Implementing Threads 2
Thread Termination

• Termination
  – thread becomes zombie
  – if joinable
    - notify waiter, if present
  – if detached
    - disappear
  • thread can’t do this by itself!

The problem here is that a terminating thread can’t free its own stack, since it would still be using that stack on return from free.
while (zombies) {
    delete(zombie);
}
Thread Yield

Current Thread

Runnable Thread

Runnable Thread

Runnable Thread

Runnable Thread
void thread_yield() {
    if (!queue_empty(runqueue)) {
        enqueue(runqueue, CurrentThread);
        thread_switch();
    }
}

Time Slicing

• Periodically
  – current thread forced to do a thread yield

```c
void ClockInterrupt(int sig) {
    thread_yield();
}
```

• Implement ClockInterrupt with VTALRM signal
Digression: Invoking the Signal Handler

- Basic idea is to set up the user stack so that the handler is called as a subroutine and so that when it returns, normal execution of the thread may continue
- Complications:
  - saving and restoring registers
  - signal mask

If a signal is to be dealt with via a signal handler, the kernel arranges so that the handler is invoked much like a subroutine, and that when it returns, the thread resumes its normal execution. To accomplish this, the state of the thread must be saved before invocation of the handler and resumed on return. A major component of the thread’s state information is the contents of the registers: a typical subroutine does not save and restore all of the registers, since some are allowed to be modified (e.g., the register used to pass back a return value); since a signal handler can be invoked at any point, something must be done to save and restore all registers. We must also deal with masking signals while in the handler: invoking a signal handler causes a specified set of signals (including the one that was just delivered) to be masked off; the thread’s original signal mask must be restored on return from the handler.
In this sequence of slides, we show how a signal handler is invoked in Linux. In this slide, our program is executing within its “main line.”
A signal has occurred. This causes the thread to be interrupted and the kernel is entered. The thread’s registers (including its instruction pointer) are saved on the thread’s kernel stack.
The kernel pushes a `struct sigframe` onto the thread’s user-mode stack, containing a copy of the thread’s user-mode registers (copied from the kernel stack — it includes the thread’s user-mode program counter, which points to instruction following the point of interruption), the signal mask, and a new return address that points to executable code (also pushed onto the stack). This executable code, which is executed on return from the signal handler, invokes the `sigreturn` system call with the signal mask and the user-mode registers as its arguments. The effect of the call is to return to the point of the interrupt with the signal mask restored.
Now the system returns our thread to user mode and it resumes execution in the appropriate signal handling routine.
When the signal handler returns, it returns to the executable code that was pushed onto the stack (within *sigframe*) by the kernel. This code, as already mentioned, invokes the *sigreturn* system call, which passes a pointer to the *sigframe* structure back to the kernel.
The `sigreturn` handler in the kernel restores the user-mode registers and signal mask from the `sigframe` structure and returns back to the point of interrupt.
Quiz 1

The description of invoking the signal handler:

a) works fine.
b) has a security problem discussed in CS 33.
c) is rendered unusable because of a solution to a security problem discussed in CS 33.
When the handler is entered for SIGVTALRM, SIGVTALRM is masked. Since the thread yields and does not return from the handler right away, we need to explicitly unmask the signal, as shown in the slide.
Setting Up Time Slicing

```c
struct sigaction timesliceact;
timesliceact.sa_handler = ClockInterrupt;
timesliceact.sa_mask = VTALRMmask;
timesliceact.sa_flags = SA_RESTART; // avoid EINTR

struct timeval interval = {0, 1};
    // every .001 milliseconds
struct itimerval timerval;
timerval.it_value = interval;
timerval.it_interval = interval;
sigaction(SIGVTALRM, &timesliceact, 0);
setitimer(ITIMER_VIRTUAL, &timerval, 0);
    // time slicing is started!
```
Async-Signal Safety

- A function is asynchronous-signal safe if it may be used in the handler for an asynchronous signal (such as SIGVTALRM)
  - malloc and free
    - no
  - mutex_lock
    - no
  - read and write
    - yes
Achieving Async-Safety

- The problem: an action in the signal handler interferes with an action in the main-line code
  - while in malloc/free, a signal occurs and the handler calls malloc/free
  - while holding the lock on a mutex, a thread is interrupted and the handler attempts to lock the mutex
- The solution: mask signals while in malloc/free and when holding locks
  - assuming signal handler calls malloc/free or mutex_lock
Caution!

- *thread_switch* is not async-signal safe
  - it's called from *thread_yield*, which is called from the signal handler for SIGVTALRM
  - must mask signals before calling it (and unmask afterwards)
Note that `pthread_sigmask` and `sigprocmask` do exactly the same thing.
Doing It Cheaply

```c
void thread_no_preempt_on() {
    thread_no_preempt = 1;
}

void thread_no_preempt_off() {
    thread_no_preempt = 0;
}

void ClockInterrupt(int sig) {
    if (thread_no_preempt)
        return;
    ...
}
```

Limitations of User Threads

- Threads are implemented strictly at user level
  - the OS kernel is unaware of their existence
- What happens if a user thread makes a blocking system call, e.g., read?

