G52OSC Labs: Processes, Process Scheduling, Concurrency and Operating System APIs

Goals

The lab sessions focus 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 during the lab sessions is similar to what was illustrated during the lectures for the process scheduler in Windows 7, but this time for Linux.

You will use your code to investigate the influence of process scheduling in a multi-core/multi-CPU environment, in particular 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**. This will help you to better understand the principles explained in the lectures, and will illustrate the theory in practice. Whilst you will develop and run all code on the school servers, the observations are valid for most of the contemporary Linux systems (using the same process schedulers).

In summary, the key goals of the labs are:

- To illustrate the use of operating system APIs (necessary for the coursework)
- To help you understand process creation in Linux
- To help you better understand process scheduling
- To illustrate the practical relevance of the concepts discussed in the lectures
- To prepare you and give you the background knowledge that is required to successfully complete the coursework and the exam

Background Information

- An additional tutorial on compiling source code in Linux using the GNU C-compiler can be found on the Moodle page.
- Additional information on Linux programming is available in the "Advanced Linux Programming" book (which is freely available online), amongst many others
- 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

Writing code

Support for programming on the school servers, i.e. coding environments, is typically limited to `emacs` or `vim`, 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 `gcc program`.

Session 1: Creating/working with Processes

Fork

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 the 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 from 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

```
#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");
}
```

Task 1:

Using the same principles as above, write a program in which the parent process creates a pre-specified number of child processes (e.g. using a fork in a for loop). This number can be specified either on the command line or as a constant in your code (e.g. `NUMBER_OF_PROCESSES`). Make sure that you start with a relatively small number processes and assign every child process a unique index, e.g. between `1` and `NUMBER_OF_PROCESSES` (you will need this index later in these lab sessions). Add a `printf` statement to the child process that displays this ID together with the child's PID. Verify that your implementation is working as expected.

You may notice that more child processes are created than the number you specified (hence, why we asked you to start with a small number of processes to minimise the load on the school's servers). If this is the case, think of a way to resolve this (i.e. to create the exact number of processes) and implement it. You can increase the number of processes once you are confident that your code is working properly.

Sample output:

```
$ ./task1
$ Hello from the child 0 with pid 18038
Hello from the child 2 with pid 18041
Hello from the child 1 with pid 18040
Hello from the child 4 with pid 18043
Hello from the child 3 with pid 18042
```

Run your program multiple times and analyse the output. Why do the child processes print their output in different orders? Why don't you see the command prompt ($) printed when all the child processes have finished?

Session 2: Overwriting Process Images

Background:

The `fork()` system call creates an exact 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 of the individual entries in the `man` pages or on the web. Can you find the different process states and the process priorities discussed in the lectures in the output?

Task 2:

Based on the example code above, **modify your code from the previous task to create an additional child process** for which you overwrite the "memory image" to run `ps -l`. The output should show all the child processes that you have created, their process state, and their priorities. What happens if you remove the sleep(1) instruction of your code from task1? Why?

Sample output:

```
PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    1 28134 28133 24653 pts/8    24653 S     5248   0:00 ./a.out XDG_SESSION_ID=2291
HOSTNAME=severn.cs.nott.ac.uk    SELINUX_ROLE_REQUESTED=    TERM=xterm    SHELL=/bin/bash
HISTSIZE=1000    SSH_CLIENT=128.243.18.129    65413    22    SELINUX_USE_CURRENT_RANGE=
QTDIR=/usr/lib64/qt-3.3    QTINC=/usr/lib64/qt-3.3/include    SSH_TTY=/dev/pts/8
QT_GRAPHICSSYSTEM_CHECKED=1 USER=pszgd
  1 28135 28133 24653 pts/8    24653 S     5248   0:00 ./a.out XDG_SESSION_ID=2291
HOSTNAME=severn.cs.nott.ac.uk    SELINUX_ROLE_REQUESTED=    TERM=xterm    SHELL=/bin/bash
HISTSIZE=1000    SSH_CLIENT=128.243.18.129    65413    22    SELINUX_USE_CURRENT_RANGE=
QTDIR=/usr/lib64/qt-3.3    QTINC=/usr/lib64/qt-3.3/include    SSH_TTY=/dev/pts/8
QT_GRAPHICSSYSTEM_CHECKED=1 USER=pszgd
…
Hello from the child 1 with pid 28135
Hello from the child 0 with pid 28134
Hello from the child 2 with pid 28136
```

