CEng 334 - Introduction to Operating Systems

Spring 2022-2023, Midterm, May 4, 2023
(5 pages, 4 questions, 100 points, 90 minutes)

METU Honor Code and Pledge

METU Honor Code: The members of the METU community are reliable, responsible, and Honorable people who embrace only the success and recognition they deserve and act with integrity in their use, evaluation, and presentation of facts, data, and documents.

I have read and understood the implications of the METU Honor Code. To be precise, I understand that this is a ***closed*** *notes exam, and I am forbidden to access any other source of information other than that provided within the exam. I understand and accept to obey all the rules announced by the course staff, and that failure to obey these will result in disciplinary action.*

IMPORTANT NOTE: This document will NOT be graded. You should write all your answers into 3 empty A4 pages. The first A4 page should contain your answers to Q1 and Q2. The second and third A4 pages should contain your answers to Q3 and Q4. Write your name, surname, and ID at the top of all pages at the beginning of the exam. The code templates given in this document are just guidance, and you'll need to copy/use them on the A4 pages to write your answers.

Some of the questions in this exam contain minor variations and are compiled separately for each student. Any cheating attempt via sharing of the answers will be detected and prosecuted in full.

QUESTION 1.(25 points)

a) (10 pts) A system has 4 active tasks. Task P is currently running on the CPU for 1 tick (unit time), its remaining CPU burst is 9 ticks, and its priority is 2. The other tasks and their scheduling information are as follows:

- Task S: Remaining CPU need: 11 ticks. Priority: 1
- Task Q: Remaining CPU need: 5 ticks. Priority: 1
- Task R: Remaining CPU need: 2 ticks. Priority: 3

Given the scheduling policy, find the task that will run on the CPU for the next tick. For policies requiring queues, assume the queue order is the same as the order above. If the running task and another task have equal conditions, the scheduler chooses the running task, avoiding context switch. Note that only a subset of the information is relevant for each of the policies.

- i) Round Robin with quantum 2:
- ii) Shortest Remaining Task First:
- iii) Multilevel Priority Queues (smaller number means higher priority, no round robin):

b) (15 pts) Consider the code below:

```
void whatthefork(){
int status;
    if (fork() != 0){
        printf("AA\n");
        if (fork() != 0){
            printf("BB\n");
            wait(&status);
            printf("%d\n",status);
        } else {
            printf("CC\n");
            exit(11);
        }
    } else {
        printf("DD\n");
        exit(22);
    }
    printf("FF\n");
}
```

Which of the following are valid printouts of this code? Wrong results will be punished with negative points.

a)	AA BB CC DD 11 FF
b)	AA BB CC 22 FF DD
c)	DD AA CC BB 22 FF
d)	DD AA BB 22 FF CC
e)	AA DD BB 11 CC FF
f)	AA BB CC 22 DD FF

QUESTION 2.(25 points)

As a freelance programmer, after finishing a programming project about databases, you are asked to include a new facility that would allow the client to plot mathematical expressions such as *sin(x)* on a pop-up window. Given that plotting a mathematical expression on a window is a completely different problem than databases, you decided to use a freely available program. You have tested the `gnuplot` from a shell as follows:

```
shell> /usr/local/bin/gnuplot
gnuplot> plot sin(x)
gnuplot> quit
shell>
```

where *gnuplot* plotted *sin(x)* on a popup window after entering the command `plot sin(x)`.

Modify your program such that `gnuplot` runs as a separate **process**, and is controlled through text commands sent from your application through a **pipe**. Use the following template and assume that all system calls succeed (hence no need to check for error conditions). The file descriptors for stdin, stdout, and stderr are stored in constants `STDIN`, `STDOUT`, and `STDERR`. Use the system call descriptions provided on the last page of the exam.

```
int main(){
    myBelovedDatabaseCode();
    //Now let's insert code for plotting sin(x)



}
```

QUESTION 3.(25 points)

a) (23 pts) You need to get 2 Hydrogen (H) atoms and 1 Oxygen (O) atom all together at the same time to make water. The atoms are threads. Each H atom invokes a function called hReady() when it is ready to react, and each O atom invokes a function called oReady() when it is ready. Write the code for hReady() and oReady() functions, which must delay until there are at least two H atoms and one O atom present. Finally, oReady() calls the function makeWater() (which just prints that water was made). After the makeWater() call, two instances of hReady() and one instance of oReady() should return.

