CS3281 / CS5281

Synchronization

Will Hedgecock
Sandeep Neema
Bryan Ward

Review

- Threads enable concurrency
    - Data sharing and non-atomic operations can lead to race conditions
- Most architectures provide atomic machine instructions that provide the building blocks for synchronization primitives
- The xchg instruction is an example of such an instruction; it does two things atomically
    - Obtain the previous value of a variable
    - Update the current value

Thread Scheduling in More Detail

- Time sharing is driven by the timer interrupt
  - Calls scheduler_tick(), which invokes the scheduler to see if another thread should run
- Figure on the right shows the functions of the scheduler
  - Time sharing: every runnable thread gets a chance to run; highest priority goes first
  - Preemption: the scheduler can preempt the currently running thread and run another
  - Load balancing: the scheduler can move runnable threads to other CPUS

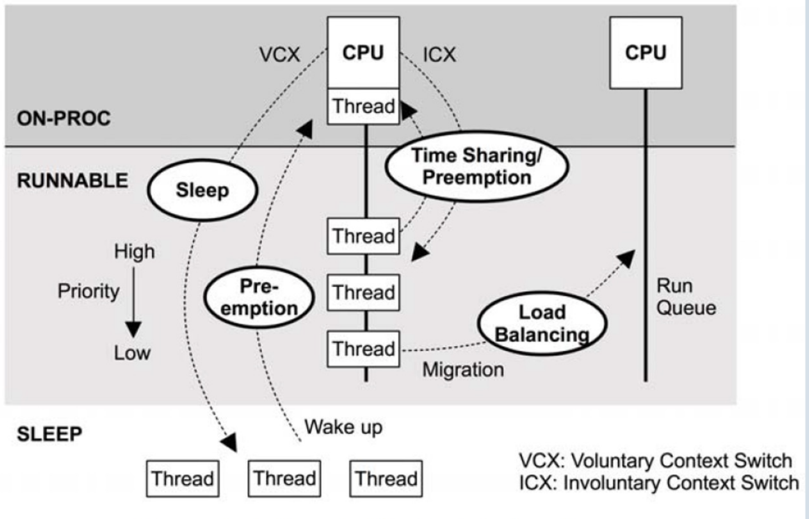


Figure 6-12 Kernel CPU scheduler functions

*Figure from *Systems Performance Enterprise and the Cloud* by Brendan Gregg

Mutexes

- Spin-locks are usually not the right solution for locking
  - They waste CPU cycles when the lock is contended
- Better solution is to *sleep* if the lock is not available
  - Have the OS wake you up when the lock is available
- Linux provides mutexes for just this purpose
  - Part of the POSIX library

