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);`

- The execution of multiple threads is interleaved

- Can have **non-preemptive threads**:
  - One thread executes exclusively until it makes a blocking call.

- Or **preemptive threads**:
  - May switch to another thread between any two instructions.

- Using multiple CPUs is inherently preemptive
  - Even if you don’t take $CPU_0$ away from thread $T$, another thread on $CPU_1$ can execute between any two instructions of $T$. 
Kernel threads

- Can implement thread create as system call
- Start with process abstraction in kernel
- Strip out unnecessary features
  - Same address space, file table, etc.
  - rfork/clone actually allow individual control
- Faster than a process, but still very heavy weight
Why kernel threads suck

• Every thread operation must go through kernel
  - create, exit, join, synchronize, or switch for any reason
  - On Athlon 3400+: syscall takes 359 cycles, fn call 6 cycles
  - Result: threads 10x-30x slower when implemented in kernel

• One-size fits all thread implementation
  - Kernel threads must please all people
  - Maybe pay for fancy features (priority, etc.) you don’t need

• General heavy-weight memory requirements
  - E.g., requires a fixed-size stack within kernel
  - Other data structures designed for heavier-weight processes
User threads

- An alternative: implement in user-level library
  - One kernel thread per process
  - create, exit, etc., just library functions
Implementing user-level threads

- Allocate a new stack for each thread
- Keep a queue of runnable threads
- Replace networking system calls (read/write/etc.)
  - If operation would block, switch and run a different thread
- Schedule periodic timer signal (setitimer)
  - Switch to another thread on timer signals (preemption)
- Multi-threaded web server example
  - Read data from connecting web browser
  - No data? Replace read schedules another thread
  - On timer or idle check which connections have new data

- How to switch threads?
Background: calling conventions

- \( sp \) register always base of stack
  - frame pointer (\( fp \)) is old \( sp \)
- Local vars in stack & registers
  - By convention, registers divided into caller- and callee-saved
- Function arguments go in callee-saved regs and on stack

\[
\begin{array}{c}
\text{Call arguments} \\
\hline
\text{return addr} \\
\hline
\text{old frame ptr} \\
\hline
\text{callee-saved registers} \\
\hline
\text{Local vars and temps} \\
\end{array}
\]
Background: procedure calls

save active caller registers

call foo

saves used callee registers

...do stuff...

restores callee registers

jumps back to pc

restore caller regs

• Some state saved on stack
  - Return address, caller-saved registers

• Some state saved not saved
  - Callee-saved regs, global variables, stack pointer
Threads vs. procedures

• Threads may resume out of order:
  - Cannot use LIFO stack to save state
  - General solution: one stack per thread

• Threads switch less often:
  - Don’t partition registers (why?)

• Threads involuntarily interrupted:
  - Synchronous: procedure call can use compiler to save state
  - Asynchronous: thread switch code saves all registers

• More than one than one thread can run
  - Scheduling: what to run next and on which CPU?
  - Procedure call scheduling obvious: run called procedure
Example user threads implementation

- Per-thread state in thread control block structure

```c
typedef struct tcb {
    unsigned long md.esp;    /* Stack pointer of thread */
    char *t_stack;            /* Bottom of thread’s stack */
    /* ... */
};
```

- Machine-dependent thread-switch function:
  - void thread_md_switch (tcb *current, tcb *next);

- Machine-dependent thread initialization function:
  - void thread_md_init (tcb *t,
                        void (*fn) (void *), void *arg);
i386 thread_md_switch

pushl %ebp; movl %esp,%ebp  # Save frame pointer
pushl %ebx; pushl %esi; pushl %edi  # Save callee-saved regs
movl 8(%ebp),%edx  # %edx = thread_current
movl 12(%ebp),%eax  # %eax = thread_next
movl %esp,(%edx)  # %edx->md Esp = %esp
movl (%eax),%esp  # %esp = %eax->md Esp

popl %edi; popl %esi; popl %ebx  # Restore callee saved regs
popl %ebp  # Restore frame pointer
ret  # Resume execution

• This is literally switch code from simple thread lib
  - Nothing magic happens here

• You will see very similar code in Pintos switch.S
Why user threads suck

- Can’t take advantage of multiprocessors
- A blocking system call blocks all threads
  - Can replace read to handle network connections
  - But usually OSes don’t let you do this for disk
  - So uncached disk read blocks all processes
- A page fault blocks all threads
- Hard to run as many threads as CPUs
  - Don’t know how many CPUs available
  - Don’t know which threads are blocked
- Possible deadlock if one thread blocks on another
  - May block entire process and make no progress
  - [More on deadlock next week.]
Uthreads on kthreads

- User threads implemented on kernel threads
  - Multiple kernel threads per process
  - create, exit, etc., still library functions as before
Problems

- Still many of the same problems as before
- Hard to keep same # ktrheads as available CPUs
  - Kernel knows how many CPUs available
  - But tries to hide this from applications with preemption
- Kernel doesn’t know about relative importance of threads
  - Might preempt kthread in which library holds important lock
Lessons

• Threads best implemented as a library
  - But kernel threads not the best interface on which to build this

• Better kernel interfaces have been suggested
  - See Scheduler Activations [Anderson et al.]
  - Maybe too complex to implement on existing OSes (some have added then removed such features)

• Standard user or kernel threads are still fine for most purposes
  - Use kernel threads if I/O concurrency main goal
  - Use user on kernel threads for highly concurrent (e.g., scientific applications) with many thread switches
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?
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?
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*

- Without SC, multiple CPUs “worse” than preemptive
  - Get results that cannot occur with any interleaving on 1 CPU

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

- 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 SC, 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:

  ```
  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;
    /* Remaining 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 */
    }
}
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 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: 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

```assembly
_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