Implementing Threads and Synchronization

Jeff Chase
Duke University
Programs run as independent processes.

Protected OS kernel mediates access to shared resources.

Each process has a private virtual address space and at least one thread.

Protected system calls

...and upcalls (e.g., signals)

Threads enter the kernel for OS services.

The kernel code and data are protected from untrusted processes.
Project 1t

- **Pretend** that your 1t code is executing within the kernel.
- **Pretend** that the 1t API methods are system calls.
- **Pretend** that your kernel runs on a uniprocessor.
  - One core; at most one thread is in **running** state at a time.
- **Pretend** that your 1t code has direct access to protected hardware functions (since it is in the kernel).
  - Enable/disable interrupts
  - *(You can’t really, because your code is executing in user mode. But we can use Unix signals to simulate timer interrupts, and we can simulate blocking them).*

- It may be make-believe, but you are building the foundation of a classical operating system kernel.
Threads in Project 1

Threads

- thread_create(func, arg);
- thread_yield();

Locks/Mutexes

- thread_lock(lockID);
- thread_unlock(lockID);

Condition Variables

- thread_wait(lockID, cvID);
- thread_signal(lockID, cvID);
- thread_broadcast(lockID, cvID);

Mesa monitors

All functions return an error code: 0 is success, else -1.
Thread control block

TCB1
PC

Ready queue
registers

Code
Stack

TCB2
PC
SP
registers

Code
Stack

TCB3
PC
SP
registers

Code
Stack

Thread 1 running

CPU
Thread control block

Thread 1 running
Creating a new thread

- Also called “forking” a thread
- Idea: create initial state, put on ready queue

1. Allocate, initialize a new TCB
2. Allocate a new stack
3. Make it look like thread was going to call a function
   - PC points to first instruction in function
   - SP points to new stack
   - Stack contains arguments passed to function
   - Project 1: use makecontext
4. Add thread to ready queue
Implementing threads

- **Thread_fork(func, args)**
  - Allocate thread control block
  - Allocate stack
  - Build stack frame for base of stack (stub)
  - Put func, args on stack
  - Put thread on ready list
  - Will run sometime later (maybe right away!)

- **stub(func, args): Pintos switch_entry**
  - Call (*func)(args)
  - Call thread_exit()
CPU Scheduling 101

The OS scheduler makes a sequence of “moves”.

- Next move: if a CPU core is idle, pick a ready thread $t$ from the ready pool and dispatch it (run it).
- Scheduler’s choice is “nondeterministic”
- Scheduler’s choice determines interleaving of execution

```
blocked
threads
Wakeup
ready pool
```

```
If timer expires, or wait/yield/terminate
GetNextToRun
SWITCH()
```
Thread states and transitions

If a thread is in the **ready** state thread, then the system may choose to run it “at any time”. When a thread is running, the system may choose to preempt it at any time. From the point of view of the program, dispatch and preemption are **nondeterministic**: we can’t know the **schedule** in advance.

These **preempt** and **dispatch** transitions are controlled by the kernel scheduler.

**Sleep** and **wakeup** transitions are initiated by calls to internal sleep/wakeup APIs by a running thread.
Timer interrupts enable timeslicing

The **system clock** (timer) interrupts each core periodically, giving control back to the kernel. The kernel may **preempt** the running thread and switch to another (an involuntary context switch).

Enables **timeslicing**

User mode

Kernel mode

Kernel "top half"

Kernel "bottom half" (interrupt handlers)

```
while(1);
...
```

Resume

Clock interrupt

Time

Time →
**Synchronization: layering**

<table>
<thead>
<tr>
<th>Concurrent Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphores</td>
</tr>
<tr>
<td>Locks</td>
</tr>
<tr>
<td>Condition Variables</td>
</tr>
<tr>
<td>Interrupt Disable</td>
</tr>
<tr>
<td>Atomic Read/Modify/Write Instructions</td>
</tr>
<tr>
<td>Multiple Processors</td>
</tr>
<tr>
<td>Hardware Interrupts</td>
</tr>
</tbody>
</table>
We need hardware support for atomic read-modify-write operations on data item X by a thread T.

