G52OSC Coursework Part 2: Process Scheduling in Linux

Requirements

This coursework focusses on the use of **operating systems APIs** (specifically, the POSIX API in Linux) to create a set of processes. You will also use the APIs to influence the way in which the processes that you have created are scheduled by the **operating system's process scheduler**. What you aim to achieve is similar to what was illustrated during the lectures for the process scheduler in Windows 7. You will use your code to investigate the influence of **hard and soft CPU affinity**, **process priorities**, and the **scheduling of your processes as a function of the number of processes you have created**. You will be required to use the insights that you gained during the **lectures** and from **relevant literature** on process scheduling in Linux to explain the behaviour that you observe. You will experiment with you code on the school's Linux servers, in particular on bann.

Copying Code and Plagiarism

You may freely copy and adapt any code samples provided below, including any lab exercises or lecture samples provided in the G52OSC course. You may freely copy code samples from the Linux/POSIX websites, which has many samples explaining how to do specific tasks. This coursework assumes that you will do so and doing so is a part of the coursework, therefore you are not passing someone else's code off as your own, thus doing so does not count as plagiarism. Note that some of the code examples provided below will omit error checking for clarity of the code. You are required to add error checking wherever necessary.

You must not copy code samples from any other source, including another student on this or any other course, or any third party. If you do so then you are attempting to pass someone else's work off as your own and this is plagiarism. The University takes plagiarism extremely seriously and this can result in getting 0 for the coursework, the entire module, or potentially much worse.

Getting Help

You MAY ask Geert De Maere or one of the lab helpers for help in understanding what the requirements mean (i.e. *what* you need to achieve) if they are not clear. Any necessary clarifications will then be added to the Moodle page so that everyone can see them.

You may NOT get help from anybody to actually do the coursework (i.e. *how* to do it), including myself or the lab helpers. You may get help on any of the code samples below, since these are designed to help you to do the coursework without giving you the answers directly.

Background Information

Note that an additional tutorial on compiling source code in Linux using the GNU c-compiler can be found on the Moodle page. If required, additional background information on process scheduling in Linux can be found online, including in the tutorial listed here: http://www.ibm.com/developerworks/library/l-completely-fair-scheduler. Additional information on Linux programming is available here: http://www.advancedlinuxprogramming.com/alp-folder/, if required.

General OS/Scheduler Background

Linux is a multi-tasking operating system. This means that, in practice, multiple processes run concurrently (by quickly alternating them) or in parallel on multiple CPUs/cores. The operating system is responsible for managing these processes. This includes creating, destroying, context switching, and scheduling them, and the management of the execution traces (threads) and the resources they use. The processes you create in this coursework will contain only one thread. The reason for this is that the school's server offers more possibilities to influence the scheduling of processes than the scheduling of threads.

Every process in Linux is characterised by a unique and non-negative integer, called the process identifier (PID). The PID is used as an index into the process table where the associated process control block is stored. Note that the process control block in "Linux terminology" is called the task control block and that processes themselves are called tasks. Only a finite number of processes can exist simultaneously within the same system, and once the PIDs have reached their maximum value, they are wrapped around. PID `1` is ``reserved'' for the `init` process (the parent of all processes in Linux).

As stated, the first process started on a Linux system is the `init` process. All other processes are created by the `init` process using system calls. The process scheduler is responsible for determining the order and the CPU/core on which these processes will run. In determining this order, the CPU scheduler takes a number of objectives and process characteristics into account (e.g. CPU bound, I/O bound, priority, CPU affinity). The process scheduler also aims to get the best possible value for a number of objectives, including throughput, utilisation, fairness, and responsiveness. Note that Linux process schedulers have evolved considerably over different versions of Linux, with each approach having its own strengths and weaknesses.

Programming in Linux

To complete the coursework successfully, you will have to use a number of operating system APIs. The use of these APIs is illustrated below. You are allowed to use the code listed below and modify it in your coursework.

