

# CS 33

## Multithreaded Programming VI

# Cancellation



# Sample Code

```
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)

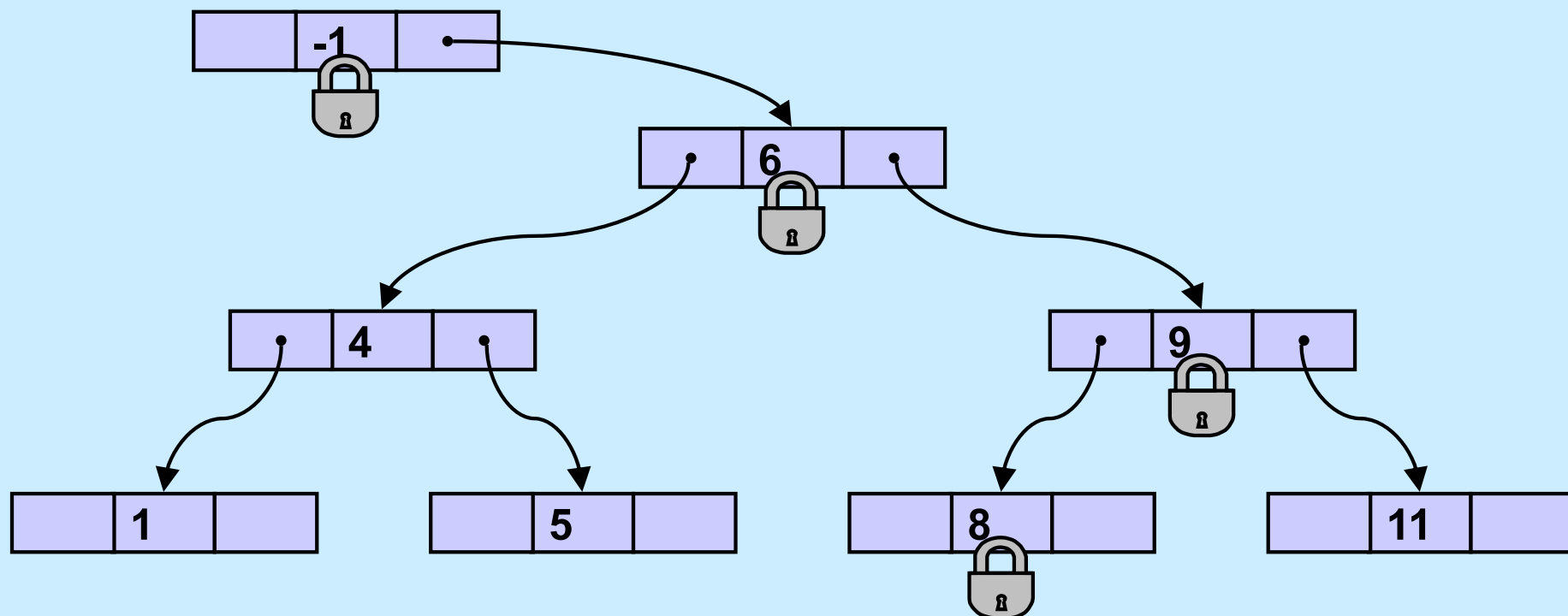
# Sample Code, Revisited

```
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 ...



# Start/Stop



- **Start/Stop interface**

```
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

```
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 3

You're in charge of designing POSIX threads. Should *pthread\_cond\_wait* be a cancellation point?

- a) no
- b) yes; cancelled threads must acquire the mutex before invoking cleanup handler
- c) yes; but they don't acquire mutex

# Start/Stop



- **Start/Stop interface**

```
void wait_for_start(state_t *s) {  
    pthread_mutex_lock(&s->mutex);  
    pthread_cleanup_push(  
        pthread_mutex_unlock, &s);  
    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

```
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);
```

# A Problem ...

- In thread 1:

```
if ((ret = open(path,  
    O_RDWR) == -1) {  
    if (errno == EINTR) {  
        ...  
    }  
    ...  
}
```

- In thread 2:

```
if ((ret = socket(AF_INET,  
    SOCK_STREAM, 0)) {  
    if (errno == ENOMEM) {  
        ...  
    }  
    ...  
}
```

**There's only one errno!**

**However, somehow it works.**

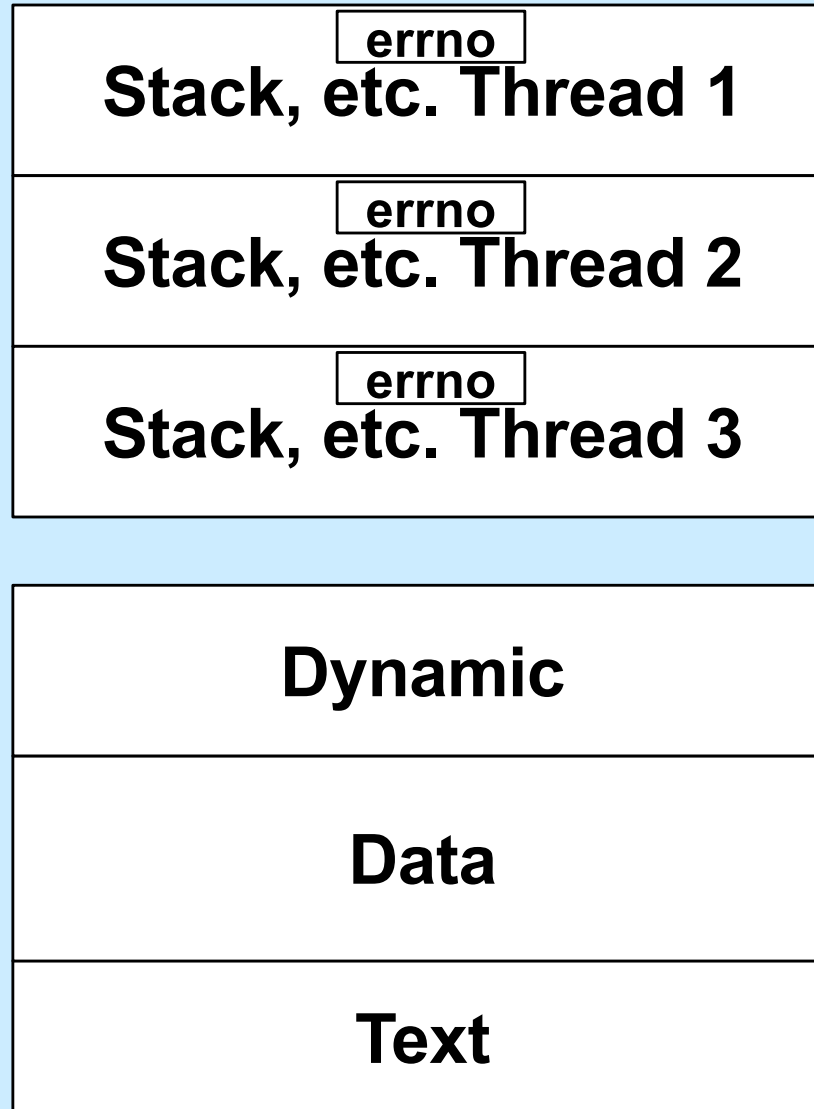
**What's done???**

# A Solution ...

```
#define errno (*__errno_location())
```

- **`__errno_location` returns an `int *` that's different for each thread**
  - thus each thread has, effectively, its own copy of `errno`

# Process Address Space





# Beyond POSIX

## TLS Extensions for ELF and gcc

- Thread Local Storage (TLS)

```
__thread int x=6;  
// Each thread has its own copy of x,  
// each initialized to 6.  
// Linker and compiler do the setup.  
// May be combined with static or extern.  
// Doesn't make sense for local variables!
```

# Example: Per-Thread Windows

```
typedef struct {
    wcontext_t win_context;
    int file_descriptor;
} win_t;

__thread static win_t my_win;

void getWindow() {
    my_win.win_context = ... ;
    my_win.file_descriptor = ... ;
}

int threadWrite(char *buf) {
    int status = write_to_window(
        &my_win, buf);

    return(status);
}
```

```
void *tfunc(void * arg) {
    getWindow();

    threadWrite("started");
    ...

    func2(...);
}
```

```
void func2(...) {
    threadWrite(
        "important msg");
    ...
}
```

# Thread Safety

- **Making the world safe for threads**

# Static Local Storage

```
char *strtok(char *str, const char *delim) {  
    static char *saveptr;  
  
    ... // find next token starting at either  
    ... // str or saveptr  
    ... // update saveptr  
  
    return (&token) ;  
}
```

# Coping

- **Use thread local storage**
- **Allocate storage internally; caller frees it**
- **Redesign the interface**

# Thread-Safe Version

```
char *strtok_r(char *str, const char *delim,  
               char **saveptr) {  
  
    ... // find next token starting at either  
    ... // str or *saveptr  
    ... // update *saveptr  
  
    return (&token) ;  
}
```

# Shared Data

- **Thread 1:**

```
printf("goto statement reached");
```

- **Thread 2:**

```
printf("Hello World\n");
```

- **Printed on display:**

```
go to Hell
```

# Coping

- **Wrap library calls with synchronization constructs**
- **Fix the libraries**



# Efficiency

- **Standard I/O example**

- `getc()` **and** `putc()`

- » **expensive and thread-safe?**

- » **cheap and not thread-safe?**

- **two versions**

- » `getc()` **and** `putc()`

- **expensive and thread-safe**

- » `getc_unlocked()` **and** `putc_unlocked()`

- **cheap and not thread-safe**

- **made thread-safe with** `flockfile()` **and** `funlockfile()`

# Efficiency

- **Naive**

```
for (i=0; i<lim; i++)  
    putc(out[i]);
```

- **Efficient**

```
flockfile(stdout);  
for (i=0; i<lim; i++)  
    putc_unlocked(out[i]);  
funlockfile(stdout);
```

# What's Thread-Safe?

- Everything except

asctime()	ecvt()	gethostent()	getutxline()	putc_unlocked()
basename()	encrypt()	getlogin()	gmtime()	putchar_unlocked()
catgets()	endgrent()	getnetbyaddr()	hcreate()	putenv()
crypt()	endpwent()	getnetbyname()	hdestroy()	pututxline()
ctime()	endutxent()	getnetent()	hsearch()	rand()
dbm_clearerr()	fcvt()	getopt()	inet_ntoa()	readdir()
dbm_close()	ftw()	getprotobyname()	l64a()	setenv()
dbm_delete()	gcvt()	getprotobyname()	lgamma()	setgrent()
dbm_error()	getc_unlocked()	getprotoent()	lgammaf()	setkey()
dbm_fetch()	getchar_unlocked()	getpwent()	lgammal()	setpwent()
dbm_firstkey()	getdate()	getpwnam()	localeconv()	setutxent()
dbm_nextkey()	getenv()	getpwuid()	localtime()	strerror()
dbm_open()	getgrent()	getservbyname()	lrand48()	strtok()
dbm_store()	getgrgid()	getservbyport()	mrnd48()	ttyname()
dirname()	getgrnam()	getservent()	nftw()	unsetenv()
dlderror()	gethostbyaddr()	getutxent()	nl_langinfo()	wcstombs()
drand48()	gethostbyname()	getutxid()	ptsname()	wctomb()

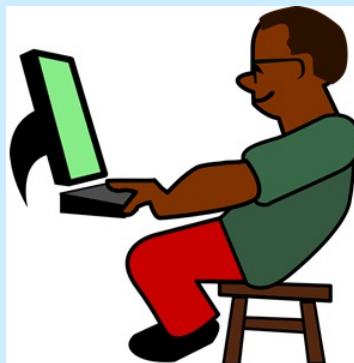
# You'll Soon Finish CS 33 ...

- You might
  - celebrate



- take another systems course

- » 32
    - » 138
    - » 166
    - » 167
    - » 168



- become a 33 TA



# Systems Courses Next Semester

- **CS 32 (Intro to Software Engineering)**
    - you've mastered low-level systems programming
    - now do things at a higher level
    - learn software-engineering techniques using Java, XML, etc.
  - **CS 138 (Distributed Systems)**
    - you now know how things work on one computer
    - what if you've got lots of computers?
    - some may have crashed, others may have been taken over by your worst (and smartest) enemy
  - **CS 166 (Computer Systems Security)**
    - liked buffer?
    - you'll really like 166
  - **CS 167/169/267 (Operating Systems)**
    - still mystified about what the OS does?
    - write your own!
-

# Systems Courses Next Semester

- **CS 168 (Computer Networks)**
  - intrigued by the little bit of networking we covered?
  - understand how it works and implement protocols yourself!

# The End

**Well, not quite ...  
Database is due on 12/17**

**Happy Coding and Happy Holidays!**