CS 33

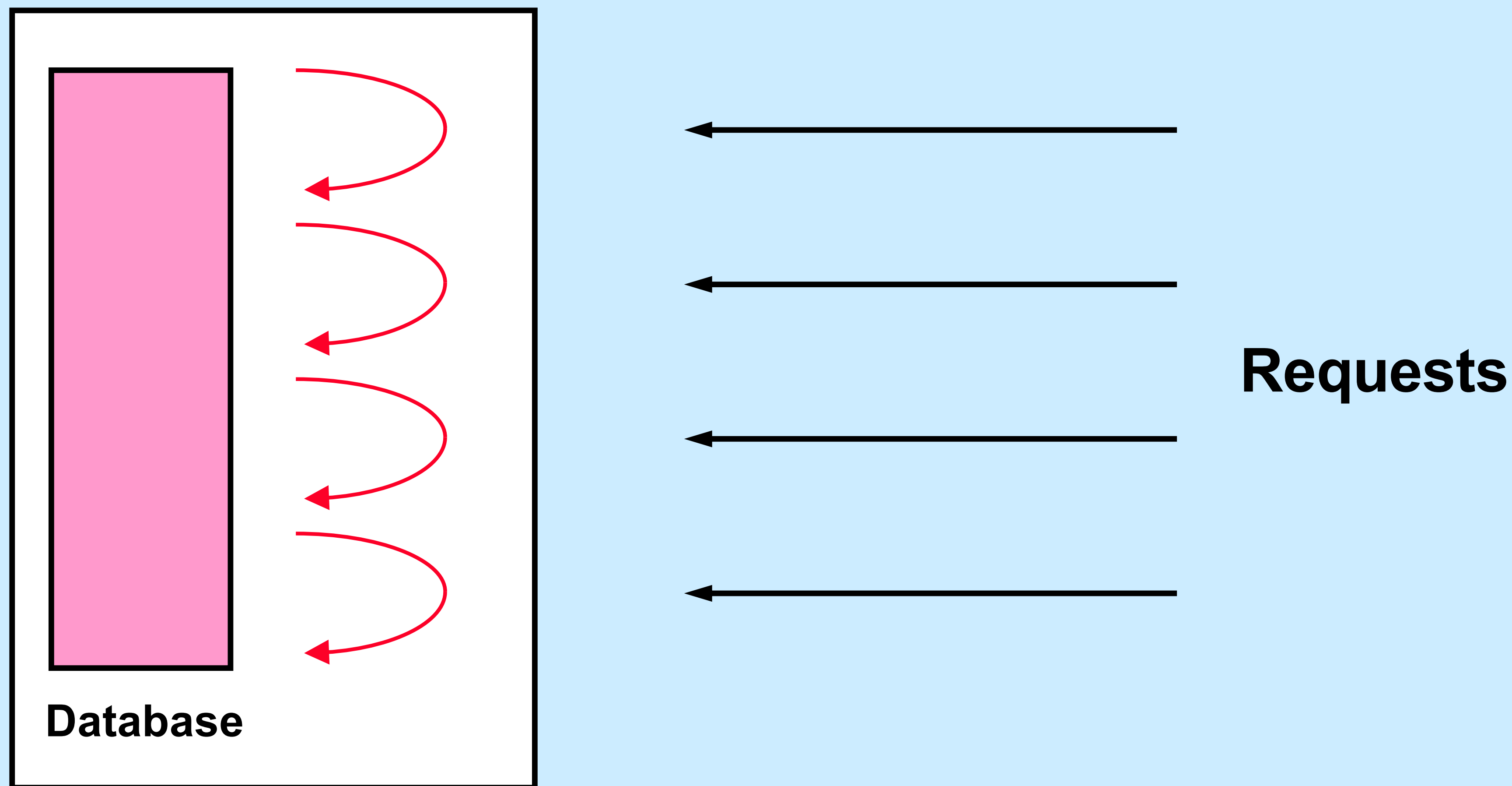
Multithreaded Programming IV

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Cancellation

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Multithreaded Database Server



 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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);
```


Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Quiz 1

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

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

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

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Cancellation Concerns

- Getting cancelled at an inopportune moment
- Cleaning up

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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)
    ```


Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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;
}
```


Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Cleaning Up

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

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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 = head->next;
    free(*headp);
    *headp = nodep;
  }
}
```

Quiz 2

This program will safely handle asynchronous cancels.

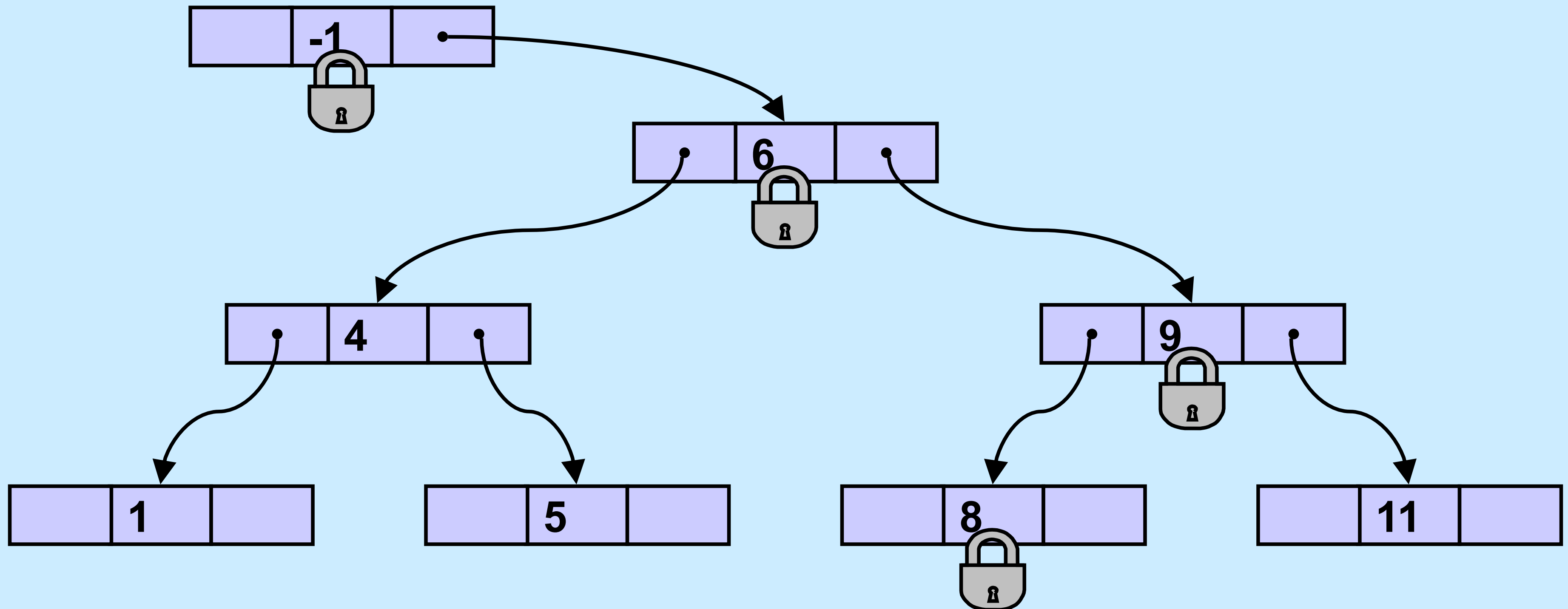
a) yes

b) yes, assuming thread-safe malloc and free

c) no

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

A More Complicated Situation …



 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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);
}
```

