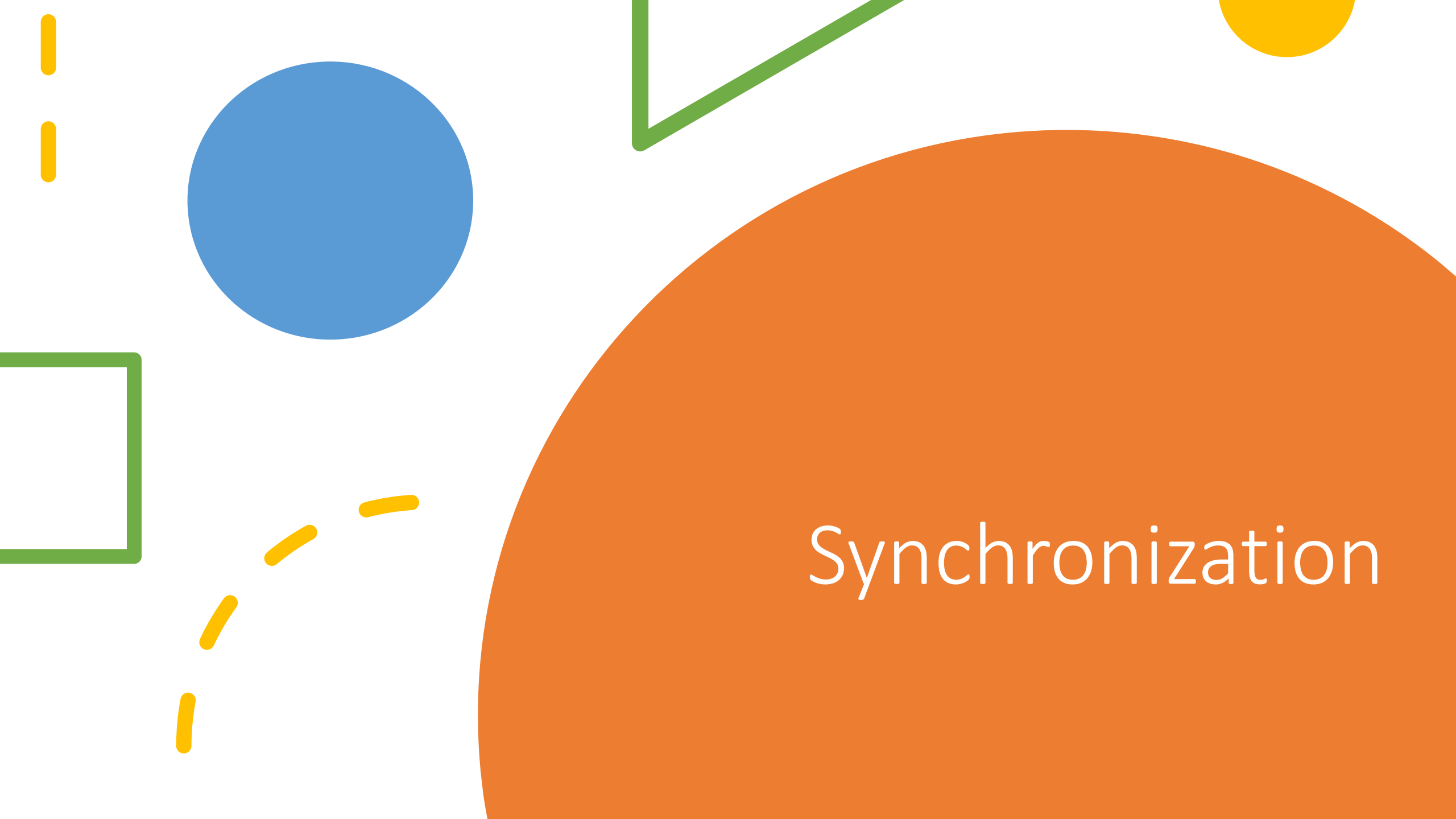

Synchronization

Creating shared memory between processes

Two processes can share memory

- Like threads.
- We wil use mmap API

#include <sys/mman.h>

- **void *mmap(void **addr*, size_t *length*, int *prot*, int *flags*, int *fd*, off_t *offset*);**
- **int munmap(void **addr*, size_t *length*);**

For processes memory is not shared unless we state otherwise

For threads memory is shared by default.

