A process is an instance of a running program
- A thread is an execution context
- Process can have one or more threads
- Threads share address space (code, data, heap), open files
- Threads have their own stack and register state

POSIX Thread APIs:
- pthread_create() - Creates a new thread
- pthread_exit() - Destroys current thread
- pthread_join() - Waits for thread to exit
int total = 0;

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

void sub() {
    int i;
    for (i=0; i<N; i++) {
        total--;
    }
}
Critical Sections: Assembly Pseudocode

```c
int total = 0;

void add() {
    int i;
    /* r8 := &total */
    for (i=0; i<N; i++) {
        lw r9, 0(r8)
        add r9, 1
        sw r9, 0(r8)
    }
}

void sub() {
    int i;
    for (i=0; i<N; i++) {
        lw r9, 0(r8)
        sub r9, 1
        sw r9, 0(r8)
    }
}
```
Thread #1
lw r9, 0(r8)
add r9, 1
sw r9, 0(r8)

Thread #2
lw r9, 0(r8)
sub r9, 1
sw r9, 0(r8)

• Increment completes, then decrements
• Result: total = zero.pnum

Critical Section: Schedule 1
Critical Section: Schedule 1

Thread #1
lw r9, 0(r8)
add r9, 1
sw r9, 0(r8)

Thread #2
lw r9, 0(r8)
sub r9, 1
sw r9, 0(r8)

- Increment completes, then decrements
- Result: total = 0
Critical Section: Schedule 2

Thread #1
lw r9, 0(r8)
add r9, 1
sw r9, 0(r8)

Thread #2
lw r9, 0(r8)
sub r9, 1
sw r9, 0(r8)
Critical Section: Schedule 2

Thread #1
lw r9, 0(r8)
add r9, 1
sw r9, 0(r8)

Thread #2
lw r9, 0(r8)
sub r9, 1
sw r9, 0(r8)

- Both load zero, then stores clobber one another
- Result: total = -1
Thread #1
lw r9, 0(r8)
add r9, 1
sw r9, 0(r8)

Thread #2
lw r9, 0(r8)
sub r9, 1
sw r9, 0(r8)

Both load zero, then stores clobber one another.
Result: total = /one.pnum/six.pnum /three.pnum/two.pnum
Critical Section: Schedule 3

Thread #1
lw r9, 0(r8)
add r9, 1
sw r9, 0(r8)

Thread #2
lw r9, 0(r8)
sub r9, 1
sw r9, 0(r8)

- Both load zero, then stores clobber one another
- Result: total = 1
Need for Synchronization

- Problem: Data Races occur when no synchronization is provided
- Options:
  - Atomic Instructions: atomically modify value
  - Locks: prevent other code from running concurrently
- ... it gets worse!
int flag1 = 0, flag2 = 0;

void p1 (void *ignored) {
    flag1 = 1;
    if (!flag2) { critical_section_1 (); }
}

void p2 (void *ignored) {
    flag2 = 1;
    if (!flag1) { critical_section_2 (); }
}

int main () {
    tid id = thread_create (p1, NULL);
    p2 ();
    thread_join (id);
}

Q: Can both critical sections run?
int data = 0, ready = 0;

void p1 (void *ignored) {
    data = 2000;
    ready = 1;
}

void p2 (void *ignored) {
    while (!ready)
    {
        use (data);
    }
}

int main () { ... }

Q: Can use be called with value 0?
int a = 0, b = 0;

void p1 (void *ignored) {
    a = 1;
}

void p2 (void *ignored) {
    if (a == 1)
        b = 1;
}

void p3 (void *ignored) {
    if (b == 1)
        use (a);
}

Q: If p1–3 run concurrently, can use be called with value 0?
Correct answers

- Program A: I don't know
- Program B: I don't know
- Program C: I don't know
- Why don't we know?
  - It depends on what machine you use
  - If a system provides sequential consistency, then answers all No
  - But not all hardware provides sequential consistency

Note: Examples, other content from /one.pnum/one.pnum / /three.pnum/two.pnum
Correct answers

- Program A: I don’t know
Correct answers

- Program A: I don’t know
- Program B: I don’t know

- Why don’t we know?
- It depends on what machine you use
- If a system provides sequential consistency, then answers all No
- But not all hardware provides sequential consistency

Note: Examples, other content from one.pnum/one.pnum / /three.pnum/two.pnum
Correct answers

- Program A: I don’t know
- Program B: I don’t know
- Program C: I don’t know

Why don’t we know?
- It depends on what machine you use
- If a system provides *sequential consistency*, then answers all No
- But not all hardware provides sequential consistency

Note: Examples, other content from [Adve & Gharachorloo]
**Definition**

*Sequential consistency*: The result of execution is as if all operations were executed in some sequential order, and the operations of each processor occurred in the order specified by the program.

