System and Device Programming
Standard Exam
04.09.2023

Ex 1 (1.5 points)

Suppose that the following program is run using the command
./pgrm 2
Indicate which are the possible outputs generated. Note that more than one response can indeed be correct and that incorrect answers may imply a penalty on the final score.

```
#define N 100

int main (int argc, char *argv[]) {
  int n;
  char str[N];
  n = atoi (argv[1]);
  setbuf(stdout,0);
  while (n>0 && !fork()) {
    fprintf (stdout, "F");
    if (fork()) {
      fprintf (stdout, "E");
      sprintf (str, "%d", n-1);
      execlp (argv[0], argv[0], str, NULL);
    } else {
      sprintf (str, "echo -n S");
      system (str);
    }
    n--;
  }
  return 1;
}
```

Choose one or more options:

1. ⬜ FFFEEESSS
2. ⬜ FEFEFESSS
3. ⬜ FEFESFESS
4. ⬜ FSFSFSEEE
5. ⬜ FESFSEFES
6. ⬜ FESFEEFSS
7. ⬜ FSEFSSFEE
8. ⬜ FSFSEEFSE

Ex 2 (1.5 points)

Suppose to run the following program. Indicate which are the possible outputs generated. Note that more than one response can indeed be correct and that incorrect answers may imply a penalty on the final score.

```
typedef struct cond_s {
  pthread_mutex_t lock;
  pthread_cond_t cond;
  int count;
  int flag;
} cond_t;
```

```
static void *TA (void *args) {
  cond_t *cond_d = (cond_t *) args;
  while (1) {
    pthread_mutex_lock (&cond_d->lock);
    cond_d->count--;
    printf ("%d ", cond_d->count);
    if (cond_d->count <= 0) {
      cond_d->flag = 1;
      pthread_cond_signal (&cond_d->cond);
      pthread_mutex_unlock (&cond_d->lock);
      break;
    }
    pthread_mutex_unlock (&cond_d->lock);
  }
  pthread_exit(0);
}

static void *TB (void *args) {
  cond_t *cond_d = (cond_t *) args;
  pthread_mutex_lock (&cond_d->lock);
  while (cond_d->flag == 0) {
    pthread_cond_wait (&cond_d->cond, &cond_d->lock);
    printf ("(%d) ", cond_d->count);
    cond_d->count--;
  }
  pthread_mutex_unlock (&cond_d->lock);
  pthread_exit(0);
}

int main () {
  cond_t cond_d;
  pthread_t tid1, tid2;
  setbuf (stdout, 0);

  pthread_mutex_init (&cond_d.lock, NULL);
  pthread_cond_init (&cond_d.cond, NULL);
  cond_d.count = 10;
  cond_d.flag = 0;

  pthread_create (&tid1, NULL, TA, (void *) &cond_d);
  pthread_create (&tid2, NULL, TB, (void *) &cond_d);
  pthread_join (tid1, NULL);
  pthread_join (tid2, NULL);
  printf ("[%d]", cond_d.count);
  pthread_exit(0);
}
```

Choose one or more options:

1. ⬜ 9 8 7 6 5 4 3 2 1 [-1]
2. ⬜ (9) 8 (7) 6 (5) 4 (3) 2 (1) 0
3. ⬜ 9 (8) 7 (6) 5 (4) 3 (2) 1 (0)
4. ⬜ 9 8 7 6 5 4 3 2 1 0 (0) [-1]
5. ⬜ 9 8 7 6 5 4 3 2 1 0
6. ⬜ 9 8 7 6 5 4 3 2 1 0 [0]
7. ⬜ 9 8 7 6 5 4 3 2 1 [0]

Ex 3 (1.5 points)

Analyze the following code snippet. Indicate how many copy assignment operators and move assignment operators are called. Note that wrong answers imply a penalty in the final score.

```
class C {
private:
   ...
public:
   ...
};

int main() {
  C e1, e2;   default constructor twice
  e2 = e1;  copy assignment
  C e3 = *new C; default constructor with a copy constructor
  e3 = e2;   copy assignment
  e3 = std::move(e1);  move assignment
  return 0;
}
```

Choose one or more options:

1. ⬜ 1 copy assignment(s) and 1 move assignment(s).
2. ⬜ 1 copy assignment(s) and 2 move assignment(s).
3. ☑ 2 copy assignment(s) and 1 move assignment(s).
4. ⬜ 2 copy assignment(s) and 2 move assignment(s).
5. ⬜ 2 copy assignment(s) and 3 move assignment(s).
6. ⬜ 3 copy assignment(s) and 3 move assignment(s).
7. ⬜ 4 copy assignment(s) and 3 move assignment(s).

Ex 4 (3.0 points)

Implement in C++ a thread pool with the following characteristics. The program initially runs `N` threads (with `N` specified on the command line). All threads wait to solve tasks in a task set. Each task corresponds to a file name that stores square matrices of real values of variable size. Each working thread must:

- Read one matrix from a file. Each file storing a matrix has a name like `fileIn-K.txt` where `K` is in the range from 1 to M (i.e., `fileIn-1.txt`, `fileIn-2.txt` , `fileIn-3.txt`, etc.), and M is the number of tasks passed on the command line.
- Compute the determinant of the matrix. To perform such a computation, each thread can call the function determinant, which receives the matrix of real values, recursively computes the determinant of the square matrix, and returns its value. This function is supposed to be already part of the library (i.e., it does **not** have to be implemented):
  `double determinant (vector<vector<double>> &matrix);`
- Store in the output file `fileOut.txt` the value `k` (i.e., the index of the input file) and the determinant value (i.e., the real number returned by the function `determinant`).

Please, notice that:

- There are `M` input files (`fileIn-1.txt`, `fileIn-2.txt` , `fileIn-3.txt`, etc.), and must be read only once by a single working thread in the thread pool.
- There is only one output file (`fileOut.txt`) common to all working threads in the pool. This file must be accessed properly as each working threads write a single (subsequent) line of it (with no specific order).
- All files are in ASCII format.

The program must use a producer and consumer paradigm to implement the pool and adopt a `deque` to implement the task queue.

Ex 5 (2.0 points)

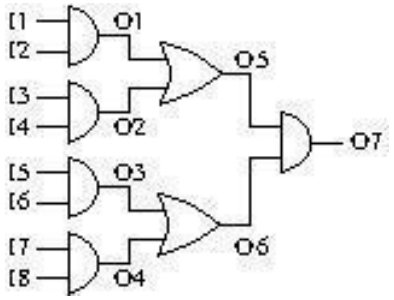
Describe how to manipulate dynamic memory in C++. More specifically, describe the use of the operators `new`, `delete`, and `make_share`, and the types `shared_ptr`, `unique_ptr`, and `weak_ptr`. Illustrate the meaning of RAII and the reason it has been introduced.

Ex 6 (2.0 points)

Describe the different techniques to implement multiplexing IO. Make an example of how to use the `select` system call, and clarify which are the advantages of the system call `select` with respect to the other possible approaches to multiplexing IO.

Ex 7 (3.5 points)

Consider the following circuit:



Realize a C++ program using tasks to compute the logic output value of the circuit (i.e., O7) when all input signals (I1, …, I8) are given. All signal values are Boolean, and AND and OR gates respect the standard Boolean logic. All tasks communicate with **promises** and **futures**. Use two tasks, one for the AND gate (instantiated five times) and one for the OR gate (instantiated two times).