# CEN 305 Operating Systems Midterm I

**April 2011**

**Duration: 120 minutes**

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Total |
|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |  |  |

**Student Name:**

**Student ID Number:**

**<u>Rules</u>**

1. **The exam is open source, all type of printed sources are allowed.**
2. **Resource sharing is forbidden.**
3. **Electronic device(phone,calculator,etc.) usage is forbidden.**
4. **You are not allowed to leave classroom before you finish the exam.**
5. **Extra paper usage is not allowed, it will not be graded.**
6. **The exam is totally 120 points.**

**Good luck. Assit. Prof. Dr. Orhan Dagdeviren**

**Question 1 (Operating System Structures, 10 pts).** What are the main advantages of the microkernel approach to system design? (5 points) What are the main advantages of the monolithic kernel design? (5 points)

Answer: Advantages of microkernel approach:

1. Easier to extend a microkernel
2. Easier to port the operating system to new architectures
3. More reliable (less code is running in kernel mode)
4. More secure

Advantages of monolithic kernel design:

1.Tightly integrated code in one address space
2 Tight integration has high potential for efficient use of resources and for efficient code
3.Easier to design, therefore faster development cycle and more potential for growth

**Question 2 (Processes, 10 points)**. What is the purpose of the command interpreter? (5 pts) Choose an operating system and show 3 commands with their functionalities (5 pts).

Answer: Command interpreter reads commands from the user or from a file of commands and executes them, usually by turning them into one or more system calls.

We choose Ubuntu Operating System:
The commands are

- ls: lists the files and directories.
- pwd: prints current working directory
- cd: changes directory

If we choose Windows Operating System:
The commands are

- dir: lists the files and directories.
- del: deletes a file
- cd: changes directory

**Question 4 (Processes, 15 points)**. Write a C program to achieve the following operations:

Fork a process to create a child process. Then use ordinary pipes (NOTE: NOT NAMED PIPES!) communicate processes. The operations are listed below:

- Child process displays: "Child process id %d parent's id %d, writing %s message to the pipe \n"
- Child process writes “Hello” message to the pipe.
- Child process displays: "Child process id %d: exiting \n"
- Child process exits.

- Parent process displays: "Parent process id %d : reading from pipe \n"
- Parent process reads from pipe.
- Parent process displays: "Parent process id %d : read this message from child:%s \n"
- Parent process waits its child to exit.
- Parent process displays: "Parent process id %d: exiting \n"

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

 int main()
 {
```

```
            int pfds[2];
            char buf[30];
            int return_pid;
            int rv;


            if (pipe(pfds) == -1) {
               perror("pipe");
               exit(1);
            }

            return_pid=fork();
            if (return_pid==0) {
              char *message="hello";
              printf("Child process id %d parent's id %d , writing %s message to the pipe
        \n",getpid(),getppid(),message);
               write(pfds[1],message, 5);
               close(pfds[0]);
               printf("Child process id %d: exiting \n",getpid());
               exit(rv);
            } else {
               printf("Parent process id %d : reading from pipe \n",getpid());
               read(pfds[0], buf, 5);
               close(pfds[1]);
               printf("Parent process id %d : read:%s \n", getpid(),buf);
               wait(&rv);
               printf("Parent process id: %d exiting \n", getpid());
            }
   }
```

**Question 5 (Threads, 10 points).** What resources are used when a thread is created? (5 pts) How do they differ from those used when a process is created? (5 pts)

Answer: Because a thread is smaller than a process, thread creation typically uses fewer resources than process creation. Creating a process requires allocating a process control block (PCB), a rather large data structure. The PCB includes a memory map, list of open files, and environment variables. Allocating and managing the memory map is typically themost time-consuming activity. Creating either a user or kernel thread involves allocating a small data structure to hold a register set, stack, and priority.

**Question 6 (Threads, 10 points).** Write a C Code to create 3 POSIX Threads (7 pts). The output of your program should be able to produce the output given below:

I am a thread, my id is 1, I am the first thread.
I am a thread, my id is 2, I am the second thread.
I am a thread, my_id is 3, I am the third thread.

Also write the commands to compile and run this program (3 pts).

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

void *print_message_function( void *ptr );

main()
{
      pthread_t thread1, thread2, thread3;
     char *message1 = "I am a thread, my id is 1,I am the first thread.";
     char *message2 = "I am a thread, my id is 2,I am the second thread.";
     char *message3 = "I am a thread, my id is 3,I am the third thread.";
     int iret1, iret2, iret3;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function,
(void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function,
(void*) message2);
     iret3 = pthread_create( &thread3, NULL, print_message_function,
(void*) message3);

     /* Wait till threads are complete before main continues. Unless we
*/
     /* wait we run the risk of executing an exit which will terminate
*/
     /* the process and all threads before the threads have completed.
*/

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     pthread_join( thread3, NULL);
     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     printf("Thread 3 returns: %d\n",iret3);
       exit(0);
}


void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);



}

Commands for compilation and run:

Compile and link:
gcc our_threads.c –o our_threads  -lpthread

Run:
./our_threads
```