– *Lamport*

- Boils down to two requirements:
  1. Maintaining *program order* on individual processors
  2. Ensuring *write atomicity*

- Without SC (Sequential Consistency), multiple CPUs can be “worse”—i.e., less intuitive—than preemptive threads
  - Result may not correspond to *any* instruction interleaving on 1 CPU

- Why doesn’t all hardware support sequential consistency?
SC thwarts hardware optimizations

- Complicates write buffers
  - E.g., read flag \( n \) before flag \( (2 - n) \) written through in Program A
- Can’t re-order overlapping write operations
  - Concurrent writes to different memory modules
  - Coalescing writes to same cache line
- Complicates non-blocking reads
  - E.g., speculatively prefetch data in Program B
- Makes cache coherence more expensive
  - Must delay write completion until invalidation/update (Program B)
  - Can’t allow overlapping updates if no globally visible order (Program C)
SC thwarts compiler optimizations

- **Code motion**
- **Caching value in register**
  - Collapse multiple loads/stores of same address into one operation
- **Common subexpression elimination**
  - Could cause memory location to be read fewer times
- **Loop blocking**
  - Re-arrange loops for better cache performance
- **Software pipelining**
  - Move instructions across iterations of a loop to overlap instruction latency with branch cost
Memory Model

- **Sequential Consistency**: statements execute in program order
- Compilers/HW reorder loads/stores for performance
- **Language-level Memory Model**
  - C/Java: sequential consistency for race free programs
  - Compiler must be aware of synchronization
  - Language provides barriers and atomics
- **Processor-level Memory Model**
  - TSO: Total Store Order - X86, SPARC (default)
  - PSO: Partial Store Order - SPARC PSO
  - RMO: Relaxed Memory Order - Alpha, POWER, ARM, PA-RISC, SPARC RMO, x86 OOS
  - Even more nuanced variations between architectures!
TSO and PSO and RMO, Oh My!

Example from SPARC V9 Architecture Manual:

P1:  st  #1,[A]   // 0  
     st  #1,[B]   // 1

P2:  ld  [A], %r1   // 2  
     ld  [B], %r2   // 3

P3:  ld  [B], %r1   // 4  
     ld  [A], %r2   // 5

- TSO: 0 occur before 1, 2 before 3, 4 before 5
- PSO: stores can occur in any order
- RMO: stores/loads can occur in any order e.g. 5,3,0,2,1,4
- Barrier instructions allow one to limit reorders
- SPARC: MEMBAR instruction
  - MEMBAR #StoreStore,#StoreLoad,#LoadStore,#LoadLoad
  - MEMBAR #Sync - Everything
• x86 uses TSO (with a few exceptions)

• **lock prefix makes a memory instruction atomic**
  - Usually locks bus for duration of instruction (expensive!)
  - Can avoid locking if memory already exclusively cached
  - All lock instructions totally ordered
  - Other memory instructions cannot be re-ordered with locked ones

• **xchg instruction is always locked (even without prefix)**

• Special barrier (or “fence”) instructions can prevent re-ordering
  - **lfence** – can’t be reordered with reads (or later writes)
  - **sfence** – can’t be reordered with writes
    (e.g., use after non-temporal stores, before setting a ready flag)
  - **mfence** – can’t be reordered with reads or writes
What about a single-instruction add?
- E.g., i386 allows single instruction `addl $1, _count`
- So implement `count++/--` with one instruction
- Now are we safe?

Not atomic on multiprocessor! (operation ≠ instruction)
- Will experience exact same race condition
- Can potentially make atomic with `lock` prefix
- But `lock` potentially very expensive
- Compiler won't generate it, assumes you don't want penalty
x86 atomicity: data races

- What about a single-instruction add?
  - E.g., i386 allows single instruction `addl $1, _count`
  - So implement `count++/--` with one instruction
  - Now are we safe?

- Not atomic on multiprocessor! (operation ≠ instruction)
  - Will experience exact same race condition
  - Can potentially make atomic with `lock` prefix
  - But `lock` potentially very expensive
  - Compiler won’t generate it, assumes you don’t want penalty

- Need solution to critical section problem
  - Place `count++` and `count--` in critical section
  - Protect critical sections from concurrent execution
• **x86:** `cmpxchg` Compare and Exchange

• **SPARC Architecture: Compare and Swap**
  - `cas addr, r1, r2`
  - if (*addr == r1) then swap *addr and r2

• **Some hardware supports double compare and swap**
  - Useful for software transactional memory, atomic doubly linked lists
**Desired properties of solution**

- **Mutual Exclusion**
  - Only one thread can be in critical section at a time

- **Progress**
  - Say no process currently in critical section (C.S.)
  - One of the processes trying to enter will eventually get in

- **Bounded waiting**
  - Once a thread $T$ starts trying to enter the critical section, there is a bound on the number of times other threads get in

- **Note progress vs. bounded waiting**
  - If no thread can enter C.S., don’t have progress
  - If thread $A$ waiting to enter C.S. while $B$ repeatedly leaves and re-enters C.S. *ad infinitum*, don’t have bounded waiting
Peterson’s solution

