Review: Thread package API

- `tid create (void (*fn) (void *), void *arg);`
  - Create a new thread, run fn with arg
- `void exit ();`
- `void join (tid thread);`
- **Threads may run simultaneously**
  - E.g., `create (p1, NULL); create (p2, NULL);`
  - p1 and p2 run interleaved or concurrently
- **Threads have same address space**
  - Usually share some memory locations between threads
  - Otherwise, could just have used processes
- **Sharing + Concurrency can lead to unexpected results**
Program A

```c
int flag1 = 0, flag2 = 0;

void p1 (void *ignored) {
    flag1 = 1;
    if (!flag2) { /* critical section */ }
}

void p2 (void *ignored) {
    flag2 = 1;
    if (!flag1) { /* critical section */ }
}

• Can both critical sections run?
```
Program B

```c
int data = 0, ready = 0;

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

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

• Can use be called with value 0?
```
Program C

```c
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);
}

• 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?
  - It depends on your hardware
  - If it provides *sequential consistency*, then answers all No
  - But not all hardware provides sequential consistency

- [BTW, examples and some other slide content from excellent Tech Report by Adve & Gharachorloo]
Sequential Consistency

• *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*

• Why doesn’t all hardware support sequential consistency?
S.C. thwarts hardware optimizations

- **Write buffers**
  - E.g., read flag\(n\) before flag\((2 - n)\) written through in Program A

- **Overlapping write operations can be reordered**
  - Concurrent writes to different memory modules
  - Coalescing writes to same cache line

- **Non-blocking reads**
  - E.g., speculatively prefetch data in Program B

- **Cache coherence**
  - Write completion only after invalidation/update (Program B)
  - Can’t have overlapping updates (Program C)
S.C. thwarts compiler optimizations

- Code motion
- Caching value in register
  - E.g., ready flag in Program B
- Common subexpression elimination
- Loop blocking
- Software pipelining
Assuming sequential consistency

• Let’s for now say we have sequential consistency
  - Apologies for starting out with trick questions
  - Just don’t forget to check the memory model in real life

• Later will see alpha which doesn’t have S.C.

• Example concurrent code: Producer/Consumer
  - buffer stores BUFFER_SIZE items
  - count is number of used slots
  - out is next empty buffer slot to fill (if any)
  - in is oldest filled slot to consume (if any)
void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */
        while (count == BUFFER_SIZE)
            ; // do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
    }
}

void consumer (void *ignored) {
    for (;;) {
        while (count == 0)
            ; // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--; /* consume the item in nextConsumed */
    }
}

• What can go wrong here?
Data races

- count may have wrong value
- Possible implementation of count++ and count--:
  - register ← count
  - register ← register + 1
  - count ← register
- Possible execution (count one less than correct):
  - register ← count
  - register ← register + 1
  - register ← count
  - register ← register − 1
  - count ← register
  - count ← register
Data races (continued)

• 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?
Data races (continued)

- **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!**
  - Will experience exact same race condition
  - Can potentially make atomic with `lock` prefix
  - But `lock` very expensive
  - Compiler won’t generate it, assumes you don’t want penalty

- **Note that without S.C., even reads can be dangerous**

- **Need solution to critical section problem**
  - Place `count++` and `count--` in critical section
Desired solution

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

- **Progress**
  - Say no process currently in critical section (C.S.)
  - Threads trying to enter C.S. can’t be blocked by those not trying
  - One of the processes trying to enter will eventually get in

- **Bounded waiting**
  - After thread $T$ starts trying to enter critical section
  - Bound on # times other threads get in
Peterson’s solution

- Still assuming sequential consistency
- Assume two threads, $T_0$ and $T_1$
- Variables
  - int turn – whose turn to enter C.S.
- Code:

```c
for (;;) { /* code in thread i */
  flag[i] = true;
  turn = 1 - i;
  while (flag[1-i] && turn == 1-i)
    ;
  /* Critical Section */
  flag[i] = false;
  /* Remainder Section */
}
```
Does Peterson’s solution work?