- **Atomic** means that no other code can operate on X while T’s operation is in progress.
  - Can’t allow any interleaving of operations on X!
- Locks provide atomic critical sections, but…
- We need hardware support to implement safe locks!
- In this discussion we continue to presume that thread primitives and locks are implemented in the kernel.
Options for hardware support for synchronization:

1. Kernel software can **disable interrupts** on T’s core.
   - Prevents an involuntary context switch (preempt-yield) to another thread on T’s core.
   - Also prevents conflict with any interrupt handler on T’s core.
   - But on multi-core systems, we also must prevent accesses to X by other cores, and disabling interrupts isn’t sufficient.

2. For multi-core systems the solution is **spinlocks**.
   - **Spinlocks** are locks that busy-wait in a loop when not free, instead of blocking (a blocking lock is called a **mutex**).
   - Use hardware-level **atomic instructions** to build spinlocks.
   - Use spinlocks internally to implement higher-level synchronization (e.g., monitors).
Spinlock: a first try

```c
int s = 0;

lock() {
    while (s == 1) {};
    ASSERT (s == 0);
    s = 1;
}

unlock() {
    ASSERT(s == 1);
    s = 0;
}
```

Spinlocks provide mutual exclusion among cores without blocking.

Global spinlock variable

Busy-wait until lock is free.

Spinlocks are useful for lightly contended critical sections where there is no risk that a thread is preempted while it is holding the lock, i.e., in the lowest levels of the kernel.
Spinlock: what went wrong

```c
int s = 0;

lock() {
    while (s == 1) {
    // Race to acquire.
    // Two (or more) cores see s == 0.
    }

    s = 1;
}

unlock();
    s = 0;
}
```
Spinlock: what went wrong

```
int s = 0;

lock() {
    while (s == 1) {
        
        s = 1;
    }
}

unlock();
    s = 0;
}
```

Race to acquire.
Two (or more) cores see s == 0.
We need an atomic “toehold”

- To implement safe mutual exclusion, we need support for some sort of “magic toehold” for synchronization.
  - The lock primitives themselves have critical sections to test and/or set the lock flags.

- Safe mutual exclusion on multicore systems requires specific hardware support: atomic instructions
  - These instructions perform an atomic read-modify-write of a memory location. We use them to implement locks.
  - If we have any of those, we can build higher-level synchronization objects like monitors or semaphores.
  - Note: we also must be careful of interrupt handlers....
Using read-modify-write instructions

- Disabling interrupts
  - Ok for uni-processor, breaks on multi-processor
  - Why?
- Could use atomic load-store to make a lock
  - Inefficient, lots of busy-waiting
- Hardware people to the rescue!
Using read-modify-write instructions

- Modern processor architectures
  - Provide an **atomic read-modify-write instruction**
- Atomically
  - Read value from memory into register
  - Write new value to memory
- Implementation details
  - Lock memory location at the memory controller
Example: test&set

Set: sets location to 1
Test: returns old value

```
test&set (X) {
    tmp = X
    X = 1
    return (tmp)
}
```

- Atomically!
- Slightly different on x86 (Exchange)
  - Atomically swaps value between register and memory
Spinlock implementation

- Use test&set
- Initially, value = 0

```c
lock () {
    while (test&set(value) == 1) {
    }
}
```
```
unlock () {
    value = 0
}
```

What happens if value = 1?
What happens if value = 0?
Atomic instructions: Test-and-Set

Spinlock::Acquire () {
    while(held);
    held = 1;
}

Wrong
load 4(SP), R2 ; load “this”
busywait:
load 4(R2), R3 ; load “held” flag
bnz R3, busywait ; spin if held wasn’t zero
store #1, 4(R2) ; held = 1

Right
load 4(SP), R2 ; load “this”
busywait:
    tsl 4(R2), R3 ; test-and-set this->held
    bnz R3, busywait ; spin if held wasn’t zero

Problem: interleaved load/test/store.

Solution: TSL atomically sets the flag and leaves the old value in a register.

One example: tsl test-and-set-lock (from an old machine)
Threads on cores

```
int x;

worker()
{
    while (1) {
        acquire L;
        x++;
        release L;
    }
}
```
int x;
worker()
while (1) {
    acquire L;
    x++;
    release L;
}
Spinlock: IA32

Idle the core for a contended lock.

Atomic exchange to ensure safe acquire of an uncontended lock.

XCHG is a variant of compare-and-swap: compare \( x \) to value in memory location \( y \); if \( x \neq *y \) then exchange \( x \) and \( *y \). Determine success/failure from subsequent value of \( x \).

