CS342 Operating Systems - Fall 2016
Project 1: Processes, IPC, and Linux Modules

```
Assigned: Oct 05, 2016 Wed
Due date: Oct 19, 2016 Wed, 23:55
```

You will do this project individually. You have to program in Linux using C programming language.

Part A: Concurrent Processes and IPC. [70 points]
Objective: Practicing process creation, process communication (by use of POSIX message queues), multi-process application development.

Develop a concurrent sort program (application) that will use multiple processes to sort a binary file of positive integers. The application will take a binary file as input and will sort it into a textfile. The binary input file will be a sequence of integers (8 byte values in a 64-bit machine; we are assuming a little-endian architecture like Intel x86). Assuming 8 byte integers, if there are 10 integers in the file, the file size will be 80.

The main process of your application will create n message queues and n child processes (worker processes): one message queue per worker process. A message queue for a worker process will be used to pass information from the worker process to the parent main process. *Note that when a child is created, its address space (memory) is populated from parent's (hence* ***initial*** *one way information passing can be done* ***once*** *from parent to child, including open file and message queue descriptors), but since the address spaces are different, after a while later, they may contain different things in their memory and they can not access their variables. Therefore, in order to exchange information, they need to use an IPC mechanism like pipes or message queues. You will use POSIX message queues in this project [4,5,6].*

After creating the worker processes, the main process will wait for sorted integers to arrive from message queues. Each worker process will read the integers from a portion of the input file and will sort them (you will use a linked list to keep integers in a worker process). Then, sorted integers will be sent via the respective message queue to the main process. The main process will receive the sorted integers from message queues and merge them *efficiently* (i.e., while receiving) into an output text file. The can be done by retrieving the minimum integer from the head of queues each time. When a worker has sent all its integers, it will terminate. Before termination, a worker can send a special integer like 0 to indicate the end of the stream. After receiving and merging all sorted integers, the main process will also terminate. Before termination, the main process will remove all the message queues (clean up).

Each worker process will process a portion of the input file. Assume there are m integers in the input file and there are n worker processes. Let $m = nk + r$, where $0 <= r <= n-1$ and $k > 0$. The first worker process will read and process the first portion of the input file, i.e. the first k integers in the file. The second worker process will read the next portion of the file, i.e., the next k integers; and so on. The last worker will process k + r integers (the last portion of the file). A worker thread will use random file access (see the lseek() system call) to jump to the start of its portion in the input file and then will read integers from there sequentially.

The program executable will be named as **csort** and will have the following parameters:

csort <n> <bin_infile> <text_outfile>

Here <n> is the worker count, <bin_infile> is binary input file and <text_outfile> is output text file. An example invocation can be:

csort 3 in.bin out.txt

An example output file content can be (assuming there are just 4 integers):

```
98
134
245
300
```

The minimum number of workers (MIN_WORKERS) is 1 and maximum (MAX_WORKERS) is 5.

Part B: Kernel Module Programming. [30 points]
Objective: Learning how to develop a Linux loadable kernel module; touching to and interacting with Linux kernel; starting writing kernel code.

Develop a simple Linux kernel module and test it (insert and remove). First learn how to write a kernel module. Read the related parts from the 9th edition of our textbook [1]. There are additional references that you can benefit from [2].

Your module will print out (using printk kernel function) as much information as possible about a process from its PCB (process control block - task_struct structure). The pid of the process will be given as input to your module at insertion time (load time). Your module will search for the PCB of the process with the given pid and will print as much information as possible by utilizing the PCB. Memory and file system related information will be printed out as well. When you use printk, the output will go to the kernel message buffer (not to the screen). You can the content of this buffer by using the dmesg command. Look to the last part of the output to see the printed information. Name your module as *modpcb*. Your module C file will be named as modpcb.c.

References:

[1]. *Operating System Concepts*, Silberschatz et al., 9th edition, Wiley, 2014, pages 94-98 and pages 156-158.
[2]. *The Linux Kernel Module Programming Guide*. http://www.tldp.org/LDP/lkmpg/2.6/lkmpg.pdf. [Accessed: Feb 11, 2016].
[3]. *Linux Device Drivers*. Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. O'Reilly, 3rd Edition. https://lwn.net/Kernel/LDD3/. [Accessed: Feb 11, 2016].

[4] *Supplementary Notes*. CS342 webpage; in Lecture notes section. http://www.cs.bilkent.edu.tr/~korpe/courses/cs342spring2016/. [Accessed: Feb 11, 2106].
[5]. *Linux Message Queue Overview*. http://linux.die.net/man/7/mq_overview. [Accessed: Feb 11, 2016].
[6]. *The Linux Programming Interface*. http://man7.org/tlpi/download/TLPI-52-POSIX_Message_Queues.pdf. [Accessed: Feb 11, 2016].

Submission:

Put the following files into a project directory named with your ID, tar the directory (using **tar xvf**), zip it (using **gzip**) and upload it to Moodle. For example, a student with ID 20140013 will create a directory named 20140013, will put the files there, tar and gzip the directory and upload the file. The uploaded file will be 20140013.tar.gz. In group projects, one of the students will upload and his ID will be used.

- csort.c: contains your C program in part A
- modpcb.c: contains the module code of part B
- Makefile: Compiles the program and the module.
- README: Your name and ID and any additional information that you want to put.

Additional Information and Clarifications:

- *Suggestion: work incrementally; step by step; implement something, test it, and when you are sure it is working move on with the next thing.*
- More **clarifications**, additional information and explanations that can be useful for you may be put to the **course website**, just near this project PDF. Check it regularly.
- Parent can create/open message queues and pass their descriptors (IDs) to the child processes automatically when creating child processes with fork() (that means fork already copies the address space of parent to the children and therefore the descriptors (IDs) are automatically passed to the children). This means, the children do not need to open the message queues again; They already have the message queue descriptors and can start accessing the queues with them.
- You can use command line utilities like ipcs, ipcrm, etc. to see the messages queues created and remove them outside of your program.
- The following web page contains a project skelaton. You can clone it into your local machine, if you wish.
  https://github.com/korpeoglu/cs342-fall2016-p1
- You can implement any sorting algorithm you wish.

Provided Code:

Below are the files that are also provided in github.

```
Makefile:

obj-m += modpcb.o  testmodule.o

all:
	gcc -g -Wall -o csort csort.c -lrt
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	rm -fr *~   csort *.o *.ko *.mod.c *.order
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

---

csort.c:

```
#include <stdlib.h>
#include <mqueue.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main()
{

	mqd_t mq;
	struct mq_attr mq_attr;

	/* just shown one message queue creation. More needed if more than
	   one child needs to be created.
	*/
	mq = mq_open("/mqname1", O_RDWR | O_CREAT, 0666, NULL);
	if (mq == -1) {
		perror("can not create msg queue\n");
		exit(1);
	}
	printf("mq created, mq id = %d\n", (int) mq);

	mq_getattr(mq, &mq_attr);
	printf("mq maximum msgsize = %d\n", (int) mq_attr.mq_msgsize);

	mq_close(mq);
	return 0;
}
```

---

modpcb.c:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
```

---

testmodule.c:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

/* This function is called when the module is loaded. */
int simple_init(void)
{
	printk(KERN_INFO "Loading Module\n");

	return 0;
}

/* This function is called when the module is removed. */
void simple_exit(void) {
	printk(KERN_INFO "Removing Module\n");
}

/* Macros for registering module entry and exit points. */
```

```
module_init( simple_init );
module_exit( simple_exit );

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple Module");
MODULE_AUTHOR("SGG");
```