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 Reaper Thread

```c
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
Invoking the Signal Handler (1)

```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)

Main Line

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

Kernel Stack

User Stack

User Registers

IP

Previous Frames

func Frame
Invoking the Signal Handler (3)

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

Handler
signalhandler(int sig) {
    ...
}

User Stack
Kernel Stack
User Registers
IP
Previous Frames
func Frame
sigframe
IP
ret code
Invoking the Signal Handler (4)

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) {
    ...
}
```

User Stack

```
Previous Frames
  func Frame
    sigframe
      IP
      ret code
    sighandler frame
```
Invoking the Signal Handler (5)

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 (6)

Main Line

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

User Stack

Handler

```c
signalhandler(
    int sig) {
    ...
}
```
End of Digression: Back to Time Slicing

- Periodically
  - current thread forced to do a thread yield

```c
void ClockInterrupt(int sig) {
    // SIGVTALRM is now masked
    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 .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!
```
Masking/Unmasking Signals

```c
sigset_t VTALRMmask;
...
sigemptyset(&VTALRMmask);
sigaddset(&VTALRMmask, SIGVTALRM);
...
sigprocmask(SIG_BLOCK, &VTALRMmask, 0);
...
sigprocmask(SIG_UNBLOCK, &VTALRMmask, 0);
```
Doing It Cheaply

```c
void uthread_no_preempt_on() {
    uthread_no_preempt = 1;
}

void uthread_no_preempt_off() {
    uthread_no_preempt = 0;
}

void ClockInterrupt(int sig) {
    if (uthread_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., \textit{read}?
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
Solution Details

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

9 void IdleThread_switch() {
10    getcontext(&IdleThread[processor_number]->context);
11    while (1) {
12        if (queue_empty(RunQueue))
13            continue;
14        setcontext(&dequeue(RunQueue)->context);
15    }
16 }
```
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?
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);
}
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
Attempt 1

```c
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);
}
```
Attempt 2

```c
#include <futex.h>

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
Multi-Core Processor: Simple View

Cores

Memory
Multi-Core Processor: More Realistic View

Cores  L1Caches  Bus  Memory
Multi-Core Processor: Even More Realistic

- Cores
- L1 Caches
- Bus
- Memory

buffers
Concurrent Reading and Writing

Thread 1:

```
i = shared_counter;
```

Thread 2:

```
shared_counter++;  
```
Mutual Exclusion w/o Mutexes

```c
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

• Don’t rely on shared memory for synchronization
• Use the synchronization primitives
Which Runs Faster?

```c
volatile int a, b;

void *thread1(void *arg) {
    int i;
    for (i=0; i<reps; i++) {
        a = 1;
    }
}

void *thread2(void *arg) {
    int i;
    for (i=0; i<reps; i++) {
        b = 1;
    }
}
```

```c
volatile int a, padding[128], b;

void *thread1(void *arg) {
    int i;
    for (i=0; i<reps; i++) {
        a = 1;
    }
}

void *thread2(void *arg) {
    int i;
    for (i=0; i<reps; i++) {
        b = 1;
    }
}
```
Cache Lines

Address

Tag | Index | Offset

Data Cache

Cache Line

Tag | Data
False Sharing

Data Cache

Tag  a  b  Cache Line

Data Cache

Tag  a  b  Cache Line