Spin_Lock:
- CMP lockvar, 0 ; Check if lock is free
- JE Get_Lock
- PAUSE ; Short delay
- JMP Spin_Lock

Get_Lock:
- MOV EAX, 1
- XCHG EAX, lockvar ; Try to get lock
- CMP EAX, 0 ; Test if successful
- JNE Spin_Lock
Atomic instructions also drive hardware memory consistency
7.1. LOCKED ATOMIC OPERATIONS
The 32-bit IA-32 processors support locked atomic operations on locations in system memory. These operations are typically used to manage shared data structures (such as semaphores, segment descriptors, system segments, or page tables) in which two or more processors may try simultaneously to modify the same field or flag.

Note that the mechanisms for handling locked atomic operations have evolved as the complexity of IA-32 processors has evolved.

Synchronization mechanisms in multiple-processor systems may depend upon a strong memory-ordering model. Here, a program can use a locking instruction such as the XCHG instruction or the LOCK prefix to insure that a read-modify-write operation on memory is carried out atomically. Locking operations typically operate like I/O operations in that they wait for all previous instructions to complete and for all buffered writes to drain to memory.

This is just an example of a principle on a particular machine (IA32): these details aren’t important.
Spelling it out

• Spinlocks are fast locks for short critical sections.
• They waste CPU time and they are dangerous: what if a thread is preempted while holding a spinlock?
• They are useful/necessary inside the kernel on multicore systems, e.g., to implement higher-level synchronization.
• But on a uniprocessor (one core, one thread runs at a time), we can use enable/disable interrupts instead.
• That is what we (pretend) to do in p1t. So you don’t need spinlocks for p1t.
• Note: on a multicore system, you need both spinlocks and interrupt disable! (Internally, within the kernel.)
Using interrupt disable-enable

• Disable-enable on a uni-processor
  • Assume atomic (can use atomic load/store)

• How do threads get switched out (2 ways)?
  • Internal events (yield, I/O request)
  • External events (interrupts, e.g., timers)

• Easy to prevent internal events
• Use disable/enable to prevent external events
Interrupts

An arriving interrupt transfers control immediately to the corresponding handler (Interrupt Service Routine). ISR runs kernel code in kernel mode in kernel space. Interrupts may be nested according to priority.
Interrupt priority: rough sketch

- **N interrupt priority classes**
- When an ISR at priority $p$ runs, CPU blocks interrupts of priority $p$ or lower.
- Kernel software can query/raise/lower the CPU **interrupt priority level (IPL)**.
  - Defer or **mask** delivery of interrupts at that IPL or lower.
  - Avoid races with higher-priority ISR by raising CPU IPL to that priority.
  - e.g., BSD Unix *spl*/*splx* primitives.
- **Summary**: Kernel code can enable/disable interrupts as needed.

**BSD example**
```
int s;
s = splhigh();
/* all interrupts disabled */
splx(s);
/* IPL is restored to s */
```
What ISRs do

• Interrupt handlers:
  – trigger involuntary thread switches (preempt)
  – bump counters, set flags
  – throw packets on queues
  – ...
  – **wakeup waiting threads**

• **Wakeup puts a thread on the ready queue.**
• **On multicore, use spinlocks for the queues**
• **But how do we synchronize with interrupt handlers?**
Wakeup from interrupt handler

Examples?

**Note:** interrupt handlers do not block: typically there is a single interrupt stack for each core that can take interrupts. If an interrupt arrived while another handler was sleeping, it would corrupt the interrupt stack.
Synchronizing with ISRs

- Interrupt delivery can cause a race if the ISR shares data (e.g., a thread queue) with the interrupted code.
- **Example**: Core at IPL=0 (thread context) holds spinlock, interrupt is raised, ISR attempts to acquire spinlock….
- **That would be bad**. Disable interrupts.

```c
int s;
s = splhigh();
/* critical section */
splx(s);
```
Spinlocks in the kernel

- We have basic mutual exclusion that is very useful inside the kernel, e.g., for access to thread queues.
  - **Spinlocks** based on atomic instructions.
  - Can synchronize access to sleep/ready queues used to implement higher-level synchronization objects.

- Don’t use spinlocks from user space! A thread holding a spinlock could be preempted at any time.
  - If a thread is preempted while holding a spinlock, then other threads/cores may waste many cycles spinning on the lock.
  - That’s a kernel/thread library integration issue: fast spinlock synchronization in user space is a research topic.

- But spinlocks are very useful in the kernel, esp. for synchronizing with interrupt handlers!
How to disable/enable to synchronize the thread library for Project 1?

