

CS 33

Multithreaded Programming IV

Quiz 1

```
void long_running_procedure( )  
{  
    pthread_mutex_lock(&m);  
    state = function(state);  
    pthread_mutex_unlock(&m);  
}
```

```
void display(state_t *statep)  
{  
    pthread_mutex_lock(&m);  
    print_state(statep)  
    pthread_mutex_unlock(&m);  
}
```

long_running_procedure is run by the main thread; ***display*** is run by the thread that is handling signals (via *sigwait*). Is there a potential deadlock resulting from their use of mutexes?

- a) No, since the functions are run by separate threads
- b) Yes, since ***display*** is called in response to a signal and thus uses the same stack as does the call to ***long_running_procedure***

Some Thread Gotchas ...

- **Exit vs. pthread_exit**
- **Handling multiple arguments**

Worker Threads

```
int main() {  
    pthread_t thread[10];  
    for (int i=0; i<10; i++)  
        pthread_create(&thread[i], 0,  
            worker, (void *)i);  
    return 0;  
}
```

Better Worker Threads

```
int main() {  
    pthread_t thread[10];  
    for (int i=0; i<10; i++)  
        pthread_create(&thread[i], 0,  
            worker, (void *)i);  
    pthread_exit(0);  
}
```

Multiple Arguments

```
void relay(int left, int right) {  
    pthread_t LRthread, RLthread;  
  
    pthread_create(&LRthread,  
        0,  
        copy,  
        left, right);           // Can't do this ...  
    pthread_create(&RLthread,  
        0,  
        copy,  
        right, left);           // Can't do this ...  
}
```

Multiple Arguments

```
typedef struct args {  
    int src;  
    int dest;  
} args_t;
```

```
void relay(int left, int right) {  
    args_t LRargs, RLargs;  
    pthread_t LRthread, RLthread;  
    ...  
    pthread_create(&LRthread, 0, copy, &LRargs);  
    pthread_create(&RLthread, 0, copy, &RLargs);  
    pthread_join(LRthread, 0);  
    pthread_join(RLthread, 0);  
}
```

Quiz 2

Does this work?

a) yes

b) no

Multiple Arguments

```
struct 2args {  
    int src;  
    int dest;  
} args;
```

```
void relay(int left, int right) {  
    pthread_t LRthread, RLthread;  
    args.src = left; args.dest = right;  
    pthread_create(&LRthread, 0, copy, &args);  
    args.src = right; args.dest = left;  
    pthread_create(&RLthread, 0, copy, &args);  
}
```

Quiz 3

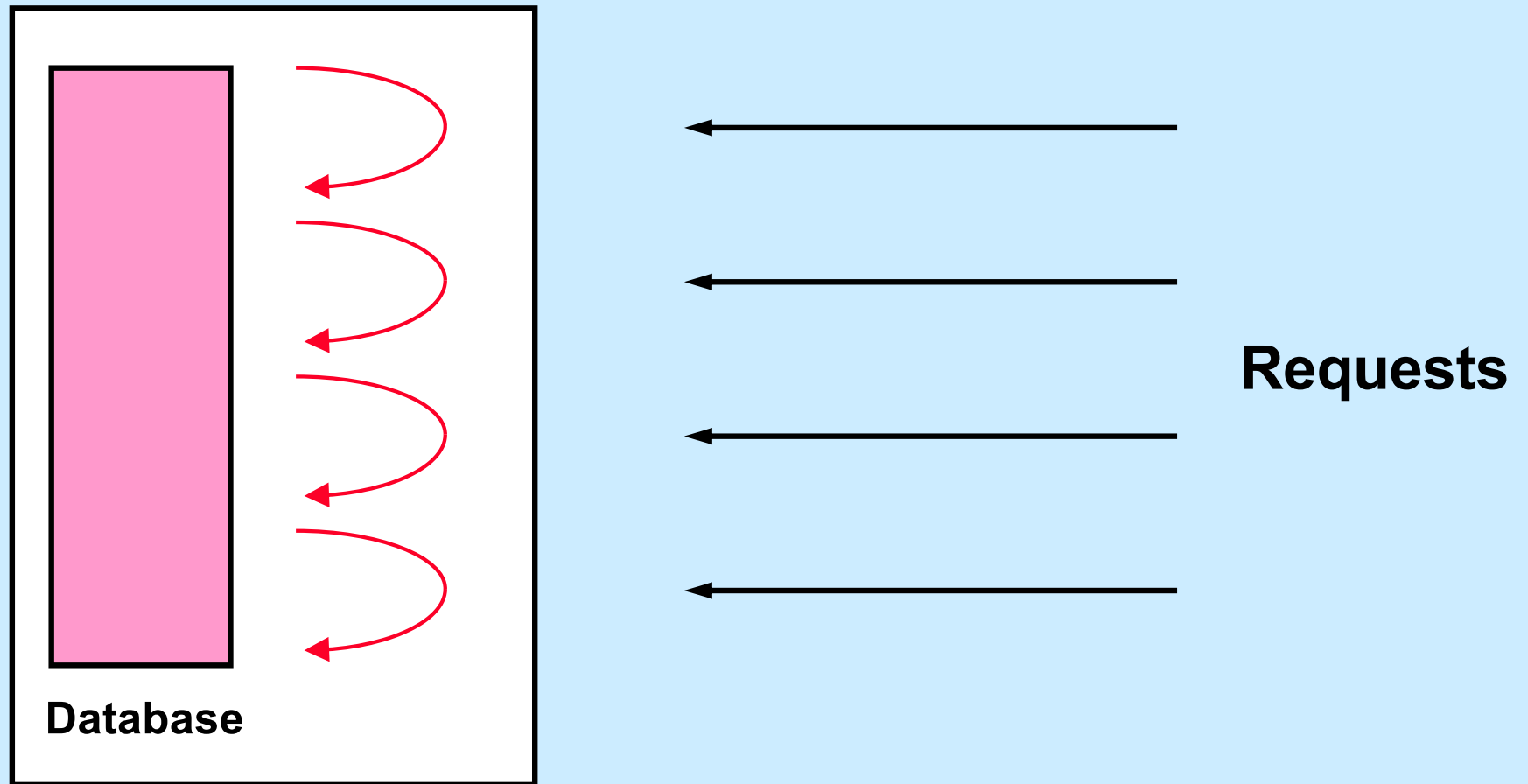
Does this work?

