Synchronization

key concepts: critical sections, mutual exclusion, test-and-set, spinlocks, blocking and blocking locks, semaphores, condition variables, deadlocks

Lesley Istead

David R. Cheriton School of Computer Science
University of Waterloo

Spring 2021

Thread Synchronization

- All threads in a concurrent program **share access** to the program's global variables and the heap.
- The part of a concurrent program in which a shared object is accessed is called a *critical section*.
- What happens if several threads try to access the same global variable or heap object at the same time?

Critical Section Example

```
/* Note the use of volatile; revisit later */
int volatile total = 0;

void add() {
   int i;
   for (i=0; i<N; i++) {
      total++;
   }
}
```

```
void sub() {
   int i;
   for (i=0; i<N; i++) {
      total--;
   }
}
```

> If one thread executes add and another executes sub what is the value of total when they have finished?

Critical Section Example (assembly detail)

```
/* Note the use of volatile */
int volatile total = 0;

void add() {
   loadaddr R8 total
   for (i=0; i<N; i++) {
      lw R9 0(R8)
      add R9 1
      sw R9 0(R8)
   }
}
```

```
void sub() {
   loadaddr R10 total
   for (i=0; i<N; i++) {
      lw R11 0(R10)
      sub R11 1
      sw R11 0(R10)
   }
}
```

Critical Section Example (Trace 1)

```
Thread 1                    Thread 2
loadaddr R8 total
lw R9 0(R8)  R9=0
add R9 1     R9=1
sw R9 0(R8) total=1
               <INTERRUPT>
                            loadaddr R10 total
                            lw R11 0(R10)   R11=1
                            sub R11 1       R11=0
                            sw R11 0(R10)   total=0
```

One possible order of execution. Final value of `total` is 0.

Critical Section Example (Trace 2)

```
Thread 1                    Thread 2
loadaddr R8 total
lw R9 0(R8)  R9=0
add R9 1     R9=1
         <INTERRUPT and context switch>
                            loadaddr R10 total
                            lw R11 0(R10)   R11=0
                            sub R11 1       R11=-1
                            sw R11 0(R10)   total=-1
                  ...
         <INTERRUPT and context switch>
sw R9 0(R8) total=1
```

One possible order of execution. Final value of `total` is 1.

Critical Section Example (Trace 3)

```
Thread 1                          Thread 2
loadaddr R8 total                 loadaddr R10 total
lw R9 0(R8)  R9=0                 lw R11 0(R10)   R11=0
add R9 1     R9=1                 sub R11 1       R11=-1
sw R9 0(R8) total=1
                                  sw R11 0(R10)   total=-1
```

Another possible order of execution, this time on two processors. Final value of `total` is -1.

Race Condition

A **race condition** is when the program result depends on the order of execution. Race conditions occur when multiple threads are reading and writing the same memory at the same time.

Sources of race conditions:

1. implementation
2. ...
3. ...

... more sources of race conditions to come!

Is there a race condition?

```
int list_remove_front(list *lp) {
   int num;
   list_element *element;
   assert(!is_empty(lp));
   element = lp->first;
   num = lp->first->item;
   if (lp->first == lp->last) {
     lp->first = lp->last = NULL;
   } else {
     lp->first = element->next;
   }
   lp->num_in_list--;
   free(element);
   return num;
}
```


Is there a race condition?

```
void list_append(list *lp, int new_item) {
   list_element *element = malloc(sizeof(list_element));
   element->item = new_item
   assert(!is_in_list(lp, new_item));
   if (is_empty(lp)) {
     lp->first = element; lp->last = element;
   } else {
     lp->last->next = element; lp->last = element;
   }
   lp->num_in_list++;
}
```

Tips for identifying race conditions

- find the critical sections
  - inspect each variable; is it possible for multiple threads to read/write it at the same time?
  - constants and memory that all threads only **read**, do not cause race conditions

What next?

After identifying the critical sections, how can you prevent race conditions?