pthread Mutexes

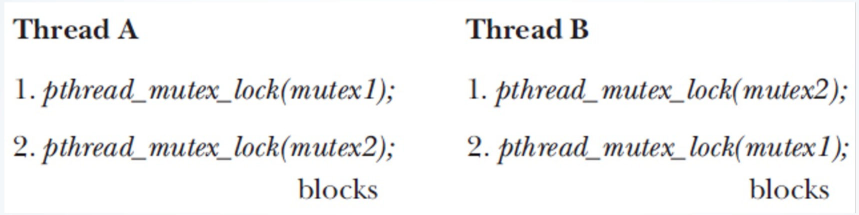
- Data type:
  - pthread_mutex_t
- Operations:
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
- Typical use:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void *thread_func(void *arg)
{
  pthread_mutex_lock(&mtx); // get the lock
  // access shared data
  pthread_mutex_unlock(&mtx); // release the lock
}
```

Deadlocks with Mutexes

- Consider the following scenario



Thread A	Thread B
1. *pthread_mutex_lock(mutex1);*	1. *pthread_mutex_lock(mutex2);*
2. *pthread_mutex_lock(mutex2);* blocks	2. *pthread_mutex_lock(mutex1);* blocks

- The result is a deadlock!
- Lesson: threads should always acquire locks in the same order

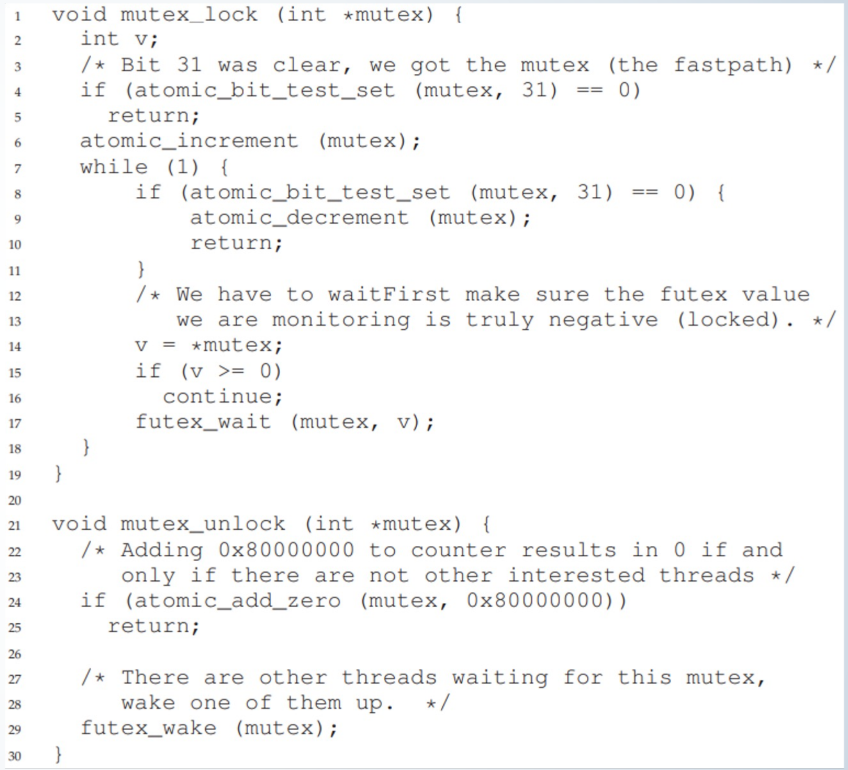
*Figure from *The Linux Programming Interface* by Michael Kerrisk

Locking the Same Lock

- What if the same thread tries to obtain the same mutex multiple times?
  - The result depends on how the mutex was initialized
- Types of pthread mutexes:
  - PTHREAD_MUTEX_DEFAULT or PTHREAD_MUTEX_NORMAL
    - Results in a deadlock if the same pthread tries to lock it a second time using the pthread_mutex_lock subroutine without first unlocking it. This is the default type.
  - PTHREAD_MUTEX_ERRORCHECK
    - Avoids deadlocks by returning a non-zero value if the same thread attempts to lock the same mutex more than once without first unlocking the mutex.
  - PTHREAD_MUTEX_RECURSIVE
    - Allows the same pthread to recursively lock the mutex using the pthread_mutex_lock subroutine without resulting in a deadlock or getting a non-zero return value from pthread_mutex_lock. The same pthread has to call the pthread_mutex_unlock subroutine the same number of times as it called pthread_mutex_lock subroutine in order to unlock the mutex for other pthreads to use.

Linux Implementation of Mutexes

- Locking:
  - Bit 31: indicates if lock is taken
    - 1: taken
    - 0 free
  - Remaining bits: number of waiters
  - Line 7: don't return from here until we get the lock
  - Lines 8-10: check lock is free and decrement # of waiters if so
    - Decrement because we incremented
  - Line 15: check if lock is taken
  - Line 17: futex system call
    - Put calling process on queue

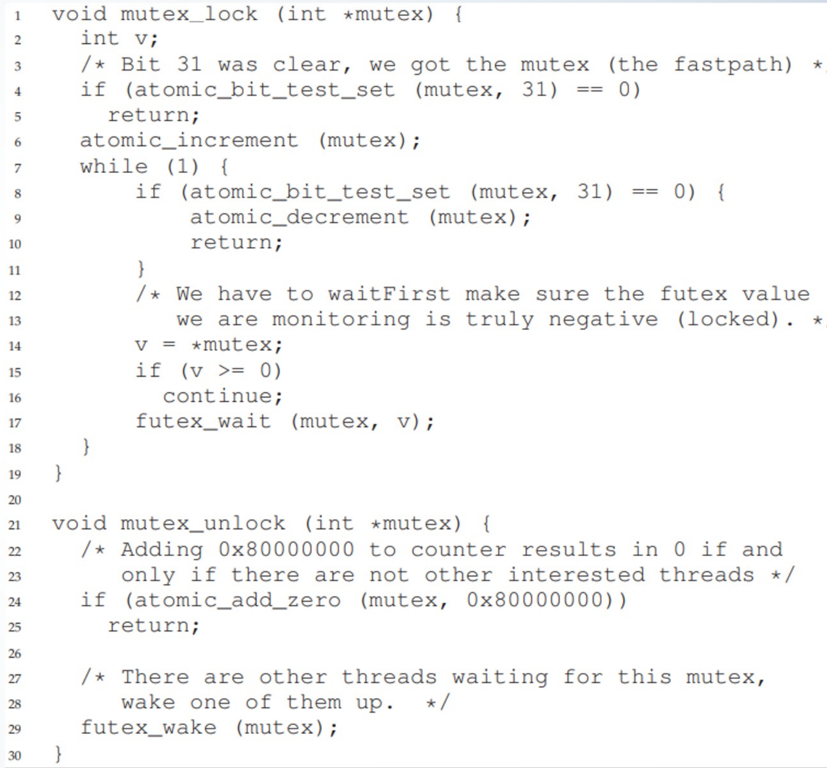