Session 3: Waiting for Processes

Background:

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 (this is often also required for threads, as in the coursework). I.e. the parent process has to suspend its execution until the child process(es) have finished. 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;
	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");
}
```

Task 3:

Modify your code above 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. You should notice now that the prompt only appears after all the child processes have printed their messages.

Sample output:

```
$ ./a.out
Hello from the parent process
Hello from the parent process
Hello from the parent process
Hello from the parent process
Hello from the child 0 with pid 20109
Hello from the child 1 with pid 20110
Hello from the child 2 with pid 20111
Hello from the child 3 with pid 20112
Bye from the parent!
$
```

Sessions 4-5: The Linux Process Scheduler

General OS/Scheduler Background

Linux is a multi-tasking operating system. This means that, in practice, multiple processes run concurrently (by quickly alternating) 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 during the lab sessions 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. Note that `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 (similar to the ones you used above). The process scheduler is responsible for determining the order as well as the CPU/core on which the processes will run. In determining this order, the CPU scheduler takes a number of objectives and process characteristics into account (e.g. CPU bound process, 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.

Logging Process Times

Background:

Like most programming languages, C can use the "system time". This time can be used to check at what times/intervals the processes you created are using the CPU, e.g. relative to the start time of the parent process. The functions/data structures that you can use 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;

}
```

Task 4:

Extend the code developed in task 3 to estimate the time that it takes to fork a process, combined with the time it takes between forking the process and running the process (i.e., the response time). Note that these are only approximations since the parent process can be, for example, suspended between instructions. However, it should give you a reasonable estimate.

Sample output:

```
Hello from the child 0 with pid 8109 at time 93
Hello from the child 1 with pid 8110 at time 101
Hello from the child 2 with pid 8111 at time 125
Hello from the child 3 with pid 8112 at time 197
```

Task 5:

Modify the code in task 4 to track the times at which the child processes are running, relative to the base time (the time at which the parent process starts). In other words, at the start of your parent process, you log what the current time is, and pass this timestamp on to all child processes who use it as “the base time” for logging their CPU activity. In each of the child processes, you retrieve the current time and take the difference between the current time and the start time of the parent process.

There are several steps in which you can achieve this (you are welcome to start with the last one if you feel confident you can do it):

1. Define an **infinite loop** in the processes that prints the process ID and time at which the process was running (the one that you assigned yourself on the screen) in the following format:
   ```
   timevalue, process index
   timevalue, process index
   timevalue, process index
   …
   ```
   You will probably see a fairly random pattern of IDs on the screen, however, when redirecting the output to a file and visualising it in a graph, patterns should start to emerge. A few warnings:
   - Make sure that you kill all child processes that you have created from the command line using, **`killall -u XXXXX`, in which XXXXX is replaced by your own username** (this will kill ALL your processes, including anything else that you have running).

- Note that, when re-directing output to a file, **the file size will increase very rapidly**! I.e., make sure that you don't leave your code running for too long, and that **you delete the files afterwards**! Or even better, **run your code and redirect only the first 10000 lines to a file using** `./task5a | head -n 10000 > output.csv`
- Generate a visualisation that shows when processes are running (e.g. using Excel)

2. Use a predefined duration for your experiments, i.e., make sure that the child processes terminate automatically when the maximum time (relative to the parent process) is exceeded.
3. Replace the messages that you print on the screen with an array in which you log for each process the times that they were running. 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 (see above, `MAX_EXPERIMENT_DURATION` set in milliseconds) and work with a granularity of milliseconds when logging at what times the child processes are running. Make sure you choose the data structure carefully so that:
    a. You can log processes that are running in parallel
    b. That you minimise any synchronisation requirements for this data-structure (in fact, choosing the correct data structure will prevent the need for synchronisation!)

    Make sure to write out the data once all child processes finish in the correct format. Try running your code with different numbers of child processes, e.g. 4 and 16, and assess what the impact is on the process behaviour and the scheduling of the processes.