Enforcing Mutual Exclusion With Locks

```
int volatile total = 0;
/* lock for total: false => free, true => locked */
bool volatile total_lock = false; // false means unlocked

void add() {                      void sub() {
   int i;                            int i;
   for (i=0; i<N; i++) {             for (i=0; i<N; i++) {
   Acquire(&total_lock);              Acquire(&total_lock);
        total++;                            total--;
   Release(&total_lock);              Release(&total_lock);
   }                                  }
}                                 }
```

Acquire/Release must ensure that only one thread at a time can hold the lock, even if both attempt to Acquire at the same time. If a thread cannot Acquire the lock immediately, it must wait until the lock is available.

Locks provide mutual exclusion and are often referred to as a **mutex**.

Lock Aquire and Release

```
Acquire(bool *lock) {
  while (*lock == true) ; /* spin until lock is free */
  *lock = true;             /* grab the lock */
}

Release(book *lock) {
  *lock = false;            /* give up the lock */
}
```

Does this work?

Lock Aquire and Release

```
Acquire(bool *lock) {
  while (*lock == true) ; /* spin until lock is free */
  *lock = true;             /* grab the lock */
}

Release(book *lock) {
  *lock = false;            /* give up the lock */
}
```

It does not! Why?

How could you fix it?

Hardware-Specific Synchronization Instructions

- provide a way to implement atomic **test-and-set** for synchronization primitives like locks
- example: the atomic x86 (and x64) `xchg` instruction:

  `xchg src,addri`

  where `src` is a register, and `addr` is a memory address. Swaps the values stored in `src` and `addr`.
- logical behavior of `xchg` is an **atomic** function that behaves like this:

```
Xchg(value,addr) {
  old = *addr;
  *addr = value;
  return(old);
}
```


x86 - Lock Aquire and Release with Xchg

```
Acquire(bool *lock) {
  while (Xchg(true,lock) == true) ;
}

Release(bool *lock) {
  *lock = false;              /* give up the lock */
}
```

If `Xchg` returns `true`, the lock was already set, and we must continue to loop. If `Xchg` returns `false`, then the lock was free, and we have now acquired it.

This construct is known as a *spin lock*, since a thread busy-waits (loops) in Acquire until the lock is free.

ARM Synchronization Instructions

- **exclusive** load (LDREX) and store (STREX) operations
  - LDREX and STREX act as a barrier; must be used together
  - LDREX loads a value from address *addr*
  - STREX will attempt to store a value to address *addr*
  - STREX will fail to store value at address *addr* if *addr* was touched between the LDREX and STREX

LDREX and STREX

STREX may fail even if the distance between LDREX and STREX is small, but should succeed after a few attempts. It is recommended to place these instructions close together (128bits).

Lock Acquire with LDREX and STREX

```
ARMTestAndSet(addr, value) {
        tmp = LDREX addr // load value
        result = STREX value, addr  // store new value
        if (result == SUCCEED) return tmp
        return TRUE
}

Acquire(bool *lock) {
        while( ARMTestAndSet(lock, true) == true ) {};
}
```

ARMTestAndSet returns TRUE if the lock is already owned, OR, if STREX fails, so that we keep trying to acquire the lock. ARMTestAndSet **ONLY** returns FALSE if the lock is available, AND, if STREX succeeds.

MIPS Synchronization Instructions

- similar to ARM, two instructions are used; `ll` and `sc`
  - `ll`, load linked, load value at address *addr*
  - `sc`, store conditional, store new value at *addr* if the value at *addr* has not changed since `ll`

sc

... returns SUCCESS if the value stored at the address has not changed since `ll`. The value stored at the address can be any 32bit value. `sc` **does not** check what that value at the address is, it only checks if it has changed.

