Questions and answers:

- What is the memory model for threads?
- How are variables mapped to memory instances?
- How many threads reference each of these instances?

Example of shared variables in a threaded C program:

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

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

/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
    peer threads access main thread's stack indirectly through global ptr variable

    char **ptr; /* global */

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

    /* thread routine */
    void *thread(void *vargp)
    {
        int myid = (int)vargp;
        static int cnt = 0;
        printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
        peer threads access main thread's stack indirectly through global ptr variable
```
Mapping variables to memory instances

Global var: 1 instance (ptr [data])
Local automatic vars: 1 instance (i.m, msgs.m)
Local automatic vars: 2 instances (myid.p0[peer thread 0's stack], myid.pl[peer thread 1's stack])
Local static var: 1 instance (cnt [data])

char **ptr; /* global */

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

/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;
    printf("[%d]: %s (cnt=%d)\n",
           myid, ptr[myid], ++cnt);
}

Shared variable analysis

Which variables are shared?

<table>
<thead>
<tr>
<th>Variable</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.pl</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:
- ptr, cnt, and msgs are shared.
- i and myid are NOT shared.

badcnt.c: An improperly synchronized threaded program

unsigned int cnt = 0; /* shared */

int main()
{
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    if (cnt != (unsigned)NITERS*2)
        printf("BOOM!  cnt=%d\n", cnt);
    else
        printf("OK  cnt=%d\n", cnt);
}

Assembly code for counter loop

C code for counter loop

for (i=0; i<NITERS; i++)
    ctr++;;

Corresponding asm code (gcc -O0 -fforce-mem)

Head (H)
Load ctr (L)
Update ctr (U)
Store ctr (S)

Tail (T)
Concurrent execution

Key idea: In general, any sequentially consistent interleaving is possible, but some are incorrect!
- $i$ denotes that thread $i$ executes instruction $I$
- %eax$_i$ is the contents of %eax in thread $i$'s context

```
<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr$_i$</th>
<th>%eax$_i$</th>
<th>%eax$_2$</th>
<th>ctr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H$_1$</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L$_1$</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U$_1$</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S$_1$</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>H$_2$</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>L$_2$</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U$_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>T$_2$</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>T$_1$</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>
```

OK

Concurrent execution (cont)

Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2.

```
<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr$_i$</th>
<th>%eax$_i$, %eax$_2$</th>
<th>ctr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H$_1$</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L$_1$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U$_1$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H$_2$</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L$_2$</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S$_1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T$_1$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U$_2$</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S$_2$</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T$_2$</td>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>
```

Oops!

Concurrent execution (cont)

How about this ordering?

```
<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr$_i$</th>
<th>%eax$_i$, %eax$_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H$_1$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>L$_1$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>H$_2$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>L$_2$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>U$_2$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S$_2$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>U$_1$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>S$_1$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>T$_1$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>T$_2$</td>
<td></td>
</tr>
</tbody>
</table>
```

We can clarify our understanding of concurrent execution with the help of the progress graph

Progress graphs

A progress graph depicts the discrete execution state space of concurrent threads.

Each axis corresponds to the sequential order of instructions in a thread.

Each point corresponds to a possible execution state (Inst$_1$, Inst$_2$).

E.g., (L$_1$, S$_2$) denotes state where thread 1 has completed L$_1$ and thread 2 has completed S$_2$.
Trajectories in progress graphs

A trajectory is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:
H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical sections and unsafe regions

L, U, and S form a critical section with respect to the shared variable cnt.

Instructions in critical sections (wrt to some shared variable) should not be interleaved.

Sets of states where such interleaving occurs form unsafe regions.

Safe and unsafe trajectories

Def: A trajectory is safe iff it doesn't touch any part of an unsafe region.

Claim: A trajectory is correct (wrt cnt) iff it is safe.

Synchronizing with semaphores

Question: How can we guarantee a safe trajectory?

- We must synchronize the threads so that they never enter an unsafe state.

Classic solution: Dijkstra's P and V operations on semaphores.

- semaphore: non-negative integer synchronization variable.
  - P(s); [while (s == 0) wait(); s--; ]
  - Dutch for "Proberen" (test)
  - V(s); [s++; ]
  - Dutch for "Verhogen" (increment)

- OS guarantees that operations between brackets [] are executed indivisibly.
  - Only one P or V operation at a time can modify s.
  - When while loop in P terminates, only that P can decrement s.

Semaphore invariant: (s >= 0)
Safe sharing with semaphores

Provide mutually exclusive access to shared variable by surrounding critical section with P and V operations on semaphore (initially set to 1).

Semaphore invariant creates a forbidden region that encloses unsafe region and is never touched by any trajectory.

POSIX semaphores

Provide mutually exclusive access to shared variable by surrounding critical section with P and V operations on semaphore (initially set to 1).

Semaphore invariant creates a forbidden region that encloses unsafe region and is never touched by any trajectory.

Sharing with POSIX semaphores