If a user-level-implemented thread makes a system call that blocks (i.e., causes the thread to go to sleep), then there’s no opportunity for a thread switch to some other user-level thread. The user-level threads are multiplexed on a single kernel (OS) thread, and if that thread blocks, then, effectively, all user-level threads block.
Quiz 2

```c
void thread_switch() {
    thread_t *NextThread, *OldCurrent;

    NextThread = dequeue(RunQueue);
    OldCurrent = CurrentThread;
    CurrentThread = NextThread;
    swapcontext(&OldCurrent->context, &NextThread->context);
}
```

Given the discussion so far, will RunQueue ever be empty?

a) yes  
b) no
Multiple Processors

```c
void thread_switch() {
    thread_t *NextThread, *OldCurrent;

    NextThread = dequeue(RunQueue);
    OldCurrent = CurrentThread;
    CurrentThread = NextThread;
    swapcontext(&OldCurrent->context, &NextThread->context);
}
```

- How do we employ multiple processors?
  - code merely switches the caller's processor to another thread
- What if the RunQueue is empty?
Solution Sketch

- Introduce idle threads, one for each processor
- Thread calling `thread_switch` switches to idle thread for its current processor
- Idle thread then switches to first thread on `RunQueue`, if any
- If `RunQueue` is empty, idle thread repeatedly checks `RunQueue` until it’s not empty, then switches to first thread
In this slide, `getcontext` saves the thread’s register context in the area pointed to by its argument. `setcontext` loads the registers with the context that’s located in the area pointed to by its argument. `processor_ID` is private to each idle thread, i.e., each idle thread has a separate copy of it; it contains the ID of the processor represented by that thread.

This code is deceptively simple-looking. Each idle thread starts off by calling `IdleThread_switch`. It saves its context at line 10, and then goes on to repeatedly check the `RunQueue`. When one finds a thread on the `RunQueue`, it switches to that thread’s context in line 14.

Threads calling `thread_switch` first save their registers by calling `getcontext` at line 3. The variable `first` (declared `volatile` to force gcc not to put it in a register, but to leave it in memory) is set to 1 so that, on the (first) return from `getcontext`, its value is still 1 and the thread continues executing sequentially. At line 7 the call to `setcontext` switches to the registers of the processor’s idle thread. That context was saved in line 10, and thus control continues at line 11, in the context of idle thread, as explained in the preceding paragraph.

When an idle thread switches to a normal thread (by calling `setcontext` at line 14), the normal thread resumes execution starting from where it had last saved its context, at line 3. Thus the thread continues at line 4. At this point, `first`, its local variable, now has a value of zero, and thus the thread returns from its original call to `thread_switch`.

Solution Details

```c
1  void thread_switch() {  
2      volatile int first = 1;  
3      getcontext(&CurrentThread[processor_ID]->context);  
4      if (!first)  
5          return;  
6      first = 0;  
7      setcontext(&IdleThread[processor_ID]->context);  
8  }

9  void IdleThread_switch() {  
10     getcontext(&IdleThread[processor_ID]->context);  
11     while (1) {  
12         if (!queue_empty(RunQueue))  
13             continue;  
14         CurrentThread[processor_ID] = dequeue(RunQueue);  
15         setcontext(&CurrentThread[processor_ID]->context);  
16     }  
17  }
```
MP Mutual Exclusion

• Two sorts
  – spin locks
    - threads wait by repeatedly testing the lock
  – blocking locks
    - threads wait by sleeping, depending on some other thread to wake them up
Hardware Support for Spin Locks

- Compare and swap instruction

```c
int CAS(int *ptr, int old, int new) {
    int tmp = *ptr;
    if (*ptr == old)
        *ptr = new;
    return tmp;
}
```

Note that the CAS instruction is implemented (in hardware) as an atomic instruction: its effect is instantaneous: nothing else can affect memory during the execution of the instruction. In the x86 and x86-64 architectures, the effect of CAS is achieved with the cmpxchng (compare and exchange) instruction with the lock prefix.
Naive Spin Lock

```c
void spin_lock(int *spin) {
    while(CAS(spin, 0, 1))
        ;
}

void spin_unlock(int *spin) {
    *spin = 0;
}
```
Better Spin Lock

```c
void spin_lock(int *spin) {
    while (1) {
        if (*spin == 0) {
            // the mutex was at least momentarily unlocked
            if (!CAS(spin, 0, 1))
                break; // we have locked the mutex
            // some other thread beat us to it, so try again
        }
    }
}
```
Blocking Locks

void blocking_lock(mutex_t *mut) {
    if (mut->holder != 0) {
        enqueue(mut->wait_queue, CurrentThread);
        thread_switch();
    } else
        mut->holder = CurrentThread;
}

void blocking_unlock(mutex_t *mut) {
    if (queue_empty(mut->wait_queue))
        mut->holder = 0;
    else {
        mut->holder =
        dequeue(mut->wait_queue);
        enqueue(RunQueue, mut->holder);
    }
}

Does it work?

