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;
}
```

Solution

Three F, three E, and three S with the F always coming first than an E or an S.

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);
}
```

Solution

TB can start after and never start the cycle; thus (0) can be displayed or not.

```
9 8 7 6 5 4 3 2 1 0 (0) [-1]
9 8 7 6 5 4 3 2 1 0 [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;
  e2 = e1;
  C e3 = *new C;
  e3 = e2;
  e3 = std::move(e1);
  return 0;
}
```

Solution

{1}[C][C]{2}[CAO]{3}[C][CC]{4}[CAO]{5}[MAO]{6}[D][D][D]

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.

Solution

DRAFT

```
#include <thread>
#include <vector>
#include <deque>

using namespace std;

deque<string> task_queue;
sem_t sem;
mutex m;

// --------------------
int main(int argc, char **argv){
  int M = atoi(argv[1]) // get M
  if(argc < M+2){
    cerr << "Wrong number of argument";
    exit(0);
  }
  // filling the task_queue with file
  for(int i = 0; i < M; i++){
    task_queue.push_back(argv[i+2]);
  }
  vector<thread> vec;
  for(int i = 0; i<M; i++){
    vec.emplace_back(thread(th_funct, argv[M+1]));
  }
  return 0;
}

// --------------
void th_funct(string filename, string output_file){
  sem_wait(&sem);
  istream fin(filename);
  ofstream fout(output_file);
  vector<vector<double>> mat;
  double val;
  int i, j;

  while(fin >> val){
    mat[i][j] = val;
  }
  unique_lock<mutex> l_wr(m);
  double res = determinant(mat);
  fout << res;
```

```
}
```

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.

Solution

Dynamic memory allocation in C++ is different than C: it is still possible to use the legacy 'malloc()' and 'free()' functions, but some new operators exist (‘'new' and 'delete').
'new' takes the name of a class (or primitive type) and other operators (in case we want to allocate space for an array or a pointer or a matrix) and returns the pointer to that object (or to the first element of the array).
'delete' deallocates the memory previously acquired by 'new'. We need to be careful when using dynamic memory allocation as arrays are not automatically recognized when calling 'delete' (delete[] should be used).
This method of memory allocation can throw exceptions in case of errors if it is not otherwise specified with the 'nothrow' keyword.
When allocating memory with 'new' and working with raw pointers it's essential to remember that the destructor for the new object will not be automatically called once the current scope has ended, and it is the programmer's responsibility to free the memory using 'delete' once the object is no longer needed.

To help in the management of pointers, the types 'shared_ptr', 'unique_ptr', and 'weak_ptr' have been introduced:

- 'unique_ptr' is used to work with pointers that will not be duplicated, so that once the destructor to this pointer is called, the memory used for the object pointed to by it will be freed;
- 'shared_ptr' works in a similar way, but it allows the object it points to be referenced by other shred or weak pointers. 'shared_ptr' keeps a counter for every other pointer referencing the object and, once all pointers have been removed and the object is no longer in use, the memory is freed;
- 'weak_ptr' is a kind of shared_ptr that does not modify the count of pointers to an object.

RAII stands for Resource Allocation Is Initialization, and it is a coding technique that aims to resolve the problems brought by pointer handling and memory management:
the main concept is that resources, whenever allocated, need to be initialized, together with a way to dispose of them once we are done with it (a destructor) that is called automatically once that resource is no longer used by the process.
The three kinds of pointers present in C++ have been introduced to follow the concept of RAII.

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.

Solution

There are different techniques to implement multiplexing I/O. The easiest one is to use threads or fork a process and make the parent and the child read from two different files. This solution is not a good implementation of multiplexing and the IO instructions are blocking. So another solution could be using Non-Blocking IO, opening the files with the NON_BLOCKING "mode" or making the accesses to that files non-blocking using the fnctl function. This technique is not very good either.
Another solution is to implement asynchronous IO using signals. Whenever an IO instruction can be executed the process is notified and blocked until the IO function is completed.
Using the select system call is the best solution to multiplexing. With the select function we can group together all the pointers to the files that we are interest in for reading/writing and then thanks to the FD_ISSET function we can check if we are ready to perform the IO instruction on single file in a non-blocking manner.

