**Politecnico di Milano**
**SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE**

**Advanced Operating Systems**
**A.A. 2017-2018 – Exam date: July, 19th 2018**

**Prof. William FORNACIARI**

Surname (readable)..... Name (readable).....

Matr..... Signature.....

| Q1 | Q2 | TOT |
|---|---|---|
| | | |

**NOTES**

**It is forbidden to refer to texts or notes of any kind as well as interact with their neighbors. Anyone found in possession of documents relating to the course, although not directly relevant to the subject of the examination will cancel the test. It is not allowed to leave during the first half hour, the task must still be returned, even if it is withdrawn. The presence of the writing (not delivered) implies the renunciation of any previous ratings.**

## Question Q1 (12 points)

You are correcting an exercise of an AOS exam. The exercise is the following:

A C++ module is composed of two files: **priorityexecutor.h** and **priorityexecutor.cpp**. These two files contain a single C++ class, named `PriorityExecutor` that must have at least the following public interface:

```
class PriorityExecutor
{
public:
        PriorityExecutor();

        void execute(std::function<void ()> task, bool highPriority);

        ~PriorityExecutor();
};
```

This class should contain a thread that executes the tasks passed to it through the `execute()` member function. Tasks have two priority levels, high and low. The tasks must not be executed directly in the `execute()` member function, but rather passed to the background thread. The background thread should execute each task in a run-to-completion fashion, and when it has finished running a task, should both high and low priority tasks be waiting for execution, must always run all the higher priority tasks first. The destructor must wait for the (eventual) current task to complete and then join the thread immediately disregarding (eventual) other tasks.

A student has provided the following solution:

**priorityexecutor.h**

```
class PriorityExecutor {
private:
        std::vector<bool> v_priority;
        std::vector<std::function<void()>> v_task;
        bool end;
        std::mutex m;
        void start();
public:
        PriorityExecutor();
        ~PriorityExecutor();
        void execute(std::function<void()> task, bool highPriority);
};
```

**priorityexecutor.cpp**

```
#include “priorityexecutor.h”

using namespace std;

PriorityExecutor::PriorityExecutor() {
	end=false;
	thread t(start);
	t.join();
}

PriorityExecutor::~PriorityExecutor() {
	end=true;
}

void PriorityExecutor::start() {
	bool p=false;
	while(!end) {
		p=false;
		for(int i=v_priority.begin();i<v_priority.size() && !p && !end;i++)
			if(v_priority[i]==true) {
				v_task[i];
				m.lock();
				v_task.erase(i);
				v_priority.erase(i);
				m.unlock();
				p=true;
			}
		} else {
			v_task[0];
			m.lock();
			v_task.pop_front();
			v_priority.pop_front();
			m.unlock();
		}
	}
}

void PriorityExecutor::execute(std::function<void()> task, bool highPriority) {
	m.lock();
	v_priority.push_back(highPriority);
	v_task.push_back(task);
	m.unlock();
}
```

Find the functional and performance errors in the student solution. You are not required to rewrite the entire program, just to pinpoint the issues, motivate them and, if applicable, suggest an alternative.

Although the examples of concurrency errors provided in the slides are very structured, having one error at a time and “put there on purpose”, real world errors, especially from inexperienced programmers are unfortunately more confused.
For simplicity the errors are categorized in classes:

- perf: performance errors, slow the program and cause excessive resource usage
- corr: correctness error, make the program not produce correct output or make it crash
- comp: errors that cause the program not to compile
- conc: concurrency errors

Here is the list of errors

- corr: the “thread t(start)” instantiates a thread object as a local variable in the constructor. Since the thread must be running also when the constructor ends, the thread must be declared at class level
- conc: the join of the thread must be moved in the destructor, otherwise the caller of the constructor will deadlock
- conc: when the missing condition variable is added, a dummy task must pushed in the queue in the destructor to prevent deadlock in the join
- conc: the end variable is not an atomic<bool>, so its use must be guarded by a mutex, both in the destructor and in start There is no problem in the constructor, since the variable is accessed before the thread is created, that is, before concurrent accesses are possible
- comp: priority.begin() retuns an iterator, not an int. Moreover, vector<T>::erase() only accepts an iterator, so the loop must be converted to using iterators to compile. Converting it to int will not work
- corr,conc: the statemets v_task[i]; and v_task[0]; do not call the function! Moreover, they access a shared variable while the mutex is not locked. Simply locking the mutex before calling the function would not work, as it would create a performance issue by preventing execute() to be called concurrently. The only solution would have been to lock the mutex, take a copy of the function in the shared variable in a local variable, unlock the mutex and then call the function from the local variable
- perf: removal from front and middle of a vector is expensive as it requires internally to shift all the other elements. A list should have been used instead of a vector
- perf: the solution scans the entire vector every time to find if there are high priority tasks. Using two lists of functions, one for low priority and one for high priority would improve performance

- perf, conc: the bakground thread spins if not task is in the queue, causing 100% CPU utilization doing no useful work. A condition variable is required
- corr: if the vector is empty, the code accesses v_task[0], a nonexistent element, this would crash the program

## Question Q2 (11 points)

1. Why operating systems provide Inter-Process Communication (IPC) mechanisms? What is the purpose of IPC?
2. List all the IPC mechanisms shown during the course, providing a description and discussing advantages and drawbacks for each of them
3. Given the picture below, assume we have several *log generator* processes producing text data including tens of characters. We want to have GUI based program (*system log viewer*) to collect all the logs coming from the different processes and show them in a single window. Which IPC mechanism would you use? Motivate your choice.