CS310 Operating Systems

Lecture 20: Need for Synchronization – 2

Too much Milk! and Lock

Ravi Mittal
IIT Goa

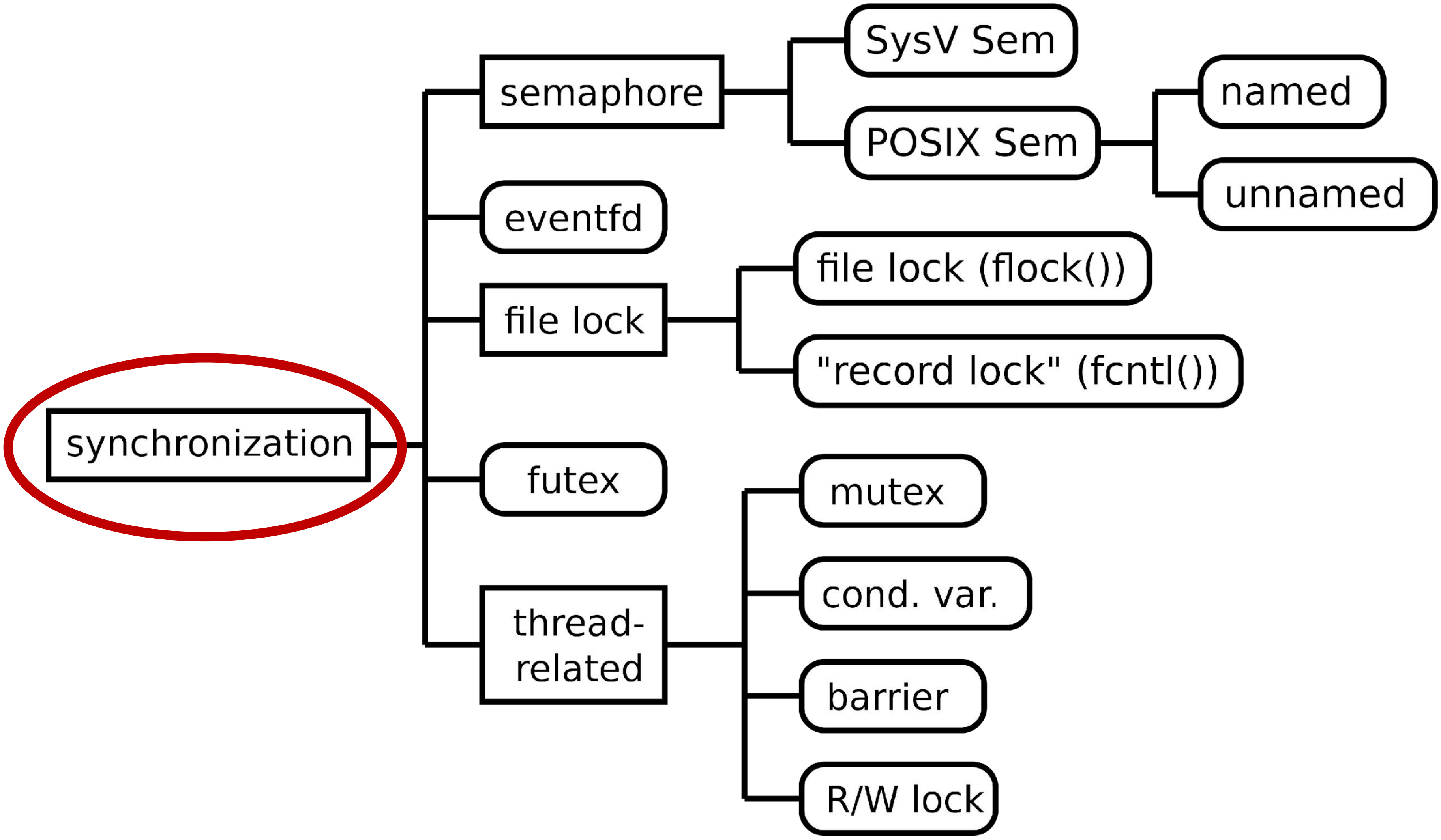
So far we have studied

- Threads
- Processes
- Concurrent execution of Threads and Processes require
  - Communication
  - Synchronization
- In the last class we looked at two cases of race condition with
  - Concurrent Processes
  - Concurrent Threads

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, University of California, Berkeley
  - Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Volume II, Chapter 5
    - Section 5.1

Needs for Synchronization



We will start with High level primitives

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Others
Hardware	Disable Ints Test&Set Compare&Swap, others

- Our focus will be on concepts

Today we will study ..