Threads
- thread_create(func, arg);
- thread_yield();

Locks/Mutexes
- thread_lock(lockID);
- thread_unlock(lockID);

Condition Variables
- thread_wait(lockID, cvID);
- thread_signal(lockID, cvID);
- thread_broadcast(lockID, cvID);

Mesa monitors

All functions return an error code: 0 is success, else -1.
The ready thread pool

For p1t, the ready thread pool is a simple FIFO queue: the **ready list** or **ready queue**. Scalable multi-core systems use multiple pools to reduce locking contention among cores. It is typical to implement a ready pool as a sequence of queues for different priority levels.

**Diagram:**
- **new threads** → **ThreadCreate** → **ready pool**
- **blocked threads** → **Wakeup**
- **ready pool**:
  - If timer expires, or running thread blocks, yields, or terminates:
    - **GetNextToRun**
  - **SWITCH()**
Locking and blocking

If thread $T$ attempts to acquire a lock that is busy (held), $T$ must spin and/or block (sleep) until the lock is free. By sleeping, $T$ frees up the core for some other use. Just spinning is wasteful!

Note: $H$ is the lock holder when $T$ attempts to acquire the lock.
A Rough Idea

Yield() {
    disable;
    next = FindNextToRun();
    ReadyToRun(this);
    Switch(this, next);
    enable;
}

Sleep() {
    disable;
    this->status = BLOCKED;
    next = FindNextToRun();
    Switch(this, next);
    enable;
}

Issues to resolve:
What if there are no ready threads?
How does a thread terminate?
How does the first thread start?
Yield

```c
yield() {
    put my TCB on ready list;
    switch();
}

switch() {
    pick a thread TCB from ready list;
    if (got thread) {
        save my context;
        load saved context for thread;
    }
}
```
Monitors 1

lock() {
  while (this monitor is not free) {
    put my TCB on this monitor lock list;
    switch(); /* sleep */
  }
  set this thread as owner of monitor;
}

unlock() {
  set this monitor free;
  get a waiter TCB from this monitor lock list;
  put waiter TCB on ready list; /* wakeup */
}

Where to enable/disable interrupts?
```c
wait() {
    unlock();
    put my TCB on this monitor wait list;
    switch();  /* sleep */
    lock();
}

notify() {
    get a waiter TCB from this monitor wait list;
    put waiter TCB on ready list;  /* wakeup */
}
```

Where to enable/disable interrupts?
Why use locks?

- If we have disable-enable, why do we need locks?
  - Program could bracket critical sections with disable-enable
  - Might not be able to give control back to thread library

```c
disable_interrupts
while (1) {}()
```

- Can’t have multiple locks (over-constrains concurrency)

- **Project 1**: only disable interrupts in thread library
Why use locks?

• How do we know if disabling interrupts is safe?
  • Need hardware support
  • CPU has to know if running code is trusted (i.e, is the OS)
  • Example of why we need the kernel

• Other things that user programs shouldn’t do?
  • Manipulate page tables
  • Reboot machine
  • Communicate directly with hardware

• Will cover later in memory lectures
switch/MIPS (old, new)
{
  old->stackTop = SP;
  save RA in old->MachineState[PC];
  save callee registers in old->MachineState

  restore callee registers from new->MachineState
  RA = new->MachineState[PC];
  SP = new->stackTop;

  return (to RA)
}

This example (from the old MIPS ISA) illustrates how context switch saves/restores the user register context for a thread, efficiently and without assigning a value directly into the PC.
Example: Switch()

switch/MIPS (old, new) {
    old->stackTop = SP;
    save RA in old->MachineState[PC];
    save callee registers in old->MachineState
    restore callee registers from new->MachineState
    RA = new->MachineState[PC];
    SP = new->stackTop;
    return (to RA)
}

RA is the return address register. It contains the address that a procedure return instruction branches to.
What to know about context switch

- The Switch/MIPS example is an illustration for those of you who are interested. It is not required to study it. But you should understand how a thread system would use it (refer to state transition diagram):
  - Switch() is a procedure that returns immediately, but it returns onto the stack of new thread, and not in the old thread that called it.
  - Switch() is called from internal routines to sleep or yield (or exit).
  - Therefore, every thread in the blocked or ready state has a frame for Switch() on top of its stack: it was the last frame pushed on the stack before the thread switched out. (Need per-thread stacks to block.)
  - When a thread switches into the running state, it always returns immediately from Switch() back to the internal sleep or yield routine, and from there back on its way to wherever it goes next.
Memory ordering

• Shared memory is complex on multicore systems.
• Does a load from a memory location (address) return the latest value written to that memory location by a store?
• What does “latest” mean in a parallel system?