```c
for (; ; ) { /* code in thread i */
    flag[i] = true;
    turn = 1 - i;
    while (flag[1-i] && turn == 1-i)
        ;
    /* Critical Section */
    flag[i] = false;
    /* Remainder Section */
}
```

- **Mutual exclusion** – can’t both be in C.S.
  - Would mean flag[0] == flag[1] == true, so turn would have allowed only one thread into C.S.

- **Progress** – If $T_0$ not in C.S., can’t block $T_1$
  - Means flag[0] == false, so $T_1$ won’t loop
  - Similarly, if $T_1$ not in C.S. can’t block $T_0$

- **Bounded waiting** – similar argument to progress
Mutexes

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

• Typically want to insulate programmer from implementing synchronization primitives

• Thread packages typically provide mutexes:
  - void lock (mutex_t m);
  - void unlock (mutex_t m);
    - Only one thread acquires $m$ at a time, others wait
    - All global data must be protected by a mutex!

• OS kernels also need some synchronization
  - May or may not look like mutexes
Improved producer

mutex_t mutex;

void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */
        lock (mutex);
        while (count == BUFFER_SIZE) {
            unlock (mutex); // <--- Why?
            yield ();
            lock (mutex);
        }

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        unlock (mutex);
    }
}
void consumer (void *ignored) {
    for (;;) {
        lock (mutex);
        while (count == 0) {
            unlock (mutex);
            yield ();
            lock (mutex);
        }
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;     
        unlock (mutex);
        /* consume the item in nextConsumed */
    }
}
Condition variables

- Busy-waiting in application is a bad idea
  - Thread consumes CPU even when can’t make progress
  - Unnecessarily slows other threads and even processes

- Better to inform scheduler of which threads can run

- Typically done with condition variables

  - `void wait (mutex_t m, cond_t c);`
    - Atomically unlock `m` and sleep until `c` signaled
    - Then reacquire `m` and resume executing

  - `void signal (cond_t c);`
  - `void broadcast (cond_t c);`
    - Wake one/all users waiting on `c`
Improved producer

mutex_t mutex;
cond_t nonempty, nonfull;

void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */

        lock (mutex);
        while (count == BUFFER_SIZE)
            wait (mutex, nonfull);

        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        count++;
        signal (nonempty);
        unlock (mutex);
    }
}
void consumer (void *ignored) {
    for (;;) {
        lock (mutex);
        while (count == 0)
            wait (mutex, nonempty);

        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--; 
        signal (nonfull);
        unlock (mutex);

        /* consume the item in nextConsumed */
    }
}

Improved consumer
Condition variables (continued)

- Why must `wait` atomically release mutex & sleep?
- Why not separate mutexes and condition variables?

```c
while (count == BUFFER_SIZE) {
    unlock (mutex);
    wait (nonfull);
    lock (mutex);
}
```
Condition variables (continued)

- Why must \texttt{wait} atomically release mutex & sleep?
- Why not separate mutexes and condition variables?

```c
while (count == BUFFER_SIZE) {
    unlock (mutex);
    wait (nonfull);
    lock (mutex);
}
```

- Can end up stuck waiting when bad interleaving

```
PRODUCER
while (count == BUFFER\_SIZE);
unlock (mutex);

CONSUMER
lock (mutex);
... count--; signal (nonfull);

wait (nonfull);
```
Implementing synchronization

- **User-visible mutex is straight-forward data structure**

  ```c
  struct mutex_t {
    bool is_locked;    /* true if locked */
    thread_id_t owner; /* thread holding lock if locked */
    thread_list_t waiters; /* threads waiting to lock */
    lower_level_lock_t lk;
  }
  ```

- **Need lower-level lock lk for mutual exclusion**
  - Otherwise, would have data races on mutex_t itself
  - E.g., two threads manipulating waiters list corrupt list

- **How to implement lower_level_lock_t?**
  - Could use Peterson’s algorithm, but typically not
  - Instead, use hardware support for synchronization
One approach: Disable interrupts

- **Does not work on multiprocessors**
  - But often most efficient solution for uniprocessors

- **For user-level threads, can use one kernel thread**
  - Context switch on timer interrupts (setitimer)
  - In critical section: Set “do not interrupt” (DNI) bit
  - If timer interrupt arrives, set “interrupted” bit
  - Manipulate protected low-level data structure
  - Clear DNI bit
  - If interrupted bit set, yield

- **In kernel, can do what old UNIX kernels did**
  - Non-preemptive threads, so count++ etc. not data race
  - *Except* memory touched in both top-half thread & interrupt
UNIX Synchronization 1

- Interface designed before multiprocessors common
- Top half kernel procedures can mask interrupts
  
  ```c
  int x = splhigh();
  /* ... */
  splx(x);
  ```

- splhigh disables all interrupts, but also splnet, splbio, splsoftnet, ...
  - C.f., Pintos intr_disable / intr_set_level

- Masking interrupts in hardware can be expensive
  - Optimistic implementation – set mask flag on splhigh, check interrupted flag on splx
UNIX Synchronization 2

• Need to relinquish CPU when waiting for events
  - Disk read, network packet arrival, pipe write, signal, etc.
• int tsleep(void *ident, int priority, ...);
  - Switches to another process
  - ident is arbitrary pointer—e.g., buffer address
  - priority is priority at which to run when woken up
  - PCATCH, if ORed into priority, means wake up on signal
  - Returns 0 if awakened, or ERESTART/EINTR on signal

• int wakeup(void *ident);
  - Awakens all processes sleeping on ident
  - Restores SPL to value when they went to sleep
    (so fine to sleep at splhigh)
For MP, need hardware support

- Need atomic read-write or read-modify-write:
  
  **Example:**
  ```c
  int test_and_set (int *lockp);
  