Use the following declarations:

```
mutex = Semaphore(1);
count = 0; //for the detection of every 2 H atoms
hWait = Semaphore(0);
oWait = Semaphore(0);
```

Use the following code template to write code for the two functions described above.

```
void hReady(){
    mutex.wait(); //this must be the first statement; fill the rest accordingly
```

```
}

void oReady(){
```

```
}
```

b) (2 pts) Is busy-waiting possible in your solution? Yes or No:

QUESTION 4.(25 points)

Given the following snapshot of a 5-process and 4-resource system, answer a-d using Banker's algorithm: Available: 2, 1, 2, 0 for R1, R2, R3, R4, respectively.

	Allocation				Max				Need			
Process	R1	R2	R3	R4	R1	R2	R3	R4	R1	R2	R3	R4
P1	0	0	1	2	0	0	3	2				
P2	2	0	0	0	2	7	5	0				
P3	0	0	3	4	6	6	5	6				
P4	2	3	5	4	4	3	5	6				
P5	0	3	3	2	0	6	5	2				

a) (4 pts) Fill in the Need values for each process.
b) (7 pts) Is this system safe or unsafe? 0 pts for answers without explanations.
c) (7 pts) If a request from a process P1 arrives for (0, 4, 2, 0), can the request be immediately granted? 0 pts for answers without explanations.
d) (7 pts) If a request from a process P2 arrives for (0, 1, 2, 0), can the request be immediately granted? 0 pts for answers without explanations.

Some system call descriptions compiled from Linux manpages

- `int fork(void);` creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent. On success, the PID of the child process is returned in the parent, and 0 is returned in the child.
- `int execl(const char *path, const char *arg, ...);` Replaces the current process image with a new process image. The const `char *arg` and subsequent ellipses can be thought of as `arg0, arg1, ..., argn`. Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments must be terminated by a NULL pointer, and, since these are variadic functions, this pointer must be cast `(char *) NULL`. The function only returns if an error has occurred.
- `int pipe(int pipefd[2]);` Creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array `pipefd` is used to return two file descriptors referring to the ends of the pipe. `pipefd[0]` refers to the read end of the pipe. `pipefd[1]` refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe. On success, zero is returned. On error, -1 is returned.
- `int dup(int oldfd); int dup2(int oldfd, int newfd);` These system calls create a copy of the file descriptor `oldfd`. `dup()` uses the lowest-numbered unused descriptor for the new descriptor. `dup2()` makes `newfd` be the copy of `oldfd`, closing `newfd` first if necessary. After a successful return from one of these system calls, the old and new file descriptors may be used interchangeably. On success, these system calls return the new descriptor. On error, -1 is returned.
- `int open(const char *pathname, int flags);` Given a pathname for a file, `open()` returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (`read(2)`, `write(2)`, etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process. The argument flags must include one of the following access modes: `O_RDONLY`, `O_WRONLY`, or `O_RDWR`. The function returns the new file descriptor, or -1 if an error occurred
- `int close(int fd);` closes a file descriptor, so that it no longer refers to any file and may be reused. The function returns zero on success. On error, -1 is returned,
- `int write(int fd, const void *buf, int count);` Write to a file descriptor. Writes up to count bytes from the buffer pointed `buf` to the file referred to by the file descriptor `fd`. The number of bytes written may be less than count. On success, the number of bytes written is returned (zero indicates nothing was written). On error, -1 is returned.
- `int read(int fd, void *buf, int count);` Attempts to read up to count bytes from file descriptor `fd` into the buffer starting at `buf`. On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.