- a) yes**
- b) no**

Cancellation



Multithreaded Database Server



Sample Code

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

pthread_cancel(thread);

Quiz 2

```
1  void *thread_code(void *arg) {
2      node_t *head = 0;
3      while (1) {
4          node_t *nodep;
5          nodep = (node_t *)malloc(sizeof(node_t));
6          nodep->next = head;
7          head = nodep;
8          if (read(0, &node->value,
9                  sizeof(node->value)) == 0) {
10             free(nodep);
11             break;
12         }
13     }
14     return head;
15 }
```

Where is it safe to terminate a thread within *thread_code*?

- a) At all lines
- b) At all lines other than 5 and 9
- c) At all lines other than 8
- d) At all lines other than 5, 8, and 9
- e) At no lines

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)`

Sample Code – Cancellation Point

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

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

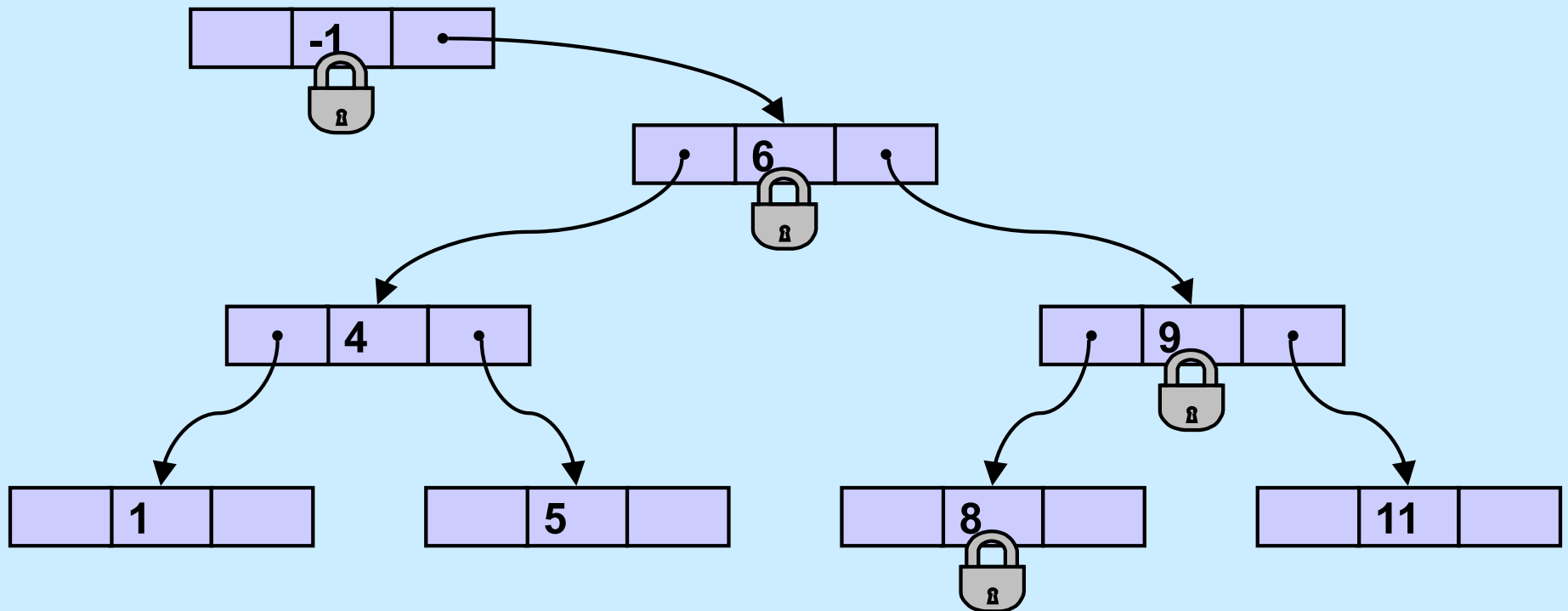
- **void** pthread_cleanup_push(**(void)** (*routine) (**void** *),
 void *arg)
- **void** pthread_cleanup_pop(**int** execute)