Message Passing between processes

- Can be done using Sockets
- (And other APIs)
- But we don’t need to checksum the data (twice in IP and TCP) and we don’t need to send acks and we don’t need to control window size!
- We can create “Sockets” that does everything accept communication and achieve much better performance
- Enter UNIX domain sockets

UNIX domain sockets

- PATH is used instead of port for bind/connect.
- Offcourse no DNS resolution
- Otherwise same as TCP/IP sockets
- See Beej guide for IPC (Chapter 11) for example

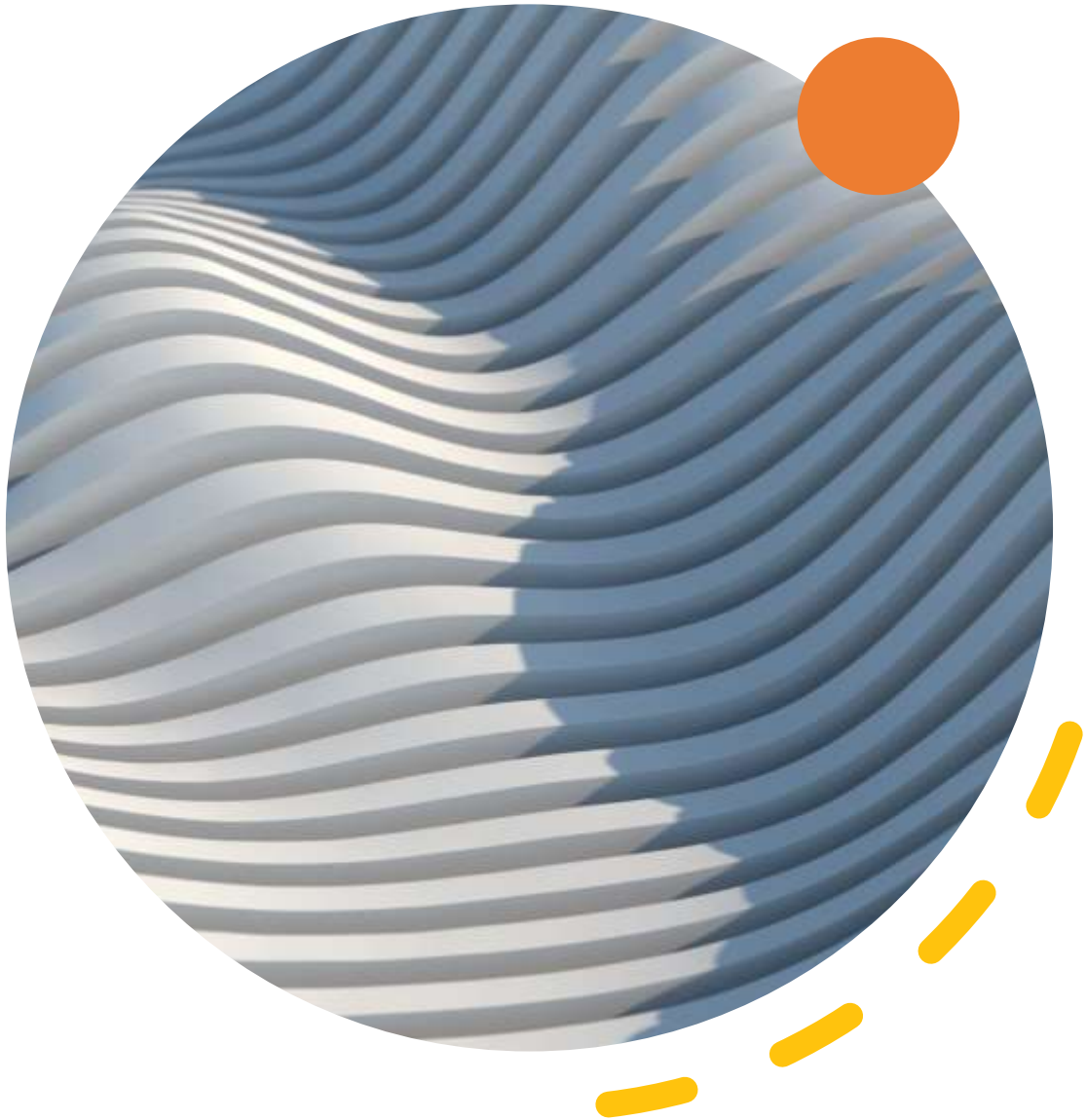
sync primitives

Only two basic types of synchronization

SOCK-ON-DOOR

PRODUCER-CONSUMER

IN ALL PROGRAMMING LANGUAGES IN ALL SCENARIOS



Sock-on-door

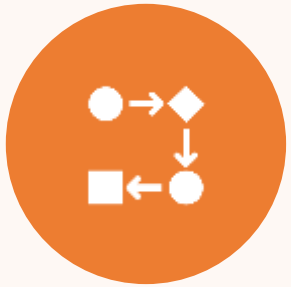
- Doorsock **(Urban dictionary)**
- A warning **signal** which is made by placing a sock over **the doorknob** of a **bedroom** door. This signifies to a roommate that you are inside the bedroom, engaged in sexual behavior.
- The sock-on-door is often called MUTEX (Mutally-exclusive)

What does sock-on-door do?

- It does not lock the door.
- It tells people who participate in the “game” that they are not supposed to go in.
- As we know from countless of college comedies at some time, some body (a foreign exchange student from Borat-land) will open the door… and then all hell breaks loose.
- Advisory locking – if you ask (or look for socks) it is locked. If you don’t ask you can access it.
- Mandatory locking - locked means locked

Sock-on-door is advisory locking

When to use sock-on-door



You have two tasks (threads or processes) – roomates.