Note that your output may sometimes not be entirely what you expected. Occasionally, you will find that the output for multiple processes is printed out randomly and on other occasions you may notice that formatting is not entirely what you would hope for. This is due to multiple processes writing to the standard output at the same time. There are two ways out of this:

- One would have to synchronise the writing (e.g. using semaphores) by allowing only one process to write to the standard output at any one time.
- Alternatively, you could make sure that only the parent process writes the information out once all child processes have finished, however, this would require the use of shared memory between the parent and child process (i.e., the child process writes the values to the parent's memory).

Sample outputs for the different stages of this task are available on Moodle.

Session 6: Shared Memory

Background:

Shared memory is a fast form of inter-process communication in which multiple processes write to the same (physical) memory, and hence, can share information with one another. Considering that multiple processes access the same memory, synchronisation must be considered and may have to be applied, e.g. using semaphores. Note that this is only required if multiple processes can access the exact same memory location simultaneously. Since shared memory is similar to accessing local memory, there is no performance penalty (unless synchronisation is required of course). Whilst the physical memory that is shared is the same (i.e. the frames), different process may have the physical memory segment attached to different logical addresses in their own address space (i.e. the pages).

Setting up shared memory consists of the following steps:

- Open a shared memory object using `shm_open`, specifying the name, optional flags, and the directory permissions, respectively
- Configure the size of the shared memory object using `ftruncate`, specifying the file descriptor, followed by size of the object.
- Map the shared memory object in to the processes' logical address space using `mmap`, specifying the address location at which to attach the memory, the size, the read/write protection, optional flags, the file descriptor for the shared object, and the offset (usually 0). Best practice is to specify `NULL` for the address location, thereby allowing Linux to decide itself where to attach the object into the logical address space, and returning the memory.
- Unlinking the shared memory segment using `shm_unlink`, specifying at least the name of the object

Additional information on any of the above functions can be easily found online, or in the man pages using, e.g. `man shm_open`. Note that you will have to specify `-lrt` on the command line to link in the appropriate library when compiling with `gcc`. An example of how to use shared memory in a single process is given below.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>

#define SIZE_OF_MEMORY sizeof(int)
#define SHARED_MEMORY_NAME "GDM123456"

int main()
{
	int shm_fd = shm_open(SHARED_MEMORY_NAME, O_RDWR | O_CREAT, 0666);
	if(shm_fd == -1)
	{
		printf("failed to open shared memory\n");
		exit(1);
	}
	if(ftruncate(shm_fd, SIZE_OF_MEMORY) == -1)
		printf("failed to set size of memory\n");
	int * i_ptr = mmap(NULL, SIZE_OF_MEMORY, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
	*i_ptr = 1000;

	if (munmap(i_ptr, SIZE_OF_MEMORY) == -1)
		perror("Error un-mmapping the file");
	shm_unlink( SHARED_MEMORY_NAME );
	shmctl(shm_fd, IPC_RMID, 0);
}
```

Task 6:

Modify your code from task 5 to use shared memory. Make sure that you choose your data structure such that you do not have to enforce mutual exclusion/synchronisation. Make the parent process print out all the values that were written to the shared memory by the child processes.

Session 7

Semaphores

Background:

Instead of using shared memory, one could use semaphores to synchronise the writing of the child processes. I.e., allow only one process to write to the standard output at any point in time. This will avoid lines from showing up in random order, and the fact that lines are occasionally printed through one another when the code is not synchronised. A full overview of using semaphores on Linux can be obtained by typing `man sem_overview` on the command line. Similar to shared memory, there are a number of steps that one has to go through to declare/access global semaphores (semaphores that are shared between processes – also called process semaphores):

- Create a new named semaphore by calling `sem_open` and specifying the semaphore's name in the format "`/semaphore_name`" (make this unique)
- Use the semaphore by calling the functions `sem_post` and `sem_wait`
- Close the semaphore once the process no longer needs it calling `sem_close`
- Deleting the semaphore from the system by calling `sem_unlink` (please do not forget to call this function since the semaphore may otherwise continue to exist in the OS, and the number of semaphores that are available is limited!)

Note that calling `sem_open` will result in a named semaphore, i.e., it has a name associated with it. This name can be shared between different processes, who can then "open" the semaphore (i.e. by retrieving a reference to the semaphore from the operating system) and use it to synchronise shared resources. The alternative to named semaphores are unnamed semaphores. These are declared in a region of memory that is shared between multiple threads (e.g. declared as a global variable or a variable in a region of shared memory – as above). You are likely to use unnamed semaphores for your coursework since the coursework uses threads. A detailed description of how to use the different functions listed above can be found in the man pages, e.g. by typing "`man sem_open`"

Task 7:

Modify the code of task 5 in such a way that the individual child processes print out their timings. Use named semaphores to make sure that only one child process prints to the standard output at any given point in time.

CPU Affinity

Background:

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);
}
```