Creating Processes

There are several ways to create new processes on Linux, including `fork()` and `clone()`. `fork()` is the most commonly known (since it was also present in Unix) and easiest approach to create new processes. However, it offers less flexibility than `clone()` (which enables to specify in detail which resources are cloned for the child process). When `fork()` is called by the parent process, it executes a system call to ask a service of the operating system, i.e., to create a new process, execute it, and carry out all internal administration that is required for that new process (e.g. creating the process control block and adding it to the process table). The `fork()` system call makes an exact copy of the current process. I.e., the child process will contain the exact same image as the parent process. After creation, the parent and child processes both continue with the first instruction immediately following the `fork()` call. Your code can distinguish between the parent and child process based on the value of the PID variable, which has not been set in the case of the child process. Hence, in the case of the child process, the variable will still contain the initial value. If, for some reason, the `fork()` call could not be carried out successfully, the PID returned to the parent process will be `-1`. The following example illustrates the use of `fork()`. More information on fork can be found here: http://linux.die.net/man/2/fork

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
	pid_t pid = 0;
	pid = fork();
	if(pid < 0) {
		printf("Could not create process\n");
		exit(1);
	} else if(pid == 0) {
		sleep(1);
		printf("Hello from the child process\n");
	} else if(pid > 0) {
		printf("Hello from the parent process\n");
	}
	printf("This code will be executed by both the child and parent\n");
}
```

Overwriting process images

The `fork()` system call creates a copy of the parent process. The memory image of the child process can be overwritten using one of the `exec()` system calls, as illustrated below. More information on the different `exec()` system calls can be found here: http://linux.die.net/man/3/exec

```c
#include <stdio.h>
#include <unistd.h>

int main()
{
	int status;
	pid_t pid = fork();
	if(pid == -1) {
		printf("fork () error\n");
	} else if(pid == 0) {
		execl("/bin/ps", "ps", "l", 0);
		printf("This code should not run");
	}
}
```

The child process in the program above executes the `ps -l` command. The output will show quite a few different fields. Have a look at the meaning is of the individual pages in the `man` pages or on the web. Can you find the different process states and priorities in the output?

Waiting for Processes

After creation, child processes usually get a life of their own. On some occasions, you would want the parent process to wait for the child until it has finished. I.e. the parent process has to suspend its execution until the child process(es) return. This can be achieved by making the parent process execute a `wait` system call that takes the child's process identifier as one of the parameters. This is illustrated below.

```
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_PROCESSES 4

int main() {
	int i, status, finishedProcessPid;
	pid_t pid;
	printf("Hello from the parent process with PID %d\n", getpid());
	pid = fork();
	if(pid < 0) {
		printf("fork error\n");
	} else if(pid == 0) {
		sleep(1);
		printf("Hello from the child process with PID %d\n", getpid());
		return;
	}
	waitpid(pid, &status, WUNTRACED);
	printf("Child process has finished\n");
}
```

CPU Affinity

In multi-core/multi-processor systems, processes/thread can run on different CPUs. Linux knows hard and soft CPU affinity. Under normal circumstances, soft affinity is used. I.e., processes can run on any available CPU/core, and migrate between CPUs to balance load. Hard affinity can be set explicitly, again by using system calls to ask the operating system's scheduler to run the process on the specified (set) of CPU(s) only. The use of hard affinity is illustrated below.

```
#define _GNU_SOURCE
#include <sched.h>

int main()
{
	cpu_set_t cpuset;
	CPU_ZERO(&cpuset);
	CPU_SET(4, &cpuset); // run on the 5th core (starts at 0)
	CPU_SET(5, &cpuset); // add core 6
	sched_setaffinity(getpid(), sizeof(cpu_set_t), &cpuset);
}
```

Setting Process Priorities

As a student user (but also for most members of staff) on the school's servers, there are fairly few ways in which you can manipulate processes and process priorities. E.g., you are not able to specify higher than usual priorities for your processes, nor can you change the type of process from the "normal class" to, e.g., the real time class. However, to investigate the influence of process priority on scheduling, it is sufficient to create different process priorities by lowering the priority of some of them. This can be achieved by the `setpriority()` system call, as illustrated below. More information on the `setpriority()` call can be found here: http://linux.die.net/man/2/setpriority

```
#include <sys/resource.h>

int main()
{
	// Sets the priority of the current process
	setpriority(PRIO_PROCESS, getpid(), 19);
}
```

Note that lowering process priority is a one way street: once you have reduced the priority of one of your processes, you cannot increase it again at a later point in time. Even the child processes inherit the parent's priority values. The restriction on setting process priorities prevents regular users from creating CPU-hogging processes that overtake all the existing ones.

Timers

Like most programming languages, C can use the "system time". This time can be used to check at what intervals the processes you created are using the CPU, e.g. relative to the start time of the process. The functions that are available on Linux are illustrated below. Note that the returned time value contains two values - the number of seconds and the number of microseconds. Hence, in order to retrieve the number of milliseconds, you will have to manipulate the times.