```
1   void mutex_lock (int *mutex) {
2     int v;
3     /* Bit 31 was clear, we got the mutex (the fastpath) */
4     if (atomic_bit_test_set (mutex, 31) == 0)
5       return;
6     atomic_increment (mutex);
7     while (1) {
8         if (atomic_bit_test_set (mutex, 31) == 0) {
9             atomic_decrement (mutex);
10            return;
11        }
12        /* We have to waitFirst make sure the futex value
13           we are monitoring is truly negative (locked). */
14        v = *mutex;
15        if (v >= 0)
16          continue;
17        futex_wait (mutex, v);
18    }
19  }
20
21  void mutex_unlock (int *mutex) {
22    /* Adding 0x80000000 to counter results in 0 if and
23       only if there are not other interested threads */
24    if (atomic_add_zero (mutex, 0x80000000))
25      return;
26
27    /* There are other threads waiting for this mutex,
28       wake one of them up.  */
29    futex_wake (mutex);
30  }
```

*Figure from *Operating Systems: Three Easy Pieces* by Arpaci-Dusseau and Arpaci-Dusseau

Linux Implementation of Mutexes

- Unlocking:
    - Lines 24-25: if *mutex == 0 after adding 0x80000000, then nobody was waiting
    - Line 29: invoke the futex system call
        - Argument of FUTEX_WAKE
- futex() is a multiplexed system call
    - Different arguments change the behavior
    - FUTEX_WAIT: go to sleep until mutex is available
    - FUTEX_WAKE: wake up someone waiting on this mutex
- Described in paper “Futexes are Tricky” by Ulrich Drepper



```
1   void mutex_lock (int *mutex) {
2     int v;
3     /* Bit 31 was clear, we got the mutex (the fastpath) */
4     if (atomic_bit_test_set (mutex, 31) == 0)
5       return;
6     atomic_increment (mutex);
7     while (1) {
8         if (atomic_bit_test_set (mutex, 31) == 0) {
9             atomic_decrement (mutex);
10            return;
11        }
12        /* We have to waitFirst make sure the futex value
13           we are monitoring is truly negative (locked). */
14        v = *mutex;
15        if (v >= 0)
16          continue;
17        futex_wait (mutex, v);
18    }
19  }
20
21  void mutex_unlock (int *mutex) {
22    /* Adding 0x80000000 to counter results in 0 if and
23       only if there are not other interested threads */
24    if (atomic_add_zero (mutex, 0x80000000))
25      return;
26
27    /* There are other threads waiting for this mutex,
28       wake one of them up.  */
29    futex_wake (mutex);
30  }
```

*Figure from *Operating Systems: Three Easy Pieces* by Arpaci-Dusseau and Arpaci-Dusseau

Producer-Consumer Problem

- Canonical example for condition variables: the producer-consumer problem
  - *Producer* threads produce elements
  - *Consumer* threads consume the elements produced by the producer threads
- A lock alone isn't a good solution:
  - It only ensures mutual exclusion
  - Consider the case where a consumer wants to run but there are no elements available:
    - Obtain lock
    - Check for elements
    - Release lock
    - Sleep
- Condition variables to the rescue!

Condition Variables

- A condition variable allows one thread to inform other threads about changes in the state of a shared variable (or other shared resource) and allows the other threads to wait (block) for such notification.
- Typical use:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_empty = PTHREAD_COND_INITIALIZER;

void *producer_func(void *arg)
{
  pthread_mutex_lock(&mtx);
  while (num_avail >= MAX_SIZE)
    pthread_cond_wait(&cond_empty, &mtx);
  num_avail++;
  pthread_mutex_unlock(&mtx);
  pthread_cond_signal(&cond_full);
}
```