Lock Acquire with `ll` and `sc`

```
MIPSTestAndSet(addr, value) {
        tmp = ll addr  // load value
        result = sc addr, value // store conditionally
        if ( result == SUCCEED ) return tmp
        return TRUE
}

Acquire(bool *lock) {
        while( MIPSTestAndSet(lock, true) == true ) {};
}
```

Initial Lock Value	Lock Value at ll	Lock Value at sc	Lock Value after sc	sc Returns	Lock State
FALSE	FALSE	FALSE	TRUE	SUCCEED	own lock
FALSE	FALSE	TRUE	TRUE	FAIL	keep spinning, no lock
TRUE	TRUE	TRUE	TRUE	SUCCEED	keep spinning, no lock
TRUE	TRUE	FALSE	FALSE	FAIL	keep spinning, no lock

Spinlocks in OS/161

A spinlock is a lock that "spins", repeatedly testing lock availability in a loop until the lock is available. Threads actively use the CPU while they "wait" for the lock. In OS/161, spinlocks are already defined.

```
struct spinlock {
  volatile spinlock_data_t lk_lock;
  struct cpu *lk_holder;
};

void spinlock_init(struct spinlock *lk}
void spinlock_acquire(struct spinlock *lk);
void spinlock_release(struct spinlock *lk);
```

spinlock_acquire calls spinlock_data_testandset in a loop until the lock is acquired.

OS/161 spinlock_acquire

```
/* return value 0 indicates lock was acquired */
spinlock_data_testandset(volatile spinlock_data_t *sd)
{
  spinlock_data_t x,y;
  y = 1;
  __asm volatile(
    /* assembly instructions  x = %0, y = %1, sd = %2 */
    ".set push;"       /* save assembler mode */
    ".set mips32;"     /* allow MIPS32 instructions */
    ".set volatile;"   /* avoid unwanted optimization */
    "ll %0, 0(%2);"    /*   x = *sd */
    "sc %1, 0(%2);"    /*   *sd = y; y = success? */
    ".set pop"         /* restore assembler mode */
    : "=r" (x), "+r" (y) : "r" (sd)); /* outputs : inputs */
  if (y == 0) { return 1; }
  return x;
}
```

C Inline Assembly

"=r" → write only, stored in a register
"+r" → read and write, stored in a register
"r" → input, stored in a register

OS/161 Locks

- In addition to spinlocks, OS/161 also has **locks**.
- Like spinlocks, locks are used to enforce mutual exclusion.

```
struct lock *mylock = lock_create("LockName");

lock_aquire(mylock);
  critical section /* e.g., call to list_remove_front */
lock_release(mylock);
```

- spinlocks spin, locks **block**:
  - a thread that calls `spinlock_acquire` spins until the lock can be acquired
  - a thread that calls `lock_acquire` **blocks** until the lock can be acquired

Locks

... can be used to protect larger critical sections without being a burden on the CPU. They are a type of **mutex**. Have owners.

Spinlocks and Locks Additional Notes

- spinlocks and locks have an **owner**; so they cannot be involuntarily released
  - a spinlock is owned by a **CPU**
  - a lock is by a **thread**
- spinlocks **disable interrupts** on their CPU
  - preemption is disabled on that CPU (hence, owned by CPU); but not others
  - minimizes spinning
  - **DO NOT** use spinlocks to protect large critical sections

Spinlock and Lock API

spinlock

- `void spinlock_init(struct spinlock *lk)`
- `void spinlock_acquire(struct spinlock *lk)`
- `void spinlock_release(struct spinlock *lk)`
- `bool spinlock_do_i_hold(struct spinlock *lk)`
- `void spinlock_cleanup(struct spinlock *lk)`

lock

- `struct lock *lock_create(const char *name)`
- `void lock_acquire(struct lock *lk)`
- `void lock_release(struct lock *lk)`
- `bool lock_do_i_hold(struct lock *lk)`
- `void lock_destroy(struct lock *lk)`

