Exercise 2: Concurrency

The purpose of the exercise is to learn about concurrency, which is important within real-time systems. Both processes and threads will be introduced, as well as important mechanisms for synchronization of processes and threads. Start Linux Mint.

Remember the `man` program, for the first part of this exercise you will use the function *fork()* so try:

```
$ man fork
```

1. Processes and Threads

A uniprocessor computer can only do one thing at the time, but since it is able to execute operations extremely fast it can share its time between different processes and threads. This is called concurrency. To switch between two processes or two threads is called context switching and is performed fast enough so it appears as they are running at the same time.

In a multiprocessor system (dual/quad core) two processes or threads can run at the same time on different cores. This allows for more "true" concurrency.

1.1. Processes

A process is an instance of a computer program. It consists of the program code and its current state. In an normal modern operating system will have many processes, but only one (or several in a multicore system) will be running at any given time.

To create a new process in a Linux C-program, you can use `fork()` or `vfork()`.

Assignment A:

Create a program which does the following:

- Contains a global and local variable.
- Creates another "instance" of itself, using `fork()`.
- Both the parent and child process should then increment each of the two variables a number of times.
- Print the variables from both processes.

Run the program with `fork()` and then with `vfork()`. Why are the results different?

1.2. POSIX Threads

Each process can contain several threads that can execute their code in parallel. Threads are defined by the POSIX standard and the threads we used in Linux are called POSIX threads. Threads within the same process share resources like memory, which means that communication between threads are simpler and that there is less overhead when switching between threads.

Although threads will be used for the remainder of this course, it is important that you understand the concept of processes, and the difference between processes and threads.

Assignment B:

You should create a program similar to the program in assignment A, just using POSIX threads instead of processes. Be aware that while `fork()` makes a copy of the running process, `pthread_create()` makes a thread that runs a specified function. The program should create two threads that both increment one local and one global variable and then print the results. The main function should wait until the threads complete using `pthread_join()`. Explain the results.

You must add `-pthread` to the linker flags in your `makefile` to link your program with the POSIX thread library.

2. Semaphores

A semaphore is a counter that keeps track of the number of available resources or a particular type. Whenever a thread starts using one of the resources the semaphore is decremented. It is then incremented when the thread is no longer using the resource. If the counter in the semaphore is zero, there are no available resources and any threads that need to use a resource must wait.

POSIX semaphores are defined by semaphore.h, and the most important functions are `sem_init()`, `sem_wait()` and `sem_post()`.

Assignment C:

Create a program that does the following:

- Define a semaphore that represents a 3 resource that the threads can use.
- Create 5 threads that each want to access one of the resources.
- When a thread is allowed access to one of the resources, only this thread can use this resource until it is done.
- You only need to set up the semaphore protecting an imaginary resource; you don't need to implement the resources (unless you really want to of course). You can run the following pseudocode which represents accessing a resource.

```
for i in 1 to 4 loop
     print "Thread number " + i;
```

```
     usleep(100000);
end loop
```

Since there are more threads than resources, the semaphore in the program must make sure that only the allowed number of threads is allowed to use the resources.

3. Mutex

A mutex is a binary semaphore that only can be unlocked by the same thread that locked it. This makes it ideally suited to avoid simultaneous use of a common resource like a global variable or I/O.

In POSIX mutexes are defined by pthread.h, and help for using them should be easy to locate on the web.

Assignment D:

Create two POSIX threads that execute the following pseudocode:

```
bool running = 1;
int var1 = 0;
int var2 = 0;

thread1:
while (running) loop
    var1 = var1 + 1;
    var2 = var1;
end loop;
exit;

thread2:
for i in 1 to 20 loop
    print "Number 1 is " + var1 + ", number 2 is " + var2 + "\n";
    usleep(100000);
end loop;
running = 0;
exit;
```

The intention of the program is that the two global variables always have the same value, why is this not the case?

Try to use a mutex to fix the program by making sure that only one of the threads can access the two variables at the same time.

4. Deadlock

Deadlock is a problem that occurs when two (or more) threads are blocking each other. This can happen if thread A has locked mutex 1 and wants to lock mutex 2, while thread B has locked mutex 2 and wants to lock mutex 1.