- Example: Too much Milk
- Lock definition and properties
- Pthread mutex
- Example: Counting with two threads

Too much Milk !

Motivating Example: "Too Much Milk"

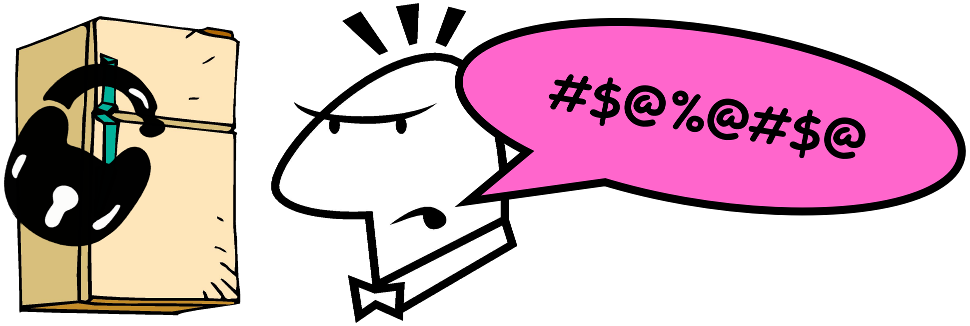
- Great thing about OS's – analogy between problems in OS and problems in real life
  - Help you understand real life problems better
  - But, computers are much stupider than people

- Example: People need to coordinate:

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Use lock to fix Milk problem

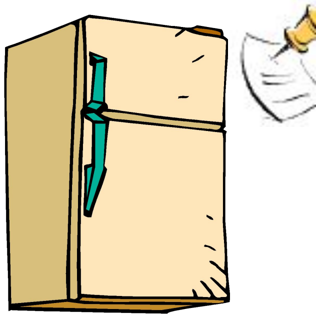
- Fix the milk problem by putting a key on the refrigerator
  - Lock it and take key if you are going to go buy milk
  - Fixes too much: roommate angry if only wants Beer



Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
- What are the correctness properties for the "Too much milk" problem???
  - Never more than one person buys
  - Someone buys if needed
- Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1 (1/3)



- **Use a note** to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this
  - Remember, only memory read/write are atomic

```
    if (noMilk) {
    if (noNote) {
       leave Note;
       buy milk;
       remove note;
    }
}
```

Does it work properly?

Too Much Milk: Solution #1 (2/3)

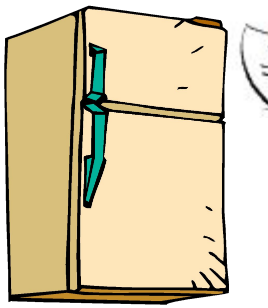
- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of "lock")
  - Remove note after buying (kind of "unlock")
  - Don't buy if note (wait)

- Suppose a computer tries this (remember, only memory read/write are atomic):

Thread A Thread B

```
if (noMilk) {
    if (noMilk) {
              if (noNote) {
        if (noNote) {
       leave Note;
          buy Milk;
          remove Note;
      }
    }
                    leave Note;
                           buy Milk;
                                         remove Note;
          }
       }
```

Too Much Milk: Solution #1 (3/3)



- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove note;
   }
}
```

- Result?
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
- Solution makes problem worse since fails intermittently
  - Makes it really hard to debug…
  - Must work despite what the dispatcher does!

Too Much Milk: Solution #1½

- Clearly the Note is not quite blocking enough
  - Let’s try to fix this by placing note first
- Another try at previous solution:

```
leave Note;
if (noMilk) {
    if (noNote) {
        buy milk;
    }
}
remove Note;
```

- What happens here?
  - Well, with human, probably nothing bad
  - With computer: no one ever buys milk

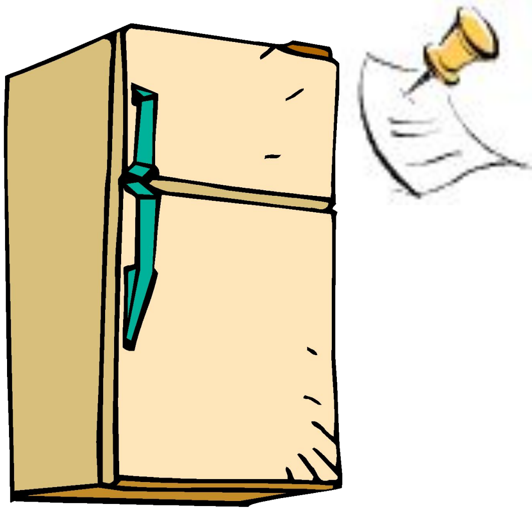
Too Much Milk: Solution #2

- How about labeled notes?
  - Now we can leave note before checking
- Algorithm looks like this:

```
        Thread A                    Thread B
    leave note A;               leave note B;
    if (noNote B) {             if (noNoteA) {
        if (noMilk) {               if (noMilk) {
            buy Milk;                   buy Milk;
        }                           }
    }                           }
    remove note A;              remove note B;
```

- Does this work?
- Possible for neither thread to buy milk
  - Context switches at exactly the wrong times can lead each to think that the other is going to buy

Too Much Milk: Solution #2: problem!



- *I'm* not getting milk, *You're* getting milk
- This kind of lockup is called starvation!

Too Much Milk Solution #3 (self reading)

- Here is a possible two-note solution:

<u>Thread A</u> <u>Thread B</u>

```
    leave note A;leave note B;
while (note B) {\\X    if (noNote A) {\\Y
   do nothing;       if (noMilk) {
}                   buy milk;
if (noMilk) {    }
   buy milk;  }
}      remove note B;
remove note A;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - If no note B, safe for A to buy,
  - Otherwise wait to find out what will happen
- At Y:
  - If no note A, safe for B to buy

Solution #3 discussion (self reading)

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
if (noMilk) {
   buy milk;
}
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex
    - Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is called "busy-waiting"
- There's a better way
  - Have hardware provide higher-level primitives than atomic load & store
  - Build even higher-level programming abstractions on this hardware support

Too Much Milk: Solution #4

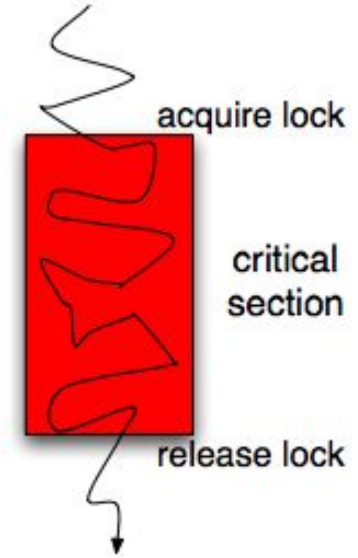
- Suppose we have some sort of implementation of a lock
  - `lock.Acquire()` – wait until lock is free, then grab
  - `lock.Release()` – Unlock, waking up anyone waiting
    - These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock
- Then, our milk problem is easy:

```
milklock.Acquire();
if (nomilk)
   buy milk;
milklock.Release();
```

Critical section

Lock Definition and use

Lock Definition



- Best mainstream solution: **Locks**
  - Implements **mutual exclusion**
    - You can’t have it if I have it, I can’t have it if you have it
- Two states of a lock: BUSY or FREE
- A lock is initially in the FREE state
- `lock.Acquire()`
  - Waits until the lock is FREE and then atomically makes the lock BUSY
  - Checking the state to see if it is FREE and setting the state to BUSY are together an atomic operation
  - Multiple threads try to acquire the lock, at most one thread will succeed
  - One thread observes that the lock is FREE and sets it to BUSY; the other threads just see that the lock is BUSY and wait

Lock Definition

- `lock.Release()`
  - makes the lock FREE
  - If there are pending acquire operations, this state change causes one of them to proceed

.

- *Only the lock "owner" should release the lock*
- Different OSs have different names for these operations of acquiring and releasing locks
  - lock()
  - Unlock()

Lock Properties

- Mutual Exclusion (safety property)
  - At most one thread holds the lock
    - Only one thread in the critical section
- Progress (liveness property)
  - If no thread holds the lock and any thread attempts to acquire the lock, then eventually some thread succeeds in acquiring the lock
- Bounded Waiting (liveness property)
  - If thread T attempts to acquire a lock, then there exists a bound on the number of times other threads can successfully acquire the lock before T does

Lock Non-Property: Thread Ordering

- The bounded waiting property guarantees that a thread will eventually get a chance to acquire the lock
- However, there is no guarantee that waiting threads acquire the lock in FIFO order

Lock Use

- Consider an update of shared variable
    - balance = balance + 1;
- This statement forms critical section. We need to protect it with a special lock variable (mutex in the example below)

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

- Note that lock is a variable that holds the state of the lock at any instant in time
- All threads accessing a critical section share a lock
- Only one thread becomes successful in holding the lock – current owner of the lock
    - Thus the thread is in critical section

Lock Use

- `lock()` and `unlock()` are routines
- The thread that acquires the lock enters the critical section
- Then, if another thread calls the lock on the same lock variable, it will not return
  - So the thread is prevented from entering critical section
  - Function `lock()` doesn't return means it doesn't come out of `lock()` function
- Note that the owner of lock calls unlock
- Thread entities are created by the programmer but scheduled by the OS
  - Locks give some control to the programmer

- `pthreads` library in Linux provides such locks

Pthread lock calls

Pthread lock calls

- `pthread_mutex_init`
  - Creates a new lock in the unlocked state
- `pthread_mutex_lock`
  - When the lock is unlocked, change the lock to the locked state and advance to the next line of code.
  - When the lock is locked, this function **blocks** execution of other threads until the lock can be acquired
- `pthread_mutex_unlock`
  - Moves the lock to the unlocked state

- Let’s use these APIs in our example.

OS Library Locks: *pthreads*

```
int pthread_mutex_init(pthread_mutex_t *mutex,
               const pthread_mutexattr_t *attr)

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Mutex Variables

- A typical sequence in the use of a mutex
  1. Create and initialize **`mutex`**
  2. Several threads attempt to lock **`mutex`**
  3. Only one succeeds and now owns **`mutex`**
  4. The owner performs some set of actions
  5. The owner unlocks **`mutex`**
  6. Another thread acquires **`mutex`** and repeats the process
  7. Finally **`mutex`** is destroyed

Example

thread3.c

```c
int ct = 0;

void *thread_start(void *ptr) {
  int countTo = *((int *)ptr);

  int i;
  for (i = 0; i < countTo; i++) {
    ct = ct + 1;
  }

  return NULL;
}

int main(int argc, char *argv[]) {
  // Parse Command Line:
  if (argc != 3)   {
    printf("Usage: %s <countTo> <thread count>\n", argv[0]);
    return 1;
  }

  const int countTo = atoi(argv[1]);
  if (countTo == 0) { printf("Valid `countTo` is required.\n"); return 1; }

  const int thread_ct = atoi(argv[2]);
  if (thread_ct == 0) { printf("Valid thread count is required.\n"); return 1; }

  // Create threads:
  int i;
  pthread_t tid[thread_ct];
  for (i = 0; i < thread_ct; i++) {
    pthread_create(&tid[i], NULL,
                                 thread_start, (void *)&countTo);
  }

  // Join threads:
  for (i = 0; i < thread_ct; i++) {
    pthread_join(tid[i], NULL);
  }

  // Display result:
  printf("Final Result: %d\n", ct);
  return 0;
}
```

thread3.c - Observations

- Command line inputs
  - countTo : Each thread counts upto CountTo
  - thread_conunt: number of threads
- For example:
  - countTo = 1000
  - thread_count = 2
  - □ final result = 2000
- However, we see that we don't get exact result for all values of countTo and thread_count
  - At architecture level ct = ct + 1 is not atomic
  - For example for the first thread – after a few counts – it is context switched after a few steps:
    - Lw s1, ct (assuming ct is in a memory location)
    - Addi s1, 1
    - Context switched…. Other threads now change ct variable
    - After some time the first thread completes operation

Synchronization using mutex

- Mutex : mutual exclusive (to each thread)

- The simplest way to protect a region of code from being accessed is through the use of a mutex lock
  - mutex stands for *mutual exclusive*

thread3.c

```c
int ct = 0;

void *thread_start(void *ptr) {
  int countTo = *((int *)ptr);

  int i;
  for (i = 0; i < countTo; i++) {
    ct = ct + 1;
  }

  return NULL;
}

int main(int argc, char *argv[]) {
  // Parse Command Line:
  if (argc != 3)  {
    printf("Usage: %s <countTo> <thread count>\n",
argv[0]);
    return 1;
  }

  const int countTo = atoi(argv[1]);
  if (countTo == 0) { printf("Valid `countTo` is
required.\n"); return 1; }

  const int thread_ct = atoi(argv[2]);
  if (thread_ct == 0) { printf("Valid thread count is
required.\n"); return 1; }

  // Create threads:
  int i;
  pthread_t tid[thread_ct];
  for (i = 0; i < thread_ct; i++) {
    pthread_create(&tid[i], NULL,
                   thread_start, (void *)&countTo);
  }

  // Join threads:
  for (i = 0; i < thread_ct; i++) {
    pthread_join(tid[i], NULL);
  }

  // Display result:
  printf("Final Result: %d\n", ct);
  return 0;
}
```

Non-atomic operation
Critical section – that should be accessed by only one thread at a time

thread3d.c

```c
pthread_mutex_t lock;
int ct = 0;

void *thread_start(void *ptr) {
  int countTo = *((int *)ptr);

  int i;
  for (i = 0; i < countTo; i++) {
    pthread_mutex_lock(&lock);
    ct = ct + 1;
    pthread_mutex_unlock(&lock);
  }

  return NULL;
}

int main(int argc, char *argv[]) {
  // Parse Command Line:
  if (argc != 3)  {
    printf("Usage: %s <countTo> <thread count>\n",
argv[0]);
    return 1;
  }

  const int countTo = atoi(argv[1]);
  if (countTo == 0) { printf("Valid `countTo` is
required.\n"); return 1; }

  const int thread_ct = atoi(argv[2]);
  if (thread_ct == 0) { printf("Valid thread count is
required.\n"); return 1; }

  // Create Lock:
  pthread_mutex_init(&lock, NULL);
```

1 Declare lock variable

3 Locking and unlocking Critical section

2 initialize lock to null (open)

Summary

- Concurrent threads are a very useful abstraction
- Concurrent threads introduce problems when accessing shared data
  - Programs must be insensitive to arbitrary interleavings
  - Without careful design, shared variables can become completely inconsistent
- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- We will study several mechanisms of implementing locks

Backup Slides

pthread lock calls

- The name that the POSIX library uses for a lock is a **mutex**

  - `pthread_mutex_init()` - get a mutex
  - `pthread_mutex_lock()` - lock a mutex (acquire it), block until available
  - `pthread_mutex_trylock()` - try to lock a mutex (acquire it), do not block if unavailable
  - `pthread_mutex_unlock()` - unlock a mutex (release it)
  - `pthread_mutex_destroy()` - destroy a mutex (remove it)

The name that the POSIX library uses for a lock is a mutex,

Pthread Synchronization

- Two primitives
  - Mutex
    - Semaphore with maximum value 1
  - Condition variable
    - Provides a shared signal
    - Combined with a mutex for synchronization

Pthread Mutex

- States
  - Locked
    - Some thread holds the mutex
  - Unlocked
    - No thread holds the mutex
- When several threads compete
  - One wins
  - The rest block
    - Queue of blocked threads

Mutex Variables

- A typical sequence in the use of a mutex
  1. Create and initialize `mutex`
  2. Several threads attempt to lock `mutex`
  3. Only one succeeds and now owns `mutex`
  4. The owner performs some set of actions
  5. The owner unlocks `mutex`
  6. Another thread acquires `mutex` and repeats the process
  7. Finally `mutex` is destroyed

Creating a mutex

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
  const pthread_mutexattr_t *attr);
```