- Still assuming sequential consistency
- Assume two threads, \(T_0\) and \(T_1\)
- Variables
  - \texttt{int not\_turn}; // not this thread’s turn to enter C.S.
  - \texttt{bool wants[2]; // wants[i] indicates if \(T_i\) wants to enter C.S.}
- Code:

```c
for (;;) { /* assume i is thread number (0 or 1) */
    wants[i] = true;
    not_turn = i;
    while (wants[1-i] && not_turn == i)
        /* other thread wants in and not our turn, so loop */;
    Critical\_section ()
    wants[i] = false;
    Remainder\_section ();
}
```
Does Peterson’s solution work?

```c
for (;;) {  // code in thread i */
  wants[i] = true;
  not_turn = i;
  while (wants[1-i] && not_turn == i)
    /* other thread wants in and not our turn, so loop */;
    Critical_section ();
  wants[i] = false;
  Remainder_section ();
}
```

- **Mutual exclusion** – can’t both be in C.S.
  - Would mean `wants[0] == wants[1] == true`, so `not_turn` would have blocked one thread from C.S.

- **Progress** – given demand, one thread can always enter C.S.
  - If `T_{1-i}` doesn’t want C.S., `wants[1-i] == false`, so `T_i` won’t loop.
  - If both threads want in, one thread is not the `not_turn` thread.

- **Bounded waiting** – similar argument to progress
  - If `T_i` wants lock and `T_{1-i}` tries to re-enter, `T_{1-i}` will set `not_turn = 1 - i`, allowing `T_i` in.
Mutexes

- Peterson expensive, only works for 2 processes
  - Can generalize to $n$, but for some fixed $n$

- Must adapt to machine memory model if not SC
  - If you need machine-specific barriers anyway, might as well take advantage of other instructions helpful for synchronization

- Want to insulate programmer from implementing synchronization primitives

- Thread packages typically provide mutexes:
  
  ```c
  void mutex_init (mutex_t *m, ...);
  void mutex_lock (mutex_t *m);
  int mutex_trylock (mutex_t *m);
  void mutex_unlock (mutex_t *m);
  ```
  - Only one thread acquires $m$ at a time, others wait
typedef struct Spinlock {
    alignas(CACHELINE) _Atomic(uint64_t) lck;
} Mutex;

void Spinlock_Init(Spinlock *m) {
    atomic_store(&m->lck, 0);
}

void Spinlock_Lock(Spinlock *m) {
    while (atomic_exchange(&m->lck, 1) == 1)
    {
    }
}

void Spinlock_Unlock(Spinlock *m) {
    atomic_store(&m->lck, 0);
}
Where’s the barriers?

// Implicit Sequential Consistency
C atomic_load(const volatile A* obj);
// Explicit Consistency
C atomic_load_explicit(const volatile A* obj, memory_order order);
// Barrier or Fence
void atomic_thread_fence(memory_order order);

enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
• Use assembly routines for compiler barriers:
  - `asm("
    :::
    "memory")`;
  - Compiler will not reorder loads/stores nor cache values

• **Use** `volatile` **keyword**
  - `volatile` originally meant for accessing device memory
  - Loads/stores to `volatile` variables will not be reordered with respect to other `volatile` operations
  - Use of `volatile` is deprecated on modern compilers
  - `volatile` operations are not atomics!
  - Use `volatile` with inline assembly to use atomics
Spinlocks in OS/161

```
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);
```

- **Spinlocks based on using** `spinlock_data_testandset`
- **Spinlocks don’t yield CPU, i.e., they spin**
- **Raise the interrupt level to prevent preemption**
/* return value 0 indicates lock was acquired */
spinlock_data_testandset(volatile spinlock_data_t *sd) {
    spinlock_data_t x,y;
y = 1;
    __asm volatile(
        ".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));
    if (y == 0) { return 1; }
    return x;
}
MIPS Atomics: Continued

- **Load Linked** `ll`: Loads a value and monitors memory for changes
- **Store Conditional** `sc`: Stores if memory didn’t change
  - `sc` can fail for multiple reasons
    - Value from `ll` was modified by another processor
    - An interrupt preempted the thread between `ll` and `sc`
- **Otherwise** `sc` will succeed returning `1`
- **On failure we can retry the operation**
- **Powerful primitives**
  - Can implement any read-modify-write operation
  - For example, atomic add or increment
  - Some architectures are implemented this way internally
Mutex Locks in OS/161

- Provide mutual exclusion like spinlocks
- Yield the CPU when waiting on the lock
- Mutex locks deal with priority inversion
  - Problem: Low priority thread sleeps while holding lock then a high priority thread wants the lock
  - Mutex locks typically boost the priority of the lower thread to unblock the higher thread

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

lock_aquire(mylock);
/* critical section */
lock_release(mylock);
```
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
- Many different wait channels holding threads sleeping for different reasons
- Similar primitives exist in most operating systems

- `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);`
  - Unblocks all threads sleeping on the wait channel
- `void wchan_wakeone(struct wchan *wc);`
  - Unblocks one thread sleeping on the wait channel
- `void wchan_lock(struct wchan *wc);`
  - Prevent operations on the wait channel
  - More on this later