Condition Variables
  o Introduction
    ▪ Mutexes are good for controlling access to a critical region
    ▪ Semaphores are good for situations where counting is needed
    ▪ Sometimes we want to block until some condition is met
    ▪ Solution: condition variables
      ▪ In terms of producer-consumer, the producer wants to sleep until the buffer is no longer full
  o Mechanism
    ▪ Same as sleep/wakeup paradigm
      ▪ Signals are used to wakeup
      ▪ No state is kept, so signals can be lost
      ▪ We can avoid the previous race conditions however
    ▪ In Linux, always used in coordination with a mutex
      ▪ Combining this with atomic operations avoids race conditions

Linux System Calls
  o pthread_cond_init(pthread_cond_t *cond, pthread_contattr_t *attr)
    ▪ initializes the condition variable cond
  o pthread_cond_destroy(pthread_cond_t *cond)
    ▪ destroys condition variable cond
    ▪ can be called only when no threads are blocked on it
      ▪ Doing otherwise is undefined
  o pthread_cond_signal(pthread_cond_t *cond)
    ▪ unblocks at least one of the threads blocked on the condition variable
    ▪ Could potentially wake up multiple threads – spurious wakeups
      ▪ Solution – use a testing loop around waiting for condition variables
    ▪ Can only fail if cond is invalid
  o pthread_cond_broadcast(pthread_cond_t *cond)
    ▪ unblocks all threads blocked on the condition variable
    ▪ Can only fail if cond is invalid
    ▪ Any thread with access to cond can call signal/broadcast
  o int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
    ▪ Block until signal is sent on condition variable cond
    ▪ Mutex MUST be locked prior to calling this function
    ▪ Operation
      ▪ When called, mutex is unlocked and block occurs atomically (avoids race condition)
      ▪ When the function returns, the condition has been signaled and mutex has been locked again
    ▪ Errors
      ▪ EINVAL – different mutexes were supplied to concurrent pthread_cond_wait calls on the same condition variable
- EPERM – Mutex was not owned by the current thread at the time of the call
  - To avoid problems from spurious wakeups, a testing loop is required on the condition
- `int pthread_cond_timedwait(pthread_cond_t *cont, pthread_mutex_t *mutex, const struct timespec *abstime)`
  - Same operation as previous, but with absolute timeout
  - Returns error and sets errno to ETIMEDOUT in case of timeout

- Example: producer-consumer problem

```c
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000 /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0; /* buffer used between producer and consumer */

void *producer(void *ptr) /* produce data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* put item in buffer */
        pthread_cond_signal(&condc); /* wake up consumer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr) /* consume data */
{
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* get exclusive access to buffer */
        while (buffer == 0) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* take item out of buffer */
        pthread_cond_signal(&condp); /* wake up producer */
        pthread_mutex_unlock(&the_mutex); /* release access to buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}