System and Device Programming
Standard Exam
20.06.2023

Ex 1 (1.5 points)

Suppose the following program is run using the command:
`./pgrm 2`
Indicate a possible program output.

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

int main (int argc, char *argv[]) {
  int i, n;
  char str[40];

  n = atoi (argv[1]);

  for (i=1; i<=n; i++) {
    if ( fork() == 0) {
      sprintf (str, "%d", n-1);
      execlp (argv[0], argv[0], str, NULL);
    }
  }

  printf ("%d", n);
  fflush (stdout);

  exit (0);
}
```

Choose one or more options:

1. ✅ 21100
2. ⬜ 221100
3. ✅ 01012
4. ⬜ 210210
5. ⬜ 110022
6. ✅ 11200
7. ⬜ 2110

Ex 2 (1.5 points)

Indicate the possible output, or outputs, that can be obtained by concurrently executing the following processes PA, PB, and PC with the reported semaphore initialization.

```
init (S1, 0);
init (S2, 1);
```

PA	***PB***	***PC***
`wait(S1);`	`wait(S2);`	`wait(S2);`

```
printf("A");
signal(S2);
wait(S2);
printf("B");
wait(S1);
printf("C");
```

```
printf("D");
signal(S1);
```

```
printf("E");
signal(S1);
```

Choose one or more options:

1. ⬜ DABC
2. ⬜ DABCE
3. ⬜ EABC
4. ✅ DAE
5. ⬜ EABCD
6. ⬜ DAD
7. ⬜ EAE
8. ⬜ DAEBC
9. ✅ EAD
10. ⬜ EADBC

Ex 3 (1.5 points)

Analyze the following code snippet in C++. When the main is executed, indicate how many (standard) constructors, copy constructors, and destructors are called.

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

void f1(C e) { ... }
void f2(C &e) { ... }

int main() {
   C e1, e2;
   f1(e1);
   f2(e2);
   C *e3 = new C;
   return 0;
}
```

Choose one or more options:

1. ✅ 3 constructors, 1 copy constructor, and 3 destructors.
2. ⬜ 1 constructor, 3 copy constructors, and 3 destructors.
3. ⬜ 3 constructors, 1 copy constructor, and 4 destructors.
4. ⬜ 2 constructors, 2 copy constructors, and 4 destructors.
5. ⬜ 3 constructors, 2 copy constructors, and 3 destructors.
6. ⬜ 1 constructor, 2 copy constructors, and 3 destructors.

Ex 4 (2.5 points)

Describe lambda expressions and why they have been introduced in C++ (which problem do they solve?). Illustrate the meaning of the capture list, the parameter list, the return type, and the body. Report some examples to illustrate these features.

Solution

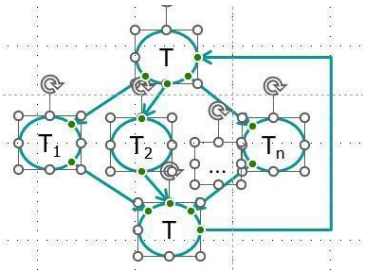
Lambda expressions are callable objects that represent a simple way to define an "inline" function. In some cases, especially when making use of generic algorithms (e.g., sort on a container of objects), we want to pass a function as a parameter in order to specify how a specific action must be performed inside another function; this function that we define doesn't need to have a name, and it doesn't need to be declared and stored in the program indefinitely, like normal function (although we can store lambda functions inside variables if we want). Lambda functions provide a simple syntax to define inline, nameless functions that may be stored in a variable, be called later, or passed to a function that will call them.
They are defined with the following syntax:

[capture_list(close square) (parameters) -> return_type { body (close curly)