They share a resource – room

One of them is engaged in an activity and needs to use the room all by himself

Needs to inform the other not to use shared resource

What do you do when you reach ”sock-on-door”

- You can wait *that’s what you usually do*
- You can do other things.
- You can not interfere and you cannot ask your roommate to “hurry-up”

How to use sock on door in THREADS environment

- POSIX 95 to the rescue
- Virtually the best and almost only standard way to Implement mutex between threads. (We will implement threads using atomic counter and TBB)

Mutex and just a regular boolean variable

- Both mutex and boolean variable support “locked” and “unlocked” states only.
- Mutex supports Test and Set atomic operation i.e. if (unlocked) lock()
- Atomic here means “indivisable” which means that either it completes or it doesn’t start but it can’t be interrupted
- With boolean variable one could consider a scenario where T1 does

if (mutex.locked == false)

BOOM context switch BOOM then T2 does

(mutex.locked == false)

(mutex.lock=true) …
* BOOM context switch BOOM* returning to T1
(mutex.lock=true) …
* nuclear cataclysim *

POSIX API

- **#include <pthread.h>**

- int pthread_mutex_destroy(pthread_mutex_t ***mutex*);**
- int pthread_mutex_init(pthread_mutex_t *restrict *mutex*,
  const pthread_mutexattr_t *restrict *attr*);**
- pthread_mutex_t *mutex* = **PTHREAD_MUTEX_INITIALIZER;**
- int pthread_mutex_lock(pthread_mutex_t ****mutex*);**
- int pthread_mutex_trylock(pthread_mutex_t ***mutex*);**
- int pthread_mutex_unlock(pthread_mutex_t ***mutex*);**

What happens when you lock a locked mutex?

- If somebody else locked it – you wait
- But if you locked it … is it unlocked with two unlocks or a single unlock
- Behaviour can be changed if mutex is recursive mutex (two unlocks required) or not-recursive (one unlock required)
- So recursive mutex basically adds “lock count”
- Type can be set using mutex attributes.
- POSIX does not define if default is recursive or unrecursive
- Default behaviour in Linux is recursive
  (BTW in Solaris it is not recursive – so make no assumptions)

Sock-on-door between processes

- There are many APIs for this problem.
- We will teach file locking API. (there are other APIs but I don’t wish to spend too much time on several APIs doing the same thing – and a rare thing)

File locking API

- **#include <fcntl.h>**
- **int fcntl(int** *fd*, **int** *cmd*, **... /*** *arg* ***/);**
- struct flock { ... short l_type; /* Type of lock: F_RDLCK, F_WRLCK, F_UNLCK */
- short l_whence; /* How to interpret l_start: SEEK_SET, SEEK_CUR, SEEK_END */
- off_t l_start; /* Starting offset for lock */
- off_t l_len; /* Number of bytes to lock */
- pid_t l_pid; /* PID of process blocking our lock (set by F_GETLK and F_OFD_GETLK) */ ...
- };

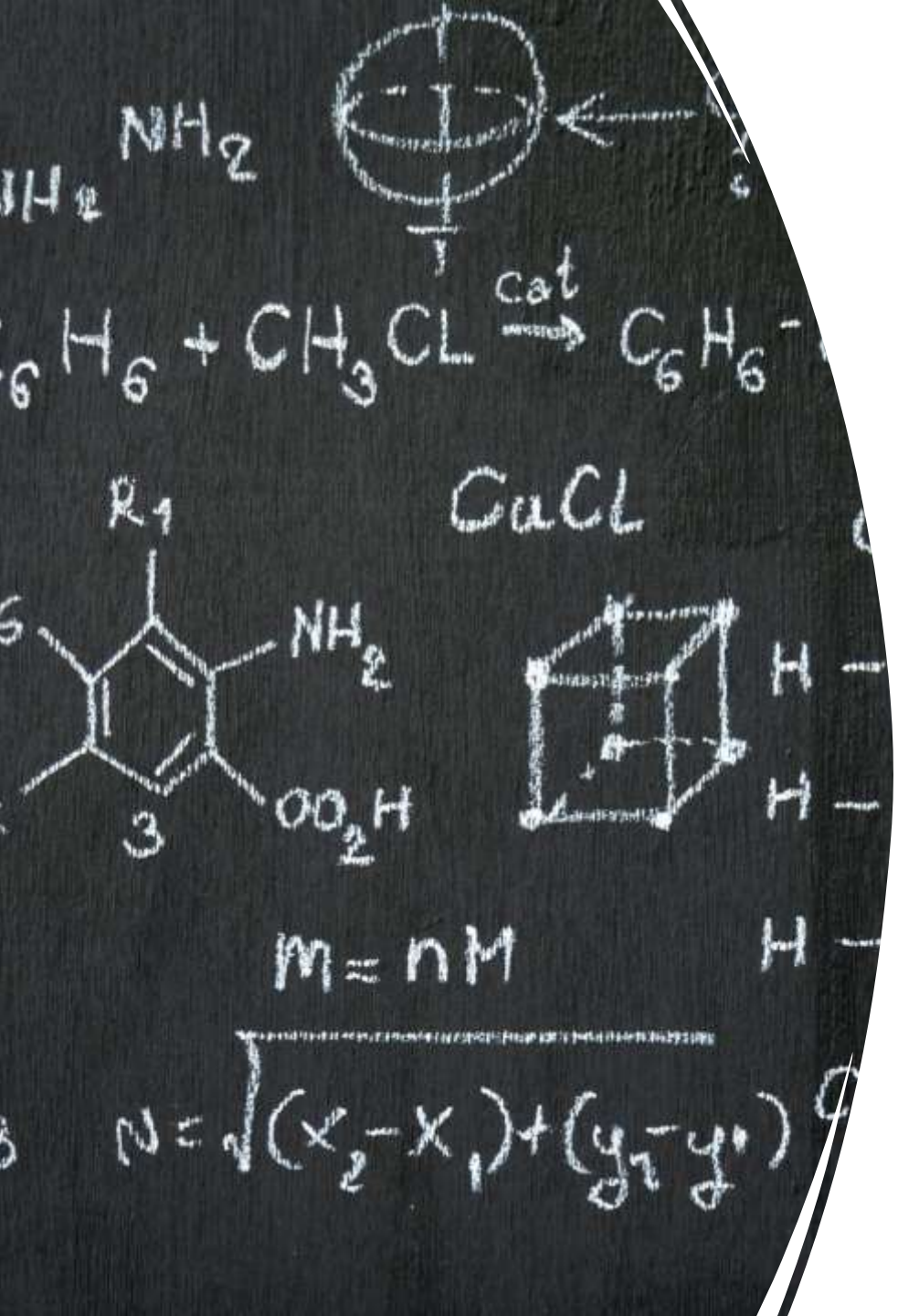
Producer consumer

- One thread makes food. (Or generate tasks to be handled e.g. requests to be handled)

- One thread eats (Or handle the tasks)

Producer-consumer

- We cannot enter the room while Taz is eating
- Taz should not be let into the room before there is actually something to eat
- The Pavlovian solution : Gasteau has a “Gong”
- When dinner is ready Gasteau bangs the Gong and leaves the room. Now Taz can eat
- Note that when Taz is ready and “acquires the lock” he locks himself out of the room. When Food is ready Gasteau bangs the Gong and Taz eats
- Gasteau and Taz are not identical “roomates” they do different things and Gasteau always prepare the food first – Taz eats second



