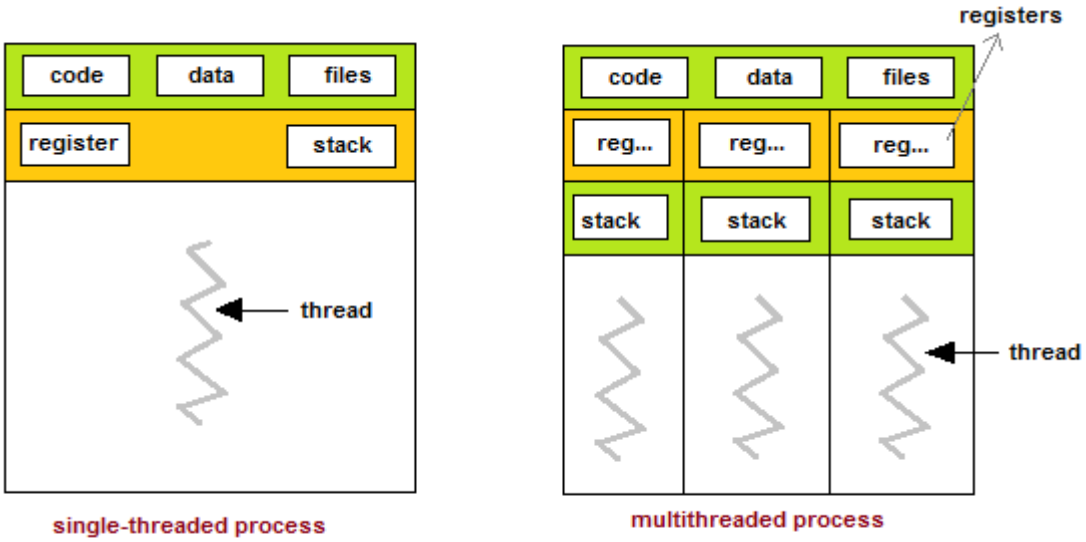
Your own Thread Library (**OTL**)

A quick google tells us that "A thread is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.)".



User threads, are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

Unix supports the POSIX standard --- in addition there are Win32 and Java Threads. This is, generally, referred to as **pthreads**, and is defined as an execution model that exists independently from a language, as well as a parallel execution model. A program can conveniently use it to control multiple different flows of work that overlap in time. For example, it allows overlapping of IO with computation. This link (https://computing.llnl.gov/tutorials/pthreads/) gives an overview of pthreads and its use.

In this assignment, you will create a user level thread level library which implies that it is supported above the kernel in user space and are managed without kernel support.

Do NOT use the PTHREADS library in any way in this assignment except to compare performance.

These are some of the functions that need to be implemented:

myThread_create (thread,attr,start_routine,arg)
myThread_exit (status)
myThread_cancel (thread)
myThread_attr_init (attr)
myThread_attr_destroy (attr)
myThread_t pthread_self(void);
myThread_yield(void)
myThread_join(int tid)

Once multiple threads are running, you will need a way of switching between different threads. To do this, you could use the library functions setjmp and longjmp. In a nutshell, setjmp saves the current state of a thread into a jmp_buf structure. longjmp uses this structure to restore (jump back) to a previously saved state. You will find these functions very useful, since they do exactly what is needed when switching from one thread to another and later resuming this thread's execution. More precisely, you can have the currently executing thread call setjmp and save the result in the thread's TCB. Then, your thread subsystem can pick another thread, and use the saved jmp_buf of that thread together with longjmp to resume the execution of the new thread.

The process of picking another thread is called scheduling. In our system, all scheduling queues should be **FIFO**. This includes the ready queue and the queue of threads waiting for a signal.

Now we need to force the thread of a program to call setjmp every once in a while (and thus, giving up control of the CPU). In particular, application developers do not need to know about your thread implementation, so it is unlikely that they will periodically call setjmp and allow your scheduler to switch to a new process. To solve this issue, we can make use of signals and alarms. More precisely, we can use the ualarm or the setitimer function to set up a periodic timer that sends a SIGALRM signal every X milliseconds (for this project, we choose X to be 50ms). Whenever the alarm goes of, the operating system will invoke the signal handler for SIGALRM. So, you can install your own, custom signal handler that performs the scheduling (switching between threads) for you. For installing this signal handler, you should use sigaction with the SA_NODEFER flag (read the man page for sigaction for details).