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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);
```


Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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???

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

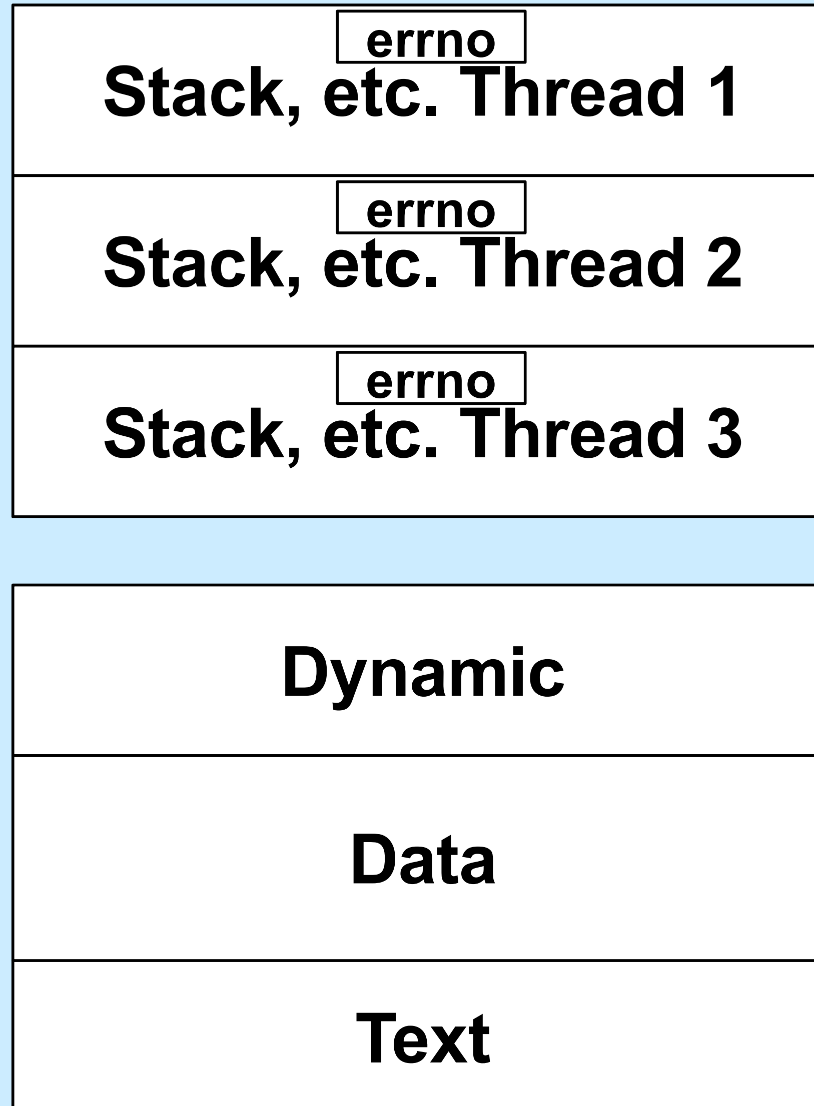
A Solution ...

