CPSC 213

Introduction to Computer Systems

Summer Session 2019, Term 2

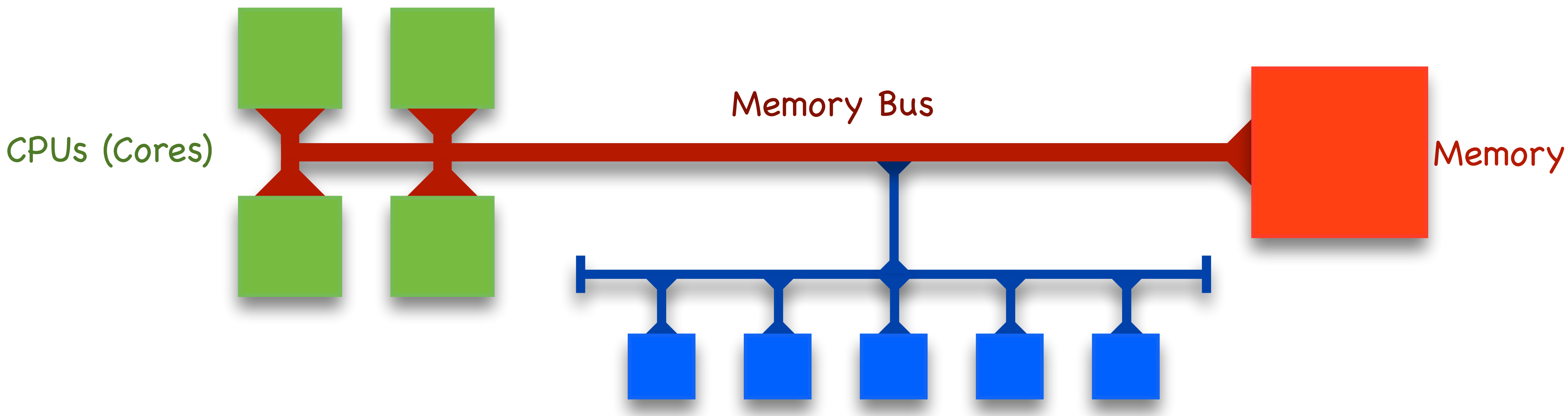
Unit 2c – Aug 1, 6, 8

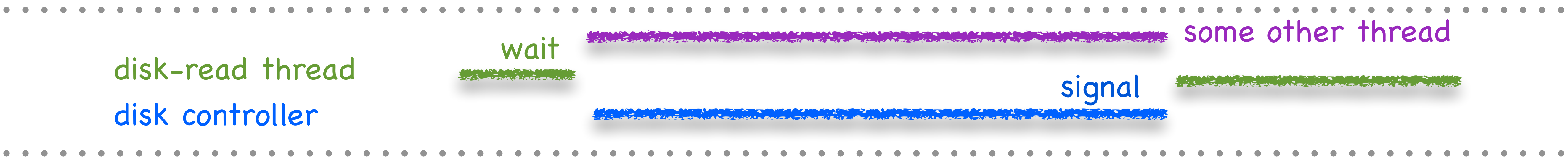
Synchronization

Overview

- Reading
  - text: 12.4-12.6, parts of 12.7
- Learning Goals
  - explain the relationship between concurrency, shared data, critical sections and mutual exclusion
  - use locks to guarantee mutual exclusion in C programs
  - identify race conditions in code
  - explain how to implement a *correct* and *efficient* spinlock
  - describe the difference between spinlocks (busy waiting) and blocking locks (blocking waiting) and identify conditions where each is favoured over the other
  - describe how blocking locks are implemented and how they use spinlocks
  - explain the difference between condition variables and monitors
  - describe why conditions are useful by giving an example of a situation where one would be used
  - use monitors and condition variables for synchronization in C programs
  - explain why it is necessary to associate a condition variable with a specific monitor and to require that the monitor be held before calling wait
  - explain how condition variables are implemented
  - describe why reader-writer monitors are useful and explain the constraints involved in their use
  - explain the difference between semaphores and monitors/condition variables
  - use semaphores for synchronization in C programs
  - explain how semaphores are implemented
  - describe what a deadlock is, how it can be caused, why it is bad, and how it can be avoided
  - give an example of the use of lock-free synchronization for updating a concurrent data structure and explain the benefit of this approach compared to using locks

Synchronization





- We invented Threads to
  - **express 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 related to controlling operation order
  - coordinating access to memory (variables) shared among 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 signal of the occurrence of events

Communicating Through Shared Data

- We have a problem if
  - threads shared a data structure
  - operations involve multiple memory accesses
  - these accesses can be arbitrarily interleaved
- For Example

