CS 33

Multithreaded Programming IV
Outline

- Unix signals
- Cancellation
Deviations

• Signals

• Cancellation
  – tamed lightning
Signals

•

– who gets them?
– who needs them?

•

– how do you respond to them?
Dealing with Signals

- Per-thread signal masks
- Per-process signal vectors
- One delivery per signal
Signals and Threads

```c
int pthread_kill(pthread_t thread, int signo);
```

– thread equivalent of `kill`

```c
int pthread_sigmask(int how,
                    const sigset_t *newmask,
                    sigset_t oldmask);
```

– thread equivalent of `sigprocmask`
Asynchronous Signals (1)

```c
int main( ) {
    void handler(int);
    signal(SIGINT, handler);

    ...
}

void handler(int sig) {
    ...
}
```
Asynchronous Signals (2)

```c
int main() {
    void handler(int);

    signal(SIGINT, handler);

    ... // complicated program

    printf("important message: ", message);

    ... // more program
}

void handler(int sig) {
    ... // deal with signal

    printf("equally important message: ", message);
}
```
Quiz 1

```c
int main( ) {
    void handler(int);
    signal(SIGINT, handler);
    ... // complicated program
    pthread_mutex_lock(&mut);
    printf("important message: ", message);
    pthread_mutex_unlock(&mut);
    ... // more program
}

void handler(int sig) {
    ... // deal with signal
    pthread_mutex_lock(&mut);
    printf("equally important message: ", message);
    pthread_mutex_unlock(&mut);
}
```

Does this work?

a) yes
b) no
Synchronizing Asynchrony

`computation_state_t` state;
`sigset_t` set;

`void *monitor(void *dummy)` {
`    int` sig;
`    while (1) {
        sigwait(&set, &sig);
        display(&state);
    }
    return (0);
}

int main()` {
    `pthread_t` thread;

    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    pthread_sigmask(SIG_BLOCK, &set, 0);
    pthread_create(&thread, 0, monitor, 0);
    long_running_procedure( );
}
Cancellation
Sample Code

```c
void *thread_code(void *arg) {
    node_t *head = 0;
    while (1) {
        node_t *nodep;
        nodep = (node_t *)malloc(sizeof(node_t));
        if (read(0, &node->value,
                    sizeof(node->value)) == 0) {
            free(nodep);
            break;
        }
        nodep->next = head;
        head = nodep;
    }
    return head;
}

pthread_cancel(thread);
```
Cancellation Concerns

- Getting cancelled at an inopportune moment
- Cleaning up
Cancellation State

- **Pending cancel**
  - `pthread_cancel(thread)`

- **Cancels enabled or disabled**
  - `int pthread_setcancelstate`
    - `{PTHREAD_CANCEL_DISABLE, PTHREAD_CANCEL_ENABLE}, &oldstate`

- **Asynchronous vs. deferred cancels**
  - `int pthread_setcanceltype`
    - `{PTHREAD_CANCEL_Asynchronous, PTHREAD_CANCEL_DEFERRED}, &oldtype`
Cancellation Points

- `aio_suspend`
- `close`
- `creat`
- `fcntl` (when `F_SETLCKW` is the command)
- `fsync`
- `mq_receive`
- `mq_send`
- `msync`
- `nanosleep`
- `open`
- `pause`
- `pthread_cond_wait`
- `pthread_cond_timedwait`
- `pthread_join`
- `pthread_testcancel`
- `read`
- `sem_wait`
- `sigwait`
- `sigwaitinfo`
- `sigsuspend`
- `sigtimedwait`
- `sleep`
- `system`
- `tcdrain`
- `wait`
- `waitpid`
- `write`
Cleaning Up

- `void pthread_cleanup_push((void)(*routine)(void *), void *arg)`
- `void pthread_cleanup_pop(int execute)`
Sample Code, Revisited

```c
void *thread_code(void *arg) {
    node_t *head = 0;
    pthread_cleanup_push(
        cleanup, &head);
    while (1) {
        node_t *nodep;
        nodep = (node_t *)
            malloc(sizeof(node_t));
        if (read(0, &node->value,
                    sizeof(node->value)) == 0) {
            free(nodep);
            break;
        }
        nodep->next = head;
        head = nodep;
    }
    pthread_cleanup_pop(0);
    return head;
}

void cleanup(void *arg) {
    node_t **headp = arg;
    while (*headp) {
        node_t *nodep = head->next;
        free(*headp);
        *headp = nodep;
    }
}
```
A More Complicated Situation …

Diagram of a complex data structure with nodes labeled -1, 4, 6, 9, 8, 11, and arrows connecting them.
Start/Stop

• Start/Stop interface

```c
void wait_for_start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  while (s->state == stopped)
    pthread_cond_wait(&s->queue, &s->mutex);
  pthread_mutex_unlock(&s->mutex);
}

void start(state_t *s) {
  pthread_mutex_lock(&s->mutex);
  s->state = started;
  pthread_cond_broadcast(&s->queue);
  pthread_mutex_unlock(&s->mutex);
}
```
Start/Stop

• Start/Stop interface

```c
void wait_for_start(state_t *s) {
    pthread_mutex_lock(&s->mutex);
    while (s->state == stopped)
        pthread_cond_wait(&s->queue, &s->mutex);
    pthread_mutex_unlock(&s->mutex);
}

void start(state_t *s) {
    pthread_mutex_lock(&s->mutex);
    s->state = started;
    pthread_cond_broadcast(&s->queue);
    pthread_mutex_unlock(&s->mutex);
}
```

Quiz 2

You’re in charge of designing POSIX threads. Should `pthread_cond_wait` be a cancellation point?

a) no
b) yes; cancelled threads must acquire mutex before invoking cleanup handler
c) yes; but they don’t acquire mutex
Start/Stop

• Start/Stop interface

```c
void wait_for_start(state_t *s) {
    pthread_mutex_lock(&s->mutex);
    pthread_cleanup_push(
        pthread_mutex_unlock, &m);
    while (s->state == stopped)
        pthread_cond_wait(&s->queue, &s->mutex);
    pthread_cleanup_pop(1);
}

void start(state_t *s) {
    pthread_mutex_lock(&s->mutex);
    s->state = started;
    pthread_cond_broadcast(&s->queue);
    pthread_mutex_unlock(&s->mutex);
}
```
Cancellation and Conditions

```c
pthread_mutex_lock(&m);
pthread_cleanup_push(pthread_mutex_unlock, &m);
while (should_wait)
    pthread_cond_wait(&cv, &m);

// ... (code perhaps containing other cancellation points)

pthread_cleanup_pop(1);
```