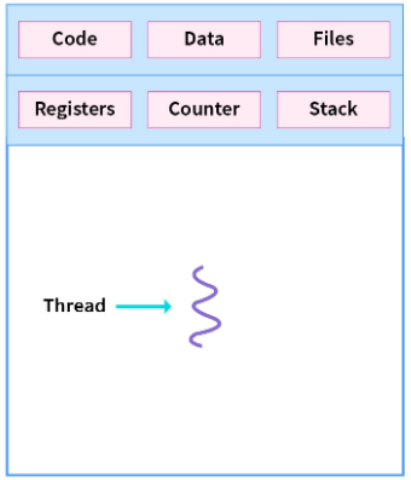
Altera Cyclone V HPS-FPGA

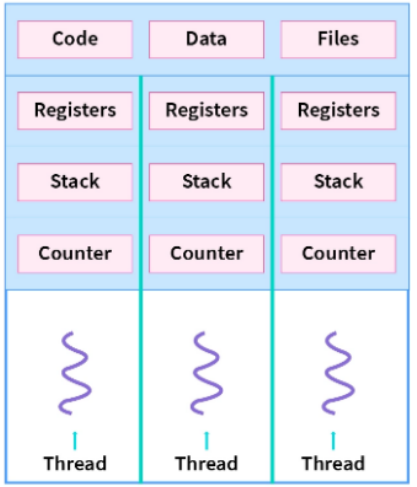
Producer / Consumer

Threads

- **Thread**: a single stream of instructions, within a process, that the scheduler can run separately and concurrently with the rest of the process.
- A **Process** can have multiple Threads (all of which share resources of the process (data and CPU) and run in the same user space.
- Each Thread can use all the global variables of the process but also have its own data, its own Stack, its own Program Counter and its own Status Registers



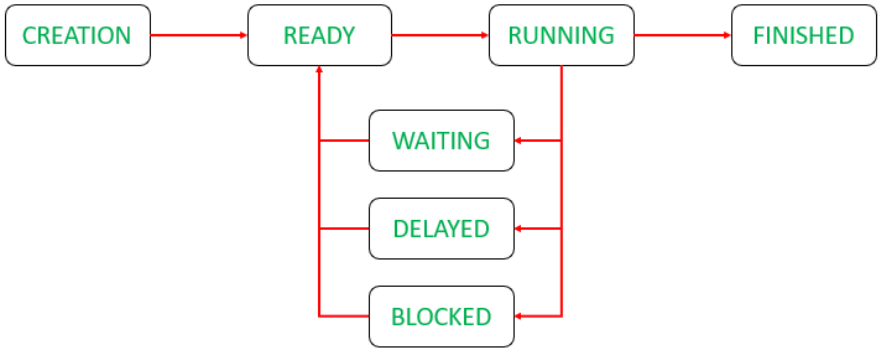
Single-threaded process



Multithreaded process

Thread States in Operating Systems

- When a thread moves through the system, it is always in one of the following states:



1. When an application is to be processed, then it **CREATEs** a thread.
2. It is then allocated the required resources(such as a network) and it comes in the **READY** queue.
3. When the thread scheduler (like a process scheduler) assign the thread with processor, it comes in **RUNNING** queue.
4. When the process needs some other event to be triggered, which is outsides it's control (like another process to be completed), it transitions from **RUNNING** to **WAITING** queue.
5. When the application has the capability to delay the processing of the thread, it when needed can delay the thread and put it to sleep for a specific amount of time. The thread then transitions from **RUNNING** to **DELAYED** queue. An example of delaying of thread is snoozing of an alarm. After it rings for the first time and is not switched off by the user, it rings again after a specific amount of time. During that time, the thread is put to sleep.
6. When thread generates an I/O request and cannot move further till it's done, it transitions from **RUNNING** to **BLOCKED** queue.
7. After the process is completed, the thread transitions from **RUNNING** to **FINISHED**.

Threads

Threads have been standardized in the IEEE POSIX 1003.1c API, and known as Pthreads.

The APIs for Pthread distinguish functions into 3 groups:

- **Thread management**: functions to create, delete, wait for the end of pthreads
- **Mutexes**: functions to support mutual exclusion synchronization (including functions to create and delete the structure for mutual exclusion of a resource, to acquire and to release that resource).
- **Condition variables**: functions to support synchronization depending on the value of variables (including functions to create and to delete the structure for synchronization, to wait, an to report variable changes.

The phtreads.h file contains pthread definitions. In compilation (and linking) using gcc add the **-lpthread** flag to use the pthread library.

Threads - Synchronization

pthread_mutex_lock(&mutex**)** — Wait for a lock on a mutex object

Mutexes are used to protect shared resources. If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it.

pthread_cond_wait(&cond, &mutex**)** — Wait on a condition variable

Cond is a condition variable is shared by threads. To change it, a thread must hold the *mutex* associated with the condition variable. The pthread_cond_wait() function releases this *mutex* before suspending the thread ("sleep") and obtains it again (which may require waiting) when it is signalled

Threads - Synchronization

pthread_cond_signal(&cond**)** — Signal Condition to One Waiting Thread

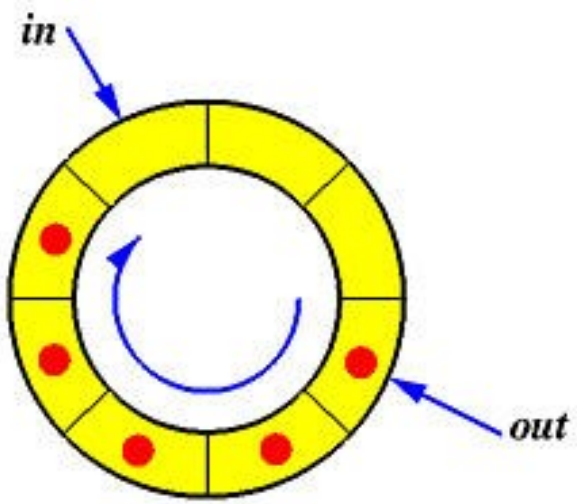
This function wakes up at least one thread that is currently waiting on the condition variable specified by *cond*. If no threads are currently blocked on the condition variable, this call has no effect.

pthread_mutex_unlock(&mutex**)** — Unlock Mutex

This function unlocks the specified mutex.

The Producer/Consumer (or Bounded-Buffer) Problem

- Suppose we have a circular buffer with two pointers in and out to indicate the next available position for depositing data and the position that contains the next data to be retrieved.
- There are two groups of threads, producers and consumers: each producer deposits a data items into the in position and advances the pointer in, and each consumer retrieves the data item in position out and advances the pointer out.



The Producer/Consumer (or Bounded-Buffer) Problem

- A producer must wait until the buffer is not full, deposit its data, and then notify the consumers that the buffer is not empty.
- A consumer, on the other hand, must wait until the buffer is not empty, retrieve a data item, and then notify the producers that the buffer is not full.
- Before a producer or a consumer can have access to the buffer, it must lock the buffer. After a producer and consumer finishes using the buffer, it must unlock the buffer.