Let's implement our own version of a pipe

```
#define PIPE_SIZE 1<<12
struct pipe {
  char buf[PIPE_SIZE];
  unsigned r, w;
  lock_t *alock;
}
void lock(lock_t* alock);
void unlock(lock_t* alock);
// we ignore the implementation of lock and unlock for now
char readc(struct pipe *p) {
  lock(&l);
  // we cannot hold the lock while the pipe is full; we wait
  while (p->r == p->w) {
    unlock(&l);
    lock(&l);
  }
  char c = p->buf[p->r++ % PIPE_SIZE];
  unlock(&l);
  return c
}
// we do a writec the same way
```

This is a fine grain lock; we could use a global lock, but that would be course grain and slow

How do we implement a lock?

We can use the following instruction

```
int xchgl(int* p, int val) {
  int old = *p;
  *p = val;
  return old;
}
```

and thus write the following cosine function:

```
void cosine(struct s *p) {
  do {
    double d1 = p->d;
    double d2 = cos(d1);
  } while(xchgl(&p->d, d1, d2) != old);
}
```

but the value can change before we run xchgl and cause us to loop infinitely!

We need to use cas!

```
bool cas(int *p, double old, double new) {
  if (*p==old) {*p=new; return 1;}
  else return 0;
}
```

We can instead use the command to build a portable

```
void cosine(struct s *p) {
  do {
    double d1 = p->d;
    double d2 = cos(d1);
  } while(!cas(&p->d, d1, d2);
}
```

We can utilize the above commands to abstract a lock API of the following sort:

```
struct s {
  double d;
  lock_t l;
}
void cosine(struct s *p) {
  lock(&s->l);
  p->d = cos(p->d);
  unlock(&p->l);
}
```

Maximum performance is found by minimizing the critical section, but we have reached the atomicity of the next layer down, so this is as well as we can do!

This style of lock is still pretty slow; a mutex is more efficient than a spin lock because we sleep while waiting

Let's try to make one!

```
//Mutex
typedef struct s{
  bool acquired;
  thread_descriptor_t *blocked;
  lock_t lock;
} bmutex_t;
// blocked forms linked list of waiting threads

void acquire(bmutex_t *b) {
  again:
  lock(&b->lock);
  if(!b->acquired) {
    b->acquired = true;
    unlock(&b->lock);
  }
  else {
    self->blocked->blocked = true;
    add_self_to_blocked_queue();
    unlock(&b->lock);
    yield();
    goto again;
  }
}
void release(bmutex *b) {
  b->locked = 0;
  unlock(&b->lock);
}
```

We can generalize a mutex to for a number of concurrent threads with a semaphore.

Semaphore

The basics:

- locked when ctr <= 0
- ctr = numbers of resources left
- N threads waiting if ctr = N
- to lock, increase ctr and place self on queue

Eggert is not a huge fan of semaphores, so they were not discussed much. They are effectively a primitive for locks and condition variables. We can, however, use these to prevent thrashing!

We can also use this to solve an earlier problem; if the pipe is empty, neither reading nor writing can be done, since it appears full. This is called the *producer/consumer problem*.

SO we abstract from a semaphore to solve it with a condition variable

Condition Variable

A condition variable contains 2 parts:

1. bool isempty
2. blocking mutex (binary semaphore)

The API is as follows:

```
int pthread_cond_wait(condvar_t *c, bmutex_t *b);
// wait until the condition becomes true
int pthread_cond_signal(condvar_t *c);
// notify the first waiting thread that c is true
int pthread_cond_broadcast(condvar_t *c);
// notify all waiting threads that c is true
// Used for when there are too many variations for separate conditions
```

A wake does not guarantee a condition is met; the thread must check again.

We implement one like so

```
struct pipe {
  char buf[BUFSIZ];
  bmutex_t b;
  int r, w;
  condvar_t nonempty;
}
int pipe_read(struct pipe *p) {
  again:
  acquire(&p->b);
  if (p->w == p->r) {
    pthread_cond_signal(&p->nonempty, &p->b);
    goto again;
  }
  char c = p->buf[p->r++%BUF_SIZE];
  release(&p->b);
  notify(&p->nonempty);
  return c;
}
```

We can use these ideas to create a few thread-safe data structures:

1. Concurrent Counters:
   - Place a lock around access and incrementation
   - Not perfectly scalable, since multiprocessing greatly increases time cost
2. Scalable Counter:
   - Each thread gets its own counter.
     - Threads share one global counter, local counters are flushed after set time
     - Value is ± n_threads*flushTime
     - Time to flush locks: high = inaccurate, low = not scalable
3. Concurrent Linked Lists:
   - Locking Options:
     (a) Hand-over-hand/lock coupling
       - Each node has its own lock
       - This is slow, so we usually do not use it

     (b) List Locking
       - One lock for head and one for tail
       - Wait if queue is full or empty
4. Concurrent Hash Table:

- One lock per bucket
- Still constant time!