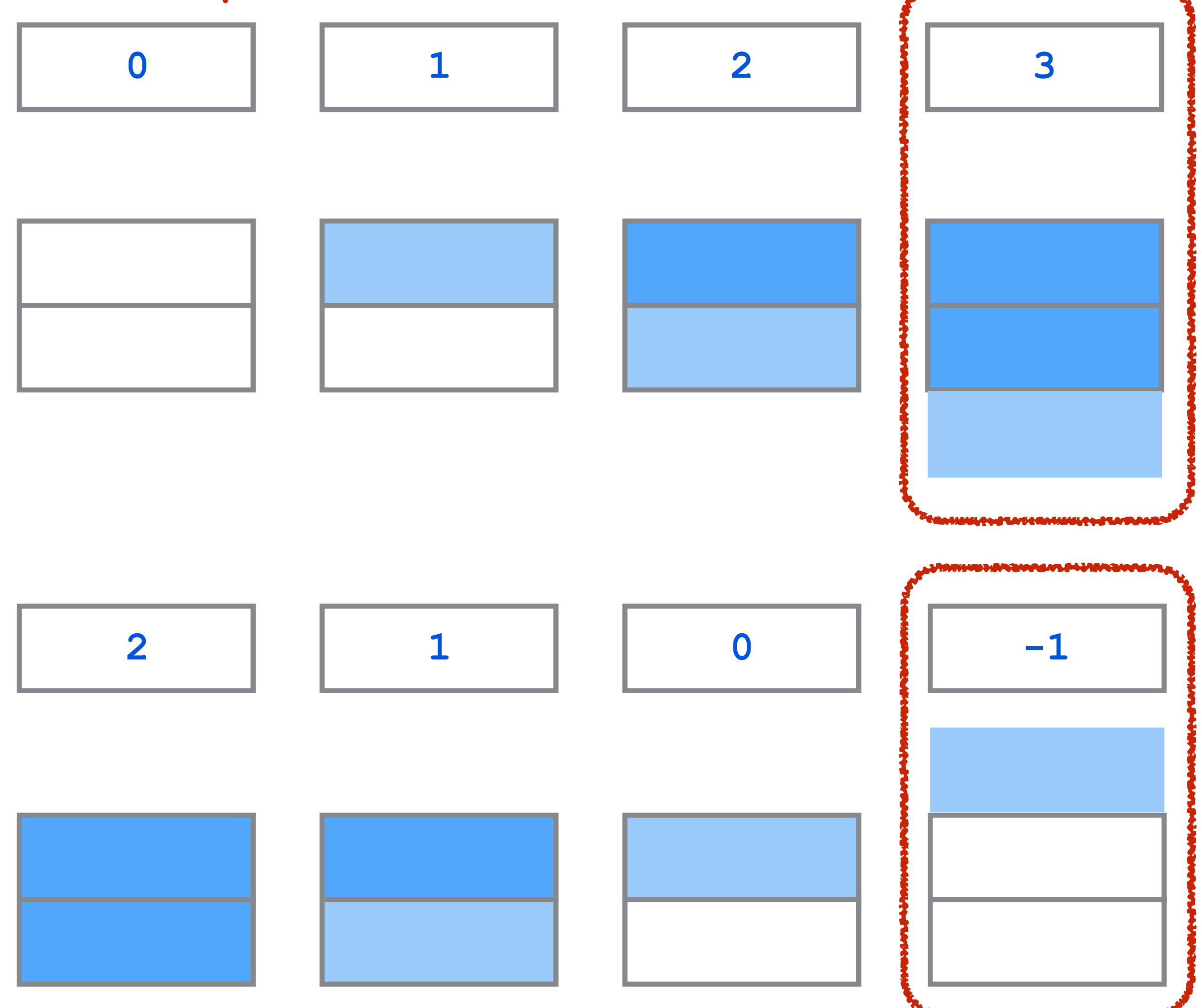
Make these sections of code **atomic**

```
int n;
int array [2];

void push (i) {
  if (n < 2) {
    array [n] = i;
    n++;
  }
}

int pop () {
  if (n > 0) {
    n--;
    return array [n];
  } else
    return -1;
}
```

Concurrent thread does same operation here



How could these bad cases happen?

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 of a shared data item are executed by one thread at a time
  - 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 (malloc (...));
}
```

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

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

- 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, 0);
uthread_join (dt, 0);
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;
}
```

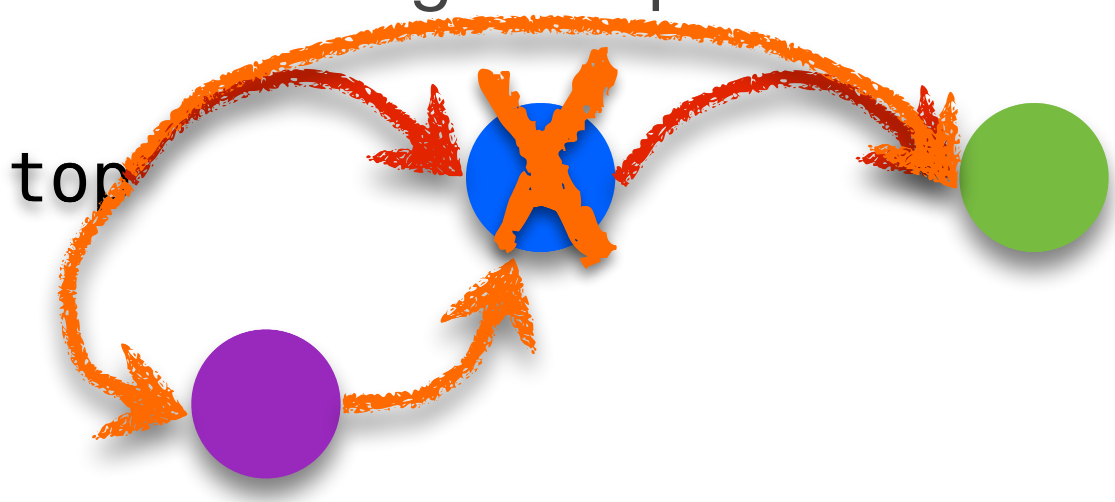
Notice:

it does work if we say **uthread_init (1)**,

but it does not work with **uthread_init (2)** or higher...

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;
}
```

  - does this work?

- We now have a race in the lock code

Thread A

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

Thread B

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

```
1. read *lock==0, exit loop
```

```
2. read *lock==0, exit loop
```

```
3. *lock = 1
4. return with lock held
```

```
5. *lock = 1, return
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)
```

lock appears free

Another thread reads lock

acquire lock

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
  - no intervening access to that location from any other thread
- 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]]` `m[r[a]] ← r[v]`	`xchg (ra), rv`

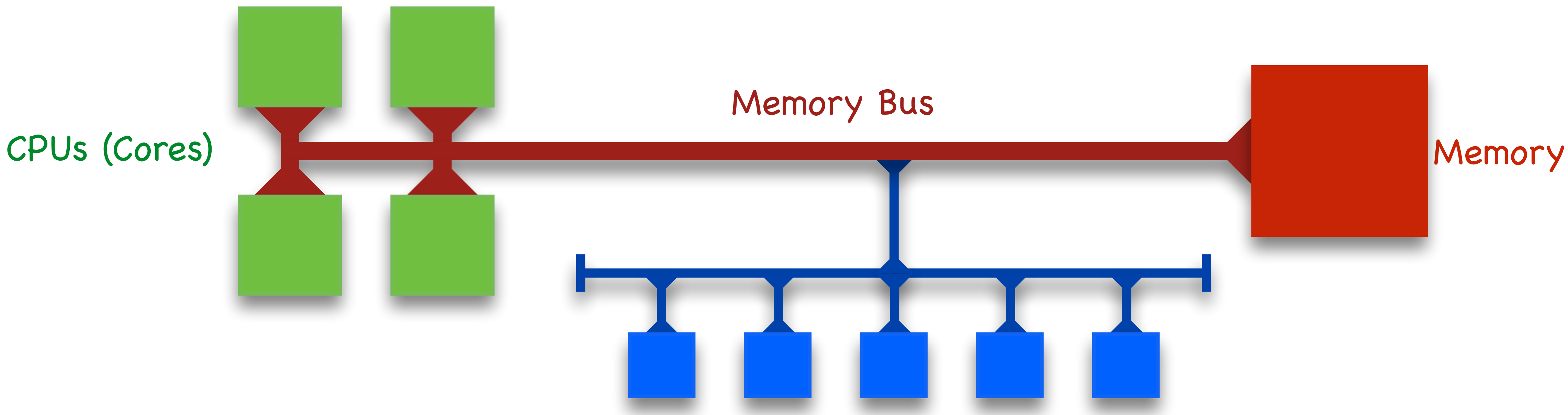
Spinlock

- A Spinlock is
  - a lock where waiter *spins,* looping on memory reads until lock is acquired
  - also called a *busy-waiting* lock
- 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 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

Spin first on normal read

- normal reads are very fast and efficient compared to 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    # r1 = &lock
loop: ld    (r1), r0     # r0 = lock
      beq   r0, try      # goto try if lock==0 (available)
      br    loop         # goto loop if lock!=0 (held)
try:  ld    $1, r0       # r0 = 1
      xchg  (r1), r0     # atomically swap r0 and lock
      beq   r0, held     # goto held lock was 0 before swap
      br    loop         # try again if another thread holds lock
held:                    # we now hold the lock
```

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 is unlocked or event is signalled
- Blocking locks for mutual exclusion
  - attempting to acquire a held lock BLOCKs calling thread
    - blocked thread's TCB is stored on lock's *waiting queue*
  - when releasing lock, UNBLOCK a waiting thread if there is one
    - remove thread block's *waiting queue* and place it on *ready queue*
- Blocking for event notification
  - wait by blocking, placing TCB on a waiting queue
  - signal a specific waiting queue by moving a thread to ready queue

Blocking vs Busy Waiting

Spinlocks

- *busy waiting*

- Pros and Cons
  - un-contended locking has low overhead
  - waiting for lock has high overhead
- 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

- *blocking waiting*

- Pros and Cons
  - un-contended locking has higher overhead
  - waiting for lock has small, fixed overhead
- Use when
  - lock may be held for some time
  - when contention is high
  - when event wait may be long

Video Playback System Example

- General problem
  - video playback has two parts: (1) fetch/decode and (2) playback
  - fetch has variable latency and so we need a buffer
    - sometimes you can fetch faster than playback rate
    - but, sometimes there are long delays
    - buffer hides the delays by fetching ahead of playback position when possible
- Bounded Buffer and Two Independent Threads
  - finite buffer of the next few video frames to play
  - maximum size is N
  - goal: keep buffer at least 50% full (lets say)
- Producer Thread
  - fetch frame from network and put them in buffer
- Consumer Thread
  - fetch frames from buffer, decode them and send them to video driver
- How are Producer and Consumer connect?
  - advantage of this approach is that they are largely decoupled; each has a separate job
  - but, it's the consumer that decides when the producer should run … HOW?

Monitors and Condition Variables

- Mutual exclusion plus inter-thread synchronization
  - introduced by Tony Hoare and Per Brinch Hansen circ. 1974
  - basis for synchronization primitives in Unix, Java etc.
- Monitor / **Mutex**
  - blocking lock to guarantee mutual exclusion
  - monitor operations were enter and exit
  - typically called a **mutex** (or just a lock) with operations **lock** and **unlock**
- Condition Variable
  - allows threads to synchronize with each other
  - **wait** blocks until a subsequent signal operation on the variable
  - **signal** unblocks waiter
  - **broadcast** unblocks all waiters
  - can only be accessed from inside of a monitor (i.e, with mutex held)

UThreads Mutex and Condition

```
struct uthread_mutex;
typedef struct uthread_mutex* uthread_mutex_t;
struct uthread_cond;
typedef struct uthread_cond*  uthread_cond_t;

uthread_mutex_t uthread_mutex_create        ();
void            uthread_mutex_lock          (uthread_mutex_t);
void            uthread_mutex_lock_readonly (uthread_mutex_t);
void            uthread_mutex_unlock        (uthread_mutex_t);
void            uthread_mutex_destroy       (uthread_mutex_t);

uthread_cond_t  uthread_cond_create         (uthread_mutex_t);
void            uthread_cond_wait           (uthread_cond_t);
void            uthread_cond_signal         (uthread_cond_t);
void            uthread_cond_broadcast      (uthread_cond_t);
void            uthread_cond_destroy        (uthread_cond_t);
```

Video Playback

```
struct video_frame;
#define N 100
struct video_frame buf [N];
int buf_length = 0;
int buf_pcur   = 0;
int buf_ccur   = 0;

uthread_mutex_t mx;
uthread_cond_t  need_frames;
```

```
void producer() {
  uthread_lock (mx);
  while (1) {
    while (buf_length < N) {
      buf [pcur] =  get_next_frame();
      buf_pcur   =  (pcur + 1) % N;
      buf_length += 1;
    }
    uthread_cond_wait (need_frames);
  }
  uthread_unlock (mx);
}
```

```
void consumer() {
  uthread_lock (mx);
  while (1) {
    assert (buf_length > 0);
    show_frame (buf [buf_ccur]);
    buf_ccur   =  (buf_ccur + 1) % N;
    buf_length -= 1;
    if (buf_length == N/2)
        uthread_cond_signal (need_frames);
  }
  uthread_unlock (mx);
}
```

Using Conditions

- Basic formulation
  - one thread acquires mutex and may wait for a condition to be established

```
uthread_mutex_lock (aMutex);
  while (!aDesiredState)
    uthread_cond_wait (aCond);
  aDesiredState = 0;
uthread_mutex_unlock (aMutex);
```

  - another thread acquires mutex, establishes condition and signals waiter, if there is one

```
uthread_mutex_lock (aMutex);
  aDesiredState = 1;
  uthread_cond_signal (aCond);
uthread_mutex_unlock (aMutex);
```

- **wait** releases the mutex and blocks thread
  - before waiter blocks, it releases mutex to allow other threads to acquire it
  - when wait unblocks, it re-acquires mutex, waiting/blocking to enter if necessary
  - note: other threads may have acquired mutex between wait call and return

- **signal** awakens at most one thread
  - waiter does not run until signaller releases the mutex explicitly
  - a third thread could intervene and acquire mutex before waiter
  - waiter must thus re-check wait condition
  - if no threads are waiting, then calling signal has no effect

Recheck Condition After Wakeup

```
lock (aMutex);
  while (!aDesiredState)
    wait (aCond);
  aDesiredState = 0;
unlock (aMutex);
```

Don't Assume Condition is Still True

```
lock (aMutex);
  if (!aDesiredState)
    wait (aCond);
  aDesiredState = 0;
unlock (aMutex);
```

- **broadcast** awakens all threads
  - may wakeup too many
  - okay since threads re-check wait condition and re-wait if necessary

```
lock (aMutex);
  aDesiredCondition += n;
  broadcast (aCond);
unlock (aMutex);
```

```
lock (aMutex);
  while (!aDesiredCondition)
    wait (aCond);
  aDesiredState --;
unlock (aMutex);
```

Video Playback (Pause on Empty)

```
struct video_frame;
#define N 100
struct video_frame buf [N];
int buf_length = 0;
int buf_pcur   = 0;
int buf_ccur   = 0;

uthread_mutex_t mx;
uthread_cond_t  need_frames;
uthread_cont_t  have_frame;
```

```
void producer() {
  uthread_lock (mx);
  while (1) {
    while (buf_length < N) {
      buf [pcur] =  get_next_frame();
      buf_pcur   =  (pcur + 1) % N;
      buf_length += 1;
      uthread_cond_signal (have_frame);
    }
    uthread_cond_wait (need_frames);
  }
  uthread_unlock (mx);
}
```

```
void consumer() {
  uthread_lock (mx);
  while (1) {
    while (buf_length == 0)
      uthread_wait (have_frame);
    show_frame (buf [buf_ccur]);
    buf_ccur   =  (buf_ccur + 1) % N;
    buf_length -= 1;
    if (buf_length < N/2)
      uthread_cond_signal (need_frames);
  }
  uthread_unlock (mx);
}
```

Why WHILE?

What if there are two concurrent consumers?

Video Playback - Full Version

```
struct video_frame;
#define N 100
struct video_frame buf [N];
int buf_length = 0;
int buf_pcur   = 0;
int buf_ccur   = 0;

uthread_mutex_t mx;
uthread_cond_t  need_frames;
uthread_cont_t  have_frame;
uthread_cont_t  show_next_frame;
```

```
void producer() {
  uthread_lock (mx);
  while (1) {
    while (buf_length < N) {
      buf [pcur] =  get_next_frame();
      buf_pcur   =  (pcur + 1) % N;
      buf_length += 1;
      uthread_cond_signal (have_frame);
    }
    uthread_cond_wait (need_frames);
  }
  uthread_unlock (mx);
}
```

One More Thing:
show_next_frame will be signalled every a new frame is required for the video driver; e.g., every 1/30 s.

```
void consumer() {
  uthread_lock (mx);
  while (1) {
    uthread_cond_wait (show_next_frame);
    while (buf_length==0)
      uthread_cond_wait (have_frame);
    show_frame (buf [buf_ccur]);
    buf_ccur   =  (buf_ccur + 1) % N;
    buf_length -= 1;
    if (buf_length < N/2)
      uthread_cond_signal (need_frames);
  }
  uthread_unlock (mx);
}
```

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
- Data Structure for Beer Pitcher
  - **glasses** will count the number of classes of beer left
  - **mx** is the mutex
  - **hasBeer** is condition indicating that there’s a least one glass of beer

Implementing Beer Drinking

- Static Declaration

```
struct BeerPitcher {
  int              glasses;
  uthread_mutex_t mx;
  uthread_cond_t  hasBeer;
};
```

- Create and initialize Instance

```
void foo() {
  struct BeerPitcher* p = malloc (sizeof (struct BeerPitcher));
  p->glasses = 0;
  p->mx      = uthread_mutex_create();
  p->hasBeer = uthread_cond_create (p->mx);
  ...
}
```

Pouring a Glass

```
void pour (struct BeerPitcher* p) {
  uthread_mutex_lock (p->mx);
    while (p->glasses == 0)
      uthread_cond_wait (p->hasBeer);
    glasses --;
  uthread_mutex_unlock (p->mx);
}
```

Refilling the Pitcher

```
void refill (struct BeerPitcher* p, int n) {
  uthread_mutex_lock (p->mx);
    p->glasses += n;
    for (int i=0; i<n; i++)
      uthread_cond_signal (p->hasBeer);
  uthread_mutex_unlock (p->mx);
}
```

If beer is very popular you might want this

```
void refill (struct BeerPitcher* p, int n) {
  uthread_mutex_lock (p->mx);
    p->glasses += n;
    uthread_cond_broadcast (p->hasBeer);
  uthread_mutex_unlock (p->mx);
}
```

If refill should wake up most of waiters, then this

Review Question

- We do this

```
void pour (…) {
  lock (p->mx);
    while (p->glasses == 0)
      wait (p->hasBeer);
    glasses --;
  unlock (p->mx);
}
```

```
void refill (…) {
  lock (p->mx);
    p->glasses += n;
    for (int i=0; i<n; i++)
      signal (p->hasBeer);
  unlock (p->mx);
}
```

- Why not this

```
void pour (…) {
  lock (p->mx);
    if (p->glasses == 0)
      wait (p->hasBeer);
    glasses --;
  unlock (p->mx);
}
```

Signal and Monitor Race

```
void pour (…) {
  lock (p->mx);
    while (p->glasses == 0)
      wait (p->hasBeer);
    glasses --;
  unlock (p->mx);
}
```

```
void refill (…) {
  lock (p->mx);
    p->glasses += n;
    for (int i=0; i<n; i++)
      signal (p->hasBeer);
  unlock (p->mx);
}
```

Thread A **Thread B** **Thread C**

1. pour acquires mutex
2. sees glasses==0
3. waits, releasing mutex

4. refill acquires mutex
5. sets glasses = 1
6. signals condition
7. releases mutex

8a. tries to acquire mutex ← race to get mutex → 8c pour acquires mutex
9a. fails, waits on mutex
9. sets glasses = 0
10. releases mutex

11. acquires mutex
12. sees glasses==0 again
13. waits, releasing mutex

Extending the Example

- What if you want to refill automatically?
  - a pitcher has capacity *maxGlasses* and current volume *glasses*
  - pouring removes one glass if there is enough beer and waits otherwise
  - refilling loops forever waiting for pitcher to be empty, when it is, it refills the pitcher to its capacity awakening any pourers

Extended Example Solution (1)

```
struct BeerPitcher {
  int                maxGlasses;
  int                glasses;
  uthread_mutex_t mx;
  uthread_cond_t  hasBeer;
  uthread_cont_t  isEmpty;
};
```

```
void foo (int n) {
  struct BeerPitcher* p = malloc (sizeof (struct BeerPitcher));
  p->maxGlasses = n;
  p->glasses    = 0;
  p->mx         = uthread_mutex_create();
  p->hasBeer    = uthread_cond_create (p->mx);
  p->isEmpty    = uthread_cond_create (p->mx);
  ...
}
```

Extended Example Solution (2)

```
void pour (struct BeerPitcher* p) {
  uthread_mutex_lock (p->mx);
    while (p->glasses == 0)
      uthread_cond_wait (p->hasBeer);
    p->glasses --;
    if (p->glasses == 0)
      uthread_cond_signal (p->isEmpty);
  uthread_mutex_unlock (p->mx);
}
```

```
void refill (struct BeerPitcher* p) {
  uthread_mutex_lock (p->mx);
    while (1) {
      while (p->glasses > 0)
        uthread_cond_wait (p->isEmpty);
      p->glasses += p->maxGlasses;
      for (int i=0; i<p->maxGlasses; i++)
        uthread_cond_signal (p->hasBeer);
    }
  uthread_mutex_unlock (p->mx);
}
```

Could we use
IF instead of **WHILE**?

Event Ordering Exercise

- Lets say we have two threads running concurrently
  - t0 calls procedure a
  - t1 calls procedure b
- We need to ensure that b is not called until a returns
  - how?

Using Condition Variables for Disk Read

- Blocking read
  - schedule read as before
  - but now block on condition variable

```
void read (char* buf, int nbytes, int blockno) {
  uthread_mutex_lock (mx);
    scheduleRead (buf, nbytes, blockno);
    uthread_cond_wait (readComplete);
  uthread_mutex_unlock (mx);
}
```

- Read completion
  - called by disk ISR as before
  - but now restarted blocked reader thread by signalling condition variable

```
void readComplete() {
  uthread_mutex_lock (mx);
    uthread_cond_signal (readComplete);
  uthread_mutex_unlock (mx);
}
```

Why must mutex be held when calling Wait?

- We do this

```
void read (char* buf, int nbytes, int blockno) {
  uthread_mutex_lock (mx);
    scheduleRead (buf, nbytes, blockno);
    uthread_cond_wait (readComplete);
  uthread_mutex_unlock (mx);
}
```

- Why not this

```
void read (char* buf, int nbytes, int blockno) {
  scheduleRead (buf, nbytes, blockno);
  uthread_cond_wait (readComplete);
}
```

- Or even this

```
void read (char* buf, int nbytes, int blockno) {
  scheduleRead (buf, nbutes, blockno);
  uthread_mutex_lock (mx);
    uthread_cond_wait (readComplete);
  uthread_mutex_unlock (mx);
}
```

Wait-Signal Race

The Problem

```
void read (char* buf, int nbytes, int blockno) {
  scheduleRead (buf, nbytes, blockno);
  uthread_cond_wait (readComplete);
}
```

- wait condition check / trigger and wait are not atomic
- signal could occur before wait
- waiter could thus miss signal

Reader	ISR
1. scheduleRead	
	2. readComplete
	3. signal cond
4. wait	

The Solution

```
void read (char* buf, int nbytes, int blockno) {
  uthread_mutex_lock (mx);
    scheduleRead (buf, nbytes, blockno);
    uthread_cond_wait (readComplete);
  uthread_mutex_unlock (mx);
}
```

- ensure that condition check /trigger and wait are atomic
- so that wait is ordered before signal

Reader	ISR
1. acquire mutex	
2. scheduleRead	
	3. readComplete
4. wait, releasing mutex	
	5. acquire mutex
	6. signal cond
	7. release mutex
8. wakeup, acquiring mutex	

Why Must Signal Be Inside Monitor?

- We do this

```
void readComplete() {
  uthread_mutex_lock (mon);
    uthread_cond_signal (cv);
  uthread_mutex_unlock (mon);
}
```

- Why not this

```
void readComplete() {
  uthread_cond_signal (cv);
}
```

Wait-Signal Race ... Again

Preventing the Race

- requires making waiter code atomic
- using monitor lock
- but, its not atomic if signal isn't inside monitor

Reader

1. acquire mutex
2. scheduleRead

5. wait, releasing mutex

ISR

3. readComplete
4. signal condition

Naked Notify

- that's what we call a signal outside of a monitor
- its sometimes necessary
  - when signal is called in a context where blocking is not allowed

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;
  thread->next = 0;
  return thread;
}
```

Adding Mutual Exclusion

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

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

Change dequeue to wait if queue is empty

- assuming that *producer* is running in another thread
  - e.g., producer enqueues video frames consumer thread dequeues them for display

```
void enqueue (uthread_queue_t* queue, uthread_t thread) {
  uthread_mutex_lock (&queue->mx);
    thread->next = 0;
    if (queue->tail)
      queue->tail->next = thread;
    queue->tail = thread;
    if (queue->head==0)
      queue->head = queue->tail;
    uthread_cond_signal (&queue->not_empty);
  uthread_mutex_unlock (&queue->mx);
}

uthread_t* dequeue (uthread_queue_t* queue) {
  uthread_t thread;
  uthread_mutex_lock (&queue->mx);
    while (queue->head==0)
      uthread_cond_wait (&queue->not_empty);
    thread = queue->head;
    queue->head = queue->head->next;
    if (queue->head==0)
      queue->tail=0;
    thread->next = 0;
  uthread_mutex_unlock (&queue->mx);
  return thread;
}
```

You have to Signal every time

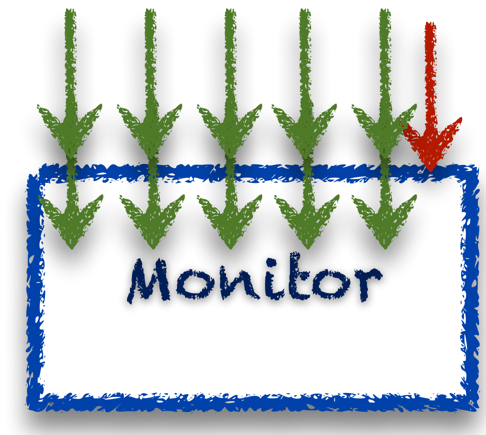
- This code seems like it would be right

```
void enqueue (uthread_queue_t* queue, uthread_t thread) {
  uthread_mutex_lock (&queue->mx);
    …
    if (queue->head == 0)
      uthread_cond_signal (&queue->not_empty);
  uthread_mutex_unlock (&queue->mx);
}
```

  - Just signal when adding to an empty queue

- But it is wrong
  - lets say there are N threads waiting in dequeue on `queue->not_empty`
  - if there are N enqueues, there MUST be N signals to wakeup the all dequeues
  - if you get two enqueues in a row before a dequeue runs, however
    - the second enqueue does not see the queue empty (i.e., `queue->head != 0`)
    - and so it does not signal `queue->not_empty` in this version of the code
  - you thus get N enqueues but fewer than N signals
    - some threads will still be waiting in dequeue, even though the queue isn't empty … a bug

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-for-writing**
  - mutex_lock ()
    - waits for monitor to be **free** then sets its state to **held-for-writing**
  - mutex_lock_read_only ()
    - waits for monitor to be **free** or **held-for-reading**, then sets is state to **held-for-reading**
    - **increment reader count**
  - mutex_unlock ()
    - if **held-for-writing**, 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
  - if monitor state is *held-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?
  - how does this effect concurrency and *throughput*?
- 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 reasonably fair implementation that is also efficient – a tradeoff
  - 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
- A Semaphore is
  - an atomic counter that can never be less than 0
  - attempting to make counter negative blocks calling thread
- P (s) – wait (s)
  - try to reduce s (*probeer te verlagen* in Dutch)
  - atomically blocks until s>0 then decrements s
- V (s) – signal (s)
  - increase s (*verhogen* in Dutch)
  - atomically increase s unblocking threads waiting in `P` as appropriate
- but
  - you can’t read the value of the counter … why not?

UThread Semaphores

```
struct uthread_sem;
typedef struct uthread_sem* uthread_sem_t;

uthread_sem_t uthread_sem_create  (int initial_value);
void          uthread_sem_destroy (uthread_sem_t);
void          uthread_sem_wait    (uthread_sem_t);
void          uthread_sem_signal  (uthread_sem_t);
```

Using Semaphores to Drink Beer

- Use semaphore to store glasses held by pitcher
  - set initial value of empty when creating it

```
uthread_sem_t glasses = uthread_sem_create (0);
```

- Pouring and refilling don’t require a monitor

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

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

Using Semaphores to Implement Monitors

- Implementing Monitors
  - initial value of semaphore is `1`
  - `lock` is `wait()`
  - `unlock` is `signal()`
- Implementing Condition Variables
  - this is very hard, as it turns out
  - 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”

Hiding Asynchrony

Blocking Synchronous Operations

- use threads to hide asynchrony
- requires request to synchronize with completion handler

Using Monitors and Condition Variables

- to avoid wait-signal race, wait and signal must be done while mutex is held
  - problematic in cases where signaller can't block

```
void read (…) {
  uthread_mutex_lock (mx);
    scheduleRead (buf, nbytes, bno);
    uthread_cond_wait (complete)
  uthread_mutex_unlock (mx);
}
```

```
void readCompletionHandler() {
  uthread_mutex_lock (mx);
    uthread_cond_signal (complete);
  uthread_mutex_unlock (mx);
}
```

Using Semaphores

- no critical section, wait-signal race problem goes away … why?
  - signaller need not block

```
void read (…) {
  scheduleRead (buf, nbytes, bno);
  uthread_sem_wait (complete);
}
```

```
void readCompletionHandler() {
  uthread_sem_signal (complete);
}
```

Queue

- With condition variables
  - loop on wait, re-testing wait condition … why?

```
int dequeue (struct Q* q) {
  uthread_mutex_lock (q->mx);
    while (q->length==0)
      uthread_cond_wait (q->notEmpty);
    …
  uthread_mutex_unlock (q->mx);
}
```

```
void enqueue (struct Q* q, int i) {
  uthread_mutex_lock (q->mx);
    …
    uthread_cond_signal (q->notEmpty);
  uthread_mutex_unlock (q->mx);
}
```

- With semaphores
  - no need to loop … why not?

```
struct Q {
  uthread_sem_t mutex;    // initialize to 1
  uthread_sem_t length;   // initialize to 0
  ...
};
```

Why is `wait(length)` outside of critical section?

```
int dequeue (struct Q* q) {
  uthread_wait (q->length);
  uthread_wait (q->mutex);
    …
  uthread_signal (q->mutex);
}
```

```
void enqueue (struct Q* q, int i) {
  uthread_wait (q->mutex);
    …
  uthread_signal (q->mutex);
  uthread_signal (q->length);
}
```

Ordering Two Threads

- If you thread A to wait for thread B
  - initialize semaphore b to 0

Thread A

```
uthread_sem_wait (b);
```

Thread B

```
uthread_sem_signal (b);
```

- Rendezvous: both threads wait for each other
  - initialize semaphores a and b to 0

Thread A

```
uthread_sem_signal (a);
uthread_sem_wait (b);
```

Thread B

```
uthread_sem_signal (b);
uthread_sem_wait (a);
```

What if you reversed the order of **wait** and **signal** on either (or both) threads?

It works fine if you reverse one of them, but if **BOTH** of them **wait** first they deadlock.

Synchronization in Java

Mutex

- 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();
```

Conditions

- **await** is wait (replaces Object wait)
- **signal** or **signalAll** (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 ()** is wait() – also **acquire (n)**
- **release ()** is signal() – also **release (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

Java `AtomicReference<V>` Class

- `boolean` **`compareAndSet`** `(V expectedValue, V newValue)`
  - atomically sets reference to **`newValue`** if and only if its current value is the **`expectedValue`**; returns **`true`** if assignment is successful and **`false`** otherwise
- `V` **`get`**`()`
  - get the current value of reference

- Use instead of mutual exclusion to eliminate data races …

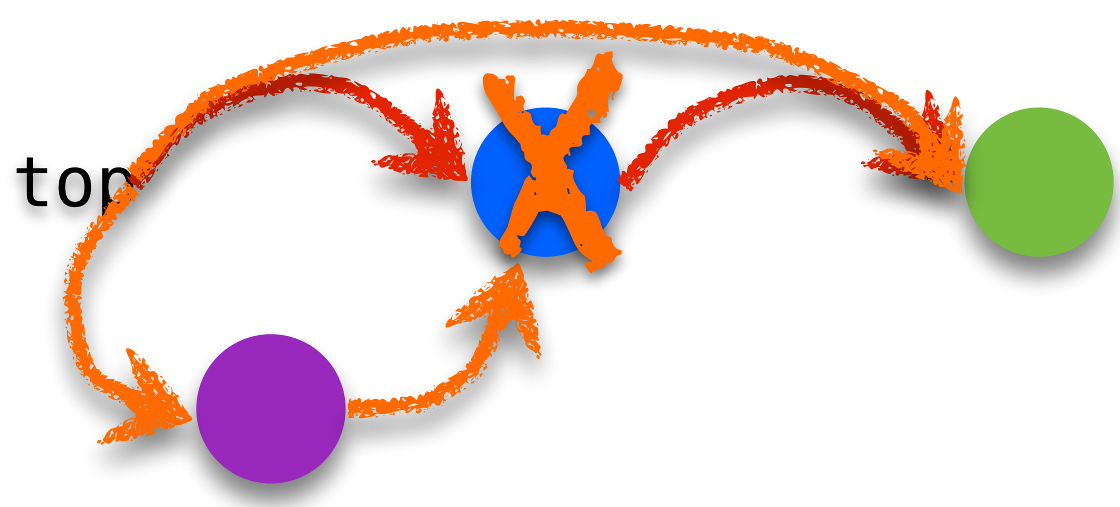
Lock-Free Atomic Stack in Java

- 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

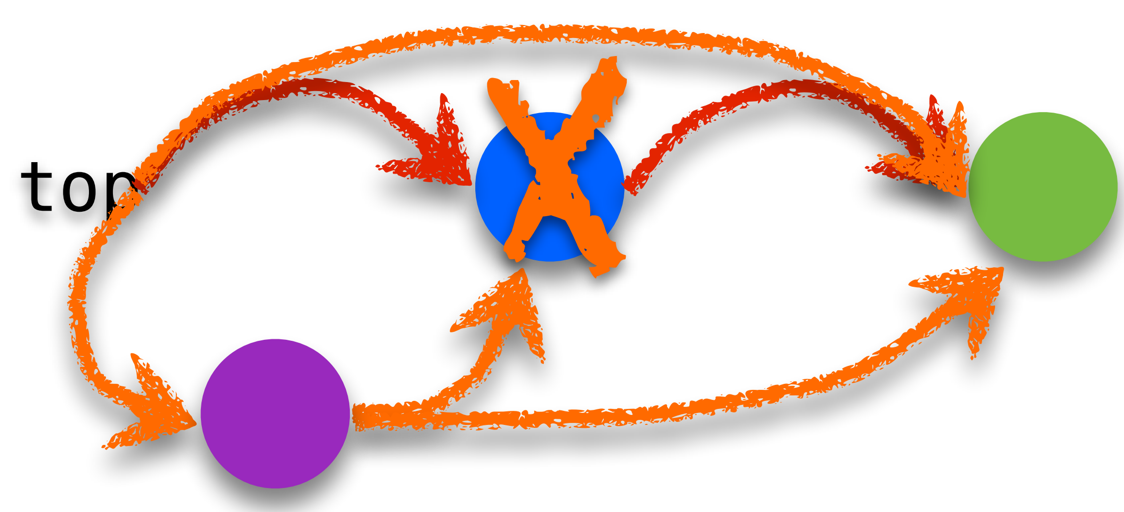


  - we solved this problem using locks to ensure mutual exclusion
  - now ... solve without locks, using **atomic compare-and-set** of top

```
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

Systems with multiple Mutexes

- We have already seen this with semaphores
- Consider a system with two mutexes: a and b

```
void foo() {
  uthread_mutex_lock   (a);
  uthread_mutex_unlock (a);
}
```

```
void bar() {
  uthread_mutex_lock   (b);
  uthread_mutex_unlock (b);
}
```

```
void x() {
  uthread_mutex_lock   (a);
    bar();
  uthread_mutex_unlock (a);
}
```

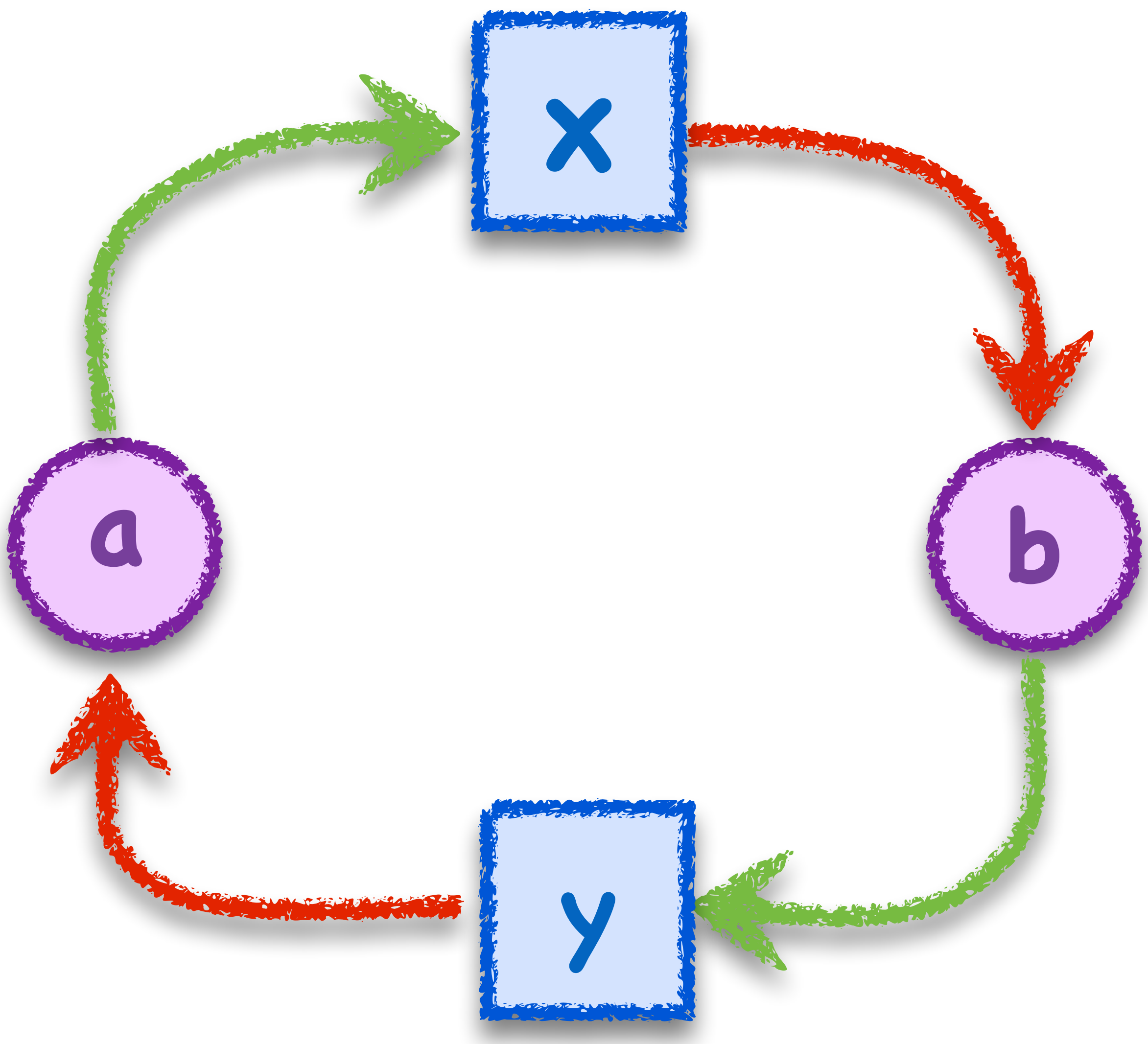
```
void y() {
  uthread_mutex_lock   (b);
    foo();
  uthread_mutex_unlock (b);
}
```

Any problems so far?

What about now?

Waiter Graph Can Show Deadlocks

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




```
void foo() {
  uthread_mutex_lock   (a);
  uthread_mutex_unlock (a);
}
```

```
void bar() {
  uthread_mutex_lock   (b);
  uthread_mutex_unlock (b);
}
```

```
void x() {
  uthread_mutex_lock   (a);
    bar();
  uthread_mutex_unlock (a);
}
```

```
void y() {
  uthread_mutex_lock   (b);
    foo();
  uthread_mutex_unlock (b);
}
```

The Dining Philosophers Problem

- Formulated by Edsger Dijkstra to explain deadlock (circa 1965)
  - 5 computers competed for access to 5 shared tape drives
  - as an exam question
- 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 (never get both forks) due to timing problem, but avoid deadlock

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
- Don’t use locks
  - for example, use atomic data structures and lock-free synchronization
- Use only one lock at a time
  - deadlock is not possible unless thread holding a lock waits (requires 2 sync variables)
- 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)

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/signal 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