Multithreaded Programming V
Alternatives to Mutexes: Atomic Instructions

- Read-modify-write performed atomically
- *Lock prefix* may be used with certain IA32 and x86-64 instructions to make this happen
  - lock incr x
  - lock add $2, x
- It’s expensive
- It’s not portable
  - no POSIX-threads way of doing it
  - Windows supports
    » InterlockedIncrement
    » InterlockedDecrement
Alternatives to Mutexes:
Spin Locks

- Consider

```c
pthread_mutex_lock(&mutex);
new->next = list_ele->next;
list_ele->next = new;
pthread_mutex_unlock(&mutex);
```

- A lot of overhead is required to put thread to sleep, then wake it up
- Rather than do that, repeatedly test mutex until it’s unlocked, then lock it
  - makes sense only on multiprocessor system
Quiz 1

```c
pthread_mutex_lock(&mutex);
new->next = list_ele->next;
list_ele->next = new;
pthread_mutex_unlock(&mutex);
```

The thread that has locked the mutex might yield the processor to another thread due to time slicing. This will necessarily cause threads waiting for the mutex to wait longer. Will this result in those threads using significant additional processor time?

a) yes  
b) no
Compare and Exchange

cmpxchg src_reg, dest

- compare contents of %rax with contents of dest
  » if equal, then dest = src_reg (and ZF = 1)
  » otherwise %rax = dest (and ZF = 0)
Spin Lock

- the spin lock is pointed to by the first arg (%rdi)
  - locked is 1, unlocked is 0

```assembly
.text
.globl slock, sunlock

slock:
loop:
  movq $0, %rax
  movq $1, %r10
  lock cmpxchg %r10, 0(%rdi)
  jne loop
  ret

sunlock:
  movq $0, 0(%rdi)
  ret
```
**Improved Spin Lock**

```assembly
.text
.globl slock, sunlock

slock:
loop:
    cmp $0, 0(%rdi)  # compare using normal instructions
    jne loop
    movq $0, %rax
    movq $1, %r10
    lock cmpxchg %r10, 0(%rdi)  # verify w/ cmpxchg
    jne loop
    ret

sunlock:
    movq $0, 0(%rdi)
    ret
```
Yet More From POSIX …

```c
int pthread_spin_init(pthread_spin_t *s,
                int pshared);
int pthread_spin_destroy(pthread_spin_t *s);
int pthread_spin_lock(pthread_spin_t *s);
int pthread_spin_trylock(pthread_spin_t *s);
int pthread_spin_unlock(pthread_spin_t *s);
```
Using Spin Locks

```c
pthread_spin_lock(&spinlock);
new->next = list_ele->next;
list_ele->next = new;
pthread_spin_unlock(&spinlock);
```
pthreads

The thread that has locked the spin lock might yield the processor to another thread due to time slicing. This will necessarily cause threads waiting for the spin lock to wait longer. Will this result in those threads using significant additional processor time?

a) yes
b) no
A Problem ...

• In thread 1:

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

• In thread 2:

```c
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

```
<table>
<thead>
<tr>
<th>Stack, etc.</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>errno</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack, etc.</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>errno</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack, etc.</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>errno</code></td>
<td></td>
</tr>
</tbody>
</table>

Dynamic

Data

Text
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

<table>
<thead>
<tr>
<th>thread 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tsd[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread 2</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tsd[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[4]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>tsd[7]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
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

```c
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);
}
```

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

    threadWrite("started");

    ... 

    func2(...);
}
```

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

    ... 
}
```
Static Local Storage

```c
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

