Exercise 1, TTK4145

Knut Brekke and Martin Skow Røed

January 13, 2014

Part I
Why concurrency?

Concurrency lets us perform certain tasks faster as we can divide the tasks into independent subtasks, so they can be performed in parallel. If you have a big asynchronous task, you can run that in a separate thread, not having to deal with the pause/resuming of the task while other things are going on.

Processes runs completely independent and isolated of each other, and can **not** access any shared data. The operating system is allocating resources (memory, CPU time etc) to each indidivual process.

Threads has its own call stack, but they can access shared data of other threads in the same process. As each thread has its own calls tack, it reads the shared data and stores it in its own memory cache. A thread can read the shared data several times. This is the basis is a concurrency problem called **visibility**.

Thread **A** and **B** both runs in the same process and can access the shared data. If A reads shared data which is later changed by B without thread A being aware of the changes, we have a visibility problem.

The other main concurrencly problem is called an **access** problem. This is when several threads try to access and change the same shared data at the same time. Both problems can obviously lead to problems for a program without giving any hints as the program may be working with false or outdated data.

Green threads are user-level threads, not scheduled by the kernel, used to simulate multi-threading on platforms that does not provide that capability. **Fibers** are a lightweight thread. While threads depends on the kernels thread scheduler, fibers yield themselves to run another fiber while executing.

- **pthread_create()** starts a new thread in the calling process.
- **threading.Thread()** starts a new thread.
- **go** starts a coroutine, which is closely related to fiber.

The **Global Intepreter Lock** (**GIL**) is solving the access problem from before. Only the thread with the GIL may operate on objects or call API functions.

Without it, multi-threading can he a huge problem, for instance when two thread simultaneously want to increment the reference count, it could end up being incremented only once instead of twice. Because of GIL, only one thread can access it at the same time. The workaround for GIL is called **multiprocessing** which spawns processes using an API similar to how we use **threading**. Multiprocessing offers both local and remote concurrency and is not influenced by the GIL as its using subprocesses instead of threads. Processes as explained earlier, can not share data with each other. The multiprocessing module however, has two ways for the processes to communicate.

Queues are almost identical to a regular queue. One process puts information in, another process can fetch it. The queue is both process and thread safe.

Pipes offers connection objects connected by a pipe which is two-way by default. Each connection object has **send** and **recv** methods. You pass one end to the process, and read from the other end in the main thread. The data in the pipe may be corrupted if two processes try to read or write to the same end of the pipe simulatenously.

From **GOLANG**'s API, "GOMAXPROCS sets the maximum number of CPUs that can be executing simultaneously and returns the previous setting".

Part II
Code

```
Python:
______________
from threading import Thread
def a():
        global i
        for k in range(1000000):
                i += 1

def b():
        global i
        for k in range(1000000):
                i -= 1

i = 0
p = Thread(target=a)
pp = Thread(target=b)
p.start()
pp.start()
p.join()
```

```
pp.join()
print "i =", i
```

```
>> 2483
```

C

```
#include <pthread.h>
#include <stdio.h>
int global_increment = 0;
void *inc_global() {
        for (int i=0; i<100; ++i)
                global_increment++;
        return NULL;
}

void *dec_global() {
        for (int i=0; i<100; ++i)
                global_increment--;
        return NULL;
}

int main() {
        pthread_t increment_global;
        pthread_t decrease_global;
        if(pthread_create(&increment_global, NULL, inc_global, NULL)) {
                fprintf(stderr, "Error creating thread 1\n");
                return 1;
        }
        if(pthread_create(&decrease_global, NULL, dec_global, NULL)) {
                fprintf(stderr, "Error creating thread 2\n");
                return 1;
        }
        if(pthread_join(increment_global, NULL)) {
                fprintf(stderr, "Error joining thread 1\n");
                return 2;
        }
        if(pthread_join(decrease_global, NULL)) {
                fprintf(stderr, "Error joining thread 2\n");
                return 2;
        }
        printf("global = %d\n", global_increment);

        return 0;
}
```

```
global = -9
```