It is common to presume that load and store ops execute sequentially on a shared memory, and a store is immediately and simultaneously visible to load at all other threads. But not on real machines.
Memory ordering

- A **load** might fetch from the local cache and not from memory.
- A **store** may buffer a value in a local cache before draining the value to memory, where other cores can access it.
- Therefore, a **load** from one core does not necessarily return the “latest” value written by a **store** from another core.

A trick called Dekker’s algorithm supports mutual exclusion on multi-core without using atomic instructions. It assumes that **load** and **store** ops on a given location execute sequentially. But they don’t.
“Sequential” Memory ordering

A machine is **sequentially consistent** iff:

- Memory operations (loads and stores) appear to execute in some sequential order on the memory, and
- Ops from the same core appear to execute in program order.

No **sequentially consistent execution** can produce the result below, yet it can occur on modern machines.

To produce this result:

- \(4 < 2\) (4 **happens-before** 2) and \(3 < 1\). No such schedule can exist unless it also reorders the accesses from T1 or T2. Then the reordered accesses are out of program order.
The first thing to understand about memory behavior on multi-core systems

- Cores must see a “consistent” view of shared memory for programs to work properly. A machine can be “consistent” even if it is not “sequential”. But what does it mean?
- Synchronization accesses tell the machine that ordering matters: a happens-before relationship exists. Machines always respect that.
  - Modern machines work for race-free programs.
  - Otherwise, all bets are off. Synchronize!

The most you should assume is that any memory store before a lock release is visible to a load on a core that has subsequently acquired the same lock.
An execution schedule defines a total order of synchronization events (at least on any given lock/monitor): the synchronization order.

Different schedules of a given program may have different synchronization orders.

Just three rules govern synchronization order:

1. Events within a thread are ordered.
3. The order is transitive: if (A < B) and (B < C) then A < C.

Purple’s unlock/release action synchronizes-with the subsequent lock/acquire.
Happens-before revisited

An execution schedule defines a partial order of program events. The ordering relation (<) is called happens-before.

Two events are concurrent if neither happens-before the other in the schedule.

Just three rules govern happens-before order:

1. Events within a thread are ordered.
3. Happens-before is transitive: if (A < B) and (B < C) then A < C.

Machines may reorder concurrent events, but they always respect happens-before ordering.
What’s a race?

• Suppose we execute program P.
• The events are synchronization accesses (lock/unlock) and loads/stores on shared memory locations, e.g., x.
• The machine and scheduler choose a schedule S
• S imposes a total order on accesses for each lock, which induces a happens-before order on all events.
• Suppose there is some x with a concurrent load and store to x. (The load and store are conflicting.)
• Then P has a race. A race is a bug. P is unsafe.
• Summary: a race occurs when two or more conflicting accesses are concurrent.
"Happens-before is the transitive closure of program order and synchronization order."

“A program is said to be correctly synchronized or data-race-free iff all sequentially consistent executions of the program are free of data races.” [According to happens-before.]
The “simple” JMM happens-before model:

- A read cannot see a write that *happens after* it.
- If a read sees a write (to an item) that happens before the read, then the write must be the last write (to that item) that happens before the read.

Augment for sane behavior for unsafe programs (loose):

- Don’t allow an *early* write that “depends on a read returning a value from a data race”.
- An *uncommitted* read must return the value of a write that happens-before the read.
The point of all that

- We use special atomic instructions to implement locks.
- E.g., a TSL or CMPXCHG on a lock variable \texttt{lockvar} is a synchronization access.
- Synchronization accesses also have special behavior with respect to the memory system.
  - Suppose core C1 executes a synchronization access to \texttt{lockvar} at time \( t_1 \), and then core C2 executes a synchronization access to \texttt{lockvar} at time \( t_2 \).
  - Then \( t_1 < t_2 \): every memory \texttt{store} that happens-before \( t_1 \) must be visible to any \texttt{load} on the same location after \( t_2 \).
- If memory always had this expensive sequential behavior, i.e., every access is a synchronization access, then we would not need atomic instructions: we could use “Dekker’s algorithm”.
- We do not discuss Dekker’s algorithm because it is not applicable to modern machines. (Look it up on wikipedia if interested.)