a place of mind
THE UNIVERSITY OF BRITISH COLUMBIA

CPSC 213
Introduction to Computer Systems

Unit 2c
Synchronization

All slides adapted from materials by Mike Feeley, Jonatan Schroeder, Robert Xiao, and Jordon Johnson

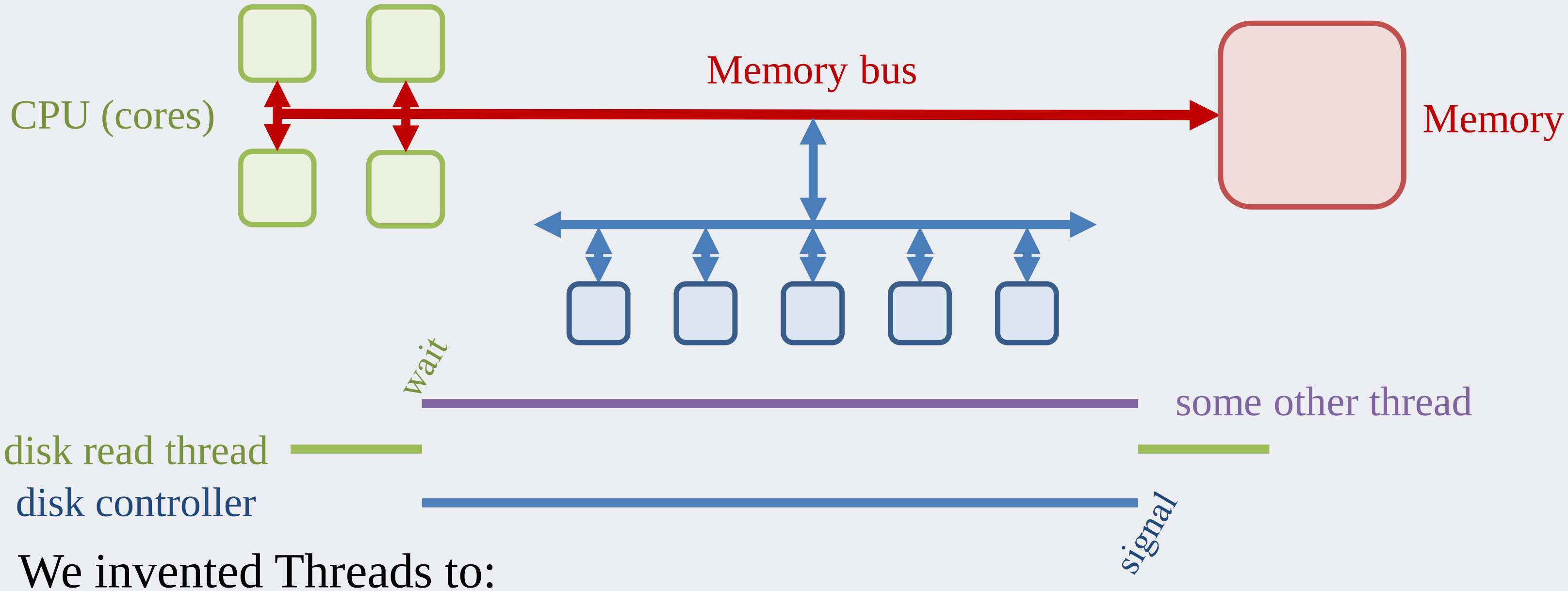
Announcements

- Google doc for lecture questions
  - See Piazza for link, section 102
  - https://docs.google.com/document/d/1G6hkekQS7mT9lFpP8AVftYao8vLRujIrRLAvOuX_07w/edit
  - Add your question anonymously (at the top)
  - Help answer questions too!

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, now we 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

- There will be problems if:
  - threads share a data structure
  - operations involve multiple memory accesses
  - these accesses can be arbitrarily interleaved
- Example: a stack implemented as an array
  - suppose two threads accessing concurrently
  - what happens if both push? Both pop? 1 push, 1 pop?

A push

1. if (n<4)

B push

2. if (n<4)
3. array[n] = i
4. n++;

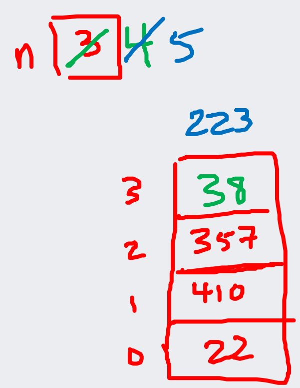
5. array[n] = i
6. n++

A

B

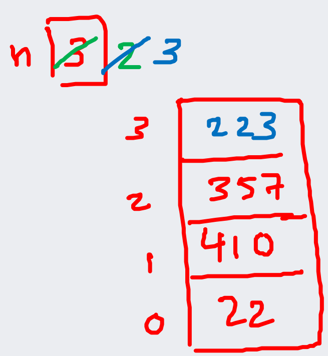
```
int n;
int array[4];

void push(i) {
  if (n < 4) {
    array[n] = i;
    n++;
  }
}

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

Both push



A pushes

B pops



return 357

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 by multiple threads (at least one updating)
  - conflicting operations on shared data structure are arbitrarily interleaved
  - unpredictable (non-deterministic) program behaviour – usually 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 only one thread at a time
  - reading and writing should be handled differently (more about this later)

A linked-list stack

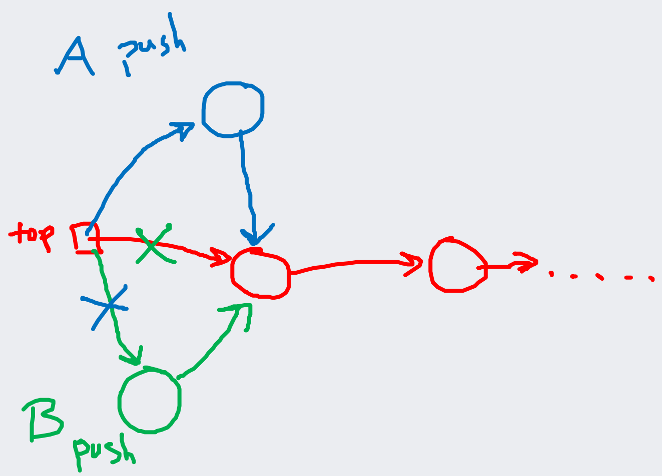
Still corrupting shared data

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

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

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

Still problems if both push? Both pop? 1 push, 1 pop?



Linked list stack

- Sequential/synchronous execution of repeated push/pop is OK…

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

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

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

Linked list stack

- …but concurrent execution doesn't always work

```
int main (void) {

  ...

  et = uthread_create(push_driver, num);
  dt = uthread_create(pop_driver, num);

  uthread_join(et, 0);
  uthread_join(dt, 0);

  assert (top == NULL);
}
```

Note: works for `uthread_init(1)` *without* pre-emption, but not guaranteed to work for `uthread_init(2)`

Linked list stack

The problem

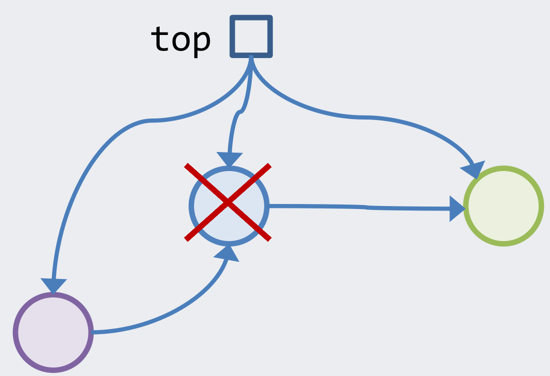
- Same situation as the array implementation
  - push and pop are critical sections on the shared stack
  - parallel execution leads to potential arbitrary interleaving of operations
  - sometimes, the interleaving corrupts the data structure

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

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

Suppose the execution occurs as follows:

```
1. e->next = top
                  2. e = top
                  3. top = top->next
                  4. return e
                  5. free(e)
6. top = e
```



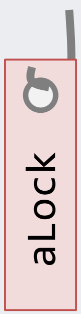
Mutual exclusion using locks

- 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

- Example: using locks on 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;
}
```



Quick questions with locks

- What happens when…

```
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;
}
```

  - Thread A calls `push_cs` while thread B is executing in `push_cs`?

    → waits for lock to release → holds the lock

  - Thread A calls `push_cs` while thread B is executing in `pop_cs`?

    → waits for lock to release → holds the lock

  - What if `push_cs` and `pop_cs` use different locks?

    synchronization problems.

  - What if `push_cs` or `pop_cs` never call unlock?

    lock is held forever, other threads cannot proceed,

Implementing simple locks

- An initial attempt
  - 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;
}
```

do nothing

claim the lock

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

Will this work?

Implementing simple locks

First attempt – problem!

- There is a race in the lock code
  - Suppose two threads both request a lock at (nearly) the same time

Thread A

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

Thread B

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

Suppose the execution occurs as follows:

Thread A	Thread B
`1. read *lock == 0, exit loop`	
	`2. read *lock == 0, exit loop`
`3. *lock = 1`	
`4. return with lock held`	
	`5. *lock = 1`
	`6. return with lock held`

Both threads think they hold the lock

Implementing simple locks

First attempt – bigger problems!

- The race exists even at the machine code level
  - two instructions are needed to acquire the lock
    - first, read to check that the lock is available
    - second, set the lock to held
  - but a read by another thread can check the lock before the first thread sets the hold

```
      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)`

We need a way to read AND write a memory location with no possibility of interruption

Atomic memory exchange

- Atomicity
  - is a general property in systems
  - where a group of operations are performed as a single, indivisible unit

- 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 the 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

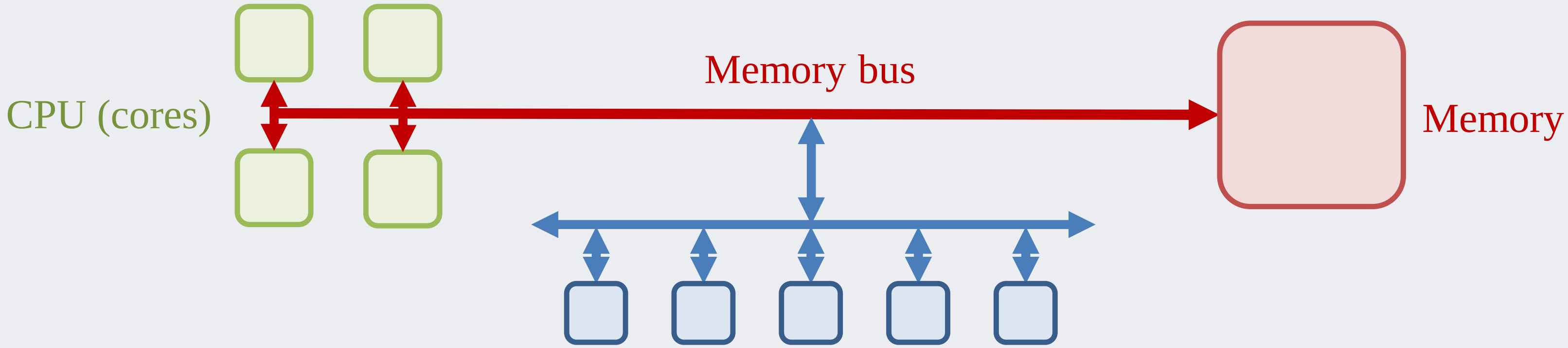
```
      ld   $lock, r1
      ld   $1,    r0
loop: xchg (r1),  r0   <- R/W
      beq  r0,  held
      br   loop
held: ...
```

Compare with:

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

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

free: st   r2,  (r1)   W
```

Problem: atomic exchange is EXPENSIVE

Implementing atomic exchange



- Cannot be implemented by CPU alone
  - must synchronize across multiple CPUs
    - accessing the same memory location at the same time

- Implemented by memory bus
  - memory bus synchronizes every CPU's 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, with higher overhead than normal read/write

Speeding up spinlocks

- Spin first on normal read
  - normal reads are 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, if 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 OK 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 might wait for a long time
  - it should block so that other threads can run
  - it will then unblock when it becomes runnable, when
    - either the lock is unlocked, or an event is signaled

- 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 blocked thread from lock'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

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);
```

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);
```

condition for acquiring lock

wait blocks the thread, release the lock

do whatever you need

  - 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);
```

set the available condition (for others)

wakes up a waiter

`wait` releases the mutex and blocks thread

- before waiter blocks, it atomically 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

Using conditions

- **`signal`** awakens at most one thread
  - waiter does not run until signaler releases the mutex explicitly
  - a third thread could intervene and acquire mutex before waiter
  - waiter must then 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 wake up too many
  - that's OK 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);
  aDesiredCondition--;
unlock (aMutex);
```

Event ordering exercise

IC_11_29 – order_starter.c

- Suppose we have two threads running concurrently
  - `t0` prints `"a"`
  - `t1` prints `"b"`
- We want to ensure the output is always `"ab"`, in that order
  - How?
    - add a mutex, condition, and a desired state variable (see earlier slides)

Video streaming with threads

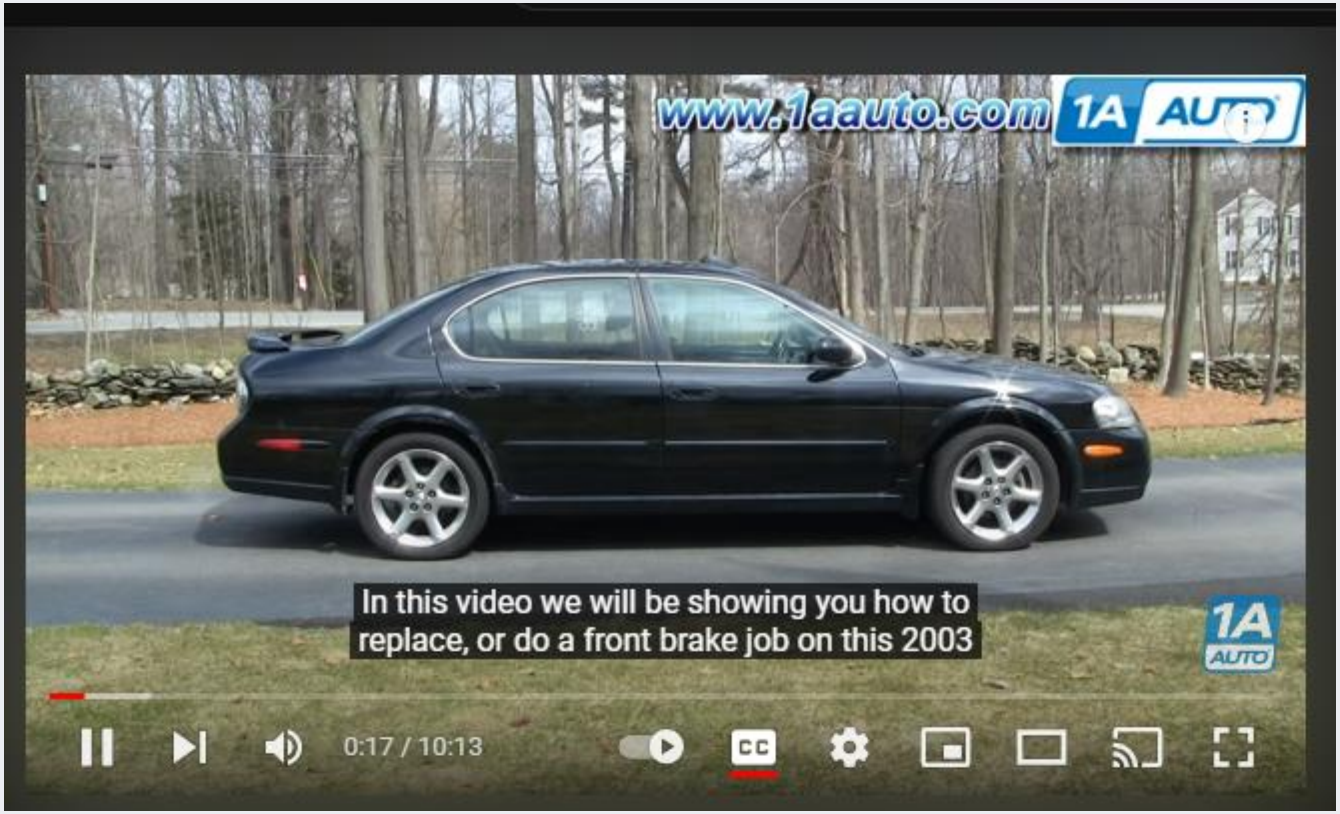


Image credit: https://www.youtube.com/@1AAuto

Video streaming with threads

- 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, other times 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 (or some other threshold of our choosing)
- Producer thread
  - fetch frame from network and put them in buffer
- Consumer thread
  - fetch frame from buffer, decode them and send them to video driver
- How are Producer and Consumer connected?
  - 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

Video streaming with threads

Step 1 – requesting frames

```
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() {
  while (1) {
    uthread_lock (mx);
      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() {
  while (1) {
    uthread_lock (mx);
      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);
  }
}
```

Video streaming with threads

Step 2 – delivering frames

```
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_cond_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_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);
}
```

Video streaming with threads

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_cond_t  have_frame;
uthread_cond_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);
}
```

```
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);
}
```

Final note:
show_next_frame will be signaled every time a new frame is required for the video driver; e.g. every 1/30 s

Producer and Consumer threads

General template

```
int canGoFlag = 1;
uthread_mutex_t mx;
uthread_cond_t canGoCond;
```

Producer (P)

```
uthread_mutex_lock(mx);
    uthread_cond_signal(canGoCond);
    canGoFlag = 1;
uthread_mutex_unlock(mx);
```

Consumer (C)

```
uthread_mutex_lock(mx);
    while (canGoFlag == 0)
        uthread_cond_wait(canGoCond);
    canGoFlag = 0;
uthread_mutex_unlock(mx);
```

- Key invariant: C does not complete until it is able to change `canGoFlag` from 0 to 1
    - so, if `canGoFlag` is 0, it waits for P to set it to 1
    - This waiting behaviour is achieved by P waiting on and C signaling the same condition

iClicker 2c.1

```
int canGoFlag = 1;
uthread_mutex_t mx;
uthread_cond_t canGoCond;
```

Producer (P)

```
uthread_mutex_lock(mx);
    uthread_cond_signal(canGoCond);
    canGoFlag = 1;
uthread_mutex_unlock(mx);
```

Consumer (C)

```
uthread_mutex_lock(mx);
    while (canGoFlag == 0)
        uthread_cond_wait(canGoCond);
    canGoFlag = 0;
uthread_mutex_unlock(mx);
```

```
void uthread_cond_wait (uthread_cond_t cond) {
  assert (cond->mutex->holder == uthread_self ());
  uthread_enqueue (&cond->waiter_queue, uthread_self());
  uthread_mutex_unlock (cond->mutex);
  uthread_block();
  uthread_mutex_lock (cond->mutex);
}
```

What would happen if `cond_wait` didn't unlock and lock as it does?

A. nothing; it's fine either way
B. it might mean that C would return when it shouldn't
C. it might mean that P would not be able to signal it correctly
D. P will never be able to signal C under any circumstances
E. I am not sure

iClicker 2c.2

```
int canGoFlag = 1;
uthread_mutex_t mx;
uthread_cond_t canGoCond;
```

Producer (P)

```
uthread_mutex_lock(mx);
    uthread_cond_signal(canGoCond);
    canGoFlag = 1;
uthread_mutex_unlock(mx);
```

Consumer (C)

```
uthread_mutex_lock(mx);
    while (canGoFlag == 0)
        uthread_cond_wait(canGoCond);
    canGoFlag = 0;
uthread_mutex_unlock(mx);
```

Why not this? (CX)

```
uthread_mutex_lock(mx);
    if (canGoFlag == 0)
        uthread_cond_wait(canGoCond);
    canGoFlag = 0;
uthread_mutex_unlock(mx);
```

Suppose P and CX are the only threads accessing the mutex. Which statement is correct?

A. CX always works
B. CX never works
C. CX works if and only if there is a single P thread
D. CX works if and only if there is a single CX thread
E. I am not sure

P1

3. acquires mutex
4. sets condition =1, signals
5. release mutex

CX

1. check condition, sees 0
2. waits release mutex

6. wakes up, attempt to acquire mutex
7. fails, goes back to sleep

9. wakes up due to signal, acquires mutex
10. proceeds as if condition==1 ✓
11. sets condition = 0
12. release mutex

P2

6. acquires mutex
7. sets condition to 1, signals
8. release mutex

P

1. check condition, sees 0
2. wait, release mutex

3. acquires mutex
4. sets condition = 1, signal
5. release mutex

Cx1

3. wakes up, attempt to get mutex
4. fails, goes back to sleep

7. wake up, acquires mutex
8. proceed as if condition == 1 ?
9. release mutex

Cx2

3. acquires mutex
4. checks condition, sees 1
5. proceeds, sets condition = 0
6. releases mutex

suppose there is a signal

Beer for everyone!

- Beer pitcher is a shared data structure with:
  - `glasses`: amount of beer left, in glasses (`int`)
  - `pour()`: pours one glass from pitcher (reduces beer left by `1`)
  - `refill()`: adds more beer to pitcher (increases beer left by `N`)

- Implementation goal
  - synchronize access to pitcher
  - pouring from empty pitcher requires waiting for it to be refilled
  - filling pitcher releases waiting threads

Beer drinking implementation

Static declaration

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

can do these with global variables

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);
  ...
}
```

Beer drinking implementation

Pouring a glass

```
void pour (struct BeerPitcher* p) {
  uthread_mutex_lock (p->mx);
    while (p->glasses == 0)
      uthread_cond_wait (p->hasBeer);
    p->glasses -= 1;
  uthread_mutex_unlock (p->mx);
}
```

similar layout to threaded consumer

loop signal vs broadcast

decide by how many waiters vs.

how much resource

Refilling the pitcher (pitcher has unlimited capacity)

```
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);
}
```

or `uthread_cond_broadcast`

Signal and mutex race

The mutex in action!

```
void pour (struct BeerPitcher* p) {
  uthread_mutex_lock (p->mx);
    while (p->glasses == 0)
      uthread_cond_wait (p->hasBeer);
    p->glasses -= 1;
  uthread_mutex_unlock (p->mx);
}
```

pour

```
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);
}
```

pour

Thread A

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

Thread B

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

race to get mutex

Thread A

8a. tries to acquire mutex
9a. fails, waits on mutex

Thread C

8c. `pour` acquires mutex
9. sets `glasses = 0`
10. releases mutex

Thread A

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

Beer for everyone, forever!

Extending the beer pouring example

- What if we 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, and when it is, it refills the pitcher to its full capacity and awakens any pourers

Static declaration

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

and an appropriate instantiation and initialization

Automatic refills

```
void pour (struct BeerPitcher* p) {
  uthread_mutex_lock (p->mx);
    while (p->glasses == 0)
      uthread_cond_wait (p->hasBeer);
    p->glasses -= 1;
    if (p->glasses == 0)
      uthread_cond_signal (p->isEmpty)
  uthread_mutex_unlock (p->mx);
}
```

same as before

NEW!

```
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 < n; i++)
        uthread_cond_signal (p->hasBeer);
    }
  uthread_mutex_unlock (p->mx);
}
```

max Glasses

} or broadcast

Back to disk reads

- Suppose we write an asynchronous disk read
  - and we want to block/wait with conditions

```
void read (char* buf, int nbytes, int blockno) {

    scheduleRead (buf, nbytes, blockno);
  uthread_mutex_lock(mx);
    uthread_cond_wait(disk_op_complete);
  uthread_mutex_unlock(mx);
}
```

something might happen here.

```
void readCompleteCalledByISR () {
  uthread_mutex_lock(mx);
    uthread_signal(disk_op_complete);
  uthread_mutex_unlock(mx);
}
```

If it takes a long time, disk_op_complete may signal before wait is called

- Wait-signal race problem:
  - wait condition check / trigger and wait are not atomic
  - signal could occur before wait, thus waiter could miss signal and never wake up

- Solution:
  - Ensure that condition check / trigger and wait are atomic
  - So that wait is ordered before signal

Reader-writer monitors

- If we classify critical sections as:
  - reader — it only reads the shared data
  - writer — it 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 states: free, held for reading, or held for writing by a single writer
  - If held for reading, multiple readers can access simultaneously
    - but, we will need to know how many readers are accessing, and when they are done

Reader-writer monitors

Operations

- `mutex_lock()`: lock for writing
  - only acquires lock if it is free
  - sets state to held for writing
- `mutex_lock_read_only()`:
  - if lock is free, set its state to held for reading
  - increments a reader count
- `mutex_unlock()`:
  - if held for writing, set state to free
  - if held for reading, decrement reader count
    - if reader count reaches zero, set state to free

Fair access to reader-writer lock

- Policy question
  - if monitor state is held for reading and…
    - Thread A calls `mutex_lock()`, and blocks while waiting for monitor to be free
    - Thread B calls `mutex_lock_read_only()`
  - What should we do?

- Option 1 – disallow new readers while a writer is waiting
  - affects a thread that could be running but now must wait
  - provides fair access to monitor (writer may have been waiting longer)
- Option 2 – allow new readers while a writer is waiting
  - increases concurrency, allows more threads to run
  - writer may need to wait for a long time to get access (starvation)

- Solution
  - tradeoffs, may depend on application

Semaphores

- Introduced by Edsger Dijkstra (THE System, ~1968)

- A semaphore is a non-negative atomic counter
  - any attempt to make counter negative will block the calling thread
  - No operation to read value; only to change it

- `P(s)` (or wait):
  - From Dutch: *prober te verlagen* – "try lowering"
  - atomically blocks until $s > 0$, then decrements s

- `V(s)` (or signal):
  - From Dutch: *verhogen* – "to increase"
  - atomically increase s, and unblock threads waiting in `P` as appropriate

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);
```

Note that there is no broadcast!

Drinking beer with semaphores

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

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

- Pour and refill no longer require a monitor
  - since the semaphore atomically changes the counter already

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

atomically decrement if it can.
otherwise sleep.
try again after waking up.

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

each signal increments by 1

Implementing monitors using semaphores

- Implementing a mutex using semaphores:
  - create semaphore with initial value 1 (free)
  - lock is `P()` / wait
  - unlock is `V()` / signal()

- Implementing condition variables using semaphores:
  - Difficult!
    - In condition variables, signal without wait will be ignored / no effect
    - In semaphores, signal can unlock a future wait
  - Further reading: *Andrew D. Birrell. "Implementing Condition Variables with Semaphores", 2003.*

Example: semaphore for disk read

- Asynchronous read request

```
void read (char* buf, int nbytes, int blockno) {
  scheduleRead(buf, nbytes, blockno);
  uthread_sem_wait(readComplete);
  ... // handle data
}
```

interruptions here are ok

- Read completion (called by disk ISR)

```
void onReadComplete() {
  uthread_sem_signal(readComplete);
}
```

No critical section, no wait-signal race problem

Ordering threads using semaphores

- If thread B must wait for thread A to finish
  - Initialize semaphore to 0, i.e. `uthread_sem_t b = uthread_sem_create(0);`
  - Thread A: `uthread_sem_signal(b);`
  - Thread B: `uthread_sem_wait(b);`

Compare with PL activity IC_11_29 (slide 25)

- If both threads need to wait for each other (rendezvous)
  - Initialize two semaphores with 0
  - Thread A:

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

wait
signal

  - Thread B:

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

wait
signal

What happens if the order of wait and signal are reversed for either (or both) threads?

