Synchronization: Advanced
Enforcing Mutual Exclusion

**Question:** How can we guarantee a safe trajectory?

**Answer:** We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.

- i.e., need to guarantee *mutually exclusive access* for each critical section.

**Classic solution:**
- Semaphores (Edsger Dijkstra)

**Other approaches (out of our scope):**
- Mutex and condition variables (Pthreads)
- Monitors (Java)
Semaphores

- **Semaphore**: non-negative global integer synchronization variable. Manipulated by $P$ and $V$ operations.

- $P(s)$
  - If $s$ is nonzero, then decrement $s$ by 1 and return immediately.
    - Test and decrement operations occur atomically (indivisibly)
  - If $s$ is zero, then suspend thread until $s$ becomes nonzero and the thread is restarted by a $V$ operation.
  - After restarting, the $P$ operation decrements $s$ and returns control to the caller.

- $V(s)$:
  - Increment $s$ by 1.
    - Increment operation occurs atomically
  - If there are any threads blocked in a $P$ operation waiting for $s$ to become non-zero, then restart exactly one of those threads, which then completes its $P$ operation by decrementing $s$.

- **Semaphore invariant**: $(s \geq 0)$
C Semaphore Operations

Pthreads functions:

```c
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */
int sem_wait(sem_t *s); /* P(s) */
int sem_post(sem_t *s); /* V(s) */
```

CS:APP wrapper functions:

```c
#include "csapp.h"

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```
badcnt.c: Improper Synchronization

/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL, thread, &niters);
    Pthread_create(&tid2, NULL, thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}

/* Thread routine */
void *thread(void *vargp)
{
    long i, niters =
    *((long *)vargp);

    for (i = 0; i < niters; i++)
        cnt++;

    return NULL;
}

How can we fix this using semaphores?

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Using Semaphores for Mutual Exclusion

Basic idea:
- Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables).
- Surround corresponding critical sections with $P(mutex)$ and $V(mutex)$ operations.

Terminology:
- **Binary semaphore**: semaphore whose value is always 0 or 1
- **Mutex**: binary semaphore used for mutual exclusion
  - P operation: “locking” the mutex
  - V operation: “unlocking” or “releasing” the mutex
  - “Holding” a mutex: locked and not yet unlocked.
- **Counting semaphore**: used as a counter for set of available resources.
**goodcnt.c: Proper Synchronization**

- Define and initialize a mutex for the shared variable `cnt`:

  ```c
  volatile long cnt = 0; /* Counter */
  sem_t mutex;         /* Semaphore that protects cnt */
  Sem_init(&mutex, 0, 1); /* mutex = 1 */
  ```

- Surround critical section with `P` and `V`:

  ```c
  for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
  }
  ```

  ```
  linux> ./goodcnt 10000
  OK cnt=20000
  linux> ./goodcnt 10000
  OK cnt=20000
  ```

*Warning: It’s orders of magnitude slower than badcnt.c.*

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Why Mutexes Work

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

Semaphore invariant creates a forbidden region that encloses unsafe region and that cannot be entered by any trajectory.

Initially \( s = 1 \)
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 😊

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Thread-Unsafe Functions (Class 1)

- **Failing to protect shared variables**
  - Fix: Use \( P \) and \( V \) semaphore operations
  - Example: `goodcnt.c`
  - Issue: Synchronization operations will 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

- Pass state as part of argument
  - and, thereby, eliminate global state

```c
/* 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 a pointer to a static variable
- Fix 1. Rewrite function so caller passes address of variable to store result
  - Requires changes in caller and callee
- Fix 2. Lock-and-copy
  - Requires simple changes in caller (and none in callee)
  - However, caller must free memory.

```c
/* lock-and-copy version */
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```
Thread-Unsafe Functions (Class 4)

- Calling thread-unsafe functions
  - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - Fix: Modify the 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`)

All functions

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

Adapted from slides by R. Bryant and D. O’Hallaron ([http://csapp.cs.cmu.edu/3e/instructors.html](http://csapp.cs.cmu.edu/3e/instructors.html))
Thread-Safe Library Functions

- All functions in the Standard C Library are thread-safe
  - Examples: `malloc`, `free`, `printf`, `scanf`
- 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>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>
One worry: Races

- A **race** occurs when 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;
}
```

N threads are sharing $i$.
Race Illustration

```c
for (i = 0; i < N; i++)
    Pthread_create(&tid[i], NULL, thread, &i);
```

- **Main thread**
  - `i = 0`
  - `i = 1`

- **Peer thread 0**
  - `myid = *((int *)vargp)`

---

Race between increment of `i` in main thread and deref of `vargp` in peer thread:

- If deref happens while `i = 0`, then OK
- Otherwise, peer thread gets wrong id value

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Could this race really occur?

### Main thread
```c
int i;
for (i = 0; i < 100; i++) {
    Pthread_create(&tid, NULL, thread,&i);
}
```

### Peer thread
```c
void *thread(void *vargp) {
    Pthread_detach(pthread_self());
    int i = *((int *)vargp);
    save_value(i);
    return NULL;
}
```

#### Race Test
- If no race, then each thread would get different value of `i`
- Set of saved values would consist of one copy each of 0 through 99

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Experimental Results

No Race

Single core laptop

Multicore server

The race can really happen!

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Race Elimination

/* Threaded program without the race */
int main()
{
    pthread_t tid[N];
    int i, *ptr;

    for (i = 0; i < N; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

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

Avoid unintended sharing of state
Another worry: Deadlock

- Def: A process is **deadlocked** iff it is waiting for a condition that will never be true

- Typical Scenario
  - Processes 1 and 2 needs two resources (A and B) to proceed
  - Process 1 acquires A, waits for B
  - Process 2 acquires B, waits for A
  - Both will wait forever!
Deadlocking With Semaphores

```c
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}

void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:
- P(s0);
- P(s1);
- cnt++;
- V(s0);
- V(s1);

Tid[1]:
- P(s1);
- P(s0);
- cnt++;
- V(s1);
- V(s0);
Deadlock Visualized in Progress Graph

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 $S_0$ or $S_1$ to become nonzero.

Other trajectories luck out and skirt the deadlock region.

Unfortunate fact: deadlock is often nondeterministic (race)

$S_0 = S_1 = 1$
Avoiding Deadlock  

Acquire shared resources in same order

```c
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1); /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1); /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}

void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:
P(s0);
P(s1);
cnt++;
V(s0);
V(s0);

Tid[1]:
P(s0);
P(s1);
cnt++;
V(s1);
V(s0);
Avoided Deadlock in Progress Graph

Processes acquire locks in same order

Order in which locks released immaterial

No way for trajectory to get stuck