Thread Blocking

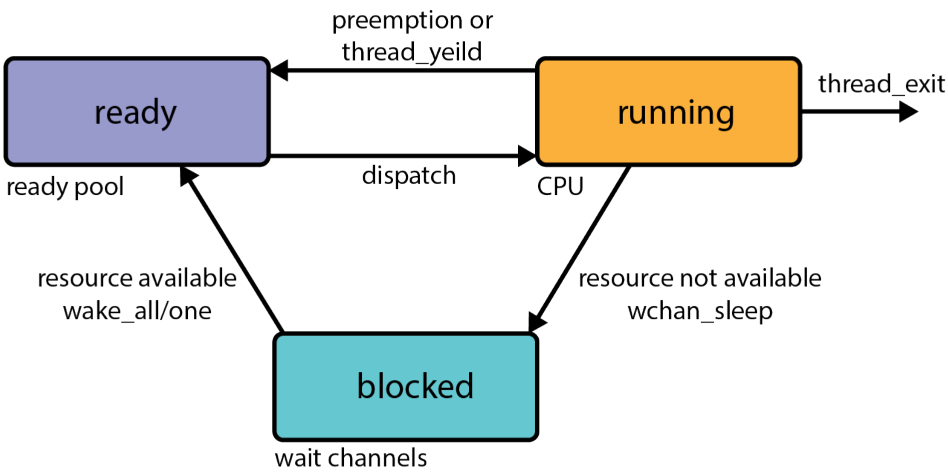
- Sometimes a thread will need to wait for something, e.g.:
  - wait for a lock to be released by another thread
  - wait for data from a (relatively) slow device
  - wait for input from a keyboard
  - wait for busy device to become idle
- When a thread blocks, it stops running:
  - the scheduler chooses a new thread to run
  - a context switch from the blocking thread to the new thread occurs,
  - the blocking thread is queued in a **wait queue** (not on the ready list)
- Eventually, a blocked thread is signaled and awakened by another thread.

Wait Channels in OS/161

- wait channels are used to implement thread blocking in OS/161
  - `void wchan_sleep(struct wchan *wc);`
    - blocks calling thread on wait channel `wc`
    - causes a context switch, like `thread_yield`
  - `void wchan_wakeall(struct wchan *wc);`
    - unblock all threads sleeping on wait channel `wc`
  - `void wchan_wakeone(struct wchan *wc);`
    - unblock one thread sleeping on wait channel `wc`
  - `void wchan_lock(struct wchan *wc);`
    - prevent operations on wait channel `wc`
    - more on this later!
- there can be many different wait channels, holding threads that are blocked for different reasons.

> wait channels in OS/161 are implemented with queues

Thread States, Revisited



running: currently executing
ready: ready to execute
blocked: waiting for something, not ready execute

> ready threads are queued on the ready queue, blocked threads are queued on wait channels

Semaphores

- A semaphore is a synchronization primitive that can be used to enforce mutual exclusion requirements. It can also be used to solve other kinds of synchronization problems.
- A semaphore is an object that has an integer value, and that supports two operations:
  - P: if the semaphore value is greater than 0, decrement the value. Otherwise, wait until the value is greater than 0 and then decrement it.
  - V: increment the value of the semaphore

> By definition, the `P` and `V` operations of a semaphore are **atomic**.

Types of Semaphores

- **binary semaphore:** a semaphore with a single resource; behaves like a lock, but does not keep track of ownership
- **counting semaphore:** a semaphore with an arbitrary number of resources
- **barrier semaphore:** a semaphore used to force one thread to wait for others to complete; initial count is typically 0

Differences between a lock and a semaphore

- `V` does not have to follow `P`
- a semaphore can start with 0 resources; calls to `V` increment the count
- semaphores do **not** have owners

> `V` does not have to follow `P`. A semaphore can start with 0 resources. This forces a thread to wait until resources are produced before continuing.

Mutual Exclusion Using a Semaphore

```
volatile int total = 0;
struct semaphore *total_sem;
total_sem = sem_create("total mutex",1);  /* initial value is 1 */

void add() {                          void sub() {
   int i;                                int i;
   for (i=0; i<N; i++) {                 for (i=0; i<N; i++) {
     P(sem);                               P(sem);
       total++;                                total--;
     V(sem);                               V(sem);
   }                                     }
}                                     }
```


Producer/Consumer Synchronization with Bounded Buffer

- suppose we have threads (producers) that add items to a buffer and threads (consumers) that remove items from the buffer
- suppose we want to ensure that consumers do not consume if the buffer is empty - instead they must wait until the buffer has something in it
- similarly, suppose the buffer has a finite capacity (N), and we need to ensure that producers must wait if the buffer is full
- this requires synchronization between consumers and producers
- semaphores can provide the necessary synchronization

