CPSC 213

Introduction to Computer Systems

Unit 2c

Synchronization
Reading

- Companion
  - 8

- Text
  - 2ed: 12.4-12.6, parts of 12.7
  - 1ed: 13.4-13.5, (no equivalent to 12.6), parts of 13.7
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

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

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

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

Sequential test works

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

```c
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

```c
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

```c
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`
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

```c
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`
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

```c
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`
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**

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

```c
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

```c
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**

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

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

**Thread B**

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

2. read *lock==0, exit loop
5. *lock = 1, return

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

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

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

free: st r2, (r1)
```

Thread A

```
ld (r1), r0
st r2, (r1)
```

Thread B

```
ld (r1), r0
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Semantics</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic exchange</td>
<td>( r[v] \leftarrow m[r[a]] )</td>
<td>xchg (ra), rv</td>
</tr>
</tbody>
</table>
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
A Spinlock is
• a lock where waiter *spins* on looping memory reads until lock is acquired
• also called “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
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
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 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

- Lock data structure

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

- The `lock` operation

```c
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);
}
```
The **unlock** operation

```c
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);
    waiter_thread->state = TS_RUNABLE;
    ready_queue_enqueue (waiter_thread);
}
```
Blocking Lock Example Scenario

Thread A

1. calls lock()
3. grabs spinlock
5. acquires blocking lock
6. releases spinlock
7. returns from lock()

9. calls unlock()
10. grabs spinlock
11. releases lock
12. restarts a Thread B
13. releases spinlock
14. returns from unlock()

Thread B

2. calls lock()
4. tries to grab spinlock, but spins
3. grabs spinlock
4. queues itself on waIter list
5. releases spinlock
6. blocks

Thread C

8. scheduled

15. yields, blocks or stops

16. scheduled
17. grabs spinlock
18. acquires blocking lock
19. releases spinlock
20. returns from lock()
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 head for some time
  - when contention is high
  - when event wait may be long
Mutual exclusion plus inter-thread synchronization
- introduced by Tony Hoare and Per Brinch Hansen circ. 1974
- basis for synchronization primitives in Java etc.

Monitor
- is a mutual-exclusion lock
- primitives are enter (lock) and exit (unlock)

Condition Variable
- allows threads to synchronize with each other
- `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)
Using Conditions

- **Basic formulation**
  - one thread enters monitor and may wait for a condition to be established
    ```java
    monitor {
        while (!x)
            wait ();
    }
    ```
  - another thread enters monitor, establishes condition and signals waiter
    ```java
    monitor {
        x = true;
        notify ();
    }
    ```

- **wait** exists the monitor and blocks thread
  - before waiter blocks, it exists 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

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

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

```c
monitor {
    while (!x)
        wait ();
}
```
```c
monitor {
    x += n;
    notify_all ();
}
```

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

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

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

**notify_all** awakens all threads

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

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

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

```java
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

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

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

    monitor {
        glasses+=n;
        notify_all ();
    }
}
```
Monitors and Condition Variables

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

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

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

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

```c
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

```c
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

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

```c
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

```c
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

```c
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 dequeue have a while loop to check for non-empty?
- Why must condition variable be associated with specific monitor?
- Why can’t 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
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 its 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
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 its 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**
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
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 its 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
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
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
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
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 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
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
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**
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**
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**
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 its 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
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 its 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
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
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 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
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
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 provided 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

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

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

- Pouring and refilling don’t require a monitor

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

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

- Getting the beer worm, 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: \[P \text{ (outstanding\_request)}\]
  - completion interrupt: \[V \text{ (outstanding\_request)}\]

- **Rendezvous**
  - two threads wait for each other before continuing
  - create a semaphore for each thread initialized to 0

```c
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

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

```c
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
  - Google “semaphores condition variables birrell”
Synchronization in Java (5)

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

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

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

- multiple-reader single writer locks

```java
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)

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

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

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

- acquire () or acquire (n) is P() or P(n)
- release () or release (n) is V() or V(n)

```java
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
Recall the problem with concurrent stack

- 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 \texttt{atomic compare-and-set} of top
Recall the problem with concurrent stack

- 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

```
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;
}
```
Recall the problem with concurrent stack

- 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
Recall the problem with concurrent stack

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

```
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;
}
```

- We solved this problem using locks to ensure mutual exclusion
- Now ... solve without locks, using **atomic compare-and-set** of top
Recall the problem with concurrent stack

- A pop could intervene between two steps of push, corrupting the linked list.

- We solved this problem using locks to ensure mutual exclusion.

- Now, let's solve without locks, using **atomic compare-and-set** of the `top` variable.

```
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;
}
```
```cpp
class Element {
    Element* next;
};

class Stack {
    AtomicReference<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));
    }
};
```
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;
            e.next = t;
        } while (!top.compareAndSet (t, e));
    }
}
class Element {
    Element* next;
}

class Stack {
    AtomicReference<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));
    }
}
class Element {
    Element* next;
}

class Stack {
    AtomicReference.atomicReferenceElement> 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));
    }
}
```cpp
class Element {
    Element* next;
}

class Stack {
    AtomicReference<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));
    }
}
```
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));
    }
}
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));
    }
}
class Element {
    Element* next;
}

class Stack {
    AtomicReference<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));
    }
}
```cpp
class Element {
    Element* next;
}

class Stack {
    AtomicReference<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));
    }
}
```
```cpp
class Element {
    Element* next;
};

class Stack {
    AtomicReference<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));
    }
}
```
class Element {
    Element* next;
}

class Stack {
    AtomicReference<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));
    }
}
```cpp
class Element {
    Element* next;
}

class Stack {
    AtomicReference<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
What should we do for a program like this

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

Here is implementation of lock, is this okay?

```c
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

```c
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

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

void bar() {
    uthread_monitor_enter (b);
    uthread_monitor_exit (b);
}
```
Systems with multiple monitors

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

```c
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);
}
```
Systems with multiple monitors

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

```c
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);
}
```

Any problems so far?
Systems with multiple monitors

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

```c
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?
We have already seen this with semaphores.

Consider a system with two monitors a, and b.

```c
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 **HOLDs** lock
  - edge from **thread** to lock if thread **WANTs** lock
  - a cycle indicates deadlock

```c
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);
}
```
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

```c
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);
}
```
Waiter Graph Can Show Deadlocks

- **Waiter graph**
  - edge from lock to thread if thread **HOLDs** lock
  - edge from thread to lock if thread **WANTs** lock
  - a cycle indicates deadlock

```c
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);
}
```
Waiter Graph Can Show Deadlocks

- Waiter graph
  - edge from lock to thread if thread **HOLDs** lock
  - edge from thread to lock if thread **WANTs** lock
  - a cycle indicates deadlock

```c
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);
}
```
Waiter Graph Can Show Deadlocks

- Waiter graph
  - edge from lock to thread if thread \texttt{HOLD}s lock
  - edge from thread to lock if thread \texttt{WANT}s lock
  - a cycle indicates deadlock

```c
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)
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