```c
/* goodcnt.c - properly synced counter program */
#include "csapp.h"
#define NITERS 10000000

unsigned int cnt; /* counter */
sem_t sem;        /* semaphore */

int main() {
    pthread_t tid1, tid2;
    Sem_init(&sem, 0, 1);
    /* create 2 threads and wait */
    ... if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
    exit(0);
}
```

Signaling with semaphores

Common synchronization pattern:

- Producer waits for slot, inserts item in buffer, and “signals” consumer.
- Consumer waits for item, removes it from buffer, and “signals” producer.

“signals” in this context has nothing to do with Unix signals

Examples

- Multimedia processing:
  - Producer creates MPEG video frames, consumer renders the frames
- Event-driven graphical user interfaces
  - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer.
  - Consumer retrieves events from buffer and paints the display.
### Producer-consumer (1-buffer)

```c
#include "csapp.h"

#define NITERS 5

struct {
    int buf; /* shared var */
    sem_t full; /* sems */
    sem_t empty;
} shared;

int main() {
    pthread_t tid_producer;
    pthread_t tid_consumer;
    Sem_init(&shared.empty, 0, 1);
    Sem_init(&shared.full, 0, 0);
    Pthread_create(&tid_producer, NULL, producer, NULL);
    Pthread_create(&tid_consumer, NULL, consumer, NULL);
    Pthread_join(tid_producer, NULL);
    Pthread_join(tid_consumer, NULL);
    exit(0);
}
```

#### Producer thread
```c
void *producer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* produce item */
        item = i;
        printf("produced %d\n", item);
        /* write item to buf */
        P(&shared.empty);
        shared.buf = item;
        V(&shared.full);
    }
    return NULL;
}
```

#### Consumer thread
```c
void *consumer(void *arg) {
    int i, item;

    for (i=0; i<NITERS; i++) {
        /* read item from buf */
        P(&shared.full);
        item = shared.buf;
        V(&shared.empty);
        /* consume item */
        printf("consumed %d\n", item);
    }
    return NULL;
}
```

---

### Limitations of semaphores

Semaphores are sound and fundamental, but they have limitations.

- Difficult to broadcast a signal to a group of threads.
  - e.g., barrier synchronization: no thread returns from the barrier function until every other thread has called the barrier function.
- Impossible to do timeout waiting.
  - e.g., wait for at most 1 second for a condition to become true.

For these we must use Pthreads **mutex** and **condition variables**.

---

### Synchronizing with mutex and condition variables

Semaphores can be used for two different kinds of synchronization:

- Safe sharing (e.g. `goodcnt.c`) and
- Signaling (e.g. `prodcons.c`).

Pthreads interface provides two different mechanisms for these functions:

- Safe sharing: operations on mutexes.
- Signaling: operations on condition variables
  - discussed in text
### Basic operations on mutex variables

- **Initializes a mutex variable** (`mutex`) with some attributes (`attr`).
  - attributes are usually NULL.
  - like initializing a mutex semaphore to 1.