```c
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

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

• Efficient

```c
flockfile(stdout);
for (i=0; i<lim; i++)
   putc_unlocked(out[i]);
funlockfile(stdout);
```
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() dnsname() dlerror() drand48() ecvt() encrypt() endgrent() endpwent() endutxent() fcvt() ftw() gcvt() getc_unlocked() getchar_unlocked() getdate() getenv() getgrent() getgrgid() getgrnam() gethostbyaddr() gethostbyname() gethostent() gethostid() getlogin() getnetbyaddr() getnetbyname() getnetent() getopt() getprotobynumber() getprotoent() getpwent() getpwnam() getpwuid() getservbyname() getservbyport() getservent() getucontext() getutxent() getutxid() getutxline() getutxline() gmeno() hcreate() hdestroy() hsearch() inet_ntoa() l64a() lgamma() lgammaf() lgammal() localeconv() localtime() malloc() malloc() mrand48() nl_langinfo() ptsname() putc_unlocked() putchar_unlocked() putenv() pututxline() readdir() setenv() setgrent() setkey() setpwent() setutxent() strftime() strerror() strtok() ttyname() unsetenv() wcscrcards() wcstombw()
Fork and Threads

[Diagram showing fork and threads]

Or

[Diagram showing fork and threads]

Or
Processes vs. Threads

Process 1

Process 2

Process 3
Communicating via Shared Memory

Process 1

Shared Memory

Process 2
Cross-Process Synchronization

```c
pthread_mutexattr_t mattr;
pthread_condattr_t cattr;
pthread_mutex_t mut;
pthread_cond_t cond;

pthread_mutexattr_init(&mattr);
pthread_condattr_init(&cattr);

pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
pthread_condattr_setpshared(&cattr, PTHREAD_PROCESS_SHARED);

pthread_mutex_init(&mut, &mattr);
pthread_cond_init(&cond, &cattr);
```
Cross-Process Producer-Consumer (1)

typedef struct buffer {
    char buf[BSIZE];
    sem_t filled;
    sem_t empty;
    int nextin;
    int nextout;
} buffer_t;
Cross-Process Producer-Consumer (2)

```c
int main() {
    buffer_t *buffer;

    if ((buffer = mmap(0, sizeof(buffer_t), PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0)) == (void *)-1) {
        perror("mmap");
        exit(1);
    }
}
```
Cross-Process Producer-Consumer (3)

buffer->nextin = buffer->nextout = 0;
sem_init(&buffer->filled, 1, 0);
sem_init(&buffer->empty, 1, BSIZE);

if (fork() == 0)
    consumer_driver(buffer);
else
    producer_driver(buffer);

return 0;
}
Cross-Process Producer-Consumer (4)

```c
void producer_driver(
    buffer_t *b) {
    int item;

    while (1) {
        item = getchar();
        if (item == EOF) {
            produce(b, '\0');
            break;
        } else {
            produce(b, (char)item);
        }
    }
}
```

```c
void consumer_driver(
    buffer_t *b) {
    char item;

    while (1) {
        if ((item = consume(b)) == '\0')
            break;
        putchar(item);
    }
}
```
Cross-Process Producer-Consumer (5)

```c
void produce(buffer_t *b, char item) {
    sem_wait(&b->empty);
    b->buf[b->nextin] = item;
    b->nextin++;
    b->nextin %= BSIZE;
    sem_post(&b->filled);
}

char consume(buffer_t *b) {
    char item;
    sem_wait(&b->filled);
    item =
        b->buf[b->nextout];
    b->nextout++;
    b->nextout %= BSIZE;
    sem_post(&b->empty);
    return item;
}
```
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:

\[ i = \text{shared_counter} \]

Thread 2:

\[ \text{shared_counter}++ \]
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;
    }
}
Quiz 3

```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;
    }
    This works on sunlab machines.
    a) true
    b) false
}
```
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);
}
```
void producer(char item) {
    while (in - out == BSIZE);
    buf[in%BSIZE] = item;
    in++;
}

This works on sunlab machines.
a) true
b) false

c 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;
    }
}

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

L1 Cache

Cache Line

Tag | Data
False Sharing

L1 Cache

Tag | a | b

Cache Line

L1 Cache

Tag | a | b

Cache Line