# CPSC 213

## Introduction to Computer Systems

*Unit 2c*

***Synchronization***

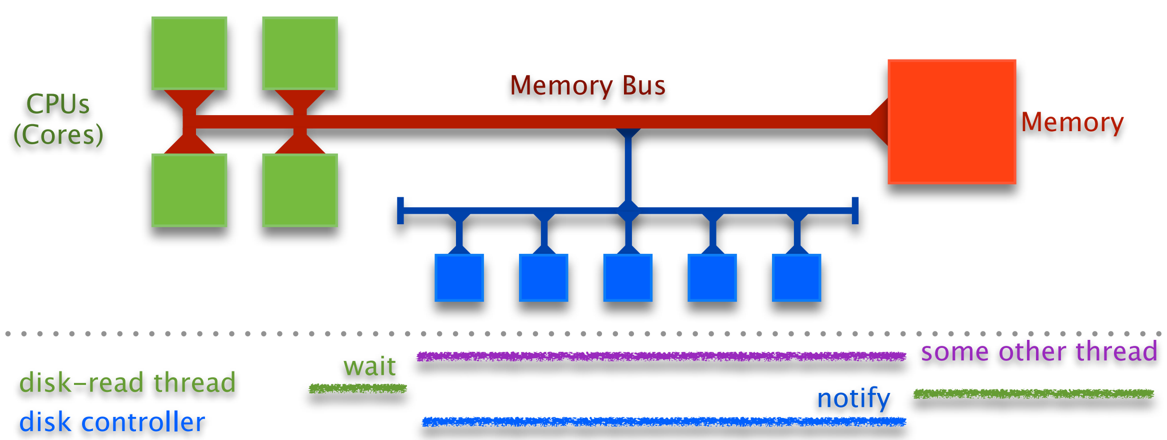


# Reading

- Companion
  - 6 *(Synchronization)*
- Text
  - *Shared Variables in a Threaded Program, Synchronizing Threads with Semaphores, Using Threads for Parallelism, Other Concurrency Issues*
  - 2ed: 12.4-12.6, parts of 12.7
  - 1ed: 13.4-13.5, (no equivalent to 12.6), parts of 13.7



# Synchronization




- We invented Threads to
  - **exploit parallelism** do things at the same time on different processors
  - **manage asynchrony** do something else while waiting for I/O Controller
- But, we now have two problems
  - coordinating access to memory (variables) shared by multiple threads
  - control flow transfers among threads (wait until notified by another thread)
- Synchronization is the mechanism threads use to
  - ensure ***mutual exclusion*** of critical sections
  - wait for and notify of the occurrence of events



# The Importance of Mutual Exclusion

- Shared data
  - data structure that could be accessed by multiple threads
  - typically concurrent access to shared data is a bug
- Critical Sections
  - sections of code that access shared data
- Race Condition
  - simultaneous access to critical section section by multiple threads
  - conflicting operations on shared data structure are arbitrarily interleaved
  - unpredictable (non-deterministic) program behaviour — usually a bug (a serious bug)
- Mutual Exclusion
  - a mechanism implemented in software (with some special hardware support)
  - to ensure critical sections are executed by one thread at a time
  - though reading and writing should be handled differently (more later)
- For example
  - consider the implementation of a shared stack by a linked list ...

## Stack implementation

```
void push_st (struct SE* e) {
  e->next = top;
  top     = e;
}
```

```
struct SE {
  struct SE* next;
};
struct SE *top=0;
```

```
struct SE* pop_st () {
  struct SE* e = top;
  top = (top)? top->next: 0;
  return e;
}
```

## Sequential test works

```
void push_driver (long int n) {
  struct SE* e;
  while (n--)
    push ((struct SE*) malloc (...));
}
```

```
void pop_driver (long int n) {
  struct SE* e;
  while (n--) {
    do {
      e = pop ();
    } while (!e);
    free (e);
  }
}
```

```
push_driver (n);
pop_driver  (n);
assert      (top==0);
```



## concurrent test doesn't always work

```
et = uthread_create ((void* (*)(void*)) push_driver, (void*) n);
dt = uthread_create ((void* (*)(void*)) pop_driver,  (void*) n);
uthread_join (et);
uthread_join (dt);
assert (top==0);
```

**malloc: *** error for object 0x1022a8fa0: pointer being freed was not allocated**

## what is wrong?

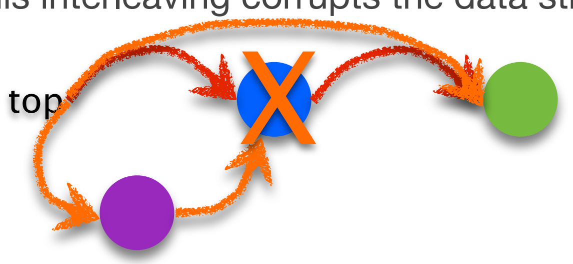
```
void push_st (struct SE* e) {
  e->next = top;
  top     = e;
}
```

```
struct SE* pop_st () {
  struct SE* e = top;
  top = (top)? top->next: 0;
  return e;
}
```



## The bug

- push and pop are critical sections on the shared stack
- they run in parallel so their operations are arbitrarily interleaved
- sometimes, this interleaving corrupts the data structure




```
void push_st (struct SE* e) {
  e->next = top;
  top     = e;
}
```

```
struct SE* pop_st () {
  struct SE* e = top;
  top = (top)? top->next: 0;
  return e;
}
```

1. e->next = top

2. e = top
3. top = top->next
4. return e
5. free e

6. top = e



# Mutual Exclusion using Locks

## lock semantics

- a lock is either *held* by a thread or *available*
- at most one thread can hold a lock at a time
- a thread attempting to acquire a lock that is already held is forced to wait

## lock primitives

- **lock** acquire lock, wait if necessary
- **unlock** release lock, allowing another thread to acquire if waiting

## using locks for the shared stack

```
void push_cs (struct SE* e) {
  lock (&aLock);
  push_st (e);
  unlock (&aLock);
}
```

```
struct SE* pop_cs () {
  struct SE* e;
  lock (&aLock);
  e = pop_st ();
  unlock (&aLock);

  return e;
}
```

# Implementing Simple Locks

- Here's a first cut
  - use a shared global variable for synchronization
  - **lock** loops until the variable is 0 and then sets it to 1
  - **unlock** sets the variable to 0

```
int lock = 0;
```

```
void lock (int* lock) {
  while (*lock==1) {}
  *lock = 1;
}
```

```
void unlock (int* lock) {
  *lock = 0;
}
```

  - why doesn't this work?



- We now have a race in the lock code

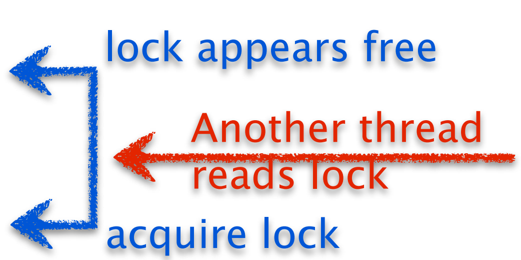
| Thread A | Thread B |
|---|---|
| `void lock (int* lock) { while (*lock==1) {} *lock = 1; }` | `void lock (int* lock) { while (*lock==1) {} *lock = 1; }` |
| 1. read *lock==0, exit loop | |
| | 2. read *lock==0, exit loop |
| 3. *lock = 1<br>4. return with lock held | |
| | 5. *lock = 1, return<br>6. return with lock held |

Both threads think they hold the lock ...



- The race exists even at the machine-code level
  - two instructions acquire lock: one to read it free, one to set it held
  - but read by another thread and interpose between these two

```
      ld $lock, r1
      ld $1,    r2

loop: ld  (r1), r0
      beq r0, free
      br  loop

free: st  r2, (r1)
```




| Thread A | Thread B |
|---|---|
| ld (r1), r0 | |
| | ld (r1), r0 |
| st r2, (r1) | |
| | st r2, (r1) |

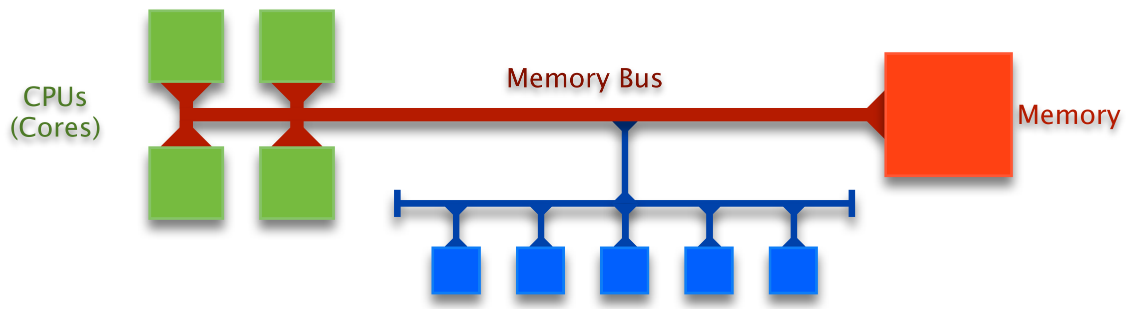


# Atomic Memory Exchange Instruction

- We need a new instruction
  - to ***atomically*** read **and** write a memory location
  - with no intervening access to that memory location from any other thread allowed
- Atomicity
  - is a general property in systems
  - where a group of operations are performed as a single, indivisible unit
- The Atomic Memory Exchange
  - one type of atomic memory instruction (there are other types)
  - group a load and store together atomically
  - exchanging the value of a register and a memory location

| Name | Semantics | Assembly |
|---|---|---|
| *atomic exchange* | r[v] ← m[r[a]]<br>m[r[a]] ← r[v] | xchg (ra), rv |

# Implementing Atomic Exchange




- Can not be implemented just by CPU
  - must synchronize across multiple CPUs
  - accessing the same memory location at the same time
- Implemented by Memory Bus
  - memory bus synchronizes every CPUs access to memory
  - the two parts of the exchange (read + write) are coupled on bus
  - bus ensures that no other memory transaction can intervene
  - this instruction is **much slower**, higher overhead than normal read or write



# Spinlock

- A Spinlock is
  - a lock where waiter *spins* on looping memory reads until lock is acquired
  - also called "busy waiting" lock
- Simple implementation using Atomic Exchange
  - spin on atomic memory operation
  - that attempts to acquire lock while
  - atomically reading its old value

```
      ld   $lock, r1
      ld   $1, r0
loop: xchg (r1), r0
      beq  r0, held
      br   loop
held:
```

  - but there is a problem: atomic-exchange is an expensive instruction



# Implementing Spinlocks Efficiently

- Spin first on fast normal read, then try slow atomic exchange
  - use normal read in loop until lock appears free
  - when lock appears free use exchange to try to grab it
  - if exchange fails then go back to normal read

```
      ld   $lock, %r1
loop: ld   (%r1), %r0
      beq  %r0, try
      br   loop
try:  ld   $1, %r0
      xchg (%r1), %r0
      beq  %r0, held
      br   loop
held:
```

- Busy-waiting pros and cons
  - Spinlocks are necessary and okay if spinner only waits a short time
  - But, using a spinlock to wait for a long time, wastes CPU cycles



# Blocking Locks

- If a thread may wait a long time
  - it should block so that other threads can run
  - it will then unblock when it becomes runnable (lock available or event notification)
- Blocking locks for mutual exclusion
  - if lock is held, locker puts itself on waiter queue and blocks
  - when lock is unlocked, unlocker restarts one thread on waiter queue
- Blocking locks for event notification
  - waiting thread puts itself on a a waiter queue and blocks
  - notifying thread restarts one thread on waiter queue (or perhaps all)
- Implementing blocking locks presents a problem
  - lock data structure includes a waiter queue and a few other things
  - data structure is shared by multiple threads; lock operations are critical sections
  - mutual exclusion can be provided by blocking locks (they aren't implemented yet)
  - and so, we need to use spinlocks to implement blocking locks (this gets tricky)

# Implementing a Blocking Lock

```
void lock (struct blocking_lock l) {
  spinlock_lock (&l->spinlock);
    while (l->held) {
      enqueue        (&waiter_queue, uthread_self ());
      spinlock_unlock (&l->spinlock);
      uthread_switch  (ready_queue_dequeue (), TS_BLOCKED);
      spinlock_lock   (&l->spinlock);
    }
    l->held = 1;
  spinlock_unlock (&l->spinlock);
}
```

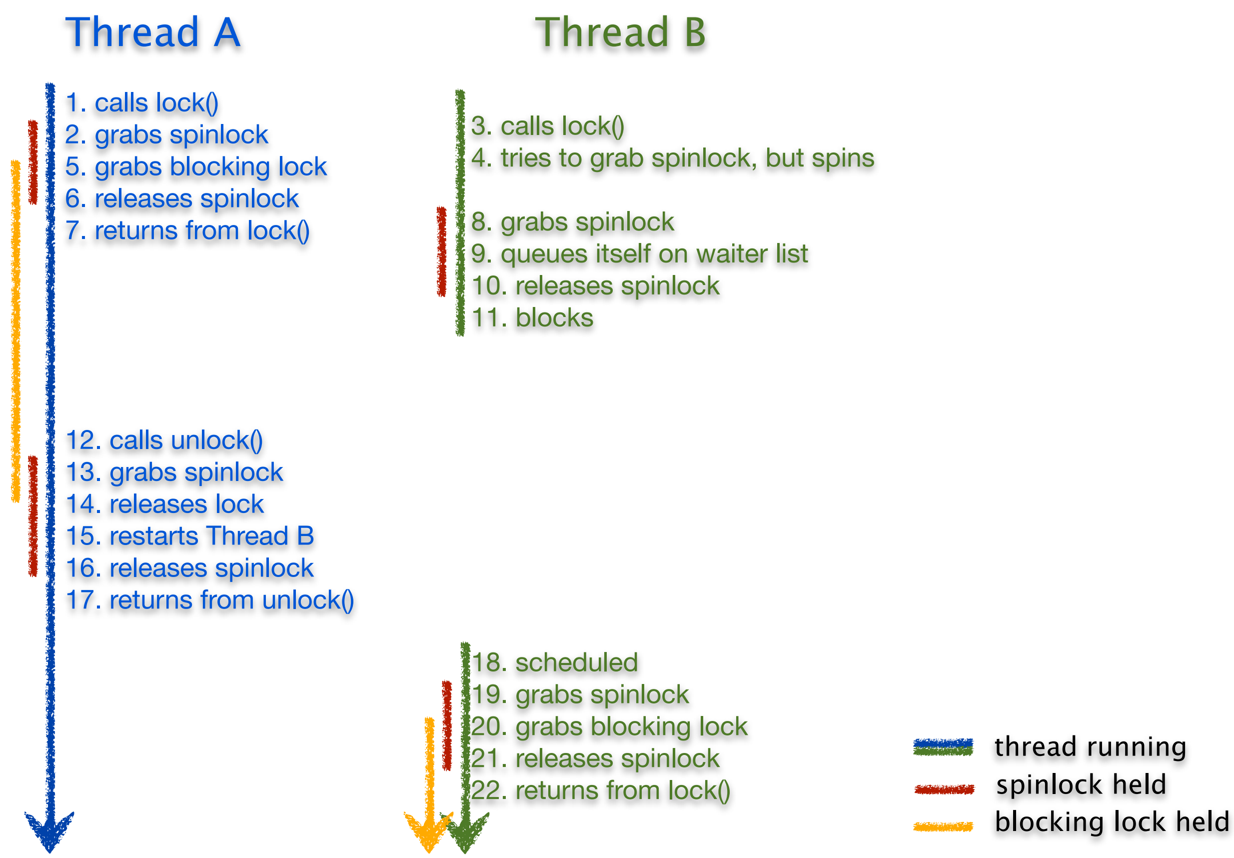
```
void unlock (struct blocking_lock l) {
  uthread_t* waiter_thread;

  spinlock_lock (&l->spinlock);
  l->held = 0;
  waiter_thread = dequeue (&l->waiter_queue);
  spinlock_unlock (&->spinlock);
  if (waiter_thread) {
    waiter_thread->state = TS_RUNNABLE;
    ready_queue_enqueue (waiter_thread);
  }
}
```

```
struct blocking_lock {
  spinlock_t         spinlock;
  int                held;
  uthread_queue_t  waiter_queue;
};
```

- Spinlock guard
  - on for critical sections
  - off before thread blocks



# Blocking Lock Example Scenario





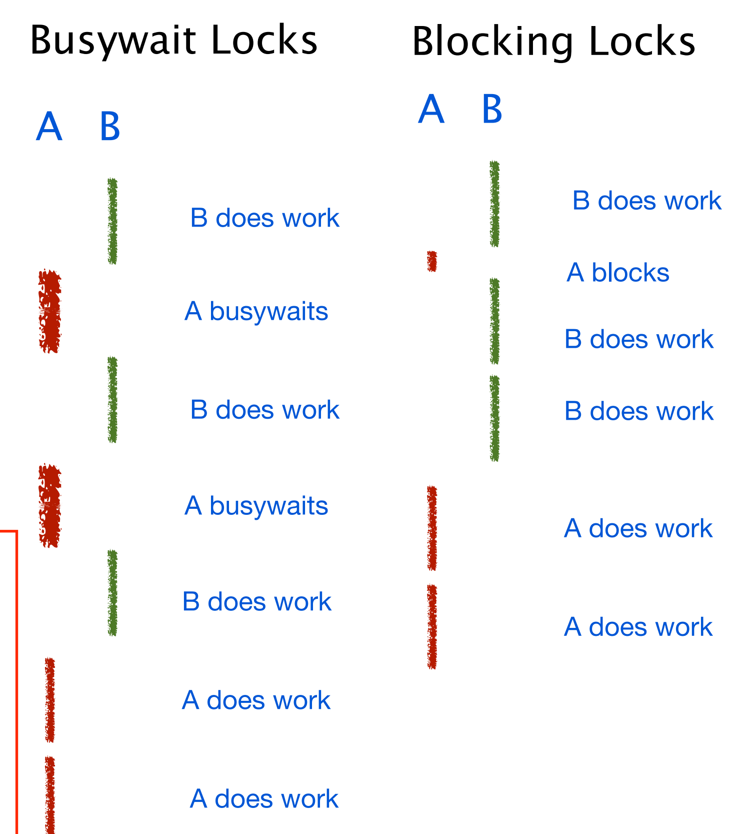

# Blocking vs Busy Waiting

- Spinlocks
  - Pros and Cons
    - uncontended locking has low overhead
    - contending for lock has high cost
  - Use when
    - critical section is small
    - contention is expected to be minimal
    - event wait is expected to be very short
    - when implementing Blocking locks
- Blocking Locks
  - Pros and Cons
    - uncontended locking has higher overhead
    - contending for lock has no cost
  - Use when
    - lock may be held for some time
    - when contention is high
    - when event wait may be long



# Busywaiting vs Blocking

- Using spinlocks to busywait for long time wastes CPU cycles
  - use for short things
    - including within implementation of blocking locks
- Using blocking locks has high overhead
  - use for long things
- Common mistake
  - assume that CPU is busywaiting during blocking locks
    - thread does not run again until after blocking lock is released



# Locks and Loops Common Mistakes

- Confusion about spinlocks inside blocking locks
  - use spinlocks in the implementation of blocking locks
  - two separate levels of lock!
    - holding spinlock guarding variable read/write
    - holding actual blocking lock
- Confusion about when spinlocks needed
  - must turn on to guard access to shared variables
  - must turn off before finishing or blocking
- Confusion about loop function
  - busywait
    - only inside spinlock
  - thread blocked inside loop body, **not** busywaiting
    - yield for blocking lock
    - re-check for desired condition: is lock available?



# Synchronization Abstractions

- Monitors and condition variables
  - monitor guarantees mutual exclusion with blocking locks
  - condition variable provides control transfer among threads with wait/notify
  - abstraction supports explicit locking
- Semaphores
  - blocking atomic counter, stop thread if counter would go negative
  - introduced to coordinate asynchronous resource use
  - abstraction implicitly supports mutex, no need for explicit locking by user
  - use to implement monitors, barriers (and condition variables, sort of)



# Monitors and Conditions

- Mutual exclusion plus inter-thread synchronization
  - introduced by Tony Hoare and Per Brinch Hansen circa 1974
  - abstraction supporting explicit locking
    - basis for synchronization primitives in Java etc.
- Monitor
  - monitor guarantees mutual exclusion with blocking locks
  - primitives are enter (lock) and exit (unlock)
- Condition Variable
  - allows threads to synchronize with each other (provides control transfer between threads):
  - **wait** blocks until a subsequent signal operation on the variable
  - **notify** unblocks waiter, but continues to hold monitor (Hansen)
  - **notify_all** unblocks all waiters and continues to hold monitor
  - can only be accessed from inside of a monitor (i.e, with monitor lock held)



# Monitors

- Provides mutual exclusion with blocking lock
  - **enter** lock
  - **exit** unlock

```
void doSomething (uthread_monitor_t* mon) {
  uthread_monitor_enter (mon);
   touchSharedMemory();
  uthread_monitor_exit (mon);
}
```

- Standard case: assume all threads could overwrite shared memory.
  - mutex: only allows access one at a time
- Special case: distinguish read-only access (readers) from threads that change shared memory values (writers).
  - mutex: allow multiple readers but only one writer

# Condition Variables

- Mechanism to transfer control back and forth between threads
  - uses monitors: CV can only be accessed when monitor lock is held
- Primitives
  - **wait** blocks until a subsequent **notify** operation on the variable
  - **notify** unblocks one waiter, continues to hold monitor
  - **notify_all** unblocks all waiters (broadcast), continues to hold monitor
- Each CV associated with a monitor
- Multiple CVs can be associated with same monitor
  - independent conditions, but guarded by same mutex lock

```
uthread_monitor_t* beer = uthread_monitor_create ();
```

```
uthread_cv_t* not_empty = uthread_cv_create (beer);
uthread_cv_t* warm      = uthread_cv_create (beer);
```



# Using Conditions

- Basic formulation
  - one thread enters monitor and may wait for a condition to be established

```
monitor {
  while (!x)
    wait ();
}
```

  - another thread enters monitor, establishes condition and signals waiter

```
monitor {
  x = true;
  notify ();
}
```

- **wait** exits the monitor and blocks thread
  - before waiter blocks, it exits monitor to allow other threads to enter
  - when wait unblocks, it re-enters monitor, waiting/blocking to enter if necessary
  - note: other threads may have been in monitor between wait call and return



- **notify** awakens one thread
  - does not release monitor
  - waiter does not run until notifier exits monitor
  - a third thread could intervene and enter monitor before waiter
  - waiter must thus re-check wait condition

```
monitor {
  x = true;
  notify ();
}
```

```
monitor {
  while (!x)
    wait ();
}
```

And not

```
monitor {
  if (!x)
    wait ();
}
```

- **notify_all** awakens all threads
  - may wake up too many
  - okay since threads re-check wait condition and re-wait if necessary

```
monitor {
  x += n;
  notify_all ();
}
```

```
monitor {
  while (!x)
    wait ();
}
```



# Drinking Beer Example

- Beer pitcher is shared data structure with these operations
  - **pour** from pitcher into glass
  - **refill** pitcher
- Implementation goal
  - synchronize access to the shared pitcher
  - pouring from an empty pitcher requires waiting for it to be filled
  - filling pitcher releases waiters

```
void pour () {
  monitor {
    while (glasses==0)
      wait ();
    glasses--;
}}
```

```
void refill (int n) {
  monitor {
    for (int i=0; i<n; i++) {
      glasses++;
      notify ();
}}}
```

or

```
monitor {
  glasses+=n;
  notify_all ();
}
```

# Wait and Notify Semantics

- Monitor automatically exited before block on wait
  - before waiter blocks, it exits monitor to allow other threads to enter
- Monitor automatically re-entered before return from wait
  - when trying to return from wait after notify, thread may block again until monitor can be entered (if monitor lock held by another thread)
- Monitor stays locked after notify: does not block
- Implication: cannot assume desired condition holds after return from blocking wait
  - other threads may have been in monitor between wait call and return
    - must explicitly re-check: usually enclose wait in while loop with condition check
    - same idea as blocking lock implementation with spinlocks!

```
void pour () {
  monitor {
    while (glasses==0)
      wait;
    glasses--;
  }}
```

```
void refill (int n) {
  monitor {
    for (int i=0; i<n; i++) {
      glasses++;
      notify;
    }}}
```



# Monitors and Condition Variables

- Programs can have multiple independent monitors
  - so a monitor implemented as a “variable” (a struct really)

```
uthread_monitor_t* beer = uthread_monitor_create ();
```

- Monitors may have multiple independent conditions
  - so a condition is also a variable, connected to its monitor

```
uthread_cv_t* not_empty = uthread_cv_create (beer);
uthread_cv_t* warm      = uthread_cv_create (beer);
```

```
void pour (int isEnglish) {
  uthread_monitor_enter (beer);
    while (glasses==0 || (isEnglish && temp<15)) {
      if (glasses==0)
        uthread_cv_wait (not_empty);
      if (isEnglish && temp < 15)
        uthread_cv_wait (warm);
    }
  glasses--;
  uthread_monitor_exit (beer);
}
```



# Using Condition Variables for Disk Read

- Blocking read
  - call async read as before
  - but now block on condition variable that is given to completion routine

```
void read (char* buf, int bufSize, int blockNo) {
  uthread_monitor_t* mon = uthread_monitor_create ();
  uthread_cv_t*      cv  = uthread_cv_create (mon);
  uthread_monitor_enter (mon);
    asyncRead (buf, bufSize, readComplete, mon, cv);
    uthread_cv_wait (cv);
  uthread_monitor_exit (mon);
}
```

- Read completion
  - called by disk ISR as before
  - but now notify the condition variable, restarting the blocked read cal

```
void readComplete (uthread_monitor_t* mon, uthread_cv_t* cv) {
  uthread_monitor_enter (mon);
    uthread_cv_notify (cv);
  uthread_monitor_exit (mon);
}
```



# Shared Queue Example

- Unsynchronized Code

```
void enqueue (uthread_queue_t* queue, uthread_t* thread) {
  thread->next = 0;
  if (queue->tail)
    queue->tail->next = thread;
  queue->tail = thread;
  if (queue->head==0)
    queue->head = queue->tail;
}

uthread_t* dequeue (uthread_queue_t* queue) {
  uthread_t* thread;
  if (queue->head) {
    thread = queue->head;
    queue->head = queue->head->next;
    if (queue->head==0)
      queue->tail=0;
  } else
    thread=0;
  return thread;
}
```

## Adding Mutual Exclusion

```
void enqueue (uthread_queue_t* queue, uthread_t* thread) {
  uthread_monitor_enter (&queue->monitor);
    thread->next = 0;
    if (queue->tail)
      queue->tail->next = thread;
    queue->tail = thread;
    if (queue->head==0)
      queue->head = queue->tail;
  uthread_monitor_exit (&queue->monitor);
}

uthread_t* dequeue (uthread_queue_t* queue) {
  uthread_t* thread;
  uthread_monitor_enter (&queue->monitor);
    if (queue->head) {
      thread = queue->head;
      queue->head = queue->head->next;
      if (queue->head==0)
        queue->tail=0;
    } else
      thread=0;
  uthread_monitor_exit (&queue->monitor);
  return thread;
}
```



## Now have dequeue wait for item if queue is empty

- classical producer-consumer model with each in different thread
  - e.g., producer enqueues video frames consumer thread dequeues them for display

```
void enqueue (uthread_queue_t* queue, uthread_t* thread) {
  uthread_monitor_enter (&queue->monitor);
    thread->next = 0;
    if (queue->tail)
      queue->tail->next = thread;
    queue->tail = thread;
    if (queue->head==0)
      queue->head = queue->tail;
    uthread_cv_notify (&queue->not_empty);
  uthread_monitor_exit (&queue->monitor);
}

uthread_t* dequeue (uthread_queue_t* queue) {
  uthread_t* thread;
  uthread_monitor_enter (&queue->monitor);
    while (queue->head==0)
      uthread_cv_wait (&queue->not_empty);
    thread = queue->head;
    queue->head = queue->head->next;
    if (queue->head==0)
      queue->tail=0;
  uthread_monitor_exit (&queue->monitor);
  return thread;
}
```



# Some Questions About Example

```
uthread_t* dequeue (uthread_queue_t* queue) {
  uthread_t* thread;
  uthread_monitor_enter (&queue->monitor);
    while (queue->head==0)
      uthread_cv_wait (&queue->not_empty);
    thread = queue->head;
    queue->head = queue->head->next;
    if (queue->head==0)
      queue->tail=0;
  uthread_monitor_exit (&queue->monitor);
  return thread;
}
```

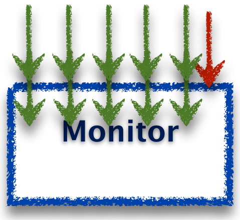
- Why is does dequeue have a while loop to check for non-empty?
- Why must condition variable be associated with specific monitor?
- Why can't we use condition variable outside of monitor?
  - this is called a *naked* use of the condition variable
  - this is actually required sometimes ... can you think where (BONUS)?
    - Experience with Processes and Monitors with Mesa, Lampson and Redell, 1980



# Implementing Condition Variables

- Some key observations
  - wait, notify and notify_all are called while monitor is held
  - the monitor must be held when they return
  - wait must release monitor before locking and re-acquire before returning
- Implementation
  - in the lab
  - look carefully at the implementations of monitor enter and exit
  - understand how these are similar to wait and notify
  - use this code as a guide
  - you also have the code for semaphores, which you might also find helpful

# Reader-Writer Monitors




- If we classify critical sections as
  - **reader** if only reads the shared data
  - **writer** if updates the shared data
- Then we can weaken the mutual exclusion constraint
  - writers require exclusive access to the monitor
  - but, a group of readers can access monitor concurrently
- Reader-Writer Monitors
  - monitor state is one of
    - **free**, **held-for-reading**, or **held**
  - monitor_enter ()
    - waits for monitor to be **free** then sets its state to **held**
  - monitor_enter_read_only ()
    - waits for monitor to be **free** or **held-for-reading**, then sets is state to **head-for-reading**
    - **increment reader count**
  - monitor_exit ()
    - if **held**, then set state to **free**
    - if **held-for-reading**, then **decrement reader count** and set state to **free if reader count is 0**



- Policy question
  - monitor state is head-for-reading
  - thread A calls monitor_enter() and blocks waiting for monitor to be free
  - thread B calls monitor_enter_read_only(); what do we do?
- Disallowing new readers while writer is waiting
  - is the fair thing to do
  - thread A has been waiting longer than B, shouldn't it get the monitor first?
- Allowing new readers while writer is waiting
  - may lead to faster programs by increasing concurrency
  - if readers must WAIT for old readers and writer to finish, less work is done
- What should we do
  - normally either provide a fair implementation
  - or allow programmer to choose (that's what Java does)



# Semaphores

- Introduced by Edsger Dijkstra for the THE System circa 1968
  - recall that he also introduced the "process" (aka "thread") for this system
  - was fearful of asynchrony, Semaphores synchronize interrupts
  - synchronization primitive provide by UNIX to applications
- A Semaphore is
  - an atomic counter that can never be less than 0
  - attempting to make counter negative blocks calling thread
- P (s)
  - try to decrement s (*prolaag* for *probeer te varlagen* in Dutch)
  - atomically blocks until s >0 then decrement s
- V (s)
  - increment s (*verhogen* in Dutch)
  - atomically increase s unblocking threads waiting in P as appropriate



# Using Semaphores to Drink Beer

- Explicit locking not required when using semaphores since atomicity built in
- Use semaphore to store glasses head by pitcher
  - set initial value of empty when creating it

```
uthread_semaphore_t* glasses = uthread_create_semaphore (0);
```

- Pouring and refilling don't require a monitor

```
void pour () {
  uthread_P (glasses);
}
```

```
void refill (int n) {
  for (int i=0; i<n; i++)
    uthread_V (glasses);
}
```

- Getting the beer warm, however, doesn't fit quite as nicely
  - need to keep track of the number of threads waiting for the warm beer
  - then call V that number of times
  - this is actually quite tricky

# Other ways to use Semaphores

- Asynchronous Operations
  - create ***outstanding_request*** semaphore
  - async_read: P (outstanding_request)
  - completion interrupt: V (outstanding_request)
- Rendezvous
  - two threads wait for each other before continuing
  - create a semaphore for each thread initialized to 0

```
void thread_a () {
  uthread_V (a);
  uthread_P (b);
}

void thread_b () {
  uthread_V (b);
  uthread_P (a);
}
```

What if you reversed order of V and P?



- Barrier (local)
  - In a system of 1 parent thread and N children threads
  - All threads must arrive at barrier before any can continue

```
void* add (void* arg) {
  struct arg_tuple* tuple = (struct arg_tuple*) arg;
  tuple->result = tuple->arg0 + tuple->arg1;
  uthread_V (tuple->barrier);
  return 0;
}
```

```
uthread_semaphore_t* barrier = uthread_semaphore_create (0);
struct arg_tuple a0 = {1,2,0,barrier};
struct arg_tuple a1 = {3,4,0,barrier};
uthread_init (1);
uthread_create (add, &a0);
uthread_create (add, &a1);
uthread_P (barrier);
uthread_P (barrier);
printf ("%d %d\n", a0.result, a1.result);
```

- Barrier (global)
  - In a system of N threads with no parent
  - All threads must arrive, before any can continue ... and should work repeatedly



- Implementing Monitors
  - initial value of semaphore is 1
  - lock is P()
  - unlock is V()
- Implementing Condition Variables
  - this is the warm beer problem
  - it took until 2003 before we actually got this right
  - for further reading
    - Andrew D. Birrell. “Implementing Condition Variables with Semaphores”, 2003.
    - Google “semaphores condition variables birrell”



# Using Semaphores

- good building block for implementing many other things
  - monitors
    - initial value of semaphore is 1
    - lock is P()
    - unlock is V()
  - condition variables (almost)
    - this is the warm beer problem
    - it took until 2003 before we actually got this right
    - for further reading
      - Andrew D. Birrell. “Implementing Condition Variables with Semaphores”, 2003.
      - Google “semaphores condition variables birrell”
  - rendezvous: two threads wait for each other before continuing
  - barriers: all threads must arrive at barrier before any can continue

# Synchronization in Java *(5)*

- Monitors using the Lock interface
  - a few variants allow interruptibility, just trying lock, ...

```
Lock l = ...;
l.lock ();
try {
  ...
} finally {
  l.unlock ();
}
```

```
Lock l = ...;
try {
  l.lockInterruptibly ();
  try {
    ...
  } finally {
    l.unlock ();
  }
} catch (InterruptedException ie) {}
```

  - multiple-reader single writer locks

```
ReadWriteLock l  = ...;
Lock         rl = l.readLock ();
Lock         wl = l.writeLock ();
```



- Condition variables
  - **await** is wait (replaces Object wait)
  - **signal** or **signalAll** is "notify" (replaces Object notify, notifyAll)

```
class Beer {
  Lock      l       = ...;
  Condition notEmpty = l.newCondition ();
  int       glasses  = 0;

  void pour () throws InterruptedException {
    l.lock ();
    try {
      while (glasses==0)
        notEmpty.await ();
      glasses--;
    } finaly {
      l.unlock ();
    }
  }

  void refill (int n) throws InterruptedException {
    l.lock ();
    try {
      glasses += n;
      notEmpty.signalAll ();
    } finaly {
      l.unlock ();
    }}}
```



- Semaphore class
  - **acquire ()** or **acquire (n)** is P() or P(n)
  - **release ()** or **release (n)** is V() or V(n)

```
class Beer {
  Semaphore glasses = new Semaphore (0);

  void pour () throws InterruptedException {
    glasses.acquire ();
  }

  void refill (int n) throws InterruptedException {
    glasses.release (n);
  }
}
```

- Lock-free Atomic Variables
  - AtomicX where X in {Boolean, Integer, IntegerArray, Reference, ...}
  - atomic operations such as getAndAdd(), compareAndSet(), ...
    - e.g., x.compareAndSet (y,z) atomically sets x=z iff x==y and returns true iff set occurred



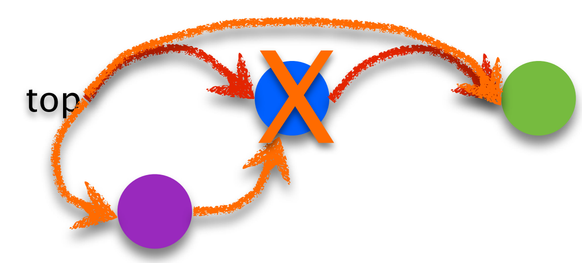
# Lock-Free Atomic Stack in Java

- Recall the problem with concurrent stack

```
void push_st (struct SE* e) {
  e->next = top;
  top     = e;
}
```

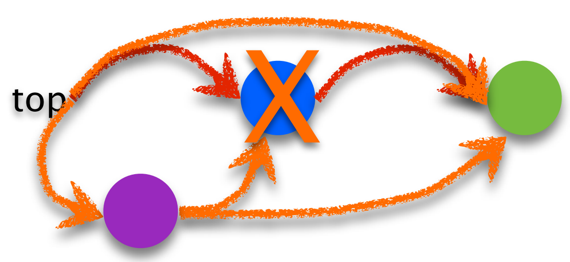
```
struct SE* pop_st () {
  struct SE* e = top;
  top = (top)? top->next: 0;
  return e;
}
```

  - a pop could intervene between two steps of push, corrupting linked list




  - we solved this problem using locks to ensure mutual exclusion
  - now ... solve without locks, using **atomic compare-and-set** of top

```
class Element {
  Element* next;
}

class Stack {
  AtomcReference<Element> top;
  Stack () {
    top.set (NULL);
  }

  void push () {
    Element t;
    Element e = new Element ();
    do {
      t = top.get ();
      e.next = t;
    } while (!top.compareAndSet (t, e));
  }
}
```






# Problems with Concurrency

- Race Condition
  - competing, unsynchronized access to shared variable
    - from multiple threads
    - at least one of the threads is attempting to update the variable
  - solved with synchronization
    - guaranteeing mutual exclusion for competing accesses
    - **but the language does not help you see what data might be shared --- can be very hard**
- Deadlock
  - multiple competing actions wait for each other preventing any to complete
  - what can cause deadlock?
    - MONITORS
    - CONDITION VARIABLES
    - SEMAPHORES



# Recursive Monitor Entry

- What should we do for a program like this

```
void foo () {
  uthread_monitor_enter (mon);
    count--;
    if (count>0)
      foo();
  uthread_monitor_exit (mon);
}
```

- Here is implementation of lock, is this okay?

```
void lock (struct blocking_lock* l) {
  spinlock_lock (&l->spinlock);
    while (l->held) {
      enqueue         (&waiter_queue, uthread_self ());
      spinlock_unlock (&l->spinlock);
      uthread_switch  (ready_queue_dequeue (), TS_BLOCKED);
      spinlock_lock   (&l->spinlock);
    }
    l->held = 1;
  spinlock_unlock (&l->spinlock);
}
```



- if we try to lock the monitor again it is a *deadlock*
  - the thread will hold the monitor when it tries to enter
  - the thread will wait for itself, and thus never wake up
- allow a thread that holds the monitor to enter again

```
void uthread_monitor_enter (uthread_monitor_t* monitor) {
  spinlock_lock (&monitor->spinlock);
  while (monitor->holder && monitor->holder!=uthread_self()) {
    enqueue         (&monitor->waiter_queue, uthread_self ());
    spinlock_unlock (&monitor->spinlock);
    uthread_stop    (TS_BLOCKED);
    spinlock_lock   (&monitor->spinlock);
  }
  monitor->holder = uthread_self ();
  spinlock_unlock (&monitor->spinlock);
}
```

# Systems with multiple monitors

- We have already seen this with semaphores
- Consider a system with two monitors, a and b

```
void foo() {
  uthread_monitor_enter (a);
  uthread_monitor_exit   (a);
}
```

```
void bar() {
  uthread_monitor_enter (b);
  uthread_monitor_exit  (b);
}
```

```
void x() {
  uthread_monitor_enter (a);
    bar();
  uthread_monitor_exit  (a);
}
```

```
void y() {
  uthread_monitor_enter (b);
    foo();
  uthread_monitor_exit  (b);
}
```

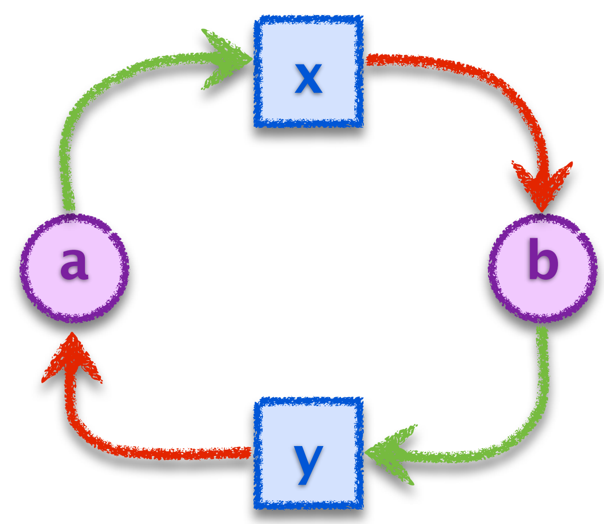
Any problems so far?

What about now?



# Waiter Graph Can Show Deadlocks

- Waiter graph
  - edge from **lock** to thread if thread **HOLD**s lock
  - edge from **thread** to lock if thread **WANT**s lock
  - a cycle indicates deadlock




```
void foo() {
  uthread_monitor_enter (a);
  uthread_monitor_exit   (a);
}
```

```
void bar() {
  uthread_monitor_enter (b);
  uthread_monitor_exit  (b);
}
```

```
void x() {
  uthread_monitor_enter (a);
    bar();
  uthread_monitor_exit  (a);
}
```

```
void y() {
  uthread_monitor_enter (b);
    foo();
  uthread_monitor_exit  (b);
}
```



# The Dining Philosophers Problem

- Formulated by Edsger Dijkstra to explain deadlock (circa 1965)
  - 5 computers competed for access to 5 shared tape drives
- Re-told by Tony Hoare
  - 5 philosophers sit at a round table with fork placed in between each
    - fork to left and right of each philosopher and each can use only these 2 forks
  - they are either eating or thinking
    - while eating they are not thinking and while thinking they are not eating
    - they never speak to each other
  - large bowl of spaghetti at centre of table requires 2 forks to serve
    - dig in ...
  - deadlock
    - every philosopher holds fork to left waiting for fork to right (or vice versa)
    - how might you solve this problem?
  - starvation (aka *livelock*)
    - philosophers still starve (ever get both forks) due to timing problem, but avoid deadlock
    - for example:



# Avoiding Deadlock

- Don't use multiple threads
  - you'll have many idle CPU cores and write asynchronous code
- Don't use shared variables
  - if threads don't access shared data, no need for synchronization
- Use only one lock at a time
  - deadlock is not possible, unless thread forgets to unlock
- Organize locks into precedence hierarchy
  - each lock is assigned a unique precedence number
  - before thread *X* acquires a lock *i*, it must hold all higher precedence locks
  - ensures that any thread holding *i* can not be waiting for *X*
- Detect and destroy
  - if you can't avoid deadlock, detect when it has occurred
  - break deadlock by terminating threads (e.g., sending them an exception)

# Deadlock and Starvation

- Solved problem: race conditions
  - solved by synchronization abstractions: locks, monitors, semaphores
- Unsolved problems when using multiple locks
  - deadlock: nothing completes because multiple competing actions wait for each other
  - starvation: some actions never complete
  - no abstraction to simply solve problem, major concern intrinsic to synchronization
  - some ways to handle/avoid:
    - precedence hierarchy of locks
    - detect and destroy: notice deadlock and terminate threads



# Synchronization Summary

- Spinlock
  - one acquirer at a time, busy-wait until acquired
  - need atomic read-write memory operation, implemented in hardware
  - use for locks held for short periods (or when minimal lock contention)
- Monitors and Condition Variables
  - blocking locks, stop thread while it is waiting
  - monitor guarantees mutual exclusion
  - condition variables wait/notify provides control transfer among threads
- Semaphores
  - blocking atomic counter, stop thread if counter would go negative
  - introduced to coordinate asynchronous resource use
  - use to implement barriers or monitors
  - use to implement something like condition variables, but not quite
- Problems, problems, problems
  - race conditions to be avoided using synchronization
  - deadlock/livelock to be avoided using synchronization carefully