Programme Code TU856,TU858
Module Code CMPU 2017

# TECHNOLOGICAL UNIVERSITY DUBLIN

**Grangegorman**

---

# TU856 BSc in Computer Science
# TU858 -BSc in Computer Science (International)

**Year 2**

---

SEMESTER 1 EXAMINATIONS 2022/23

---

## CMPU2017 -Operating Systems 2

Internal Examiners

Mr. Denis Manley
Dr Paul Doyle

External Examiners

Ms. Sanita Tifenale

Answer Questions 1 and any two others

Question 1 is worth 40 marks, all the rest are worth 30.

1

a) Given the following arrival times and CPU time for 4 processes determine the average turnaround time for:

*i.* A Round Robin schedule algorithm with a time slice of 6 *ms* **(4 marks)**

**ii.** *The Shortest Remaining Time.* **(4 marks**

| Arrival Time | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Job | A | B | C | D |
| CUP cycle time | 8 | 4 | 9 | 5 |

b) What are the four conditions that are required for Deadlock to occur? **(4 marks)**

c) Explain, using a suitable example, the steps required to push a node onto a stack. **(8 marks)**

d) Explain, using suitable examples, the steps required to add a node to an ordered linked list. **(10 marks)**

e) Will the following code delete nodes from a queue with 2 nodes in the queue to a non-empty queue? Clearly explain your reasoning. **(10 marks)**

The function call and the parameters passed to the enqueuer function are:

```
QueueNode* headPtr = NULL; // initialize headPtr
QueueNode* tailPtr = NULL; // initialize tailPtr
char item; // char input by user

// if queue is not empty
if (headPtr != NULL) {
   item = dequeue(&headPtr, &tailPtr);
}
printQueue(headPtr);
break;
```

The code for the dequeue function is:

```
// remove node from queue head
char dequeue(QueueNode* *hPtr, QueueNode* *tPtr)
{

   char value = (*hPtr)->data;


   QueueNode* tempPtr = *hPtr;
   hPtr = (*hPtr)->nextPtr;

   // if queue is empty
   if (*hPtr == NULL) {
      *tPtr = NULL;
   }

   free(tempPtr);

   return value;
}
```

2

a) Briefly describe the elements of a process control block (PCB) **(4 marks)**

b) In Linux a process is created using the *fork(), wait() and exec()* command. Explain, how the fork() command and wait() command produce a child process in the following code: **(9 marks)**

denis.manley@apollo: ~/CT2/week7

```
int main(int argc, char **argv)
{
    printf("--beginning of program\n");
    int counter = 0;
    int second;

    pid_t pid = fork();

    if (pid == 0)
    {

        int max_c = atoi(argv[1]);

        int i,x;
        for (i = 0; i < 5; ++i)
        {
            printf("child process: counter=%d\n", ++counter);
            for(x=0; x<max_c;x++);
        }
    }
    else if (pid > 0)
    {
        int j,x, status;
        int  max_p = atoi(argv[2]);

        pid = wait(&status);
        for (j=0; j < 5; ++j)
        {
            printf("parent process: counter=%d\n", ++counter);
            for (x=0;x<max_p;x++);
        }
    }
    else
    {
        // fork failed
        printf("fork() failed!\n");
        return 1;
    }
    printf("--end of program--\n");
    return 0;
}
```

9,0-1 72%

c) In the code from part b what is the purpose of the command line arguments: 1000 and 10000? **(5 marks)**

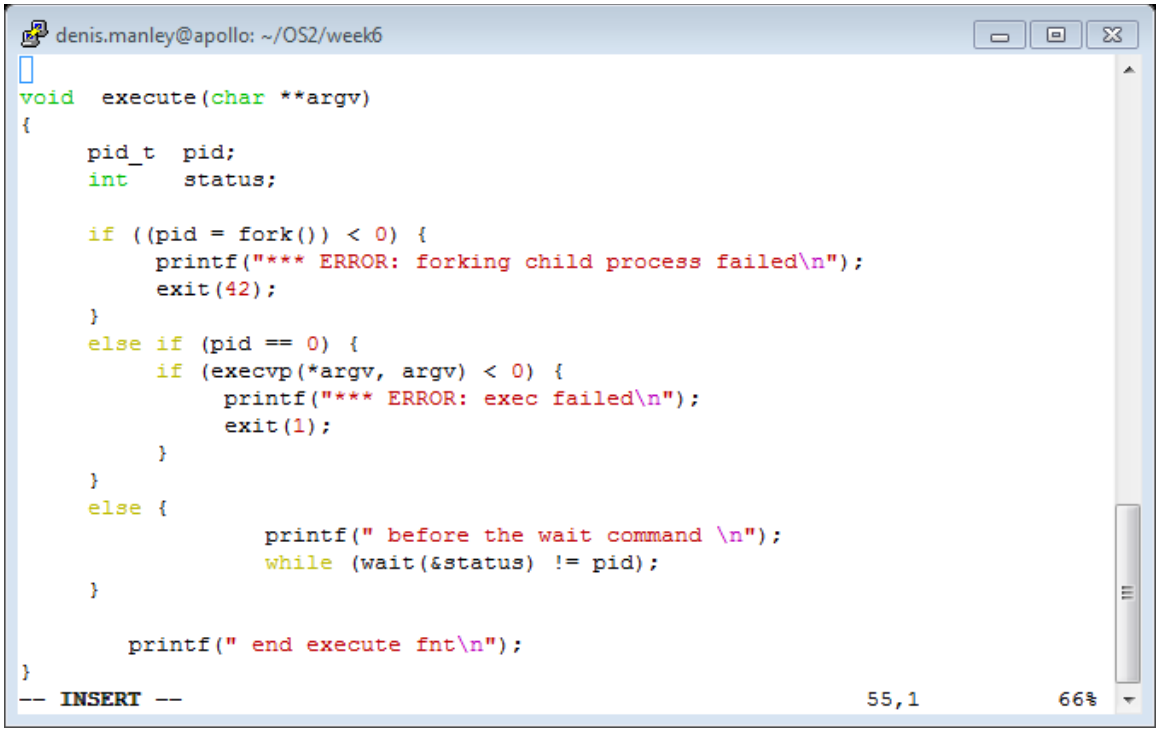
d) The execvp linux command has the following function prototype format