```
#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**

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

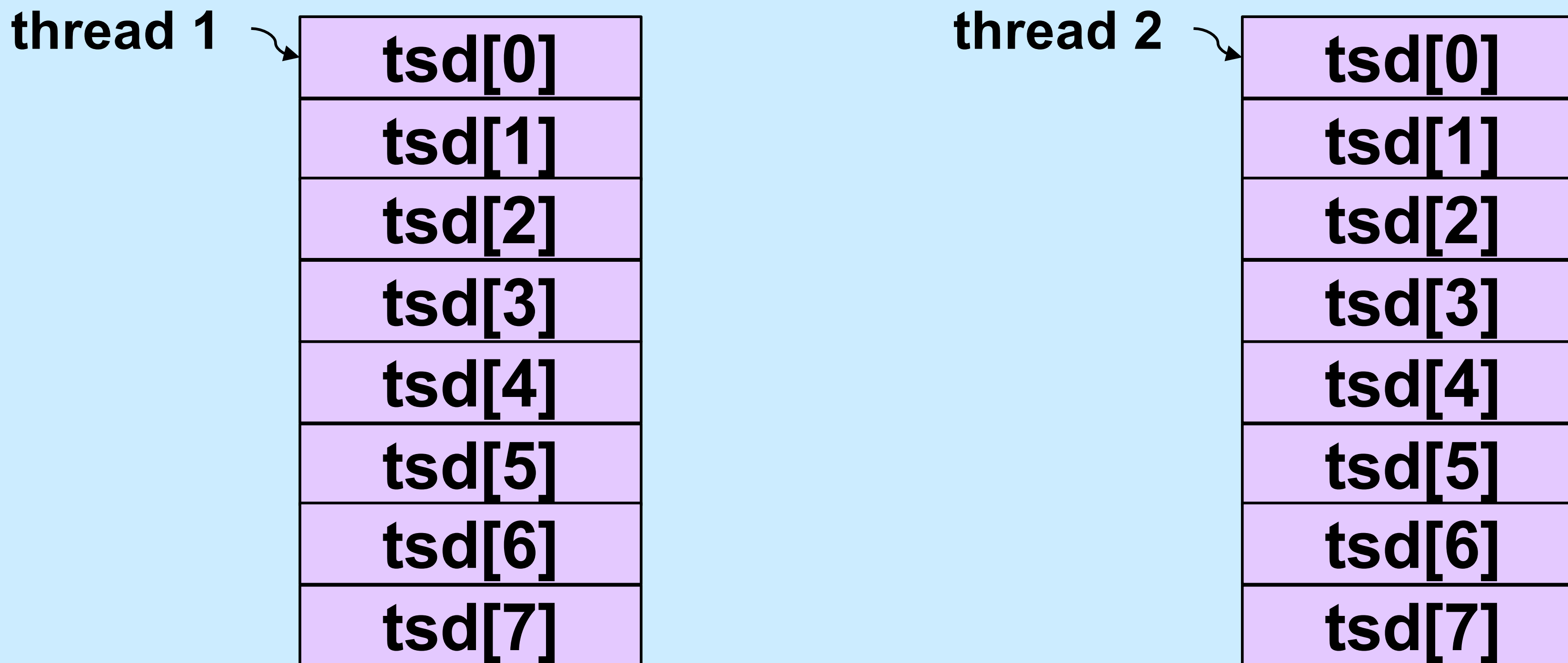
Process Address Space



 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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



Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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**

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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 (int)pthread_getspecific(errno_key);

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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Quiz 4

Earlier we saw that on Linux, errno is defined as

```
(*__errno_location())
```

This allows *errno* to be assigned to as well as read from. Could we arrange to do this using an implementation based on *pthread_getspecific* and *pthread_setspecific*?

a) No

b) Yes—easily

c) Yes—not so easily (it involves malloc and free)

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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.
  // x may be assigned to and copied from.
  // Linker and compiler do the setup.
  // May be combined with static or extern.
  // Doesn't make sense for local variables!
```


Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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_decriptor = … ;
}

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");
  …
}
```


Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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);
}
```


Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Coping

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

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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);
}
```


Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Shared Data

- **Thread 1:**

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

- **Thread 2:**

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

- **Printed on display:**

```
go to Hell
```

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

Coping

- Wrap library calls with synchronization constructs
- Fix the libraries

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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()`

Copyright © 2024 Thomas W. Doeppner. All rights reserved.

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);
```

 Copyright © 2024 Thomas W. Doeppner. All rights reserved.

What's Thread-Safe?

- **Everything except**

asctime()
basename()
catgets()
crypt()
ctime()
dbm_clearerr()
dbm_close()
dbm_delete()
dbm_error()
dbm_fetch()
dbm_firstkey()
dbm_nextkey()
dbm_open()
dbm_store()
dirname()
dlerror()
drand48()

ecvt()
encrypt()
endgrent()
endpwent()
endutxent()
fcvt()
ftw()
gcvt()
getc_unlocked()
getchar_unlocked()
getdate()
getenv()
getgrent()
getgrgid()
getgrnam()
gethostbyaddr()
gethostbyname()

gethostent()
getlogin()
getnetbyaddr()
getnetbyname()
getnetent()
getopt()
getprotobyname()
getprotobynumber()
getprotoent()
getpwent()
getpwnam()
getpwuid()
getservbyname()
getservbyport()
getservent()
getutxent()
getutxid()

getutxline()
gmtime()
hcreate()
hdestroy()
hsearch()
inet_ntoa()
l64a()
lgamma()
lgammaf()
lgammal()
localeconv()
localtime()
lrand48()
mrand48()
nftw()
nl_langinfo()
ptsname()

putc_unlocked()
putchar_unlocked()
putenv()
pututxline()
rand()
readdir()
setenv()
setgrent()
setkey()
setpwent()
setutxent()
strerror()
strtok()
ttyname()
unsetenv()
wcstombs()
wctomb()

Copyright © 2024 Thomas W. Doeppner. All rights reserved.