Bounded Buffer Producer/Consumer with Semaphores

```
struct semaphore *Items,*Spaces;
Items = sem_create("Buffer Items", 0);  /* initially = 0 */
Spaces = sem_create("Buffer Spaces", N);/* initially = N */

Producer's Pseudo-code:
  P(Spaces);
  add item to the buffer
  V(Items);

Consumer's Pseudo-code:
  P(Items);
  remove item from the buffer
  V(Spaces);
```

There is still a race condition in this code. What is it? How can you fix it?

Bounded Buffer Producer/Consumer with Semaphores

Discussion:

- consumers will wait for items to be produced
- producers will wait for spaces to be available
- producers and consumers can **both** access the bounded buffer at the **same** time
  - a third synchronization primitive is required to protect the buffer
  - a lock or binary semaphore is sufficient

Semaphore Implementation

```
P(struct semaphore * sem) {
    spinlock_acquire(&sem->sem_lock);
    while (sem->sem_count == 0) {
        wchan_lock(sem->sem_wchan);
        spinlock_release(&sem->sem_lock);
        wchan_sleep(sem->sem_wchan);
        spinlock_acquire(&sem->sem_lock);
    }
    sem->sem_count--;
    spinlock_release(&sem->sem_lock);
}

V(struct semaphore * sem) {
    spinlock_acquire(&sem->sem_lock);
    sem->count ++;
    wchan_wakeone(sem->sem_wchan);
    spinlock_release(&sem->sem_lock);
```

Notes:

- semaphores do not have owners
- the wait channel must be locked before releasing the spinlock.

Incorrect Semaphore Implementation Trace

Suppose `spinlock_release` preceeded `wchan_lock`, `count`$= 0$.

Thread 1	Thread 2
calls `P()`	...
...	
`count==0`	
`spinlock_release`	
context switch →	

The semaphore has no resources, Thread 1 will need to wait for a resource. But, before Thread 1 sleeps, there is a context switch.

Incorrect Semaphore Implementation Trace

Thread 1	Thread 2
calls P()	...
...	
count==0	
spinlock_release	
context switch →	
	V()
	...
	count++
	wchan_wakeone
	...
	← context switch

Thread 2 produces a resource by calling V. At this point, count= 1.

Incorrect Semaphore Implementation Trace

Thread 1	Thread 2
calls P()	...
...	
count==0	
spinlock_release	
context switch →	
	V()
	...
	count++
	wchan_wakeone
	...
	← context switch
wchan_lock	
wchan_sleep	

Thread 1 is now blocked on a semaphore that **HAS RESOURCES**.

Correct Semaphore Implementation Trace

Suppose `wchan_lock` preceeds `spinlock_release`, `count`= 0.

Thread 1	Thread 2
calls `P()`	...
...	
`count==0`	
`wchan_lock`	
`spinlock_release`	
context switch →	

The semaphore has no resources, Thread 1 will need to wait for a resource. But, before Thread 1 sleeps, there is a context switch.

Correct Semaphore Implementation Trace

Thread 1	Thread 2
calls `P()`	...
...	
`count==0`	
`wchan_lock`	
`spinlock_release`	
context switch →	
	calls `V()`
	`spinlock_acquire`
	`count++`
	`wchan_wakeone`
	← context switch

Thread 1 owns the wait channel, so Thread 2 will spin/block inside of `wchan_wakeone`.

Correct Semaphore Implementation Trace

Thread 1	Thread 2
calls `P()`	...
...	
`count==0`	
`wchan_lock`	
`spinlock_release`	
context switch →	
	calls `V()`
	`spinlock_acquire`
	`count++`
	`wchan_wakeone`
	← context switch
`wchan_sleep`	
context switch →	

Thread 1 is now sleeping on the semaphores wait channel. Thread 2 will wake.

