 2 then 3 instances simultaneously on the same core, and explain the new times.

(5.2) Take back your program calling `sleep` and explain the times measured by `/usr/bin/time`.

(5.3) Write a program making logical inputs-outputs (small reads and writes in a pipe) and explain the times observed by `/usr/bin/time`. Vary the size of writings and readings, plot the evolution of the time proportion of system/user, and explain. What happens with very large writings? Deduct the size of the buffer from the Linux pipes.

(5.4) You will read a file on the disk (or on the network) but not yet in memory. You can use a large local file not yet read since the start of the machine (movies, music, in `ls -lSr /usr/lib`, ...). If you are root, you can also empty the cache of files with:

```
sudo bash -c "echo 1 >/proc/sys/vm/drop_caches"
```

If we have a NFS, we can create a large file on another machine with:

```
dd if=/dev/zero of=file count=100000 bs=1000
```

Once the file not present in memory chosen, read it with:

```
/usr/bin/time cat file > /dev/null
```

Explain the observed times during the first run. Remember what is `/dev/null` and why you redirected the output to it. Repeat this command several times and explain what you are observing. How does the behavior change if you write in a real file?

6 CPU burner

With all the previous works, write a **CPU burner**, a program that wastes a fixed proportion (for example 40%) of a given processor, regardless of its activity (100% used or not, by a single process or several, ...). Show the good behavior with `top` on different examples, and explain the limits of the program.