The capture list can be used to "capture" variables from the enclosing scope of the lambda in order to use them inside the body. They can be passed by value (i.e., simply specifying their name), which means that the value used inside the lambda will always be the same and equal to the one that the variable had at the moment of the lambda's creation or by reference (prepending a '&' before the name), so that the value used inside the body is always the same as the actual value of the variable, even when it changes. The capture list must always be specified. We can quickly capture all variables by reference by just including a '&' inside the brackets or '=' to capture them all by value; we can even add some extra variables after these symbols to make exceptions (ex. &, v1, v2 captures all variables by reference except for v1 and v2).
The parameter list is the list of the parameters taken by the lambda.
The return type specifies which type is returned by the lambda. It can be omitted if the lambda doesn't return anything (void) or if it only has 1 return statement in the body (the compiler can infer the return type); if more than 1 return statement is included, then it is mandatory.
Finally, the body contains the actual instructions of the lambda and is precisely the same as for everyday functions.

Ex 5 (3.0 points)

Use barriers implemented with semaphores, mutexes, and counters to implement the following synchronization scheme:



In the previous figure, T and all T_i are different cyclic (i.e., running a primary cycle) threads.

Solution

```
#include <iostream>
#include <thread>
#include <mutex>
#include <semaphore.h>

#define NTHREAD 5

using namespace std;

int count;
sem_t sem;
sem_t sem2;
```

```
mutex m;
sem_t sem_T;

// Thread T
void fT() {
    while (true) {
        sem_wait(&sem_T);
        // T doing something...
        {
            lock_guard<mutex> lock(m);
            cout << "Thread T run complete" << endl;
            for (int i = 0; i < NTHREAD; ++i)
                sem_post(&sem2);
        }
    }
}

// Thread Ti
void fTi(int id) {
while (true) {
// Wait for thread T to unlock me
    sem_wait(&sem2);
    // Ti doing something...
    // Barrier 1
    {
        lock_guard<mutex> lock(m);
        cout << "Thread " << id << " run complete" << endl;
        ++count;
        if (count == NTHREAD) {
            for (int i = 0; i < NTHREAD; ++i)
                sem_post(&sem);
        }
    }
    sem_wait(&sem);
    // Barrier 2
        {
            lock_guard <mutex> lock(m);
            count--;
            if (count == 0) {
                sem_post(&sem_T);
            }
        }
    }
}

int main() {
    count = 0;
    sem_init(&sem, 0, 0);
    sem_init(&sem2, 0, 0);
    sem_init(&sem_T, 0, 1);

    thread T, Ti[NTHREAD];
    T = thread(fT);
    for (int i=0; i<NTHREAD; ++i)
        Ti[i] = thread(fTi, i+1);
```

```
    for (auto & i : Ti) // never reached
        i.join();
    T.join();

    return 0;
}
```

Ex 6 (3.0 points)

Use tasks promises and future to implement three threads T_1, T_2, and T_3 executed directly by the main thread such that:

- T_1 reads a string from standard input, transfers that string to T_2, and terminates.
- T_2 transforms the string received from T_1 into capital letters, transfers it to T_3, and terminates.
- T_3 displays the string received from T_2 on standard output and terminates.

Indicate how it is possible to make the three threads cyclic, i.e., repeat the process reported in the previous itemization (the C code in not necessary, but it is requested a description in English language).

Solution

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

using namespace std;

string f1() {
    string a;
    cout << "Enter a string >";
    cin >> a;
    return a;
}

string f2(const string &a) {
    string b;
    for (char c: a)
        b += static_cast<char>(toupper(c));
    return b;
}

void f3(const string &a) {
    cout << a << endl;
}

int main() {
    promise<string> p1, p2;
    auto fut1 = p1.get_future();
    auto fut2 = p2.get_future();

    thread T1([&p1] { p1.set_value(f1()); });
    thread T2([&p2, &fut1] {p2.set_value(f2(fut1.get()));});
    thread T3([&fut2] {f3(fut2.get()); });