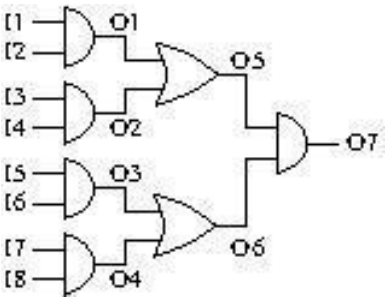
Example of how to use select system call:

```
fd_set rfds;
int maxfdp1, fd1, fd2, nbytes;
char *buf;
```

```
fd1 = open("file1", 666);
fd2 = open("file2", 666);
if(fd1 > fd2)
      maxfdp1 = fd1 + 1;
else
      maxfdp1 = fd2 + 1;
FD_ZERO (rfds);
FD_SET (fd1, rfds);
FD_SET (fd2, rfds);
select (maxfdp1, rfds, NULL, NULL, NULL, NULL);
while(...) {
   if(FD_ISSET(fd1, rdfs)) {
      // read from fd1
      read (fd1, buf, nbytes);
      ...
   }
   if(FD_ISSET(fd2, rdfs)) {
      // read from fd2
      read (fd2, buf, nbytes);
      ...
   }
   ...
}
```

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).

Solution 1

```
#include <iostream>
#include <future>
#include <vector>

using namespace std;

bool f_and (bool in1, bool in2) {
  bool out = in1 & in2;
  cout << in1 << " AND " << in2 << " = " << out << endl;
  return out;
}
```

```
bool f_or (future<bool> s1f, future<bool> s2f) {
  bool in1 = s1f.get();
  bool in2 = s2f.get();
  bool out = in1 | in2;
  cout << in1 << " OR " << in2 << " = " << out << endl;
  return out;
}

int main () {
  bool in1 = false;
  bool in2 = true;
  bool in3 = true;
  bool in4 = true;
  bool in5 = false;
  bool in6 = false;
  bool in7 = false;
  bool in8 = false;

  future<bool> s1f = async (f_and, in1, in2);
  future<bool> s2f = async (f_and, in3, in4);
  future<bool> s5f = async (f_or, move(s1f), move(s2f));

  future<bool> s3f = async (f_and, in5, in6);
  future<bool> s4f = async (f_and, in7, in8);
  future<bool> s6f = async (f_or, move(s3f), move(s4f));

  bool s5p = s5f.get();
  bool s6p = s6f.get();

  future<bool> s7f = async (f_and, s5p, s6p);

  bool out = s7f.get();
  cout << "OUT = " << out << endl;

  return 0;
}
```

Solution 2

```
#include <iostream>
#include <future>

using namespace std;

void and_f(bool a, bool b, promise<bool> res) {
    res.set_value(a && b);
}
```

```
void or_f(bool a, bool b, promise<bool> res) {
    res.set_value(a || b);
}

int main() {
    bool i[8] = {true, false, true, true, true, true, false, false};

    promise<bool> pr[7];
    future<bool> o[7];
    for (int j = 0; j < 7; ++j) {
        o[j] = pr[j].get_future();
    }

    thread t[7];
    t[0] = thread(and_f, i[0], i[1], std::move(pr[0]));
    t[1] = thread(and_f, i[2], i[3], std::move(pr[1]));
    t[2] = thread(and_f, i[4], i[5], std::move(pr[2]));
    t[3] = thread(and_f, i[6], i[7], std::move(pr[3]));
    t[4] = thread(or_f,  o[0].get(), o[1].get(), std::move(pr[4]));
    t[5] = thread(or_f,  o[2].get(), o[3].get(), std::move(pr[5]));
    t[6] = thread(and_f, o[4].get(), o[5].get(), std::move(pr[6]));

    cout << "result is " << o[6].get() << endl;

    for (auto &j: t) {
        j.join();
    }

    return 0;
}
```