Note that we require that your thread system supports thread preemption and switches between multiple threads that are ready. It is **not** okay to run each individual thread to completion before giving the next one a chance to execute.

To ensure atomicity of multiple operations, users will use locks that you provide. To implement those locks, you COULD use the "compare and swap" routine provided by the x86.

This gives you the background to implement the *myThreadswitch*() function which does the scheduling for you.

Tests

- **Single thread**: A program that creates and thread (which prints something out) and then waits for it to complete.
- **Multiple threads**: creates multiple threads and then waits for all of them to complete. The threads should be run in FIFO order. There may be a large number of these, too - - which could stress whether you clean up the thread stack when done.
- **Multiple threads + yield**: creates multiple threads which frequently yield to each other. Testing whether yield works and FIFO ordering is preserved.
- **Multiple threads + lock**: creates threads and has them call lock/unlock on a single lock around a critical section.
- **Multiple threads + lock + yield**: creates threads, which grab lock then yield. Makes sure that lock is still held properly across yielding. Also makes sure that lock queue is managed in FIFO order.
- **Multiple threads + condition variable**: creates two threads which use a condition variable to wait/signal each other.

• **Multiple threads + condition variable + yield**: creates threads which use condition variables and locks to test whether condition variable queues are managed in FIFO order.

You will test your implementations using the following application programs:

- Multi threaded Matrix Multiplication with N threads
    - Plot a graph with respect to time taken versus N (=1 to 40)
    - Calculate the average context switch overhead with our thread library
    - Repeat the experiment with pThreads and compare the average context switch overheads inboth cases.
- Bounded Buffer (Producer Consumer) with N containers, M producers and M Consumers (M<N)
    - In this simulation, you should make each producer and consumer print when it acquires a container or deposits a container.
    - You must also maintain a shared buffer that tracks and prints how many containers have been produced and consumed. The producers and consumers should each stop after 10*N items have been either produced or consumed.

Your thread library needs to be built into a **dynamically-linked shared library**.

Building a dynamically-linked library

Your thread library needs to be built into a dynamically-linked shared library. The library should be called *libmyOwnthread.so*

There are two basic steps to building a dynamically-linked shared library:

- *Compile each .c file*
- *For each .c file that comprises your library (e.g., userthread.c), you should compile as follows:*
    - *gcc -Wall -fpic -c userthread.c*
    - *Here, the -Wall flag prints all warnings, as usual; the -fpic flag tells the compiler to use "position-independent" code, which is good to use when building shared libraries; finally, the -c flag tells the compiler to create an object file (in this case, userthread.o).*
- *Link the object files together to create a shared library*
    - *Once you've built all your .o files, you need to make the shared library:*
        - *gcc-o libmyOwnthread.so userthread.o -shared*
- *Linking a program with your library*
    - *Let's say you have a program, test.c, that wants to use this thread library. First, test.c should include the header file "userthread.h"*
    - *To build and test, you need to link it with your library (assuming all of your code is in the same directory)*
        - gcc -o test test.c -L. -lmyOwnthread

- *Of course for the assignment, you must use these commands in the Makefile context and have the library in its proper place.*

What do you turn in:

When we look at your your code, we will expect to see the following files in your repository (there may be more):

- myOwnthread.h, the header file that defines the functions exported by your library
- The multiple source code and header files that implement the functionality exported by userthread.h
- A Makefile to build your code. Typing make should build libmyOwnThread.so.
- libmyOwnThread.so should be compiled with the following flags: -Wall -fpic
- The application src codes
- The makefile should have multiple targets
    - make primes
    - make boundedBuffer
    - makeall
    - make clean
- A file called README which includes any additional nodes on your program that you think are important.
- You should not turn in any binary or .o files. Only turn in source code and header files