**Question 7. (Process Scheduling, 32 points)** Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

| Process | Burst Time | Priority | Arrival Time |
|---|---|---|---|
| P1 | 10 | 3 | 0 |
| P2 | 1 | 1 | 1 |
| P3 | 2 | 3 | 4 |
| P4 | 1 | 4 | 6 |

a) (12 pts) Draw six timelines illustrating the execution of these processes using

1. First-Come-First-Served
2. Shortest Job First
3. Preemptive Shortest Job First
4. Non-preemptive Priority based scheduling (a smaller priority number implies a higher priority)
5. Preemptive priority based scheduling
6. Round Robin (quantum=1) scheduling.

b) (10 pts) What is the turnaround time of each process for each of the scheduling algorithms in part (a)?
c) (10 pts) What is the waiting time of each process for each of the scheduling algorithm in part (a)?

**Question 8. (Process Sychronization 10 points)** What is busy waiting related to process synchronization? (4 pts) Show an example code in which busy waiting occurs (3 pts) and show how busy waiting is eliminated (3 pts).

**Busy-waiting** is a technique in which a process repeatedly checks to see if a condition is true, for example whether a lock is available.

Code in which busy waiting occurs:

```
int value = 0; // Free
        Acquire() {
        while (TestAndSet(value)); // while busy
}
        Release() {
        value = 0;
}
```

Code in which busy waiting is reduced:

```
Acquire() {
        // Short busy-wait time
        while (TestAndSet(guard));
        if (lock == BUSY) {
                put thread on wait queue;
                go to sleep() & guard = 0;
        } else {
                lock = BUSY;
                guard = 0;
        }
}
Release() {
        // Short busy-wait time
        while (TestAndSet(guard));
        if anyone on wait queue {
                take thread off wait queue
                Place on ready queue
   (wake up that thread);
        } else {
            lock = FREE;
        }
        guard = 0;
```

**Question 9. (Process Sychronization 23 points)** There are 4 processes which produce 4 different outputs infinitely. Process 1 produces Output 1. Process 2 inputs Output 1 and produces Output 2. Process 3 inputs Output 1 and produces Output 3. Process 4 inputs Output 2 and Output 3 and produces Output 4. These processes run concurrently and the buffers are unbounded. Synchronize these processes using semaphores and write the algorithm of each process (each process is 5 pts). Comment on your solution's correctness and show the by giving an example (3 pts).

Answer: We will use 4 semaphores: output1Produced, output2Produced, output3Produced, output4Produced.

```
Process 1:
while(1)
{
  produceOutput1();
  signal(output1Produced);
}

Process 2:
while(1)
{
  wait(output1Produced);
  produceOutput2();
  signal(output2Produced);
}

Process 3:
while(1)
{
  wait(output1Produced);
  produceOutput3();
  signal(output3Produced);
}


Process 4:
while(1)
{
  wait(output2Produced);
  wait(output3Produced);
  produceOutput4();
}
```

Assume the example:
All process started.
Process 2, 3 and 4 wait.
Process 1 produces output1.
Process 1 signals output1Produced.
Process 3 produces output3.
Process 1 produces output1.
Process 1 signals output1Produced.
Process 2 produces output2.
Process 4 first executes wait(output2Produced); then executes wait(output3Produced);
Process 4 produces output4.