Task 8

Modify your code from task 5 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) by setting hard CPU affinity. Rerun the code, and generate a data file containing the process times. Generate a visualisation that shows when processes are running (e.g. using excel). You should recognise the characteristics of the Linux scheduler that were discussed during the lectures. As always, make sure that the parent process waits for the child processes to finish.

Session 8: Process Priorities

Background:

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. Lowering process priorities 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 restrictions on setting process priorities prevents regular users from creating CPU-hogging processes that overtake all the existing ones. 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);
}
```

Task 9:

Using your code from task 6 (with all processes running on a single CPU), modify the process priorities to be 0, +5, +10, and +15 (i.e. 4 processes). Re-run your code and analyse the results. Generate a data file and visualisation similar to the one above and try to explain the results using the principles discussed during the lectures.

Task 10:

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, and in this case allows more detailed analysis of the process times compared to graphs generated in, e.g., Excel.. 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)

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`)

Observe the scheduling behaviour and link it with the principles discussed during the lectures. 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 more basic scheduling algorithms discussed during the lectures (e.g. round robin, priority queues), and if not, explain how you believe the scheduling of your processes occurs. Repeat the above experiments for 15 processes.

Extra session: Operating System APIs for File systems

Files

Most of the I/O operations in Linux can be performed using only five functions: `open`, `read`, `write`, `lseek`, and `close`. The functions described here are often referred to as unbuffered I/O, which means that each read or write invokes a system call in the kernel.

To the kernel, all open files are referred to by **file descriptors**. A file descriptor is a non-negative integer. When we open an existing file or create a new file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with the file descriptor that was returned by open or create as an argument to either read or write.

POSIX-compliant applications use symbolic constants STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO to refer to the standard input of a process, the standard output, and the standard error, respectively. These constants are defined in the <unistd.h> header.

Familiarise yourself with these five operations using the man pages. Then, run the following program and check the resulting output. Use `od -c outputFile.dat` to look at the contents of the file as characters.

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
#include<stdlib.h>

char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";
int main(void)
{
int fd;
if( (fd=open("archivo",O_CREAT|O_TRUNC|O_WRONLY,S_IRUSR|S_IWUSR))<0) {
printf("\nError %d to open",errno);
exit(-1);
}
if(write(fd,buf1,10) != 10) {
perror("\nError in first write");
exit(-1);
}
if(lseek(fd, 16384,SEEK_SET) < 0) {
perror("\nError in lseek");
exit(-1);
}
if(write(fd,buf2,10) != 10) {
perror("\nError in second write");
exit(-1);
```

```
}
return 0;
}
```

Note that with lseek, the file's offset can be greater than the file's current size, in which case the next write to the file will extend the file. This is referred to as creating a hole in a file and is allowed. Any bytes in a file that have not been written are read back as 0.

A hole in a file isn't required to have storage backing it on disk. Depending on the file system implementation, when you write after seeking past the end of the file, new disk blocks might be allocated to store the data, but there is no need to allocate disk blocks for the data between the old end of file and the location where you start writing.

To prove that there is really a hole in the file, let's compare the file created with a file of the same size, but without holes:

```
$ ls -ls outFile2.dat
20 -rw-rw-r--. 1 pszit pszit 16394 Sep 15 15:44 outFile2.dat

$ cat outputFile.dat > outputFile2.dat

$ ls -ls out*
20 -rw-rw-r--. 1 pszit pszit 16394 Sep 15 15:48 outputFile2.dat
 8 -rw-------. 1 pszit pszit 16394 Sep 15 15:43 outputFile.dat
```

Why both files are the same size? How many disk blocks have these files and why? According to this, what is the block size of our Operating System?