```
#include <sys/time.h>
#include <stdio.h>

long int getDifferenceInMilliSeconds(struct timeval start, struct timeval end);
long int getDifferenceInMicroSeconds(struct timeval start, struct timeval end);

int main()
{
	int i;
	struct timeval startTime, currentTime;
	gettimeofday(&startTime, NULL);
	sleep(1);
	gettimeofday(&currentTime,NULL);
	printf("Difference in milli-seconds %d\n", getDifferenceInMilliSeconds(startTime,
currentTime));
	printf("Difference in micro-seconds %d\n", getDifferenceInMicroSeconds(startTime,
currentTime));
}

long int getDifferenceInMilliSeconds(struct timeval start, struct timeval end)
{
	int seconds  = end.tv_sec  - start.tv_sec;
	int useconds = end.tv_usec - start.tv_usec;
	int mtime = (seconds * 1000 + useconds / 1000.0);
	return mtime;
}

long int getDifferenceInMicroSeconds(struct timeval start, struct timeval end)
{
	int seconds  = end.tv_sec  - start.tv_sec;
```

```
        int useconds = end.tv_usec - start.tv_usec;
        int mtime = (seconds * 1000000 + useconds);
        return mtime;

}
```

Assignment

This coursework should be implemented on the school servers. Support for programming, i.e. coding environments, is typically limited to `emacs`, `vi`, or `vim` on these machines, but you are more than welcome to use any of your own environments to develop the code, and compile it on the Linux servers using the `gcc` compiler.

Your task consists of writing a program that creates a specified number of child processes that keep track of their own CPU usage (e.g. having a loop in them that logs when they are running). You will then modify this program to test several aspects of CPU scheduling under Linux. You will implement your code in incremental steps, as specified below. Note that, if you decided to use shared data structures, you will be required to synchronise access, but only if it is strictly necessary.

- **REQUIREMENT 1** Write a program in which the parent process creates a specified number of child processes. The number of child processes to create can be specified either on the command line or as a constant in your code (e.g. `NUMBER_OF_PROCESSES`). Add a few small `printf` statements to your code to verify that your implementation is working as expected (remove them again when they are no longer needed). Make sure that every child process has a unique index associated to it, e.g. between `1` and `MAX_NUMBER_OF_PROCESSES`. You may need this index later in this coursework to visualise the CPU timings.
  You may note that more child processes are created than the number you specified. For this reason, start by creating a very small number of child processes (e.g. 4 at most). This helps to minimise the load on the school servers. You can increase the number of processes once you are fairly confident that your code is working properly.
  There are several ways in which you can ensure that you create the exact number of child processes required. Explain the method that you have used in the documentation of your code (5 lines max).
- **REQUIREMENT 2** Once the program above is working, modify the code to ensure that the parent process (and only the parent process) waits for all child processes to finish before continuing (note that all child processes have to be able to run in parallel). When this is implemented, the command prompt should only display after all processes have finished, including the child processes. Explain in the documentation how you made sure that it is only the parent process that is waiting for the child processes, and not the child processes for each other (5 lines max).
- **REQUIREMENT 3** Once the program is working, add code to track the CPU usage of your processes relative to the start time of your parent process. E.g., at the start of your parent process, you log what the current time is, and pass this timestamp on to the child processes who use it as ”the base time” for logging their CPU activity. Keep track of the “run-times” for each of the child processes in an array. To avoid generating too much data (which becomes difficult to analyse), make sure that the child processes only run for a pre-specified and configurable amount of time (set in milliseconds) and work with a granularity of milliseconds when logging at what times the child processes are running. One way of achieving this is by working with an array that contains one element for every millisecond, starting at `0` (corresponding to the base time), ending at `MAX_EXPERIMENT_DURATION`. This array contains, e.g., the process index when the child process was running at the corresponding point in time, `0` otherwise. Make sure that the children’s CPU times (i.e. the arrays) are written out before they finish, using the format shown below.

timevalue, process index
timevalue, process index
timevalue, process index
timevalue, process index
timevalue, process index
timevalue, process index
timevalue, process index
…

Note that this can be achieved by writing them to a file, or by simply re-directing the output of your program to a file. Submit a sample of your output for 4 processes (named `OUTPUT1.CSV`) with your code, together with a visualisation (named `GRAPH1`), e.g. generated in MS Excel or any other program of your choice. The graph should be submitted in `jpg` or `png` format.

- **REQUIREMENT 4** Modify the above code to run on one single CPU/core/logical core (choose one of the 8 cores at random to prevent all of you working on one individual core), i.e. by setting hard CPU affinity. Rerun the code, and generate a data file (named `OUTPUT2.CSV`) and visualisation (named `GRAPH2`) similar to the one above, and in the same file format.
- **REQUIREMENT 5** Now that all processes are running on a single CPU, modify the process priorities to be `0`, `+5`, `+10`, and `+15`. Re-run your code, and analyse the results. Generate a data file (named `OUTPUT3.CSV`) and visualisation (named `GRAPH3`) similar to the one above, and in the same file format.
- **REQUIREMENT 6** Once the above code is working, modify your program to add an additional child process that runs the `ps -l` command while the child processes are running. The output should show all the child processes that you have created, their process state, and their priorities. Submit the output together with your code (named `PROCESSLIST1.TXT`)
- **REQUIREMENT 7** Once the above code is working, modify your program to generate one single visualisation the CPU timings for the child processes in scalable vector format. Note that scalable vector graphics is an XML/HTML like format that can be used to draw scalable shapes by writing simple files. This is often a much easier approach to generating visualisations or graphs compared to generating bitmap files. A simple example can be found here: http://www.w3schools.com/svg/svg_rect.asp. Save the generated SVG/HTML code as an `HTML` file. This will allow you to open it in regular browsers which recognise and display SVG images (Internet Explorer, Google Chrome, and Firefox all recognise SVG, however Firefox seems to be the slowest).
- **REQUIREMENT 8** Now rerun the code above for 4 processes (anything between 1000 and 10000 milliseconds should provide sufficient data):
    - Without CPU affinity and equal priorities (name the generated file `TIMINGS1.HTML`)
    - With CPU affinity and equal priorities (name the generated file `TIMINGS2.HTML`)
    - With CPU affinity and non-equal priorities (name the generated file `TIMINGS3.HTML`)

  Explain in a separate document the observed scheduling behaviour. Explain whether the scheduling of your processes is pre-emptive or non-preemptive, whether you think that starvation could occur, and if not, how it is prevented. Do you recognise any of the scheduling algorithms discussed during the lectures, and if not, briefly explain how you believe the scheduling of your processes happens. Keep your explanation short and to the point (15 lines max). Repeat the above experiments for 15 processes, and upload the resulting files together with your code (named `TIMINGS4.HTML`, `TIMINGS5.HTML`, and `TIMINGS6.HTML`, respectively).

Submission

Ensure that you have fully completed the documentation file, since it will be used for the marking.

Create a single `.zip` file containing your source code (1 file), documentation file (1 file), the output files (3 `.csv` files and 3 graphs in `.jpg` or `.png` format), the process list (1 txt file), and visualisations that you have generated (6 `.html` files, containing SVG images). We will need to rebuild your code to test your implementation. You will submit your single zip file through the submission system at: https://tma.cs.nott.ac.uk

The deadline for submission is 7pm, Wednesday 29th April (Part 1 and Part 2 at the same time)

Marking criteria

In each case the marking will be on the basis of how well you explain your observations, your implementation, and the correctness of your observations. To assess these, we will consider both the submitted code and the explanations you provided in your documentation file. In particular, we will check that your code works correctly, will look for specific inefficiencies and check your explanations to verify that you understood what you did and that you did it for the correct reasons.

When considering how this will be marked, please be aware that it is important that:

A) We can compile and run your program.
B) Your program does not crash.
C) We can understand the code that you have written. Use the documentation to explain this.
D) We understand why you wrote the code in the way that you did. Again use the documentation to explain this.
E) Your code works correctly.
F) Your code does not unnecessarily complicated or inefficient.
G) You have demonstrated your understanding of process scheduling in your explanations.