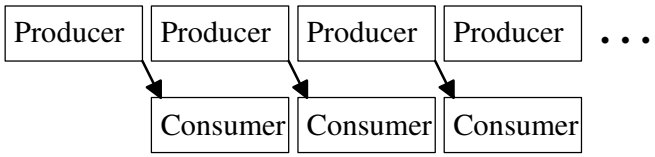
Common synchronization patterns between threads:
critical section - shared variable(s) updated must be done mutually exclusively. In pthreads, we can use a mutex to synchronize them without "busy-wait". Waiting threads are not using the CPU/core, but on an "unlock" we have no control which gets the woke-up.

1) Another common synchronization pattern is *bounded buffer* where two threads are linked by a processing pipeline where:

- a producer thread produces some data ("work"), and
- a consumer thread consumes the data (do the "work")

Without concurrency they could alternate:

With concurrency they could pipeline:



a) What is the advantage of the pipelined version?

b) In the pipelined version, how does the rate the producer produces data compare to the rate the consumer consumes it?

2) A bounded buffer can be used to store some (limited by some bound) data between the producer and consumer threads:

a) What is the advantage of the bounded buffer approach over pipelining the producer data directly to the consumer thread?

b) In the bounded buffer approach when would the producer thread need to block?

c) In the bounded buffer approach when would the consumer thread need to block?

3) Another common synchronization pattern is *readers/writers locks* which are similar to mutexs, except threads specify whether they are locking for writing (updating) or only reading the shared data structure.
a) Since writer threads must update the shared data structure mutually exclusively, what concurrency would readers/writers locks provide?

b) Consider the scenario where one (or more) reader thread(s) hold the lock and there are waiting writer thread(s) when a new reader thread "arrives." Two possibilities are:
- Allow the new reader to immediately start reading
- Make the new reader wait until after the waiting writers

What are the advantages and disadvantages of each approach?

c) Many systems including POSIX provide an implementation of readers/writers locks (`pthread_rwlock_init`, `pthread_rwlock_rdlock`, `pthread_rwlock_wrlock`, `pthread_rwlock_unlock`). However, the choice of prioritizing a new reader vs. waiting writer(s) is open to system implementation. What are the advantages and disadvantages of not specifying this choice?

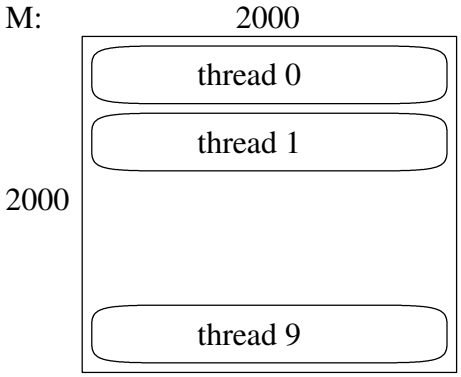
d) POSIX has specialized forms of readers/writers locks for files: `fcntl` procedure (complex) and simpler `flock` interface. Why are files likely choice for applying readers/writers locks toward?

4) Another common synchronization pattern are *synchronization barriers* where we want all threads need to finish some phase of a task before any can continue. This pattern is often used in scientific calculuations on large matrices (2D, 3D, etc. arrays). Barrier are initialized with the number of threads for which to wait. As threads arrive at the barrier and issue a "wait," all threads are blocked until the last thread performs a wait.

Sequential code:

```
for time = 0 to 1000 do
    for row = 0 to 1999 do
        for column = 0 to 1999 do
            M[row, column] = ...
```

Might assign 10 threads each 200 rows to calculate, but they all need to complete a time interval before any thread can move on to the next time interval.



Sketch the multi-thread code focusing on where the barrier initialization and wait would be performed.

To implement these synchronization patterns we need mechanisms (tools) to allow threads to wait until circumstances are appropriate for it to proceed. Two common mechanisms are:

I. *condition variables* used with monitors (or mutexes used in a monitor style) have operations

- wait - executed by a thread finding circumstances ("conditions") not to its liking, so it sleeps/blocks until another thread does a signal/notify operation
- signal/notify - unblocks a waiting thread and moves it to the run queue
- broadcast/notify-all - unblocks all waiting threads

General pseudocode syntax:

```
lock mutex
if condition is favorable then
        signal thread(s)
else
        unlock mutex and block
        /* when thread is unblocked, mutex is relocked */
end if
unlock mutex
```

5) Consider a *barrier synchronization* with global integers `threadCount` (# of total threads) and `counter`(count of threads that have reached the barrier. Complete if-statement check for "favorable condition".

```
int threadCount;
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t condVar;

int main(...) {
    pthread_cond_init(&condVar, NULL);
...

    pthread_cond_destroy(&condVar);
} // end main

void* threadWork(...) {
      ...
      /* Barrier */
      pthread_mutex_lock(&mutex);
      counter++;
      if (                                                ) {
            counter = 0;
            pthread_cond_broadcast(&condVar);
      } else {
            while(pthread_cond_wait(&condVar, &mutex) != 0);
      } // end if
      pthread_mutex_unlock(&mutex);
      ...
} // end threadWork
```

Notes:

- the `pthread_cond_wait` will unlock the mutex parameter and cause the executing thread to block until:
  - `pthread_cond_broadcast` or `pthread_cond_signal` by another thread in which case `pthread_cond_wait` returns a 0
  - other system events ("bugs") have been know to cause threads to be woke-up, but with non-0 return value. Hence the while-loop around the `pthread_cond_wait`

6) Outline how pthread mutex(es) and condition variable(s) can be used to implement a bounded buffer that can be used to store some (limited by some bound) data between the producer threads and consumer threads: