Concurrency on x86-64; 
Threads Programming Tips

Brad Karp
UCL Computer Science

CS 0019
21st March 2019

(lecture notes derived from material from Eddie Kohler, David Mazières, Phil Gibbons, Dave O’Hallaron, and Randy Bryant)
Outline

- Implementing synchronization primitives on x86-64
  - Compare-and-swap (lock cmpxchg) instruction
  - x86-64 low-level behavior: what’s atomic, what isn’t?
- Gotchas to avoid when programming with pthreads
- Further Reading
How Do We Implement Synchronization in Assembly?

- Traditional assembly-level building block: compare-and-swap (CAS) instruction
  - Atomic
  - Can be used to build all synchronization primitives we’ve discussed

CAS intuition:
- Thread reads shared state from memory; this is **expected value** if no other thread modifies this memory “underneath” our thread
- Thread computes desired **updated value** for shared state (e.g., for counter, increment of counter)
- Thread executes CAS instruction which **atomically**
  - checks if memory still has expected value (no other thread has changed it)
  - iff yes, updates memory to updated value, returns “success”
  - if no, returns “failure”
How Do We Implement Synchronization in Assembly? (II)

- CAS definition (pseudocode):

```c
// hardware does entirety of below ATOMICALLY!
int CAS_instr(uint64_t *value, uint64_t expected, uint64_t desired) {
    if (*value == expected) {
        *value = desired;
        return 1;
    }
    else {
        return 0;
    }
}
```

- Typically call CAS in `while` loop: upon failure, reread `*value`; recompute `desired`, try again
- CAS is flexible: allows arbitrary computation on a memory value; each thread can do different computation on that value...
Abstract CAS Example: Add 17

```c
volatile uint64_t theval = 0;

void threadfunc(void *arg) {
    uint64_t *pv = (uint64_t *) arg;
    uint64_t expected, desired;

    while (1) {
        expected = *pv;
        desired = expected + 17;
        if (CAS_instr(pv, expected, desired))
            break;
    }
}

// [create N pthreads each running threadfunc(…, &theval)…]
```

- Upon return, `theval` will have been atomically increased by 17
- Loop retries if `theval` has changed underneath thread
**x86-64’s CAS: lock cmpxchg**

- Same functionality as CAS; slightly different interface
  - Returns prior value of shared memory location
  - Caller determines success/failure based on whether returned value equals expected
- As usual, different variants for different types (b, l, q)
- Pseudocode:

```c
// The entirety of the below is executed ATOMICALLY
TYPE cmpxchg(TYPE* object, TYPE expected, TYPE desired) {
    TYPE actual = *object;
    if (actual == expected)
        *object = desired;
    return actual;
}
```

- C11 offers `__atomic_compare_exchange_strong()` as portable API for CPU’s CAS-like instruction
x86-64 Assembly vs. Concurrency and Shared Memory

- Instructions with atomic effect include:
  - Loading memory → register an **aligned** {1-, 2-, 4-, 8-}byte value
  - Storing register → memory an **aligned** {1-, 2-, 4-, 8-}byte value

- Note, e.g., that if many threads concurrently store 8-byte aligned values to same address, no partial/”blended” results

- But beware: only true for aligned accesses; if memory spans cache-line boundary, no atomicity! (Why?)

- Instructions without atomic effect include:
  - `incq (%rax)`; increments 64-bit int stored at address in `%rax`
  - instruction in fact must load, increment, store
  - if target address aligned, load and store each have atomic effect, but set of three ops as a whole does not!
  - x86-64 instructions that include both reads and writes are **non-atomic by default**
x86-64 and Concurrency (cont’d)

- But building locks requires atomic instructions!
- x86-64 supports atomic instructions with optional `lock` prefix
  - “Locks the bus” during instruction’s execution (coordinates among cores and their caches; effectively does mutual exclusion in hardware for operations on target address)
  - e.g., `lock incq (%rax)` has atomic effect
- On x86-64, `lock cmpxchg` is “universal” building block for atomic functions
  - But N.B. that there are other instructions with atomic effect, and `lock cmpxchg` isn’t always the fastest alternative for all function implementations!
  - Achieving highest performance *and* desired app semantics for concurrent access to shared memory is tricky
  - Can even trade-off strength of semantics and performance (beyond scope of this lecture...)
Outline

- Implementing synchronization primitives on x86-64
- Gotchas to avoid when programming with pthreads
- Further Reading
Shared Variables in Threaded C Programs

Question: Which variables in a threaded C program are shared?
- The answer is not as simple as “global variables are shared’ and “stack variables are private”

Def: A variable $x$ is shared if and only if multiple threads reference some instance of $x$.

Requires answers to the following questions:
- What is the memory model for threads?
- How are instances of variables mapped to memory?
- How many threads might reference each of these instances?
Threads Memory Model: Conceptual

- Multiple threads run within the context of a single process
- Each thread has its own separate thread context
  - Thread ID, stack, stack pointer, PC, condition codes, and GP registers
- All threads share the remaining process context
  - Code, data, heap, and shared library segments of the process virtual address space
  - Open files and installed handlers

---

**Thread 1** (private)

- stack 1

**Thread 1 context:**
- Data registers
- Condition codes
- $SP_1$
- $PC_1$

**Thread 2** (private)

- stack 2

**Thread 2 context:**
- Data registers
- Condition codes
- $SP_2$
- $PC_2$

**Shared code and data**
- shared libraries
- run-time heap
- read/write data
- read-only code/data
Threads Memory Model: Actual

Thread 1 context:
- Data registers
- Condition codes
- SP$_1$
- PC$_1$

Thread 2 context:
- Data registers
- Condition codes
- SP$_2$
- PC$_2$

Virtual Address Space
- Shared code and data
  - shared libraries
  - run-time heap
  - read/write data
  - read-only code/data
Threads Memory Model: Actual

Thread 1 context:
- Data registers
- Condition codes
- \(SP_1\)
- \(PC_1\)

Thread 2 context:
- Data registers
- Condition codes
- \(SP_2\)
- \(PC_2\)

Virtual Address Space
- Shared code and data
  - shared libraries
  - run-time heap
  - read/write data
  - read-only code/data

(stack 1)

(stack 2)
Separation of data is not strictly enforced:
- Register values are truly separate and protected, but...
- Any thread can read and write the stack of any other thread

Mismatch between conceptual and operational models is source of confusion and errors!
Example Program to Illustrate Sharing

```c
char **ptr; /* global var */

int main(int argc, char *argv[]) {  
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL,
                       thread,
                       (void *)i);
    Pthread_exit(NULL);

    void *thread(void *vargp)
    {
        long myid = (long)vargp;
        static int cnt = 0;

        printf("[\%ld]:  %s (cnt=%d)\n",  
                myid, ptr[myid], ++cnt);
        return NULL;
    }

    sharing.c
```
Example Program to Illustrate Sharing

```c
char **ptr; /* global var */

int main(int argc, char *argv[]) {
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp) {
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

A common but inelegant way to pass a single argument to a thread routine
Example Program to Illustrate Sharing

```c
char **ptr; /* global var */

int main(int argc, char *argv[]) {
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL,
                        thread, (void *)i);

    Pthread_exit(NULL);
}
```

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

Peer threads reference main thread’s stack indirectly through global `ptr` variable

A common but inelegant way to pass a single argument to a thread routine
Mapping Variable Instances to Memory

- **Global variables**
  - *Def:* Variable declared outside of a function
  - Virtual memory contains exactly one instance of any global variable

- **Local variables**
  - *Def:* Variable declared inside function without `static` attribute
  - Each thread stack contains one instance of each local variable

- **Local static variables**
  - *Def:* Variable declared inside function with the `static` attribute
  - Virtual memory contains exactly one instance of any local static variable
Mapping Variable Instances to Memory

```c
char **ptr;  /* global var */

int main(int main, char *argv[]) {
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL,
                        thread,
                        (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp) {
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```

sharing.c
int main(int main, char *argv[]) {
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

Mapping Variable Instances to Memory

** Global var: 1 instance (ptr [data])

```
char **ptr; /* global var */

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

sharing.c
Mapping Variable Instances to Memory

**Global var:** 1 instance (ptr [data])

```c
char **ptr; /* global var */
```

**Local vars:** 1 instance (i.m, msgs.m)

```c
int main(int main, char *argv[]) {
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}
```

```c
void *thread(void *vargp) {
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

sharing.c
Mapping Variable Instances to Memory

**Global var:** 1 instance (ptr [data])

```c
char **ptr; /* global var */

int main(int main, char *argv[]) {
  long i;
  pthread_t tid;
  char *msgs[2] = {
    "Hello from foo",
    "Hello from bar"
  };
  ptr = msgs;
  for (i = 0; i < 2; i++)
    Pthread_create(&tid, NULL, thread, (void *)i);
  Pthread_exit(NULL);
}
```

**Local vars:** 1 instance (i.m, msgs.m)

**Local var:** 2 instances (myid.p0 [peer thread 0’s stack], myid.p1 [peer thread 1’s stack])

```c
void *thread(void *vargp)
{
  long myid = (long)vargp;
  static int cnt = 0;
  printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
  return NULL;
}
```

sharing.c
Mapping Variable Instances to Memory

*Global var:* 1 instance (`ptr [data]`)

*Local vars:* 1 instance (`i.m, msgs.m`)

```c
char **ptr; /* global var */

int main(int main, char **argv)
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL,
                        thread,
                        (void *)&i);

    Pthread_exit(NULL);
}
```

*Local var:* 2 instances (myid.p0 [peer thread 0’s stack], myid.p1 [peer thread 1’s stack])

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```

*Local static var:* 1 instance (`cnt [data]`)
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cnt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msgs.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */

int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo", "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cnt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msgs.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo", "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}
```

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>cnt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msgs.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                     "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL, thread,(void *)i);
    Pthread_exit(NULL);
}
```

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msgs.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                     "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL, thread,(void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
**Shared Variable Analysis**

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msgs.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo", "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL, thread,(void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>i.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msgs.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo", "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread,(void *)i);
    Pthread_exit(NULL);}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>msgs.m</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo", "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread,(void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

char **ptr; /* global var */

```c
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo", "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread,(void *)i);
    Pthread_exit(NULL);
}
```

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
## Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>myid.p0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                     "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL, thread,(void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
## Shared Variable Analysis

### Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>myid.p0</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>myid.p1</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                     "Hello from bar" };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL, thread,(void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp) {
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p0</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>myid.p1</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                     "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL, thread,(void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>myid.p0</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */

int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo", "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread,(void *)i);
    Pthread_exit(NULL);
}
```

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>myid.p0</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p1</td>
<td>yes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo",
                     "Hello from bar" };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                        NULL, thread,(void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
**Shared Variable Analysis**

■ Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m <em>(i)</em></td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m <em>(msgs)</em></td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>myid.p0</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */
int main(int main, char **argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar"};
  ptr = msgs;
  for (i = 0; i < 2; i++)
    Mutex_create(&tid,
                 NULL, thread,(void *)i);
  Pthread_exit(NULL);
}

void *thread(void *vargp)
{
  long myid = (long)vargp;
  static int cnt = 0;
  printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
  return NULL;
}
```
## Shared Variable Analysis

- Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p0</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"Hello from foo", "Hello from bar"};
  ptr = msgs;
  for (i = 0; i < 2; i++)
    Pthread_create(&tid,
                   NULL, thread,(void *)i);
  Pthread_exit(NULL);
}
```

```
void *thread(void *vargp)
{
  long myid = (long)vargp;
  static int cnt = 0;
  printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
  return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p0</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */
int main(int main, char *argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar"};

  ptr = msgs;
  for (i = 0; i < 2; i++)
    Pthread_create(&tid,
                   NULL, thread,(void *)i);
  Pthread_exit(NULL);
}

void *thread(void *vargp) {
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
  return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p0</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo", "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
                       NULL, thread,(void *)&i);
    Pthread_exit(NULL);
}
```

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n",
            myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>myid.p0</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

```
char **ptr; /* global var */
int main(int main, char *argv[]) {
    long i; pthread_t tid;
    char *msgs[2] = {"Hello from foo", "Hello from bar"};
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread,(void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;
    printf("[%ld]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p0</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

```c
char **ptr; /* global var */
int main(int main, char *argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"Hello from foo",
                   "Hello from bar" };

  ptr = msgs;
  for (i = 0; i < 2; i++)
    Pthread_create(&tid,
                   NULL, thread,(void *)i);
  Pthread_exit(NULL);
}
```

```c
void *thread(void *vargp)
{
  long myid = (long)vargp;
  static int cnt = 0;

  printf("[%ld]: %s (cnt=%d)\n",
         myid, ptr[myid], ++cnt);
  return NULL;
}
```
Shared Variable Analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable instance</th>
<th>Referenced by main thread?</th>
<th>Referenced by peer thread 0?</th>
<th>Referenced by peer thread 1?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>cnt</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>i.m</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>msgs.m</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>myid.p0</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>myid.p1</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Answer: A variable \( x \) is shared iff multiple threads reference at least one instance of \( x \). Thus:

- \( \text{ptr, cnt, and msgs are shared} \)
- \( \text{i and myid are not shared} \)

Moral: \textit{beware unintended sharing; take care to synchronize all shared accesses!}
Crucial Concept: Thread Safety

- Functions called from a thread must be thread-safe

Def: A function is thread-safe iff it will always produce correct results when called repeatedly from multiple concurrent threads

- Classes of thread-unsafe functions:
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer to a static variable
  - Class 4: Functions that call thread-unsafe functions
Thread-Unsafe Functions (Class 1)

- Failing to protect shared variables
  - Fix: use synchronization primitives, e.g., spinlock, mutex, condition variable...
  - Pitfall: synchronization operations invariably slow down code
Thread-Unsafe Functions
(Class 2)

Relying on persistent state across multiple function invocations

- Example: Random number generator that relies on static state

```c
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```
Thread-Safe Random Number Generator

- Fix: pass state as argument
  - and thereby eliminate static state

```
/* rand_r - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- Consequence: programmer using `rand_r()` must maintain seed
Thread-Unsafe Functions (Class 3)

- Returning pointer to static variable
  - Fix 1: Rewrite function so caller passes *address of variable* to store result
    - Changes in caller and callee
  - Fix 2: Lock-and-copy
    - Simple changes in caller (and none in callee)
    - However, caller must free memory

```c
/* Convert integer to string */
char *itoa(int x)
{
    static char buf[11];
    sprintf(buf, "%d", x);
    return buf;
}
```

Warning: Some functions like `gethostbyname()` require a *deep copy*. Use reentrant `gethostbyname_r()` version instead.
Thread-Unsafe Functions (Class 3)

- Returning pointer to static variable
- Fix 1: Rewrite function so caller passes address of variable to store result
  - Changes in caller and callee
- Fix 2: Lock-and-copy
  - Simple changes in caller (and none in callee)
  - However, caller must free memory

```c
/* Convert integer to string */
char *itoa(int x)
{
    static char buf[11];
    sprintf(buf, "%d", x);
    return buf;
}
```

```c
char *lc_itoa(int x, char *dest)
{
    pthread_lock_mutex(&mutex);
    strcpy(dest, itoa(x));
    pthread_unlock_mutex(&mutex);
    return dest;
}
```

Warning: Some functions like `gethostbyname()` require a deep copy. Use reentrant `gethostbyname_r()` version instead.
Thread-Unsafe Functions (Class 4)

- Calling thread-unsafe functions
  - Calling one thread-unsafe function makes the entire calling function thread-unsafe

- Fix: Modify function so it calls only thread-safe functions 😊
Reentrant Functions

- Def: A function is reentrant iff it accesses no shared variables when called by multiple threads
  - Important subset of thread-safe functions
    - Require no synchronization operations
    - Only way to make a Class 2 function thread-safe is to make it reentrant (e.g., `rand_r`)

<table>
<thead>
<tr>
<th>All functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread-safe functions</td>
</tr>
<tr>
<td>Reentrant functions</td>
</tr>
<tr>
<td>Thread-unsafe functions</td>
</tr>
</tbody>
</table>
Thread-Safe Library Functions

- All functions in the Standard C Library (at the back of K&R text) are thread-safe
  - Examples: malloc(), free(), printf(), scanf()

- Most UNIX-specific C library calls are thread-safe, with a few exceptions:

<table>
<thead>
<tr>
<th>Thread-unsafe function</th>
<th>Class</th>
<th>Reentrant version</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime()</td>
<td>3</td>
<td>asctime_r()</td>
</tr>
<tr>
<td>ctime()</td>
<td>3</td>
<td>ctime_r()</td>
</tr>
<tr>
<td>gethostbyaddr()</td>
<td>3</td>
<td>gethostbyaddr_r()</td>
</tr>
<tr>
<td>gethostbyname()</td>
<td>3</td>
<td>gethostbyname_r()</td>
</tr>
<tr>
<td>inet_ntoa()</td>
<td>3</td>
<td>(none)</td>
</tr>
<tr>
<td>localtime()</td>
<td>3</td>
<td>localtime_r()</td>
</tr>
<tr>
<td>rand()</td>
<td>2</td>
<td>rand_r()</td>
</tr>
</tbody>
</table>
Further Reading (unexaminable)

- Intel’s weak memory semantics are subtle, and known as Total Store Ordering (TSO); full details explored in Sewell’s:
  - [http://www.cl.cam.ac.uk/~pes20/weakmemory/cacm.pdf](http://www.cl.cam.ac.uk/~pes20/weakmemory/cacm.pdf)
- The scalability of lock implementations, especially in kernel code, is of great import to performance; a classic exploration from MIT’s PDOS group, showing performance howlers in Linux:
- C11 supports a vast array of options for specifying the memory semantics you want Atomics to use; an informative exploration (for C++11) by Preshing:
- Designing high-performance applications that run on many cores and share memory is hard, and an ongoing systems research topic. Kohler et al.’s Masstree is an impressively fast (key, value) store for multi-core machines: