Implementing Threads 2
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
Invoking the Signal Handler (1)

Main Line

```c
func(int a1, int a2) {
    int i, j = 2;
    for (i=a1; i<a2; i++) {
        j = j*2;
        j = j/127;
        ...
    }
}
```

Previous Frames

func Frame

User Stack
main line

```c
func(int a1, int a2) {
    int i, j = 2;
    for (i=a1; i<a2; i++) {
        j = j*2;
        j = j/127;
        ...
    }
}
```

Invoking the Signal Handler (2)
Invoking the Signal Handler (3)

Main Line

```c
func(int a1, int a2) {
    int i, j = 2;
    for (i=a1; i<a2; i++) {
        j = j*2;
        j = j/127;
        ...
    }
}
```

Handler

```c
signalhandler(int sig) {
    ...
}
```
Invoking the Signal Handler (4)

Main Line

```go
func(int a1, int a2) {
    int i, j = 2;
    for (i=a1; i<a2; i++) {
        j = j*2;
        j = j/127;
        ...
    }
}
```

Handler

```go
signalhandler(int sig) {
    ...
}
```

User Stack
Invoking the Signal Handler (5)

Main Line

```go
func(int a1, int a2) {
    int i, j = 2;
    for (i=a1; i<a2; i++) {
        j = j*2;
        j = j/127;
        ...
    }
}
```

Handler

```go
signalhandler(int sig) {
    ...
}
```
Invoking the Signal Handler (6)

Main Line
```go
func(int a1, int a2) {
    int i, j = 2;
    for (i=a1; i<a2; i++) {
        j = j*2;
        j = j/127;
    }
    ...
}
```

Handler
```go
signalhandler(int sig) {
    ...
}
```
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.
Time Slicing

• Periodically
  – current thread forced to do a thread yield

```c
void ClockInterrupt(int sig) {
  // SIGVTALRM is now masked
  pthread_sigmask(SIG_UNBLOCK, &VTALRMmask, 0);
  // SIGVTALRM is now unmasked
  thread_yield();
  // thread resumes here
}
```

• Implement ClockInterrupt with VTALRM signal
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 microsecond
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)
Masking/Unmasking Signals

```c
sigset_t VTALRMmask;
...
sigemptyset(&VTALRMmask);
sigaddset(&VTALRMmask, SIGVTALRM);
...
pthread_sigmask(SIG_BLOCK, &VTALRMmask, 0);
...
pthread_sigmask(SIG_UNBLOCK, &VTALRMmask, 0);
```
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;

    ...
}
```

```c
thread_no_preempt_on();
thread_switch();
thread_no_preempt_off();
```
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*?
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 \texttt{thread\_switch} switches to idle thread for its current processor
• Idle thread then switches to first thread on \textit{RunQueue}, if any
• If \textit{RunQueue} is empty, idle thread repeatedly checks \textit{RunQueue} until it’s not empty, then switches to first thread
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;
}
```
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

```c
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?
Working Blocking Locks (?)

```c
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);
    }
}
```

```c
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) never works
Futexes

- Safe, *efficient* kernel conditional queueing in Linux
- All operations performed atomically
  - `futex_wait(futex_t *futex, int val)`
    - *if* `futex->val` *is equal to* `val`, *then* sleep
    - *otherwise* return
  - `futex_wake(futex_t *futex)`
    - *wake up one thread from* `futex`'s *wait queue, if there are any waiting threads*
Ancillary Functions

- **int atomic_inc(int *val)**
  - add 1 to *val, return its original value
- **int atomic_dec(int *val)**
  - subtract 1 from *val, return its original value
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);
    }
}

Quiz 4
Does it work?

da) Yes
b) No
MP Memory Issues

• Naive view is that all processors in MP system see same memory contents at all times
  – they don’t
Multi-Core Processor: Simple View

Cores

Memory
Multi-Core Processor: More Realistic View

- Cores
- Caches
- Bus
- Memory
Multi-Core Processor: Even More Realistic

Diagram showing the structure of a multi-core processor with cores, caches, a bus, and memory.
Concurrent Reading and Writing

Thread 1:  
\[ i = \text{shared\_counter}; \]

Thread 2:  
\[ \text{shared\_counter}++; \]
void peterson(long me) {
    static long loser; // shared
    static long active[2] = {0, 0}; // shared
    long other = 1 - me; // private
    active[me] = 1;
    loser = me;
    while (loser == me && active[other]) {
    }
    // critical section
    active[me] = 0;
}
Busy-Waiting Producer/Consumer

```c
void producer(char item) {
    while (in - out == BSIZE) ;
    buf[in%BSIZE] = item;
    in++;
}

char consumer() {
    char item;
    while (in - out == 0) ;
    item = buf[out%BSIZE];
    out++;
    return(item);
}
```
Coping

• Use what’s available in the architecture to make sure all cores have the same view of memory (when necessary)
  – lock prefix on x86
  – mfence x86 instruction

• Use the synchronization primitives
  – presumably the implementers knew what they were doing