Hint: the answer is not yes.
void blocking_lock(mutex_t *mut) {
spin_lock(mut->spinlock);
if (mut->holder != 0) {
    enqueue(mut->wait_queue, CurrentThread);
    spin_unlock(mut->spinlock);
    thread_switch();
} else {
    mut->holder = CurrentThread;
    spin_unlock(mut->spinlock);
}
}

void blocking_unlock(mutex_t *mut) {
spin_lock(mut->spinlock);
if (queue_empty(mut->wait_queue)) {
    mut->holder = 0;
} else {
    mut->holder = dequeue(mut->wait_queue);
    enqueue(RunQueue, mut->holder);
}
spin_unlock(mut->spinlock);
}

Quiz 3
This
a) always works
b) occasionally doesn’t work
c) hardly ever works
For details on futexes, avoid the Linux man pages, but look at http://people.redhat.com/drepper/futex.pdf, from which this material was obtained. Note that there's actually just one futex system call; whether it's a wait or a wakeup is specified by an argument.
These functions are available on most architectures, particularly on the x86. Note that their effect must be *atomic*: everything happens at once.
void lock(futex_t *futex) {
    int c;
    while ((c = atomic_inc(&futex->val)) != 0)
        futex_wait(futex, c+1);
}

void unlock(futex_t *futex) {
    futex->val = 0;
    futex_wake(futex);
}
void lock(futex_t *futex) {
    int c;
    if ((c = CAS(&futex->val, 0, 1) != 0) 
        do {
            if (c == 2 || (CAS(&futex->val, 1, 2) != 0))
                futex_wait(futex, 2);
            while ((c = CAS(&futex->val, 0, 2)) != 0))
        }
}

void unlock(futex_t *futex) {
    if (atomic_dec(&futex->val) != 1) {
        futex->val = 0;
        futex_wake(futex);
    }
}
MP Memory Issues

- Naive view is that all processors in MP system see same memory contents at all times
  - they don’t
This slide illustrates a simplistic view of the architecture of a multi-core processor: a number of processors are all directly connected to the same memory (which they share). If one core (or processor) stores into a storage location and immediately thereafter another core loads from the same storage location, the second core loads exactly what the first core stored.
Real multi-core processors employ a hierarchy of caches between the cores and memory, will the caches sharing the bus with the memory controller. (The Intel I-5 architecture, used on SunLab machines, has three levels of caches: two L1 caches (one each for data and instructions) per core, one L2 cache per core, and an L3 cache shared by all four cores. For the purposes of this discussion, it suffices to think of just one cache for each core.) An elaborate cache-coherency protocol is used so that the caches are consistent: no two caches have different versions of the same data item and if some a particular cache contains a data item that different (and newer) than what’s in memory, other caches when they attempt to load that item will get the newest version.
This slide shows an even more realistic model, pretty much the same as what we saw is actually used in recent Intel processors. Between each core and its caches is a store buffer. Stores by a core go into the buffer. Sometime later the effect of the store reaches the cache. In the meantime, the core is issuing further instructions. Loads by the core are handled from the buffer if the data is still there; otherwise they are taken from the caches, or perhaps from memory.

In all instances of this model the effect of a store, as seen by other cores, is delayed. In some instances of this model the order of stores made by one core might be perceived differently by other cores. Architectures with the former property are said to have *delayed stores*; architectures with the latter are said to have *reordered stores* (an architecture could well have both properties).
In this example, one thread running on one processor is loading from an integer in storage; another thread running on another processor is loading from and then storing into an integer in storage. Can this be done safely without explicit synchronization?

On most architectures, the answer is yes. If the integer in question is aligned on a natural (e.g., eight-byte) boundary, then the hardware (perhaps the cache) insures that loads and stores of the integer are atomic.

However, one cannot assume that this is the case on all architectures. Thus a portable program must use explicit synchronization (e.g., a mutex) in this situation.
Shown on the slide is Peterson’s algorithm for handling mutual exclusion for two threads without explicit synchronization. (The me argument for one thread is 0 and for the other is 1.) This program works given the first two shared-memory models. Does it work with delayed-store architectures?

This example is a solution, employing “busy waiting,” to the producer-consumer problem for one consumer and one producer. It works for the first two shared-memory models, and even for delayed-store architectures. But does it work on reordered-store architectures?

This solution to the producer-consumer problem is from “Proving the Correctness of Multiprocess Programs,” by L. Lamport, IEEE Transactions on Software Engineering, SE-3(2) 1977: 125-143.
The point of the previous several slides is that one cannot rely on expected properties of shared memory to eliminate explicit synchronization. Shared memory can behave in some very unexpected ways. However, it is the responsibility of the implementers of the various synchronization primitives to make certain not only that they behave correctly, but also that they synchronize memory with respect to other threads.