POSIX

PRIMER:

On Unix flavors (a la BSD, AT&T Unix), the concept of a process mapped to "a running instance of a program". We use the following definition for process

"An address space and a group of resources dedicated to running a program. "

The resources associated with a Unix process include:

- address space
- CPU registers
- process ID
- file descriptors
- stack

Let us try to enumerate the ways we can implement concurrency in such a model. It is clear to see that we will need to operate two instances of the program. For synchronizing computation, we will need to build some infrastructure for inter-process communication and work our way up from there. We observe that this approach looks unwieldy. Also, providing infrastructure for IPC considering the resources listed above is no trivial task and it is clear that this approach will put a heavy penalty on computation.

A CHEAPER APPROACH:

A new fundamental idea that was proposed involved having multiple threads of control instead of just one (the process model limited us to 1 thread of control). This new approach now involves divvying a limited set of available resources up among multiple threads of control and then using these threads to implement concurrency.

What are the limited resources we will need to implement a single "thread of control". Clearly, the basic set of resources are:

- Registers
- Stack

Using just registers and stack, we can represent an entire computation state (remember the data-structure from the section on Generators - we worked with a data-structure that took a snapshot of stack and registers and restoring computational state involved instantiating the values in this data-structure). The obvious other conclusion we can draw is that the other resources are global and all these threads of execution see any changes made to them.

We now see that this model has achieved concurrency without the added overhead that the original approach (of using multiple individual processes) came with.

The **pthread library** is an implementation of this model.

RESOURCE MODEL:

As stated above, the POSIX model is that threads maintain individual registers and stacks but all other resources are shared. Any modification we make to these global resources are visible to all the threads.

MAPPING MODEL:

POSIX is just a specification. How a pthread maps to the OS scheduling primitive is up to the implementation. We have 1:1 mapping available in some kernels, X:1 and X:Y are also available in operating systems (commercial and free today). Nevertheless, the behavior of the pthreads API is consistent across implementations.

PRIMITIVES:

To be able to use the POSIX API, we need to be familiar with the following primitives:

- Thread Creation:
  - Function name: `pthread create`

    - Returns:
        - `int`
    - Arguments:
        - `pthread_t *thread`: A pthread object that contains the pthread id.
        - `pthread_attr_t *attr`: thread attributes
        - `void *(*start_routine)(void *)`: The routine that is the body of code executed by this thread of execution.
        - `void *arg`: The routine's arguments
- Thread Epilogue:
    - We have two primitives for this stage of a thread:
        - `pthread_exit`
        - `pthread_join`
- Exit:
    - Arguments:
        - `void *value_ptr`: If we want to pass data to another thread, then we use a pointer to it.
- Join:
    - Suspends execution and waits for another thread to exit.
    - Arguments:
        - `pthread_t thread`: The thread whose completion we are waiting for.
        - `void **value_ptr`: The `value_ptr` passed by the caller of `pthread_exit` is available here.
- Let us now see a simple example that uses these primitives. The given example just prints out the integer argument supplied to it.

```
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>

/* download from:
http://github.com/shriphani/cs390-posix */

#define NUM_THREADS 5

void *routine(void *arg)
{
    int val = *(int *)arg;
    printf("thread no. %d\n", val);
    pthread_exit(NULL);
}

int main(int argc, char **argv)
{
    pthread_t thr[NUM_THREADS];
    int i, rc;

    for (i=0;i<NUM_THREADS;i++)
    {
        pthread_create(&thr[i], NULL, routine,
&(i));
    }

    for (i=0;i<NUM_THREADS;i++)
    {
        pthread_join(thr[i], NULL);
    }

    return EXIT_SUCCESS;
}
```

One obvious upshot of the pthread model is that we need a mechanism to secure accesses to the global resources that all threads see. Let us look at a couple of techniques provided by the pthread library:

- Mutexes:

- They provide a locking mechanism. The locks here are used to synchronize accesses to data. The idea is that when one thread is operating on the data, another shouldn't be manipulating it. The API for using mutexes looks like:
  - ```
    int pthread_mutex_init(pthread_mutex_t
    *mutex, const pthread_mutexattr_t
    *mutexattr);
    ```
- For locking and unlocking access to the data, we have:
  - ```
    int pthread_mutex_lock(pthread_mutex_t
    *mutex);
    int
    pthread_mutex_trylock(pthread_mutex_t
    *mutex);
    int pthread_mutex_unlock(pthread_mutex_t
    *mutex);
    ```
- Let us now see a simple example. We use a global variable (an integer) that each thread increments.
- Here is a pseudocode implementation:
  - ```
    i <- 0
    i_lock;

    proc:
        lock(i_lock);
        i++;
        print i;
        unlock(i_lock);

    main:
        init 5 threads;
        execute 5 threads;
    ```
- A C version of the above pseudocode is provided below:
  - ```
    #include<stdio.h>
    #include<stdlib.h>
    #include<pthread.h>