Task 11

Starting from the code above, write a program that takes as argument a pathname of a file, opens the corresponding file and using a fixed block size of 80 bytes and creates an output file that looks like this:

Block 1
// First 80 Bytes
Block 2
// Next 80 Bytes.

Next, modify this program to include an extra line indicating the number of blocks at the beginning of the program. So, it should looks like:

The number of block is:
Block 1
// First 80 Bytes
Block 2
// Next 80 Bytes.

File attributes and types

Given a pathname, the statfunction returns a structure of information about
the named file. The fstat function obtains information about the file that is
already open on the descriptor filedes. The lstatfunction is similar to stat, but when the named file is a symbolic link, lstat returns information about the symbolic link, not the file referenced by the symbolic link.

The function fills a structure that looks like this:

```
struct stat {
mode_t st_mode; /* file type & mode (permissions) */
ino_t st_ino; /* i-node number (serial number) */
dev_t st_dev; /* device number (file system) */
dev_t st_rdev; /* device number for special files */
nlink_t st_nlink; /* number of links */
uid_t st_uid; /* user ID of owner */
gid_t st_gid; /* group ID of owner*/
off_t st_size; /* size in bytes, for regular files */
time_t st_atime; /* time of last access */
time_t st_mtime; /* time of last modification */
time_t st_ctime; /* time of last filestatus change */
blksize_t st_blksize; /* best I/O block size */
blkcnt_t st_blocks; /* number of disk blocks allocated */
};
```

The biggest user of the stat functions is probably the ls –l command, to learn all the information about a file.

Multiple functions may be used to identify file types, check the following program and understand its behavior.

```
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
#include<stdlib.h>

int main(int argc, char *argv[])
{
    int  i;
    struct stat buf;
    char *ptr;

    for (i = 1; i < argc; i++) {
        printf("%s: ", argv[i]);
        if (lstat(argv[i], &buf) < 0) {
         printf("\nlstat error");
            continue;
         }
         if (S_ISREG(buf.st_mode))
            ptr = "regular";
         else if (S_ISDIR(buf.st_mode))
            ptr = "directory";
         else if (S_ISCHR(buf.st_mode))
            ptr = "character special";
         else if (S_ISBLK(buf.st_mode))
            ptr = "block special";
         else if (S_ISFIFO(buf.st_mode))
            ptr = "fifo";
         else if (S_ISLNK(buf.st_mode))
            ptr = "symbolic link";
         else if (S_ISSOCK(buf.st_mode))
            ptr = "socket";
         else
            ptr = "** unknown mode **";
         printf("%s\n", ptr);
  }
   exit(0);
}
```

Symbolic links

A symbolic link is an indirect pointer to a file, unlike the hard links from the previous section, which pointed directly to the i-node of the file. Symbolic links
were introduced to get around the limitations of hard links.

It is possible to introduce loops into the file system by using symbolic links. Consider the following commands:

```
$ mkdir foo      # make a new directory
$ touch foo/a    # create a 0-length file
$ ln -s foo/ foo/testdir  # create a symbolic link
$ ls -l foo
```

This creates a directory foo that contains the file a and a symbolic link that points to foo. A loop of this form is easy to remove. We are able to unlink the file foo/testdir, as unlink does not follow a symbolic link. But if we create a hard link that forms a loop of this type, its removal is much more difficult. This is why the link function will not form a hard link to a directory unless the process has superuser privileges.

Task 12

Create a simple program using the function ftw() (http://linux.die.net/man/3/ftw) to descend through a file hierarchy, printing each pathname encountered. Run this program on the path created with the loop. What output would you expect? What did you get?

Directories

Directories can be read by anyone who has access permission to read the directory. But only the kernel can write to a directory, to preserve file system sanity. In <sys/types.h> and <dirent.h>, you can find the most common function to work directories.

```
DIR *opendir(char *dirname)
struct dirent *readdir(DIR *dirp)
int closedir(DIR *dirp)
void seekdir(DIR *dirp, log loc)
long telldir(DIR *dirp)
void rewinddir(DIR *dirp)
```

Task 13

Using these function, write a program that descend through a file hierarchy and count how many regular file it finds in its way. For these files, the program will provide the i-node number. At the end of the program the total size occupied by all the regular files has to be shown.