*int execvp(char *prog, char * argv[])*

Explain, using an example, what this function will do and what would data be stored in the prog parameter and the arg parameter is the commandline *cp file.c file2.c* .
**(4 marks)**

**e)**

The following code uses fork(), wait() and exec() functions. Explain what the code will do if the commandline arguments stored in arvg contains mv file1.c file2.c. where mv is the linux rename function. **(8 marks)**



3

a) Distinguish between single and multi-threading processes. **(4 marks)**

b) In C, a thread is created using the following code:

*int pthread_create(pthread_t *tidp, pthread_attr_t *attr, *start_rtn, void * arg)*

Clearly explain what each of the arguments in the thread create function mean. **(8 marks)**

c) Explain, in your own words, the following code: **(10 marks)**

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

int value;
void *my_thread(void *param);             /* the thread */

main (int argc, char *argv[])
{
        pthread_t tid;          /* thread identifier */
        int retcode;


        if (argc != 4) {
                printf ("program exiting: incorrect number of command line arguments\n");
                exit(0);
        }
/*create Thread */
        retcode = pthread_create(&tid,NULL,my_thread,argv[3]);
        if (retcode != 0) {
                fprintf (stderr, "Unable to create thread\n");
                exit (1);
        }


        pthread_join(tid,NULL);

        printf("The value returned by the thread  is %d",value);
        printf ("\nThe end of the program\n");

        pthread_exit(0);
}


 /* explain that this thread does */
void *my_thread(void *param)
{
        int i = atoi(param);
        printf("I am the child thread passed the value %d \n",i);

        value = i*i*i*i;

        pthread_exit(0);
}
```

d) What would be the output of the code in part c if the following are input at the command prompt? **(6 marks)**

(a.) ./a.out
(b.) ./a.out 6 7 5
(c.) ./a.out 6 5 7

e) What would be the two outcomes in the above program if the *pthread_join* command was removed and the command line input was *./a.out 6 5 7*? Explain the reason for your answer. **(2 marks)**

4:

a) Explain, using an example, why it is critical to ensure that concurrency is carefully controlled for processes accessing the same data item; in other words the *race* problem. **(6 marks)**

b) Two ways to prevent the race problem are Test and Set and Wait and Signal. Distinguish between each approach. **(4 marks)**

c) Explain, in detail, what the following two threads are doing: **(12 Marks)**

if

t-count =5

count = 6

```
void *signal_fnt(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

   if (count == COUNT_LIMIT) {
      printf("inc_count(): thread %ld, count = %d  Threshold reached. ",
             my_id, count);
      pthread_cond_signal(&count_threshold_cv);
      printf("Just sent signal.\n");
      }
    printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
           my_id, count);
    pthread_mutex_unlock(&count_mutex);

   sleep(1);
    }
  pthread_exit(NULL);
}
```

```
void *wait_fnt(void *t)
{
  long my_id = (long)t;

  printf("Starting watch_count(): thread %ld\n", my_id);

 pthread_mutex_lock(&count_mutex);
  while (count < COUNT_LIMIT) {
                printf("thread %ld  Going into wait...\n", my_id,count);

        pthread_cond_wait(&count_threshold_cv, &count_mutex);
                printf("thread %ld Condition signal received. Count= %d\n", my_id,count);
                printf("thread %ld Updating the value of count %ld...\n", my_id,count);

    count += 100;
                printf("watch_count(): thread %ld count now = %d .\n", my_id, count);
    }

  printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);

 pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

d) If *main()* has the following thread create calls:

```
int i, rc;
long t1=1, t2=2, t3=3;
pthread_t threads[3];


//create Three threads one to use the wait function and two for the signal function
pthread_create(&threads[0], &attr, wait_fnt, (void *)t1);
pthread_create(&threads[1], &attr, signal_fnt, (void *)t2);
pthread_create(&threads[2], &attr, signal_fnt, (void *)t3);

/* Wait for all 3 threads to complete */
for (i = 0; i < NUM_THREADS; i++) {
  pthread_join(threads[i], NULL);
}
printf ("Main(): Waited and joined with %d threads. Final value of count = %d. Done.\n",
        NUM_THREADS, count);
```

Give a sample output of this multithread program and explain your reasoning.

**(8 marks)**