Assignment E:

The dining philosophers' problem is a classic example of a deadlock. Read more about this problem on http://en.wikipedia.org/wiki/Dining_philosophers. You should create a program that implements the Dining Philosophers problem.

You should use mutexes to represent forks, and each philosopher should be represented by one thread. You should get a deadlock, where all philosophers end up with one fork.

Change your dining philosopher program so there are no deadlocks. There are several ways to do this; you can choose the solution you want.

Deadlocks can also exsist in the natural world….

5. Approving the exercise

For this exercise you need to show the following to a student assistant to get the exercise approved.

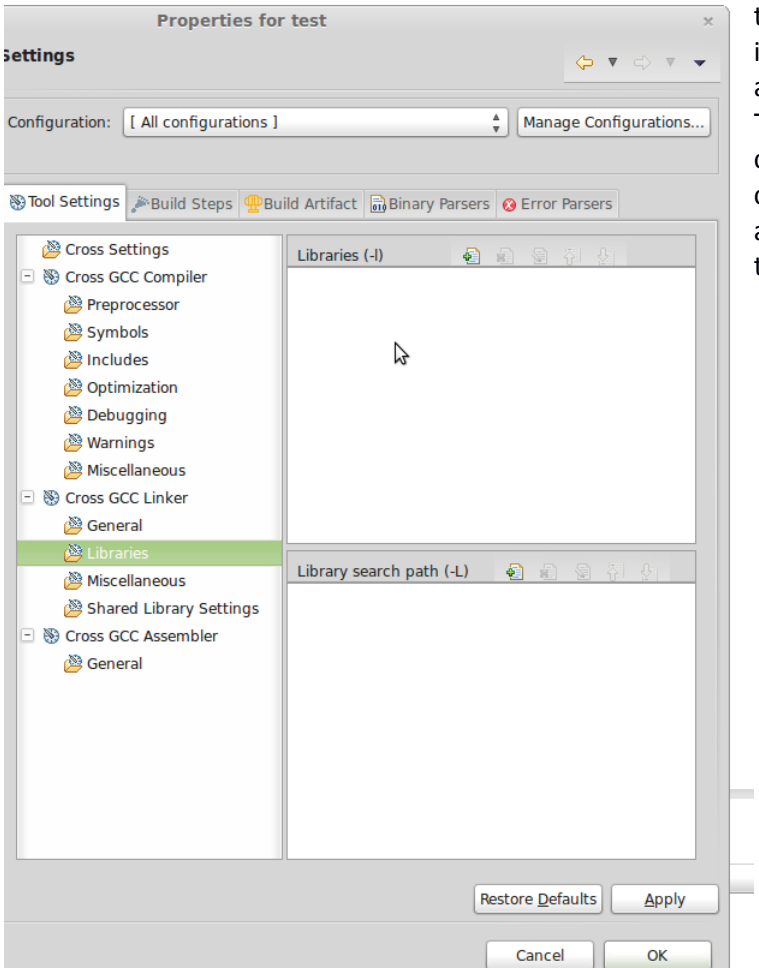
1. Run the programs from assignments A and B with fork, vfork and threads.
    - What is the difference between fork and vfork?
    - How do POSIX threads handle local and global variables?

2. Run the program from assignment C.
    - What happens with the threads that does not get access to a resource right away?
    - Are the resources shared fairly among the threads?
3. Run the program from assignment D.
    - What happens when running the threads before adding a mutex, and why?
4. Run the program from assignment E.
    - How did the deadlock happen?
    - How did you solve it?
    - Can you think of other methods to solve it?

6. Appendix: Eclipse Configuration

If you are using Eclipse you must add the same linker flags to the build configuration as are added to the Makefile. You need to open project properties, and navigate to the page as seen in the figure below. Add your library with the green + on a document icon right of "Libraries (-l)". When linking pthread, you should only write `pthread`, not include the `-l`. This same approach is used for other libraries as well.

Eclipse has a built in system for indicating build errors in the editor. Unfortunately, this system often misbehave when additional libraries are used, especially the rt library. Code that require the libraries are often shown as errors even after the library flags have been added. This does not interfere with the build process, and you will be able to build and run programs.



One method for (at least temporarily) hide these incorrect errors is to close and reopen the project. This can be done by right-clicking the project and choose "Close project", and then "Open project" to reopen.