Correct Semaphore Implementation Trace

Thread 1	Thread 2
calls `P()`	...
...	
`count==0`	
`wchan_lock`	
`spinlock_release`	
context switch →	
	calls `V()`
	`spinlock_acquire`
	`count++`
	`wchan_wakeone`
	← context switch
`wchan_sleep`	
context switch →	
	`spinlock_release`

Thread 2 wakes—and Thread 1 is moved from the wait channel to the ready queue, Thread 2 finishes execution of `V()`.

Condition Variables

- OS/161 supports another common synchronization primitive: *condition variables*
- each condition variable is intended to work together with a lock: condition variables are only used **from within the critical section that is protected by the lock**
- three operations are possible on a condition variable:

  wait: This causes the calling thread to block, and it releases the lock associated with the condition variable. Once the thread is unblocked it reacquires the lock.
  signal: If threads are blocked on the signaled condition variable, then one of those threads is unblocked.
  broadcast: Like signal, but unblocks all threads that are blocked on the condition variable.

Using Condition Variables

- Condition variables get their name because they allow threads to wait for arbitrary conditions to become true inside of a critical section.
- Normally, each condition variable corresponds to a particular condition that is of interest to an application. For example, in the bounded buffer producer/consumer example on the following slides, the two conditions are:
  - $count > 0$ (there are items in the buffer)
  - $count < N$ (there is free space in the buffer)
- when a condition is not true, a thread can `wait` on the corresponding condition variable until it becomes true
- when a thread detects that a condition is true, it uses `signal` or `broadcast` to notify any threads that may be waiting

> Note that signalling (or broadcasting to) a condition variable that has no waiters has *no effect*. Signals do not accumulate.

Condition Variable Example

```
int volatile numberOfGeese = 100;
lock geeseMutex;

int SafeToWalk() {
    lock_acquire(geeseMutex);
    if (numberOfGeese > 0) {
        ... wait? ...
    }
}
```

Thread must wait for `numberOfGeese > 0` before continuing. **BUT** thread owns `geeseMutex`, which protects access to `numberOfGeese`.

Condition Variable Example - Solution 1

```
int volatile numberOfGeese = 100;
lock geeseMutex;

int SafeToWalk() {
    lock_acquire(geeseMutex);
    while (numberOfGeese > 0) {
        lock_release(geeseMutex);
        lock_acquire(geeseMutex);
    }
}
```

Releasing and re-acquiring `geeseMutex` provides an opportunity for a context switch to occur and another thread might then acquire the lock and modify `numberOfGeese`. **BUT** the thread should not be waiting for the lock, it should be waiting for the condition to be true.

Condition Variable Example - Solution 2

```
int volatile numberOfGeese = 100;
lock geeseMutex;
cv zeroGeese;

int SafeToWalk() {
    lock_acquire(geeseMutex);
    while (numberOfGeese > 0) {
        cv_wait(zeroGeese, geeseMutex);
    }
}
```

Use a condition variable. `cv_wait` will handle releasing and re-acquring the lock passed in (`geeseMutex`, in this case), it also puts the calling thread onto the conditions wait channel to block. `cv_signal` and `cv_broadcast` are used to wake threads waiting on the cv.

Waiting on Condition Variables

- when a blocked thread is unblocked (by `signal` or `broadcast`), it reacquires the lock before returning from the `wait` call
- a thread is in the critical section when it calls `wait`, and it will be in the critical section when `wait` returns. However, in between the call and the return, while the caller is blocked, the caller is out of the critical section, and other threads may enter.

This describes Mesa-style condition variables, which are used in OS/161. There are alternative condition variable semantics (Hoare semantics), which differ from the semantics described here.

