UNIVERSITY OF LONDON

BSc EXAMINATION 2022

For Internal Students of
Royal Holloway

DO NOT TURN OVER UNTIL TOLD TO BEGIN

CS2850: Operating Systems
CS2850R: Operating Systems – for FIRSTSIT/RESIT CANDIDATES

Time Allowed: **TWO hours**

Please answer **ALL** questions

1. (a) What is the difference between the kernel and user CPU execution modes? Give an example of an operation that can only be executed in kernel mode. [4 marks]

   (b) Briefly describe the Short-Job First scheduling algorithm for batch systems. Give one advantage of using this algorithm over the First-Come First-Served algorithm. [4 marks]

   (c) Consider the following page table that shows the R and M bits for each memory page. Indicate which page would be selected for eviction by the Not-Frequently Used (NFU) page replacement algorithm, briefly justifying your answer.

Page	R	M
1	1	1
2	1	0
3	0	1
4	1	1

   [4 marks]

   (d) Consider the program that consists of the following two concurrent processes that share the variable `x`:

```
P1:   x = 7;                    P2:   x = 3;
      if(x < 5)                       if(x > 4)
        x = x + 3;                      x = x - 9;
```

   How many different values may the variable `x` have when the program terminates? Give a possible execution sequence for each case. [8 marks]

2. (a) There are four properties that are required for any correct algorithm to solve the mutual exclusion problem. Explain the importance of the property "No assumptions may be made about speeds or the number of CPUs." in this context. [4 marks]

   (b) Consider the following proposed solution to the mutual exclusion problem for two processes. Suppose there are two shared variables `v1` and `v2`, each initialised to either `0` or `1` in a random way.

```
P1:
while(1) {
    while(v1 == v2);
    critical_region1();
    v1 = v2;
    non_critical_region1();
}
```

```
P2:
while(1) {
    while(v1 != v2);
    critical_region2();
    v2 = 1 - v1;
    non_critical_region2();
}
```

   i. Briefly describe the operations of the algorithm, giving particular emphasis to any dependencies between the two processes. Explain why the algorithm does enforce strict alternation. [6 marks]

   ii. This algorithm does not comply with one of the conditions required for solving the mutual exclusion problem. Indicate which one, and give one example of execution order violating that condition. [5 marks]

   iii. Write a revised version of the program using a semaphore. Your solution should fully comply with the conditions required for solving the mutual exclusion problem. You can use pseudo-code in your answer. [5 marks]

3. (a) What is a File Allocation Table (FAT) in the context of file systems? Give one advantage of using a FAT over a linked-list implementation of files. [4 marks]

   (b) Consider a virtual memory system with 16-bit virtual addresses, 14-bit physical addresses, and page size 2K. Briefly explain how the translation from virtual to physical addresses is carried out by the MMU in this specific context. Your explanation should consider the case in which the target page is not in physical memory. [8 marks]

   (c) Consider the following C program

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main() {
5    int x = 3;
6
7    if (!fork()) {
8      x = 5;
9    }
10
11   printf("&x=%p, x=%d\n", (void *) &x, x);
12 }
```

   i. Briefly describe the operations performed by the Linux operating system when line 7 is executed. Indicate which process data structures are created and how they are initialised. [7 marks]

   ii. Suppose that a run of the program produces the following output.

```
&x=0x7fff6ee32544, x=5
&x=0x7fff6ee32544, x=3
```

   Explain why the values of `&x` are the same in the two lines, whereas those of `x` are different. Your explanation should refer to how the address spaces of the involved processes are modified by the program.

   [6 marks]

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 10

void copyString(char *in, char *out) {
 int i = 0;
 while (*(in +i) != '\0' && i < SIZE - 1){
  *(out + i) = *(in + i);
  i++;
 }
 *(out + i ) = '\0';
}

int main() {
 char *s = malloc(SIZE);
 char *t = "CS2850 Operating Systems";
 copyString(t, s);
 printf("s=%s\n", s);
 printf("t=%s\n", t);
}
```

Figure 1: A dynamically allocated string: C-code

4. **A dynamically allocated string (15 marks)** Read the program in Figure 1 and answer the following questions.

   (a) Briefly describe what the program is supposed to do and explain why we can say that the string is dynamically allocated. [5 marks]

   (b) Why can you pass the input and output strings to `copyString` without using the address operator, `&`? How does the main program know that the content of `s` has changed? May the fact that `SIZE` is smaller than 25 (the length of `t`) cause segmentation errors? [3 marks]

   (c) What is the output of the program if you add

   ```
   char c1 = *(s + 3) - '2';
   char c2 = *(s + 3 - 2);
   printf("c1=%d, c2=%c\n", c1, c2);
   ```

   at the end of the provided code? Justify your answer by explaining what is the different between the first and second statements. **Hint:** remember

that, in ASCII, digits are encoded one after the other in increasing order, e.g. $ASCII('7') = ASCII('6') + 1$. [5 marks]

(d) Would the program compile without errors using

```
gcc -Werror -Wall -Wpedantic
```

and do you expect Valgrind to produce any warning messages? Justify your answers. [2 marks]

5. **Interactive sum (20 marks)**

(a) The UNIX shell is an interactive program that allows the user to communicate with their machine. Explain how you could implement a prototype of the UNIX shell in C using `fork()`. You do not need to write a complete C code to answer this part of the question. [5 marks]

(b) The program below is supposed to mimic the behaviour of the UNIX shell and compute the sum of an arbitrary number of integers provided by the user one after the other. For each character in the user input, the parent process should generate a child process that

- tests if the character is a digit, e.g. `'2'` or `'8'`,
- adds the corresponding values to the total (or 0 if the character is not a digit),
- returns the updated value of the total.

The parent should wait for the child to terminate, decode the exit value of the child, and update the value of the total accordingly. Complete the C code in Figure 2 without changing its structure, e.g. the final version should have the same number of lines as the template, and without modifying the given parts of the statements. [10 marks]

(c) Change `readAndSum` so that the new version

- checks if the input character is a lower case letter
- adds to the total the ASCII value of the corresponding upper case letter.

Hint: ASCII lower-case letters come just after the upper-case letters (both in alphabetic order). As a consequence, the difference between the ASCII value of any lower-case letter and the corresponding upper-case letter is a constant. [5 marks]

```
#include <stdio.h>
#include ...
#include <unistd.h>

void readAndSum(int *sum, char *c) {
 if ('0' < *c && *c <= '9')
  *sum = *sum + *c - '0';
}

int main() {
 int sum = ... ;
 int status;
 char c = '\0';
 while (c != '=') {
  while((c = getchar()) != '=') {
   if (... fork()) {
    readAndSum(...);
    return sum;
   }
  wait(...);
  sum = WEXITSTATUS(status);
  }
 }
 printf("sum=...\n", sum);
}
```

Figure 2: Interactive sum: C-code template

NEXT PAGE

END