```c
int pthread_mutex_init(pthread_mutex_t *mutex,
                 pthread_mutexattr_t *attr)
```

- **Indivisibly waits for** `mutex` **to be unlocked and then locks it.**
  - like `P(mutex)`

```c
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- **Unlocks** `mutex`.
  - like `V(mutex)`

```c
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

### Thread-safe functions

**Functions called from a thread must be thread-safe.**

We identify four (non-disjoint) classes of thread-unsafe functions:

- **Class 1: Failing to protect shared variables.**
- **Class 2: Relying on persistent state across invocations.**
- **Class 3: Returning a pointer to a static variable.**
- **Class 4: Calling thread-unsafe functions.**

### Thread-unsafe functions

**Class 1: Failing to protect shared variables.**

- Fix: use Pthreads lock/unlock functions or P/V operations.
- Issue: synchronization operations will slow down code.
- Example: `goodcnt.c`

### Thread-safe functions (cont)

**Class 2: Relying on persistent state across multiple function invocations.**

- The `my_read()` function called by `readline()` buffers input in a static array.

```c
static ssize_t my_read(int fd, char *ptr)
{
    static int read_cnt = 0;
    static char *read_ptr,
    static char *read_buf[MAXLINE];
    ...
}
```

- Fix: Rewrite function so that caller passes in all necessary state.

```c
static ssize_t my_read_r(Rline *rptr, char *ptr)
{
    ... 
}
```
Thread-safe functions (cont)

Class 3: Returning a pointer to a static variable.

- **Fixes:**
  - 1. Rewrite so caller passes pointer to struct.
    - Issue: Requires changes in caller and callee.
  - 2. "Lock-and-copy"
    - Issue: Requires only simple changes in caller (and none in callee)
    - However, caller must free memory.

```c
struct hostent
  *gethostbyname(char name)
  {
  static struct hostent h;
  <contact DNS and fill in h>
  return &h;
  }

dest = malloc(...);
gethostbyname_r(name, dest);
```

Thread-safe functions

Class 4: Calling thread-unsafe functions.

- **Fix:** Modify the function so it calls only thread-safe functions
  - Example: `readline_r()` is a thread-safe version of `readline()` that calls the thread-safe `my_read_r()` function.

Reentrant functions

A function is **reentrant** iff it accesses NO shared variables when called from multiple threads.

- Reentrant functions are a proper subset of the set of thread-safe functions.

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

- **NOTE:** The fixes to Class 2 and 3 thread-unsafe functions require modifying the function to make it reentrant.

Thread-safe library functions

All functions in the Standard C Library (at the back of your K&R text) are thread-safe.

Most Unix system 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><code>asctime</code></td>
<td>3</td>
<td><code>asctime_r</code></td>
</tr>
<tr>
<td><code>ctime</code></td>
<td>3</td>
<td><code>ctime_r</code></td>
</tr>
<tr>
<td><code>gethostbyaddr</code></td>
<td>3</td>
<td><code>gethostbyaddr_r</code></td>
</tr>
<tr>
<td><code>gethostbyname</code></td>
<td>3</td>
<td><code>gethostbyname_r</code></td>
</tr>
<tr>
<td><code>inet_ntoa</code></td>
<td>3</td>
<td>(none)</td>
</tr>
<tr>
<td><code>localtime</code></td>
<td>3</td>
<td><code>localtime_r</code></td>
</tr>
<tr>
<td><code>rand</code></td>
<td>2</td>
<td><code>rand_r</code></td>
</tr>
</tbody>
</table>
Races

A race occurs when the correctness of the program depends on one thread reaching point x before another thread reaches point y.

```c
/* a threaded program with a race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        pthread_join(tid[i], NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

Threads summary

Threads provide another mechanism for writing concurrent programs.

Threads are growing in popularity

- Somewhat cheaper than processes.
- Easy to share data between threads.

However, the ease of sharing has a cost:

- Easy to introduce subtle synchronization errors.
- Tread carefully with threads!

For more info:


Deadlock

Locking introduces the potential for deadlock: waiting for a condition that will never be true.

Any trajectory that enters the deadlock region will eventually reach the deadlock state, waiting for either s or t to become nonzero.

Other trajectories luck out and skirt the deadlock region.

Unfortunate fact: deadlock is often non-deterministic.