    T1.join();
    T2.join();
    T3.join();
```

```
    return 0;
}
```

To make the thread cyclic, if the number of iteration is known, one may use vector of future/promises and suitable forms of synchronization (e.g., semaphores or mutexes and condition variables).
In the case of an indefinitely cyclic thread (e.g., while (true) {...}), one must make sure that a new future and new promises are created at each cycle since it is not possible to set the value of a promise more than once. Otherwise, other forms of communication between threads may be used.
CYCLIC THREAD VERSION WITH PROMISE-FUTURE

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

using namespace std;

mutex m1, m2, m3;
condition_variable cv1, cv2, cv3;
bool ready1, ready2, ready3;

void f1(shared_ptr<promise<string>> &p, shared_ptr<future<string>> &f) {
    bool first = true;
    string a;
    while (a != "exit") {
        unique_lock<mutex> lock(m3);
        while (!ready3)
            cv3.wait(lock);
        if (!first) {
            p.reset(new promise<string>());
            auto ft = p->get_future();
            f.reset(new future<string>(std::move(ft)));
        }
        ready3 = false;
        ready1 = true;
        cv1.notify_one();

        cout << "Enter a string >";
        cin >> a;
        p->set_value(a);
        first = false;
    }
}

void f2(shared_ptr<promise<string>> &p, shared_ptr<future<string>> &f,
shared_ptr<future<string>> &f2) {
    bool first = true;
    string b;
    while (b != "EXIT") {
        unique_lock<mutex> lock(m1);
        while (!ready1)
            cv1.wait(lock);
        if (!first) {
            b.clear();
            p.reset(new promise<string>());
            auto ft = p->get_future();
            f2.reset(new future<string>(std::move(ft)));
        }
```

```
        ready1 = false;
        ready2 = true;
        cv2.notify_one();

        for (char c: f->get())
            b += static_cast<char>(toupper(c));
        p->set_value(b);

        first = false;
    }
}

void f3(shared_ptr<future<string>> &f) {
    string a;
    while (a != "EXIT") {
        unique_lock<mutex> lock(m2);
        while (!ready2)
            cv2.wait(lock);
        a = f->get();
        cout << a << endl;
        ready2 = false;
        ready3 = true;
        cv3.notify_one();
    }
}

int main() {
    auto p1 = make_shared<promise<string>>();
    auto p2 = make_shared<promise<string>>();
    auto ft1 = p1->get_future();
    auto ft2 = p2->get_future();

    auto fut1 = make_shared<future<string>>(std::move(ft1));
    auto fut2 = make_shared<future<string>>(std::move(ft2));

    ready1 = false;
    ready2 = false;
    ready3 = true;
    thread T1(f1, ref(p1), ref(fut1));
    thread T2(f2, ref(p2), ref(fut1), ref(fut2));
    thread T3(f3, ref(fut2));
    T1.join();
    T2.join();
    T3.join();
    return 0;
}
```

Ex 7 (2.0 points)

Indicates the main differences between FIFO, message queues, and sockets. Describe when it is better to use each one of the methods instead of the others and which are the main steps to implement it (the C code in not necessary, but it is requested a description in English language).

Solution

FIFOs are an extension of traditional pipes and are sometimes called named pipes. They allow communication among unrelated processes and last as long as the system does. They can be deleted if no longer used. A

FIFO is a type of file (in the local storage), and creating a FIFO is similar to creating a file (with a pathname in the filesystem). Once a FIFO has been created, processes can open it and perform R/W operations on it.

FIFOs (and pipes) are used to pass streams of anonymous bytes, whereas with a message queue, it is possible to pass structured data. A message queue is a linked list of messages; messages are stored within the kernel; the queue is identified through an identification key. We must generate a key with the system call ftok, and then use msgget, msgsnd, and msgrcv to create or open a message queue, msgsnd and receive messages.

Sockets allow communication between processes running on different computers (connected to a common network). They can be used to communicate using many different network protocols (e.g., TCP/IP). To use sockets we must create them (system call socket), create a connection between the socket of the process requesting the service (the client) and the process providing the service (the server) (system call connect), and accept the connection (bind and listen).