  - Sets *lockp = 1 and returns old value
  ```

- **Now can implement spinlocks:**
  ```c
  #define lock(lockp) while (test_and_set (lockp))
  #define unlock(lockp) *lockp = 0
  ```

- **Spinlocks used at low level to implement mutexes**
  - Using spinlocks directly would waste CPU time, especially if thread holding lock doesn’t have a CPU
  - Critical section in mutex implementation very short, so OK

- **But gratuitous context switch has cost**
  - On MP, sometimes good to spin for a bit, then yield
Synchronization on x86

- Test-and-set only one possible hardware approach
- x86 `xchg` instruction, exchanges reg with mem
  - Can just use to implement test-and-set

  ```
  _test_and_set:
    movl 8(%esp), %edx
    movl $1, %eax
    xchg %eax, (%edx)
    ret
  ```

- CPU locks memory system around read and write
  - I.e., `xchg` always acts like it has lock prefix
  - Prevents other uses of the bus (e.g., DMA)

- Operates at memory bus speed, not CPU speed
  - Much slower than cached read/buffered write
Synchronization on alpha

- Another approach: load locked, store conditional

- \texttt{ldl\_l} – load locked
  \texttt{stl\_c} – store but sets reg to 0 if not atomic w. \texttt{ldl\_l}

\texttt{test\_and\_set:}
  \begin{verbatim}
  ldq_l v0, 0(a0)
bne v0, 1f
  addq zero, 1, v0
  stq_c v0, 0(a0)
  beq v0, _test_and_set
  mb
  addq zero, zero, v0
  
  1:
  ret zero, (ra), 1
  \end{verbatim}

- Note: Alpha does not have sequential consistency
  - Yet want all processors to think that memory accesses happened after acquiring lock, before releasing
  - \texttt{mb}, memory barrier instruction, ensures this
Other thread package features

- Alerts – cause exception in a thread
- Trylock – don’t block if can’t acquire mutex
- Timedwait – timeout on condition variable
- Shared locks – concurrent read accesses to data
- Thread priorities – control scheduling policy
- Thread-specific global data
- **Different synchronization primitives**
  - Monitors
  - Semaphores
  - Reader/writer (shared) locks
Monitors

• **Programming language construct**
  - Possibly less error prone than raw mutexes, but less flexible too
  - Basically a class where only one procedure executes at a time
    ```
    monitor monitor-name
    {
        // shared variable declarations
        procedure P1 (...) { ... }
    
    ...
    procedure Pn (...) { ... }

    Initialization code (...) { ... }
    }
    ```

• **Can implement mutex w. monitor or vice versa**
  - But monitor alone doesn’t give you condition variables
  - Need some other way to interact w. scheduler
  - Use *conditions* kind of like condition variables
Monitor implementation

- Queue of threads waiting to get in
  - Might be protected by spinlock
- Queues associated with conditions
Semaphores

• A *Semaphore* is initialized with an integer $N$

• Provides two functions:
  - `wait (S)` (originally called `P`)
  - `signal (S)` (originally called `V`)

• **Guarantees** `wait` will return only $N$ more times than `signal` called
  - Example: If $N == 1$, then semaphore is a mutex

• *Semaphores* allow elegant solutions to some problems
Semaphore producer/consumer

- **Semaphore mutex initialized to 1**
  - To protect buffer, in, out...

- **Semaphore full initialized to 0**
  - To block consumer when buffer empty

- **Semaphore empty initialized to N**
  - To block producer when queue full
void producer (void *ignored) {
    for (;;) {
        /* produce an item and put in nextProduced */
        wait (empty);
        wait (mutex);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        signal (mutex);
        signal (full);
    }
}

void consumer (void *ignored) {
    for (;;) {
        wait (full);
        wait (mutex);
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        signal (mutex);
        signal (empty);
        /* consume the item in nextConsumed */
    }
}
Readers-Writers Problem

- Multiple threads may access data
  - *Readers* – will only observe, not modify data
  - *Writers* – will change the data

- **Goal: allow multiple readers or one single writer**
  - Thus, lock can be *shared* amongst concurrent readers

- **Can implement with other primitives**
  - Keep integer i – # or readers or -1 if held by writer
Implementing shared locks

```c
struct sharedlk {
    int i;
    mutex_t m;
    cond_t c;
};

void AcquireExclusive (sharedlk *sl) {
    lock (sl->m);
    while (sl->i) { wait (sl->m, sl->c); }
    sl->i = -1;
    unlock (sl->m);
}

void AcquireShared (sharedlk *sl) {
    lock (sl->m);
    while (sl->i < 0) { wait (sl->m, sl->c); }
    sl->i++;
    unlock (sl->m);
}
```
shared locks (continued)

void ReleaseShared (sharedlk *sl) {
    lock (sl->m);
    if (!--sl->i) signal (sl->c);
    unlock (sl->m);
}

void ReleaseExclusive (sharedlk *sl) {
    lock (sl->m);
    sl->i = 0;
    broadcast (sl->c);
    unlock (sl->m);
}

• **Note: Must deal with starvation**