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));
        nodep->next = head;
        head = nodep;
        if (read(0, &nodep->value,
            sizeof(nodep->value)) == 0) {
            free(nodep);
            break;
        }
    }
    pthread_cleanup_pop(0);
    return head;
}
```

```
void cleanup(void *arg) {
    node_t **headp = arg;
    while(*headp) {
        node_t *nodep = headp->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);  
}
```



Not a Quiz

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

Cancellation and Conditions

```
pthread_mutex_lock(&m);  
pthread_cleanup_push(cleanup_handler, &m);  
while(should_wait)  
    pthread_cond_wait(&cv, &m);  
  
read(0, buffer, len);    // read is a cancellation point  
  
pthread_cleanup_pop(1);
```

Quiz 3

- **Start/Stop interface**

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

What should be used for *cleanup_func* and *cleanup_arg*?

- a) *pthread_mutex_unlock* and *&s->mutex*
- b) that and more
- c) there's no need for a cleanup function

Quiz 4

- **Start/Stop interface**

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

What should be used for *cleanup_func* and *cleanup_arg*?

- a) *pthread_mutex_unlock* and *&s->mutex*
- b) that and more
- c) there's no need for a cleanup function

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.

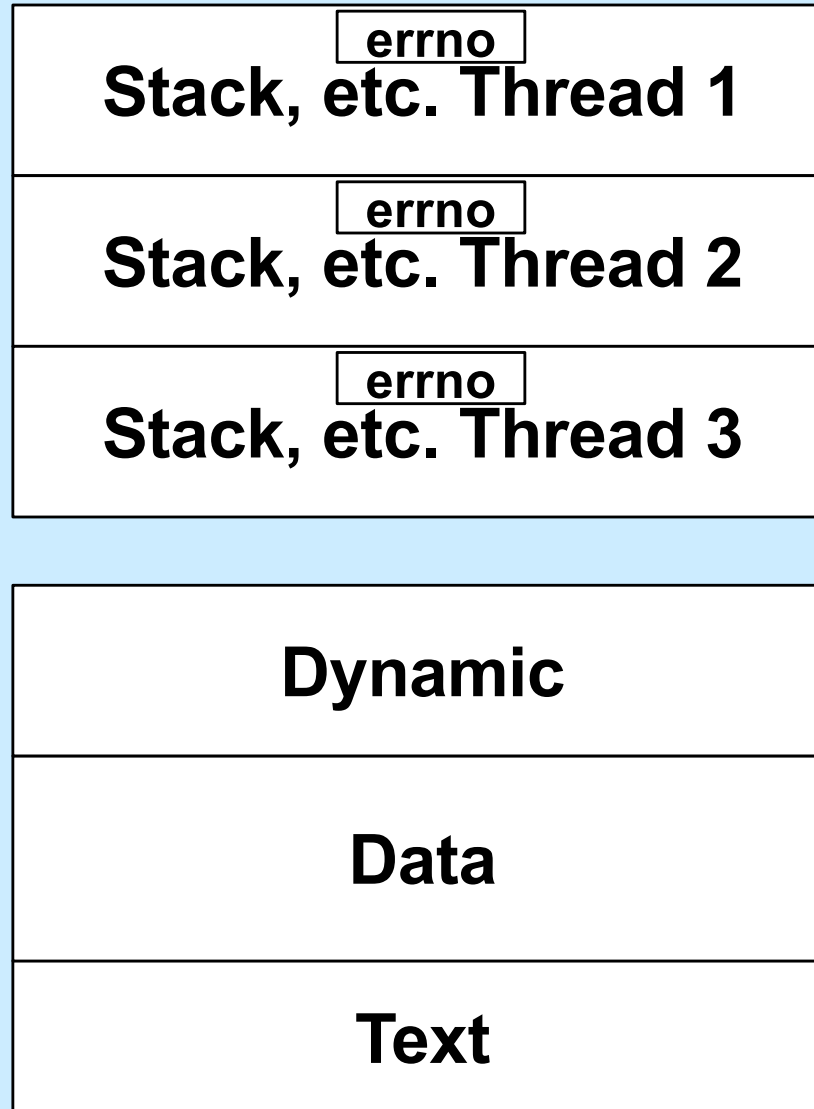
What's done???

A Solution ...

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

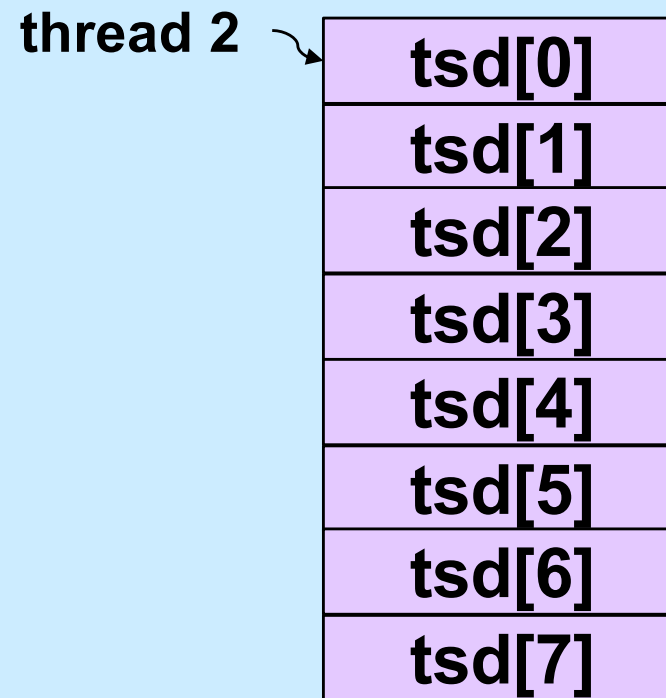
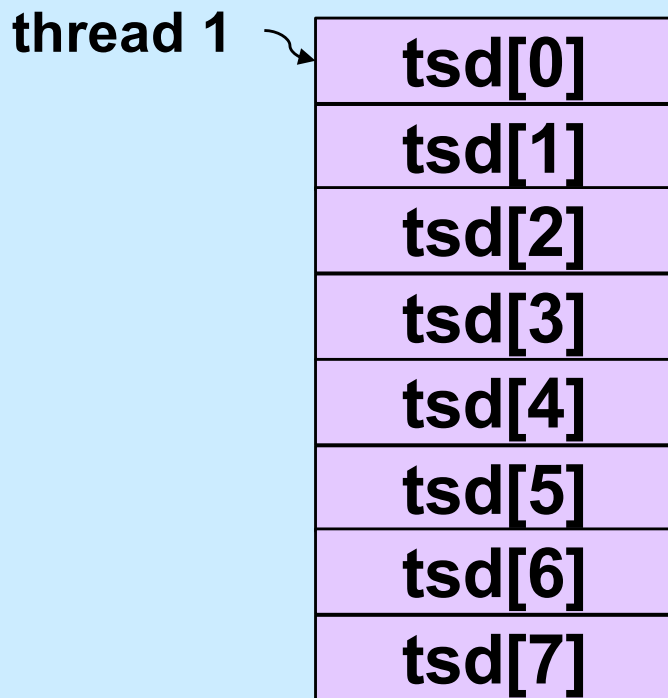
- **`__errno_location` returns an `int *` that's different for each thread**
 - thus each thread has, effectively, its own copy of `errno`

Process Address Space



Generalizing

- ***Thread-specific data*** (sometimes called ***thread-local storage***)
 - data that's referred to by global variables, but each thread has its own private copy



Some Machinery

- `pthread_key_create(&key, cleanup_routine)`
 - **allocates a slot in the TSD arrays**
 - **provides a function to cleanup when threads terminate**
- `value = pthread_getspecific(key)`
 - **fetches from the calling thread's array**
- `pthread_setspecific(key, value)`
 - **stores into the calling thread's array**

errno (Again)

```
// executed before threads are created
pthread_key_t errno_key;
pthread_key_create(&errno_key, NULL);

// redefine errno to use thread-specific value
#define errno pthread_getspecific(errno_key);

// set current thread's errno
pthread_set_specific(errno_key, (void *) ENOMEM);
```

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");
    ...
}
```


Static Local Storage and Threads

```
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()