    #define NUM_THREADS 5

    int i = 0;
    pthread_mutex_t i_lock;

    /* download from:
    http://github.com/shriphani/cs390-posix
    */

    void *routine(void *arg)
    {
        int val = *((int *)arg);
        pthread_mutex_lock(&i_lock);
        i++;
        printf("thread #%d updated i to
    %d\n", val, i);
        pthread_mutex_unlock(&i_lock);

        pthread_exit(NULL);
    }

    int main()
    {
        pthread_t thr[NUM_THREADS];
        int i, rc;

        pthread_mutex_init(&i_lock, NULL);

        for (i=0;i<NUM_THREADS;i++)
        {
            pthread_create(&thr[i], NULL,
    routine, &(i));
    ```

```
    }

    for (i=0;i<NUM_THREADS;i++)
    {
        pthread_join(thr[i], NULL);
    }

    return EXIT_SUCCESS;
}
```

- We need to keep a few things in mind when we are working with mutexes however. The API (although simple) does allow one to produce buggy code with great ease. First:
  - Locks and the data items they are protecting are completely decoupled from each other.
  - Deadlocks are possible. Here is an example of how to produce one:
    - ```
      Thr1                Thr2
      lock A
                          lock B
      lock B //locked
                          lock A //locked
      ```
  - The solution for such problems is to ensure that threads grab locks in a consistent global order.

- Conditional Variables:
  - Let us now look at another situation where concurrency is a valid execution model. Let us say two threads are playing a game of tic-tac-toe.
  - Thread1 places the X's in the grid and Thread2 places the O's in the grid.
  - In this model, we want a thread to sleep if it is not its turn. The thread should only wake up when it is supposed to.
  - The mutex primitive described above does not provide such a mechanism. The pthread library provides another primitive for this purpose.
  - ```
    pthread_cond_init(pthread_cond_t *cv);
    pthread_cond_wait(pthread_cond_t *cv,
    pthread_mutex_t *lock);
    pthread_cond_signal(pthread_cond_t *cv);
    ```
  - Usage:
    - Use `pthread_cond_init(pthread_cond_t *cv);` to initialize the condition variable.
    - A thread that decides it is time to wait on something invokes `pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *lock);`
    - Before we wait, we must grab the lock that we pass to the wait call.
    - So there is an implicit unlock involved in this contract where one thread blocks.
    - Now, if we want to signal an event, we use the `pthread_cond_signal(pthread_cond_t *cv);`
    - Once this call is made, the other thread that blocked, wakes up and waits till it can grab the mutex it gave up before going to sleep.
  - Let us build a simple program that makes use of these functions. We will implement Shannon's Useless Machine
  - The semantics of this program are very simple:
    - One thread sets the switch to ON, signals the other thread and goes off to sleep.
    - The other thread sets the switch to OFF and signals the first one and then itself goes to sleep.
  - So, this simple program can be implemented with extreme ease:
    - ```
      #include<stdio.h>
      #include<stdlib.h>
      #include<pthread.h>

      /* download from:
      http://github.com/shriphani/cs390-posix
      */
      ```
      ```

```
pthread_mutex_t switch_lock;
pthread_cond_t  switch_cond;

int SWITCH = 0;

/**
*
**/
void *set_low(void *args)
{
  while (1)
    {
      if (SWITCH)
        {
          SWITCH = 0;
          printf("OFF\n");

pthread_cond_signal(&switch_cond);

pthread_cond_wait(&switch_cond,
&switch_lock);
        }
    }

}

void *set_high(void *args)
{

  while (1)
    {
      if (!SWITCH)
        {
          SWITCH = 1;
          printf("ON\n");

pthread_cond_signal(&switch_cond);

pthread_cond_wait(&switch_cond,
&switch_lock);
        }
    }

}

/**
*
**/
int main()
{
  // init
  pthread_mutex_init(&switch_lock,
NULL);
  pthread_cond_init(&switch_cond, NULL);

  pthread_t p1, p2;
  pthread_create(&p1, NULL, set_high,
NULL);
  pthread_create(&p2, NULL, set_low,
NULL);

  pthread_join(p1, NULL);
  pthread_join(p2, NULL);

  return EXIT_SUCCESS;
}
```

- This concludes our posix tutorial.