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 if `x` calls foo?

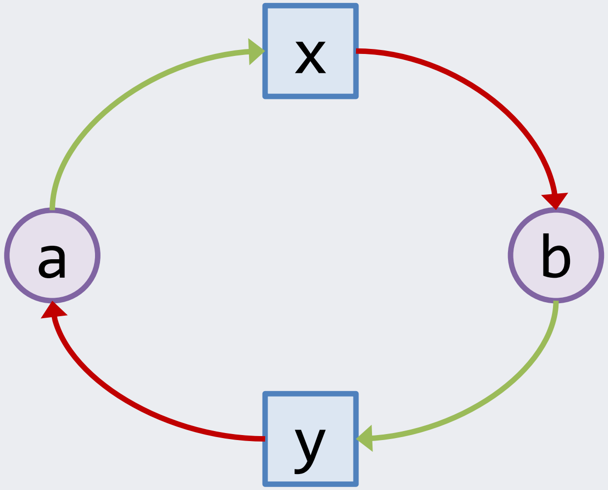
What about now?

OK if a single thread runs x, or y

but threads for x and y at the same time will deadlock

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 is WAITING for 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 around 1965 as an exam problem

- Description:
  - 5 philosophers sit at a round table with one fork placed in between each
    - so there is a fork to the left and right of each philosopher

  - each philosopher is either eating, or thinking
    - if eating, they are not thinking, and while thinking, they are not eating
    - they never speak to each other

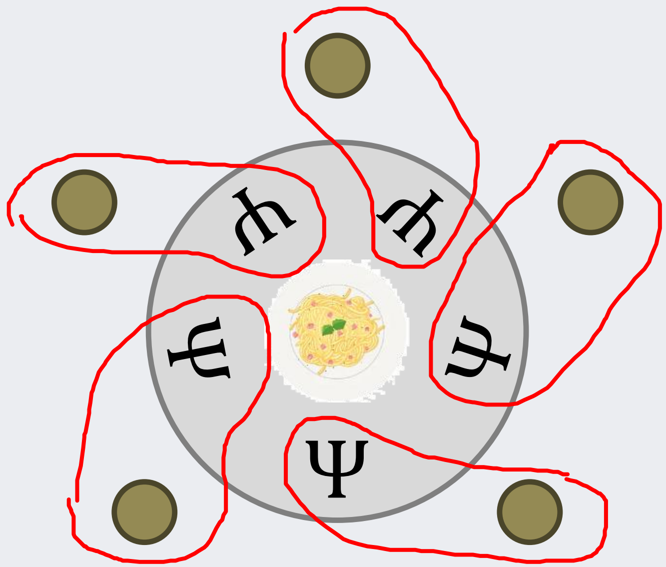
  - a large bowl of spaghetti in the middle of the table requires 2 forks to serve
    - if a philosopher wants to eat, he grabs his 2 adjacent forks to do so
    - if another philosopher is using the fork, then the other must wait

The dining philosophers problem

Deadlock

- Assume that philosophers always start with the left fork

- Also assume that all philosophers decide to start eating at the same time
  - Everyone is able to get the left fork
  - But everyone waits for the right fork
  - DEADLOCK



The dining philosophers problem

Livelock

- Assume that, if philosophers can't get the second fork, they release the first fork, then wait on the second
  - if all of them do this at the same time, they will now hold the right fork
  - but none can proceed because they can't get the left fork

- If the process is repeated, and all philosophers are synchronized
  - Philosophers will repeatedly get one fork at a time
  - All are busy (never idle waiting), but still are unable to eat
  - LIVELOCK
    - threads respond to actions by other threads, but other threads also respond the same way
    - nobody can make progress

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)
- Mutex and Condition Variables
  - blocking locks, stop thread while it is waiting
  - mutex 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
  - can be used to implement a mutex or condition variable (nearly the same semantics)
    - other uses include: turnstiles, thread ordering, and rendezvous
- Problems, problems, problems
  - race conditions to be avoided using synchronization
  - deadlock/livelock to be avoided using synchronization carefully