Bounded Buffer Producer Using Locks and Condition Variables

```
int volatile count = 0;  /* must initially be 0 */
struct lock *mutex;     /* for mutual exclusion */
struct cv *notfull, *notempty; /* condition variables */

/* Initialization Note: the lock and cv's must be created
 * using lock_create() and cv_create() before Produce()
 * and Consume() are called */

Produce(itemType item) {
  lock_acquire(mutex);
  while (count == N) {
     cv_wait(notfull, mutex); /* wait until buffer is not full */
  }
  add item to buffer (call list_append())
  count = count + 1;
  cv_signal(notempty, mutex); /* signal that buffer is not empty */
  lock_release(mutex);
}
```


Bounded Buffer Consumer Using Locks and Condition Variables

```
itemType Consume() {
  lock_acquire(mutex);
  while (count == 0) {
     cv_wait(notempty, mutex); /* wait until buffer is not emtpy */
  }
  remove item from buffer (call list_remove_front())
  count = count - 1;
  cv_signal(notfull, mutex); /* signal that buffer is not full */
  lock_release(mutex);
  return(item);
}
```

Both `Produce()` and `Consume()` call `cv_wait()` inside of a `while` loop. Why?

Volatile and Other Sources of Race Conditions

- notice throughout these slides that shared variables were declared `volatile`
- race conditions can occur for reasons **other** beyond the programmers control, specifically:
  - **compiler**
  - **CPU**

  both can introduce race conditions
- in both cases, compiler and CPU introduce race conditions due to optimizations
- **memory models** describe how thread access to memory in shared regions behave
  - a memory model tells the compiler and CPU which optimizations can be performed

Volatile

- it is faster to access values from a register, than from memory
- compilers optimize for this; storing values in registers for as long as possible
- consider:

```
int sharedTotal = 0;
int FuncA() {
    ... code that uses sharedTotal ...
}
int FuncB() {
    ... code that uses sharedTotal ...
}
```

  if the compiler optimizes `sharedTotal` into register `R3` in `FuncA`, and register `R8` in `FuncB`, which register has the correct value for `sharedTotal`?
- `volatile` disables this optimization, forcing a value to be loaded/stored to memory with each use, it also prevents the compiler from re-ordering loads and stores for that variable
- shared variables should be declared `volatile` in your code

Other Language and Instruction Level Instructions

- many languages support multi-threading with memory models and language-level synchronization functions (i.e., locks)
  - compiler is aware of critical sections via language-level synchronization functions; does not perform optimizations which cause race conditions
  - the version of C used by OS/161 does not support this
- the CPU also has a memory model as it also re-orders loads and stores to improve performance
  - modern architectures provide barrier or fence instructions to disable and reenable these CPU-level optimizations to prevent race conditions at this level
  - the MIPS R3000 CPU used in this course does not have or require these instructions

Deadlocks

Consider the following pseudocode:

```
lock lockA, lockB;
int FuncA() {                  int FuncB() {
    lock_acquire(lockA)            lock_acquire(lockB)
    lock_acquire(lockB)            lock_acquire(lockA)
    ...                              ...
    lock_release(lockA)            lock_release(lockB)
    lock_release(lockB)            lock_release(lockA)
}
```

- What if:
  - Thread 1 executes `lock_acquire(lockA)`
  - Thread 2 executes `lock_acquire(lockB)`
  - Thread 1 executes `lock_acquire(lockB)`
  - Thread 2 executes `lock_acquire(lockA)`

Deadlocks

Consider the following pseudocode:

```
lock lockA, lockB;
int FuncA() {                      int FuncB() {
    lock_acquire(lockA)                lock_acquire(lockB)
    lock_acquire(lockB)                lock_acquire(lockA)
    ...                                  ...
    lock_release(lockA)                lock_release(lockB)
    lock_release(lockB)                lock_release(lockA)
}
```

- What if:
  - Thread 1 executes `lock_acquire(lockA)`
  - Thread 2 executes `lock_acquire(lockB)`
  - Thread 1 executes `lock_acquire(lockB)`
  - Thread 2 executes `lock_acquire(lockA)`

> Thread 1 and 2 are **deadlocked**. Neither thread can make progress. Waiting will not resolve the deadlock, the threads a permanently stuck.

Two Techniques for Deadlock Prevention

No Hold and Wait: prevent a thread from requesting resources if it currently has resources allocated to it. A thread may hold several resources, but to do so it must make a single request for all of them.

Resource Ordering: Order (e.g., number) the resource types, and require that each thread acquire resources in increasing resource type order. That is, a thread may make no requests for resources of type less than or equal to i if it is holding resources of type i.