Condition Variables (cont.)

- The mutex associated with a condition variable is for mutual exclusion
- The condition variable is for signaling
- Important: always check the condition in a while loop! From the previous slide

```
void *consumer_func(void *arg)
{
  pthread_mutex_lock(&mtx);
  while (num_avail <= 0)
    pthread_cond_wait(&cond_full, &mtx);
  // consumer data and process
  num_avail--;
  pthread_mutex_unlock(&mtx);
  pthread_cond_signal(&cond_empty);
}
```

This atomically:
1. Unlocks the mutex
2. Waits on the condition variable

When execution reaches here, you have obtained the mutex, so you must unlock it

Condition Variable Operations

- Basic operations: signaling and waiting
  - Signaling can be “broadcast” (wake up everyone) or one “signal” (wake up one waiter)
- Functions return 0 on success, non-zero on error

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

In-Class Exercise

- Use condition variables in the producer/consumer code in the repo

Semaphores

- Semaphores are another synchronization primitive that can be used for both signaling and mutual exclusion
  - Originally proposed by Dijkstra
- In real life: semaphore is a system of signals for communicating visually
  - Usually with flags or lights
- Conceptually: a data type with integer value and two operations:
  - P(): decrement value of integer; block if it would go below 0
  - V(): increment the value of the integer; wake a waiting thread (if any)

P for "prolaag", a contraction of "probeer" (Dutch for "try") and "verlaag" ("decrease");
V for the Dutch word "verhoog" which means "increase"

Semaphores

- Like an integer with three differences:
  - When you create it, you can initialize its value to any integer
    - Afterwards you can only increase by one and decrease by one
  - When a thread decrements the semaphore, if the result is negative, the thread is blocked and cannot continue until another thread increments the semaphore
  - When a thread increments the semaphore, if other threads are waiting, one waiting thread gets unblocked
- This implies:
  - There’s no way to know if a thread will block before it decrements a semaphore
  - After a thread increments a semaphore, you don’t know whether it or waiting thread that woke up will continue running
  - When you increment a semaphore, you don’t know whether there are zero or one unblocked threads

Semaphores

- Values:
  - Positive: number of threads that can decrement without blocking
  - Negative: number of threads that are blocked and waiting
  - Zero: No thread waiting, but trying to decrement will block
- A mutex can be implemented as a binary semaphore
  - Initialize the semaphore to 1
- “Counting” problems can initialize the semaphore to an arbitrary value
- Linux implementation: can be used across processes
  - Mutexes and condition variables only usable between threads of same process
- Data type: sem_t and associated functions
  - Similar to POSIX mutexes and condition variables

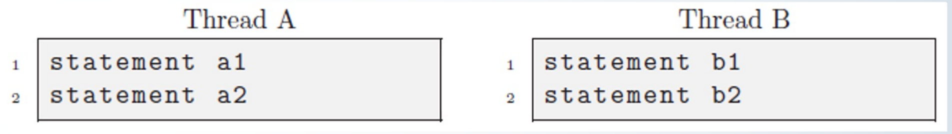
POSIX “Named” Semaphore Operations

- Named semaphores can be used across processes (just open using the same name that begins with a “/”)
  - Contrast to an unnamed semaphore: it just resides at an agreed upon location in memory

```
sem_t *sem_open(const char *, int, …, int value); // create or open an existing semaphore
int sem_close(sem_t *); // close the semaphore - do not delete it
int sem_unlink(const char *);  // delete the named semaphore
int sem_post(sem_t *); // increment value by 1
int sem_getvalue(sem_t *restrict, int *restrict); // get value of semaphore
int sem_wait(sem_t *); // decrement value by 1; block if current value == 0
int sem_timedwait(sem_t *restrict, const struct timespec *restrict); // timed version of wait
int sem_trywait(sem_t *); // non-blocking version of wait
```

Rendezvous

- Common synchronization pattern
- Two threads (A and B)
  - A has to wait for B
  - B has to wait for A
- In other words:
  - a1 happens before b2
  - b1 happens before a2



Thread A

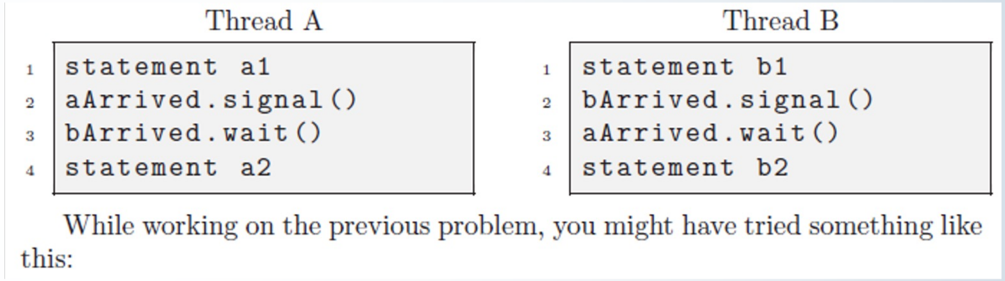
```
statement a1
statement a2
```

Thread B

```
statement b1
statement b2
```

Rendezvous Solution

- Common synchronization pattern
- Two threads (A and B)
  - A has to wait for B
  - B has to wait for A
- In other words:
  - a1 happens before b2
  - b1 happens before a2
- Solution on the right
  - Tell the other thread you've arrived
  - Then wait on the other thread



Thread A

```
statement a1
aArrived.signal()
bArrived.wait()
statement a2
```

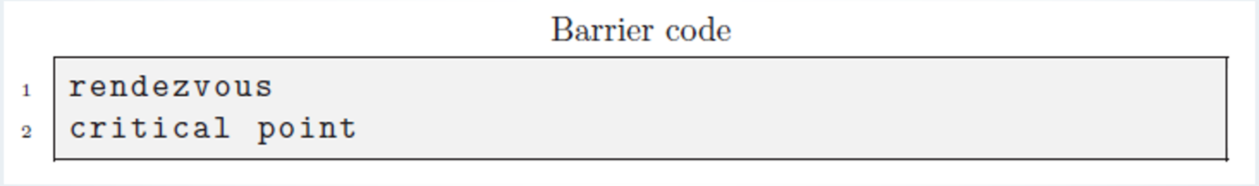
Thread B

```
statement b1
bArrived.signal()
aArrived.wait()
statement b2
```

While working on the previous problem, you might have tried something like this:

Barriers

- Another common synchronization pattern
  - “Generalizes” the Rendezvous to arbitrary number of threads
- In other words, no thread reaches the critical point until everyone has executed the rendezvous



Barrier code

```
rendezvous
critical point
```

- We’ll look at the solution next time
  - Also in “The Little Book of Semaphores” (https://greenteapress.com/wp/semaphores/)

Synchronization Implementation

- How are all of these synchronization primitives implemented?
- If no sleeping involved: with atomic machine instructions
  - Example: spin-locks use the atomic compare and exchange instruction
- If sleeping involved: with the *futex* system call
- With pthreads, the mutex, condition variable, and semaphore operations are all defined in the C library
  - “Fast” path (i.e., the lock is free): no need to transition to kernel-space (i.e., no system call)
  - “Slow” path (i.e., lock is contended): the C library invokes the futex system call

Synchronization Implementation (cont.)

- How is the futex system call implemented?
  - Quickly check again if the resource is free
  - If not, the kernel puts the calling process on a queue associated with the lock/semaphore/condition variable
  - The kernel then switches to another process
    - Putting the process on the queue and switching to another needs to happen atomically so a wake-up isn't missed;
      - Corner cases often complicate the implementation
- When another thread/process releases the lock:
  - Check if there's anyone on the queue associated with that lock
  - If so, wake them up