NOTE: With these, we still want to avoid premature optimization

There is still an optimization we can perform for small critical sections for a speedup; we cheat our way to improving locks with machine code

```
lock:
  movl $1, %eax

unlock:
  xrelease movl $0, mutex

try:
  xacquire lock xchgl %eax, mutex
  cmp $0, %eax
  jne try
  ret
# store edits into a cache & write into memory only on success
```

A few observations:

- This relies on caching for speed, so critical sections must be sufficiently small
- Note that if we did not use test and sleep, we could get a race condition
- This makes a single-threaded program slower due to overhead

BUT WAIT, we can still get bugs based on usage

```
bool copyc(struct pipe *p, struct pipe *q){
  bool ok = 0;
  acquire(&p->b);
  acquire(&q->b);
  if(p->w - p->r == 0 && q->w - q->r != 1024){
    q->buff[q->w++%1024] = p->buf[p->r++%1024];
    ok = 1;
  }
  release(&p->b);
  release(&q->b);
  return ok;
}
// multiple threads could end up waiting on each other, and the program halts
```

This situation is called a

0.1 Deadlock

There are four condition for deadlock:

1. *circular wait* := there is a cycle of dependencies
2. *mutual exclusion* := threads claim exclusive control of required resources
3. *no preemption of locks* := resources cannot be forcefully taken from threads
4. *hold & wait* := threads hold resources while waiting for more

We have a few ways to avoid deadlocks (besides just using lock-free and wait-free data structures)

1. Dynamic Deadlock Detection
   - The OS checks for possible deadlock in any thread on request & EDEADLOCK on fail
   - This pushes deadlock handling onto the app; if there is a deadlock, the OS just gives up
2. Lock Ordering

- assign a priority to each lock (often by address)
- always lock objects in order of priority
- wait on first lock failure but release all and try again if a later one fails

3. Intervention
   - kill deadlocked threads
   - is used much too often

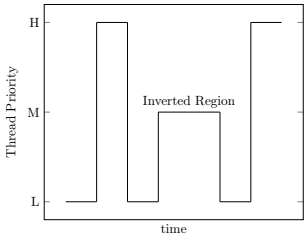
There are some types of deadlock that don't fit neatly into any solution

1. Parent/Child Communication
   If the two threads communicate via two pipes and both write a lot and don't read, both may wait on a full pipe!

2. Priority Inversion

   - High priority thread needs a lock
   - Low priority thread has that lock
   - Low priority thread is waiting on mid priority

   Then the high priority is waiting on the mid and low!

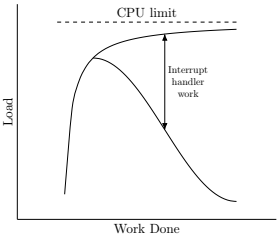


   SOLUTION:
   threads gain the priority of the highest priority dependent thread

We can also be stopped by *live-lock* := when the thread is trying to run but failing.

For example, an inundated router may spend most of its time accepting requests, since they come via interrupt, and never actually fulfill any requests. We have two possible solutions:

1. if router fills, toss requests until the router hits a low threshold
2. if requests are above a threshold, block interrupts



We discuss two of the most common non-deadlock bugs:

1. *atomicity violation* := code has an implied atomicity which is violated. This involves an *atomicity assumption* of a non-atomic action. This can be solved by locks!

```
// Thread 1::
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}

// Thread 2::
thd->proc_info = NULL;
```

2. *order violation* := the desired order of two memory accesses is flipped. This usually involves trying to act on a variable that is inialized in another thread. This can be solved by semaphores.

```
// Thread 1::
void init() {
  nThread = PR_CreateThread(nMain, ...);
}

// Thread 2::
void nMain(...) {
  nState = nThread->State;
}
```