- Initialize a pthread mutex: the mutex is initially unlocked
- Returns
  - 0 on success
  - Error number on failure
    - **`EAGAIN:`** The system lacked the necessary resources; **`ENOMEM:`** Insufficient memory ; **`EPERM:`** Caller does not have privileges; **`EBUSY:`** An attempt to re-initialise a mutex; **`EINVAL:`** The value specified by attr is invalid
- Parameters
  - **`mutex`**: Target mutex
  - **`attr`**:
    - NULL: the default mutex attributes are used
    - Non-NULL: initializes with specified attributes

Creating a mutex

- Default attributes
  - Use **`PTHREAD_MUTEX_INITIALIZER`**
    - Statically allocated
    - Equivalent to dynamic initialization by a call to **`pthread_mutex_init()`** with parameter **`attr`** specified as NULL
    - No error checks are performed

Destroying a mutex

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Destroy a pthread mutex
- Returns
    - 0 on success
    - Error number on failure
        - **EBUSY:** An attempt to re-initialise a mutex; **EINVAL:** The value specified by attr is invalid
- Parameters
    - **mutex**: Target mutex

Locking/unlocking a mutex

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Returns
    - 0 on success
    - Error number on failure
        - **`EBUSY:`** already locked; **`EINVAL:`** Not an initialised mutex; **`EDEADLK:`** The current thread already owns the mutex; **`EPERM:`** The current thread does not own the mutex

How to implement Locks?

- How can we build multi-instruction atomic operations?
  - Recall: dispatcher gets control in two ways
    - Internal: Thread does something to relinquish the CPU
    - External: Interrupts cause dispatcher to take CPU
  - On a uniprocessor, can avoid context-switching by:
    - Avoiding internal events
    - Preventing external events by disabling interrupts

Naïve use of Interrupt Enable/Disable

- Naïve Implementation of locks:

```
LockAcquire { disable Ints; }
LockRelease { enable Ints; }
```

- Problems with this approach:
  - Can't let user do this!
  - Consider following:

```
  LockAcquire();
While(TRUE) {;}   // infinite loop
```

- Critical Sections might be arbitrarily long
- What happens with I/O or other important events?
  - Flight control

backup

Too Much Milk Solution #3 (self reading)

- Here is a possible two-note solution:

Thread A   Thread B

```
    leave note A;leave note B;
while (note B) {\\X    if (noNote A) {\\Y
   do nothing;      if (noMilk) {
}               buy milk;
if (noMilk) {    }
   buy milk;  }
}      remove note B;
remove note A;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - If no note B, safe for A to buy,
  - Otherwise wait to find out what will happen
- At Y:
  - If no note A, safe for B to buy

Solution #3 discussion (self reading)

- Our solution protects a single "Critical-Section" piece of code for each thread:

```
      if (noMilk) {
         buy milk;
   }
```

- Solution #3 works, but it's really unsatisfactory
  - Really complex
    - Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is called "busy-waiting"
- There's a better way
  - Have hardware provide higher-level primitives than atomic load & store
  - Build even higher-level programming abstractions on this hardware support

Case 1

- “`leave note A`” happens before “`if (noNote A)`”

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Case 1

- “`leave note A`” happens before “`if (noNote A)`”

```
leave note A;
while (note B) {\\X
    do nothing;
};
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Wait for note B to be remove

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happene
d
before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Case 2

- “`if (noNote A)`” happens before “`leave note A`”

```
leave note A;
while (note B) {\\X
    do nothing;
};
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

happene
d
before

```
leave note A;
while (note B) {\\X
    do nothing;
};
```

Wait for
note B to
be remove

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Atomic Operations

- Indivisible operations that cannot be interleaved with or split by other operations

- What is the need for atomic operations?