Producer-Consumer Locking between threads

- **#include <pthread.h>**

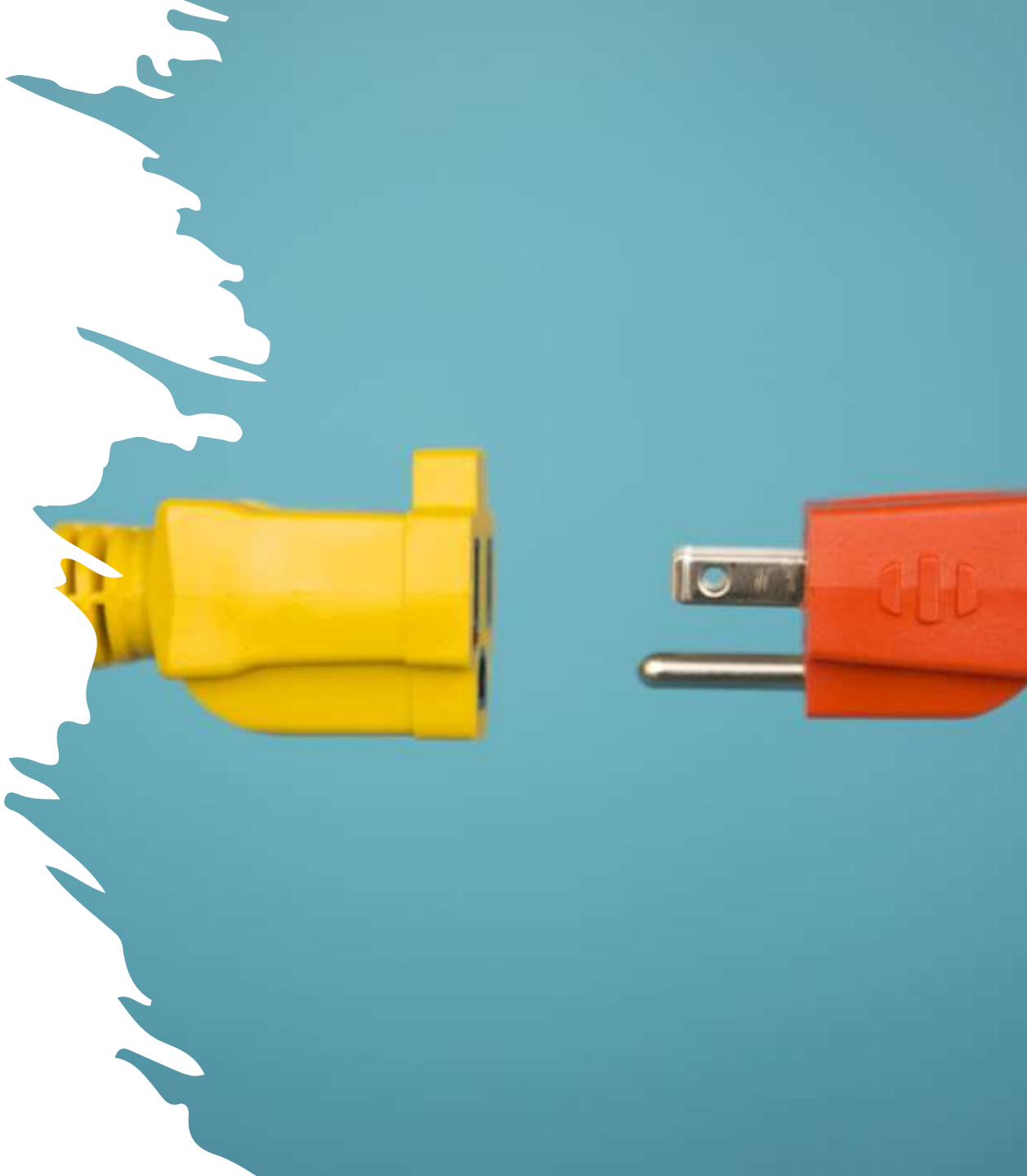
- int pthread_cond_destroy(pthread_cond_t *_cond_**);**
- int pthread_cond_init(pthread_cond_t *restrict _cond_**,**
  const pthread_condattr_t *restrict _attr_**);**
- pthread_cond_t _cond_ **= PTHREAD_COND_INITIALIZER;**
- int pthread_cond_timedwait(pthread_cond_t *restrict _cond_**,**
  pthread_mutex_t *restrict _mutex_**,**
  const struct timespec *restrict _abstime_**);**
- int pthread_cond_wait(pthread_cond_t *restrict _cond_**,**
  pthread_mutex_t *restrict _mutex_**)**
- int pthread_cond_broadcast(pthread_cond_t *_cond_**);**
  int pthread_cond_signal(pthread_cond_t *_cond_**);**

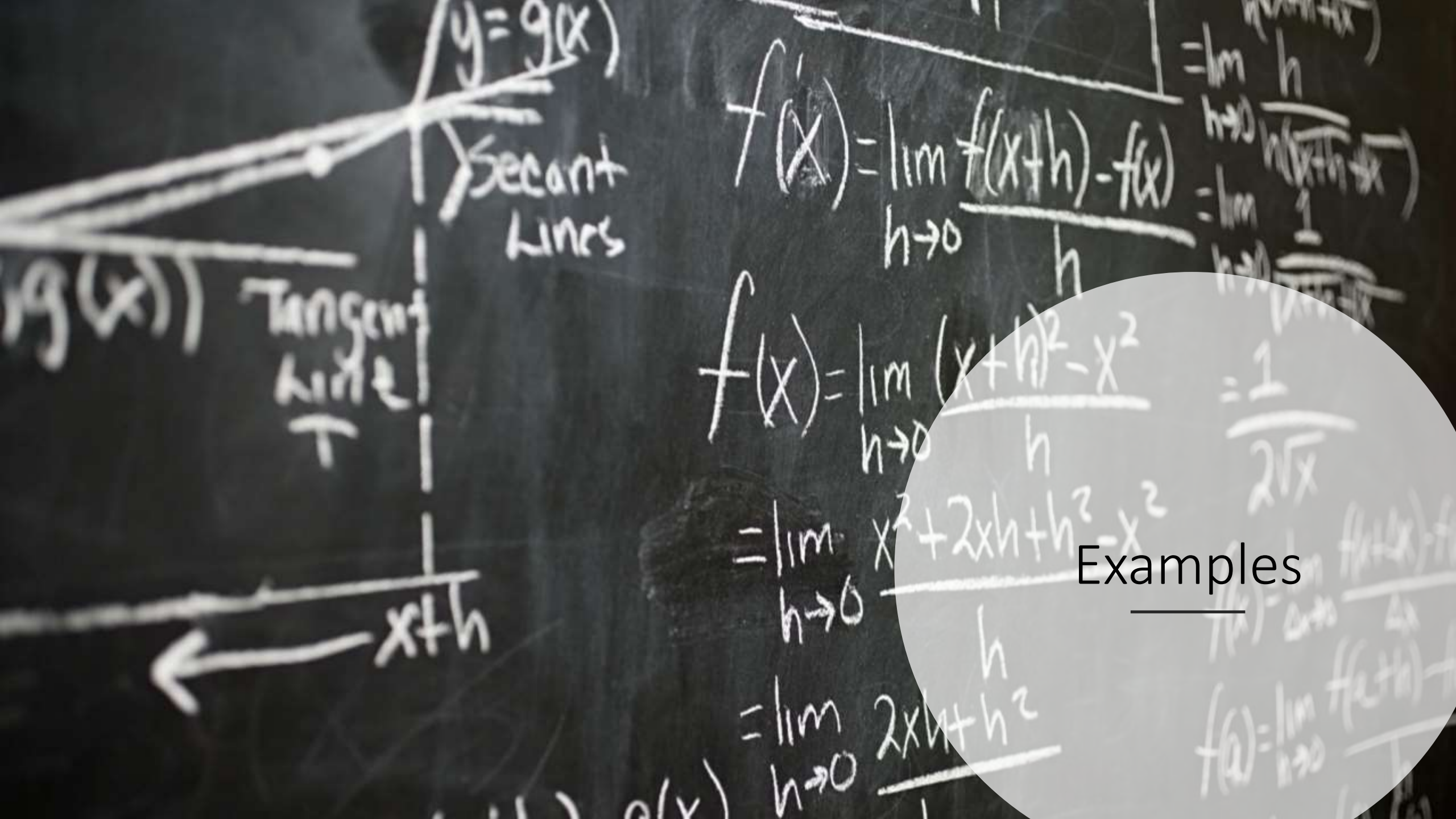
Bits and bytes

- Cond requires a mutex to wait. When calling cond wait we release the mutex.
- When returning from cond we require two things
  - Somebody called pthread_cond_signal
  - The mutex can be reacquired
- We will not return until both conditions were fulfilled.
- Also
- Signal wakes one thread waiting on cond
- Broadcast awakens all threads
- Conds are not using counter. If you signal but nobody is waiting the signal is lost. Use sockets if it matters.

Producer consumer locking between processes

- Again multiple APIs exist but since you already know sockets we will use sockets.
- Producer sends 1 byte to consumer when food is ready
- Consumer recv 1 byte and blocks until it is received (and we can now eat)
- Since we actually send bytes sockets can be used between thread and implement a counter
- Since we use sockets on the same host UDS can be used more efficiently but we will do that later




y=g(x)
Secant Lines
Tangent Line
x+h
f'(x)=lim h→0 f(x+h)-f(x)/h
f(x)=lim h→0 (x+h)²-x²/h
=lim h→0 x²+2xh+h²-x²/h
=lim h→0 2xh+h²/h
=1/2√x

Examples

On the web

What	URL
Using mmap to copy file and share memory	https://developpaper.com/using-mmap-to-copy-large-files-single-process-and-multi-process/
Using UNIX domain sockets	https://beej.us/guide/bgipc/html/multi/unixsock.html
Fcntl file locking example	https://www.informit.com/articles/article.aspx?p=23618&seqNum=4
POSIX mutex	https://riptutorial.com/posix/example/15910/simple-mutex-usage
POSIX cond	https://hpc-tutorials.llnl.gov/posix/example_using_cond_vars/