Systems Programming

Synchronization: Advanced

Byoungyoung Lee

Seoul National University

byoungyoung@snu.ac.kr

https://lifeasageek.github.io

Lecture slides are prepared based on materials provided by CSAPP authors and James Bornholt's CSE 451 materials.

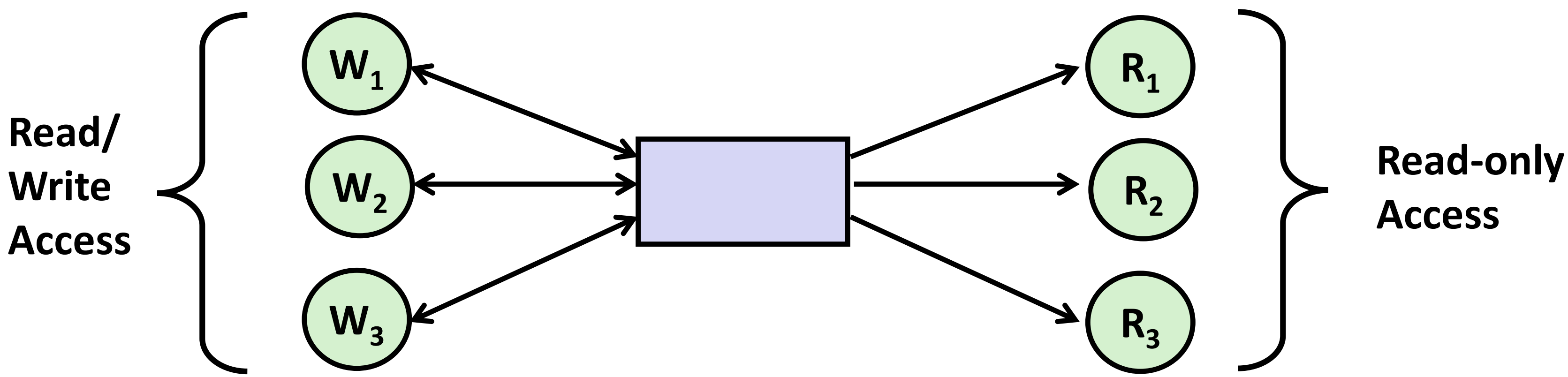
Note about Examples

- **Lecture examples will use semaphores for both counting and mutual exclusion**
  - Code is much shorter than using pthread_mutex

Today

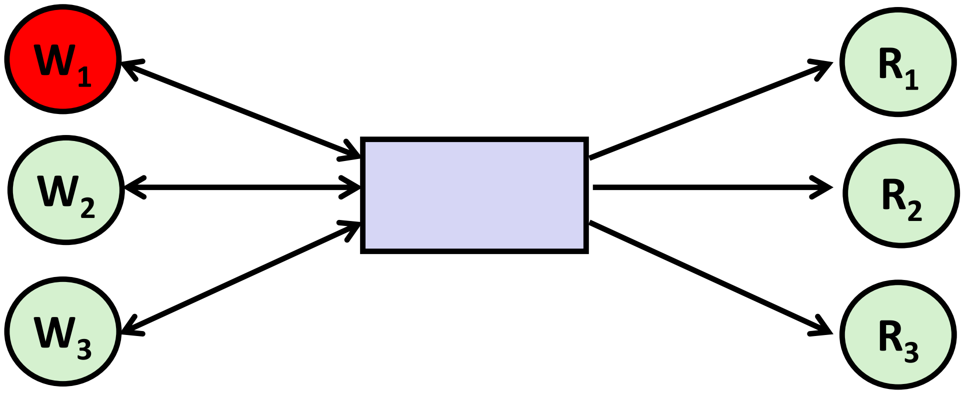
- **Using semaphores to schedule shared resources**
  - **Readers-writers problem**
- Other concurrency issues
  - Thread safety
  - Races
  - Deadlocks
  - Interactions between threads and signal handling

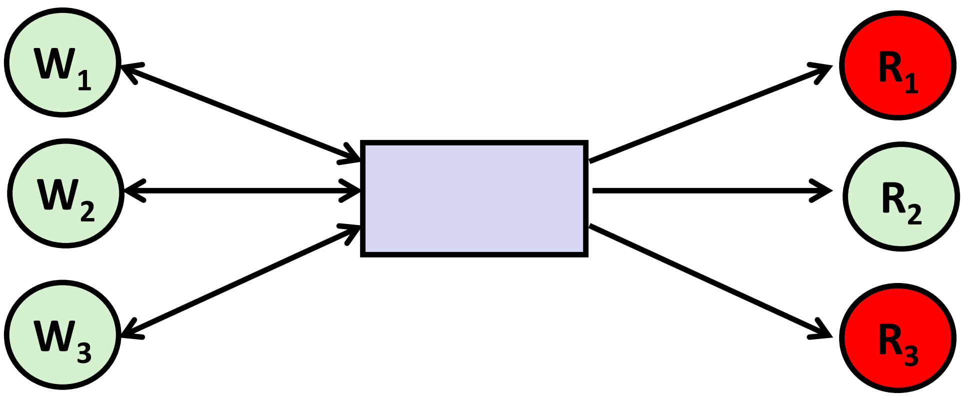
Readers-Writers Problem



- **Problem statement:**
  - *Reader* threads only read the object
  - *Writer* threads modify the object (read/write access)
  - Writers must have exclusive access to the object
  - Unlimited number of readers can access the object
- **Occurs frequently in real systems, e.g.,**
  - Online airline reservation system
  - Multithreaded caching Web proxy

Readers/Writers Examples





Variants of Readers-Writers

- ***First readers-writers problem*** **(favors readers)**
  - No reader should be kept waiting unless a writer has already been granted permission to use the object.
  - A reader that arrives after a waiting writer gets priority over the writer.

- ***Second readers-writers problem*** **(favors writers)**
  - Once a writer is ready to write, it performs its write as soon as possible
  - A reader that arrives after a writer must wait, even if the writer is also waiting.

- ***Starvation*** **(where a thread waits indefinitely) is possible in both cases.**

Solution to First Readers-Writers Problem

Readers:

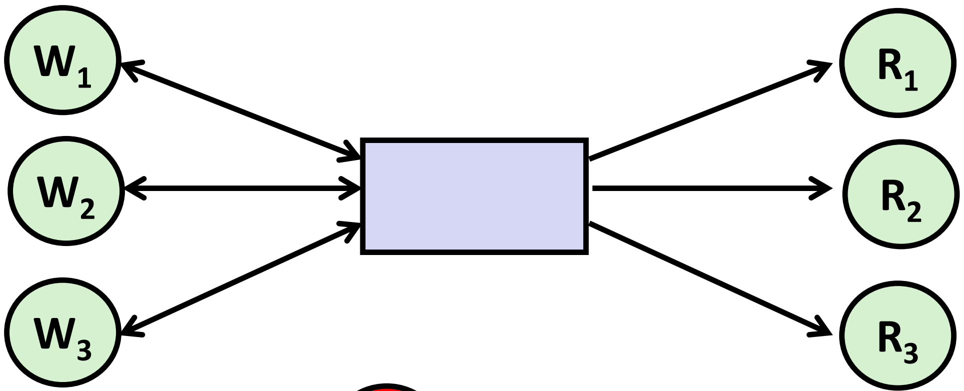
```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    WAIT(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      WAIT(&w);       /* Take the
                   priority over writer */
    POST(&mutex);

    /* Reading happens here */

    WAIT(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      POST(&w);
    POST(&mutex);
  }
}
```

Writers:

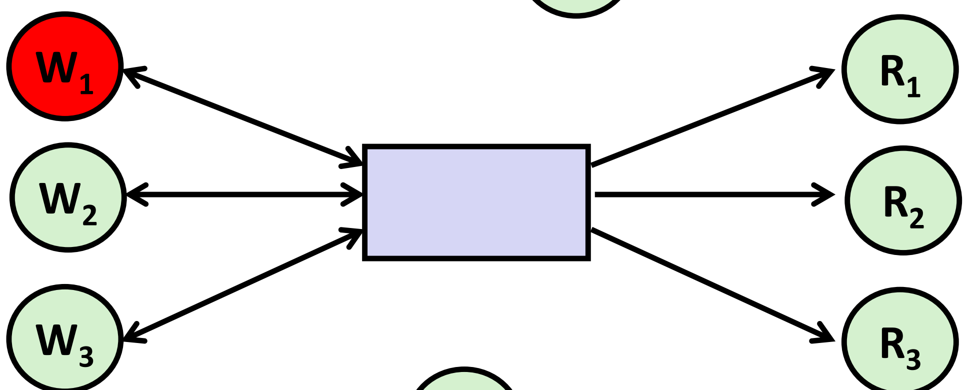
```
void writer(void)
{
  while (1) {
    WAIT(&w);

    /* Writing here */

    POST(&w);
  }
}
```

rw1.c

Readers/Writers Examples

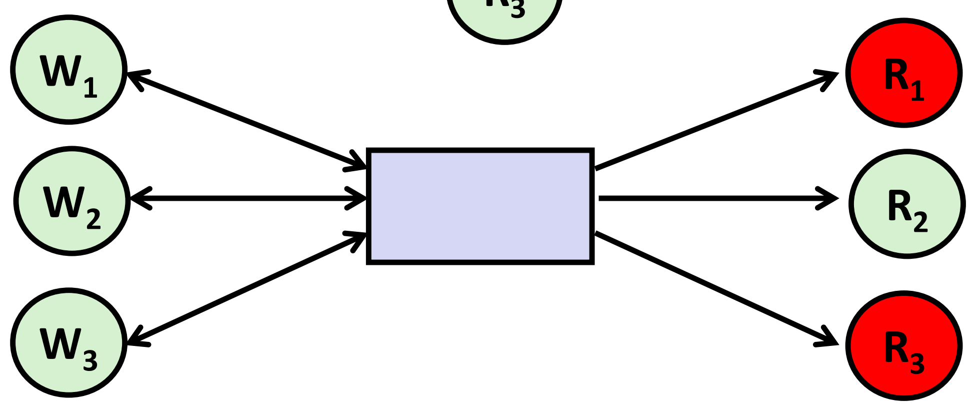


w = 1
readcnt = 0



w = 0
readcnt = 0

w = 0
readcnt = 2



Solution to First Readers-Writers Problem

Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    WAIT(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      WAIT(&w);
    POST(&mutex);

    /* Reading happens here */

    WAIT(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      POST(&w);
    POST(&mutex);
  }
}
```

Writers:

```
void writer(void)
{
  while (1) {
    WAIT(&w);

    /* Writing here */

    POST(&w);
  }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Q. what's the processing order?

Solution to First Readers-Writers Problem

Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    WAIT(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      WAIT(&w);
    POST(&mutex);

R1 → /* Reading happens here */

    WAIT(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      POST(&w);
    POST(&mutex);
  }
}
```

Writers:

```
void writer(void)
{
  while (1) {
    WAIT(&w);

    /* Writing here */

    POST(&w);
  }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0

Solution to First Readers-Writers Problem

Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    WAIT(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      WAIT(&w);
    POST(&mutex);

    /* Reading happens here */

    WAIT(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      POST(&w);
    POST(&mutex);
  }
}
```

R2 → (at `if (readcnt == 1)`)

R1 → (at `/* Reading happens here */`)

Writers:

```
void writer(void)
{
  while (1) {
    WAIT(&w);

    /* Writing here */

    POST(&w);
  }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0

Solution to First Readers-Writers Problem

Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    WAIT(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      WAIT(&w);
    POST(&mutex);
R2 →
R1 →
    /* Reading happens here */

    WAIT(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      POST(&w);
    POST(&mutex);
  }
}
```

Writers:

```
void writer(void)
{
  while (1) {    ← W1
    WAIT(&w);

    /* Writing here */

    POST(&w);
  }
}
```

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0

Solution to First Readers-Writers Problem

Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    WAIT(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      WAIT(&w);
    POST(&mutex);

    /* Reading happens here */

    WAIT(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      POST(&w);
    POST(&mutex);
  }
}
```

R2 → /* Reading happens here */

R1 → }

Writers:

```
void writer(void)
{
  while (1) {
    WAIT(&w);

    /* Writing here */

    POST(&w);
  }
}
```

W1 → WAIT(&w);

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0

Solution to First Readers-Writers Problem

Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    WAIT(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      WAIT(&w);
    POST(&mutex);

    /* Reading happens here */

    WAIT(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      POST(&w);
    POST(&mutex);
  }
}
```

R3 → `if (readcnt == 1)`

R2 → before `WAIT(&mutex);` (after reading)

R1 → closing `}` of while loop

Writers:

```
void writer(void)
{
  while (1) {
    WAIT(&w);

    /* Writing here */

    POST(&w);
  }
}
```

W1 ← `WAIT(&w);`

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 2
W == 0

Solution to First Readers-Writers Problem

Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    WAIT(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      WAIT(&w);
    POST(&mutex);

    /* Reading happens here */

    WAIT(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      POST(&w);
    POST(&mutex);
  }
}
```

R3 → (at "Reading happens here")

R2 → (at final POST(&mutex))

Writers:

```
void writer(void)
{
  while (1) {
    WAIT(&w);

    /* Writing here */

    POST(&w);
  }
}
```

W1 → (at WAIT(&w))

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 1
W == 0

Solution to First Readers-Writers Problem

Readers:

```
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
  while (1) {
    WAIT(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      WAIT(&w);
    POST(&mutex);

    /* Reading happens here */

    WAIT(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      POST(&w);
    POST(&mutex);
  }
}
```

R3 → (at `POST(&mutex);`)

Writers:

```
void writer(void)
{
  while (1) {
    WAIT(&w);

    /* Writing here */

    POST(&w);
  }
}
```

W1 → (at `WAIT(&w);`)

rw1.c

Arrivals: R1 R2 W1 R3

Readcnt == 0
W == 1

Today

- **Using semaphores to schedule shared resources**
  - Readers-writers problem
- **Other concurrency issues**
  - **Races**
  - Deadlocks
  - Thread safety
  - Interactions between threads and signal handling

One Worry: Races

- A *race* occurs when correctness depends on the orders of thread execution

```
/* a threaded program with a race */
int main(int argc, char** argv) {
    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);
    return 0;
}

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

race.c

Race example: CVE-2019-2025

Thread 1 [binder_transaction()]	Thread 2 [binder_thread_write()]
t->buffer=binder_alloc_new_buf();	
	if(t->buffer->allow_user_free == 1) – (1)
t->buffer->allow_user_free = 0 -- (2)	
	binder_free_buf(proc, t->buffer) – (3)
copy_from_user(t->buffer->data, user, size) -- (4)	

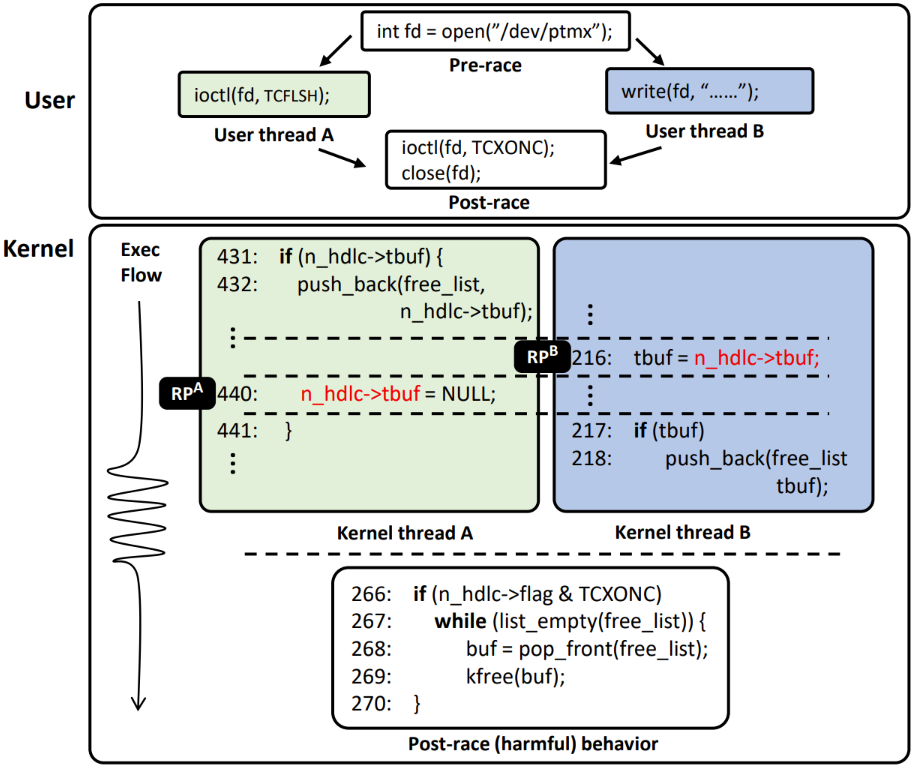
Race Elimination

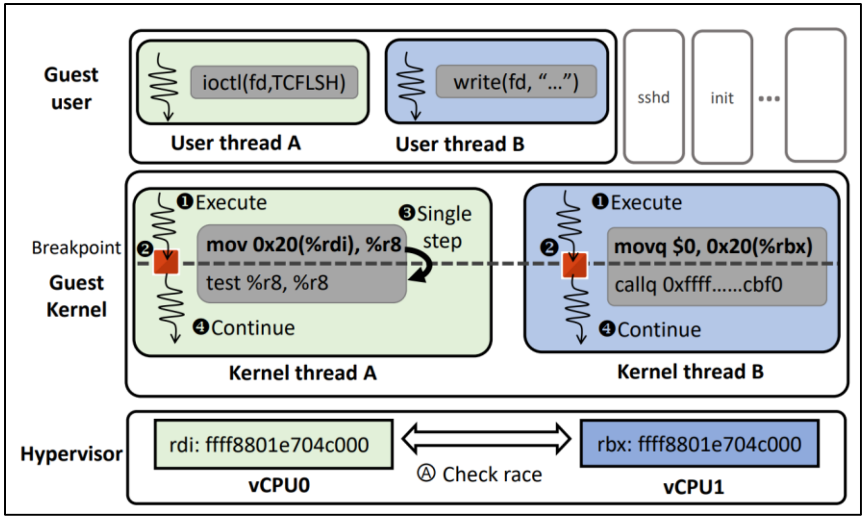
- **Don't share state**
  - E.g., use malloc to generate separate copy of argument for each thread

- **Use synchronization primitives to control access to shared state**
  - Each shared variable may use individual mutex/semaphore.

Race Detection

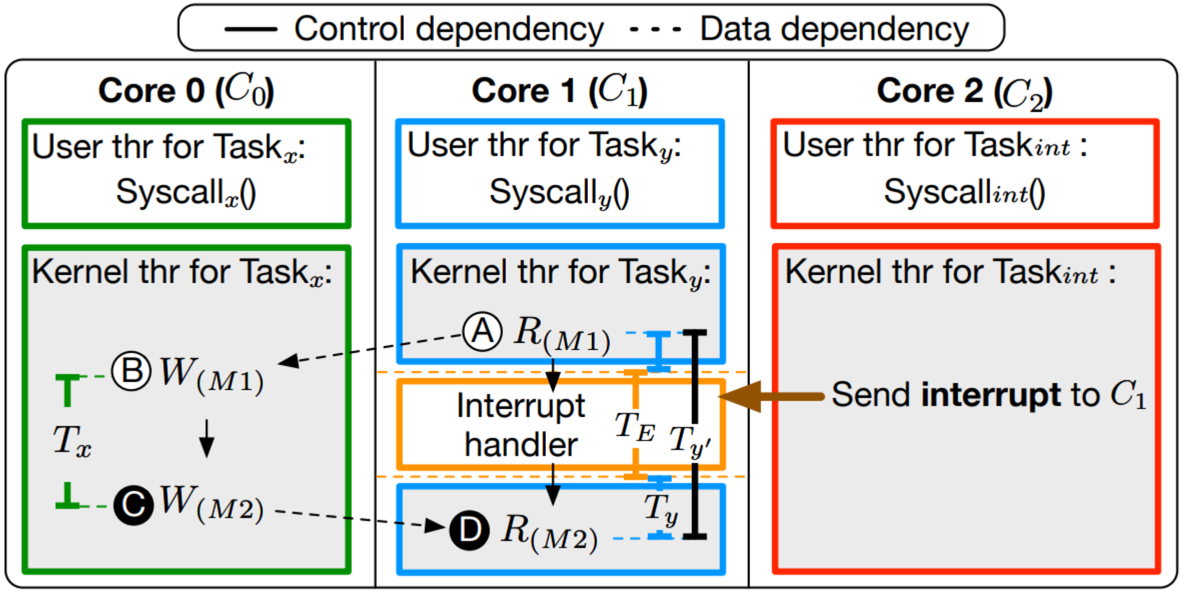
- Razzer [IEEE S&P 19]
  - (https://lifeasageek.github.io/papers/jeong-razzer.pdf)





Race Exploitation

- **ExpRace [USENIX Security 21, BlackHat USA 20]**
  - https://lifeasageek.github.io/papers/yoochan-exprace.pdf



Today

- **Using semaphores to schedule shared resources**
  - Producer-consumer problem
- **Other concurrency issues**
  - **Races**
  - **Deadlocks**
  - Thread safety
  - Interactions between threads and signal handling

A 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 need 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!

A Worry: Deadlock

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

- **More fully (and beyond the scope of this course), a deadlock has four requirements**
  - Mutual exclusion
    - Only one process can use the resource at a time
  - Hold and wait
    - A process **holds** at least one resource, and further requests for another resource held by another process (i.e., **wait**)
  - Circular waiting
  - No pre-emption
    - A resource is voluntarily released by the process holding the resource

Deadlocking With Semaphores

```
int main(int argc, char** argv)
{
    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);
    return 0;
}
```

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

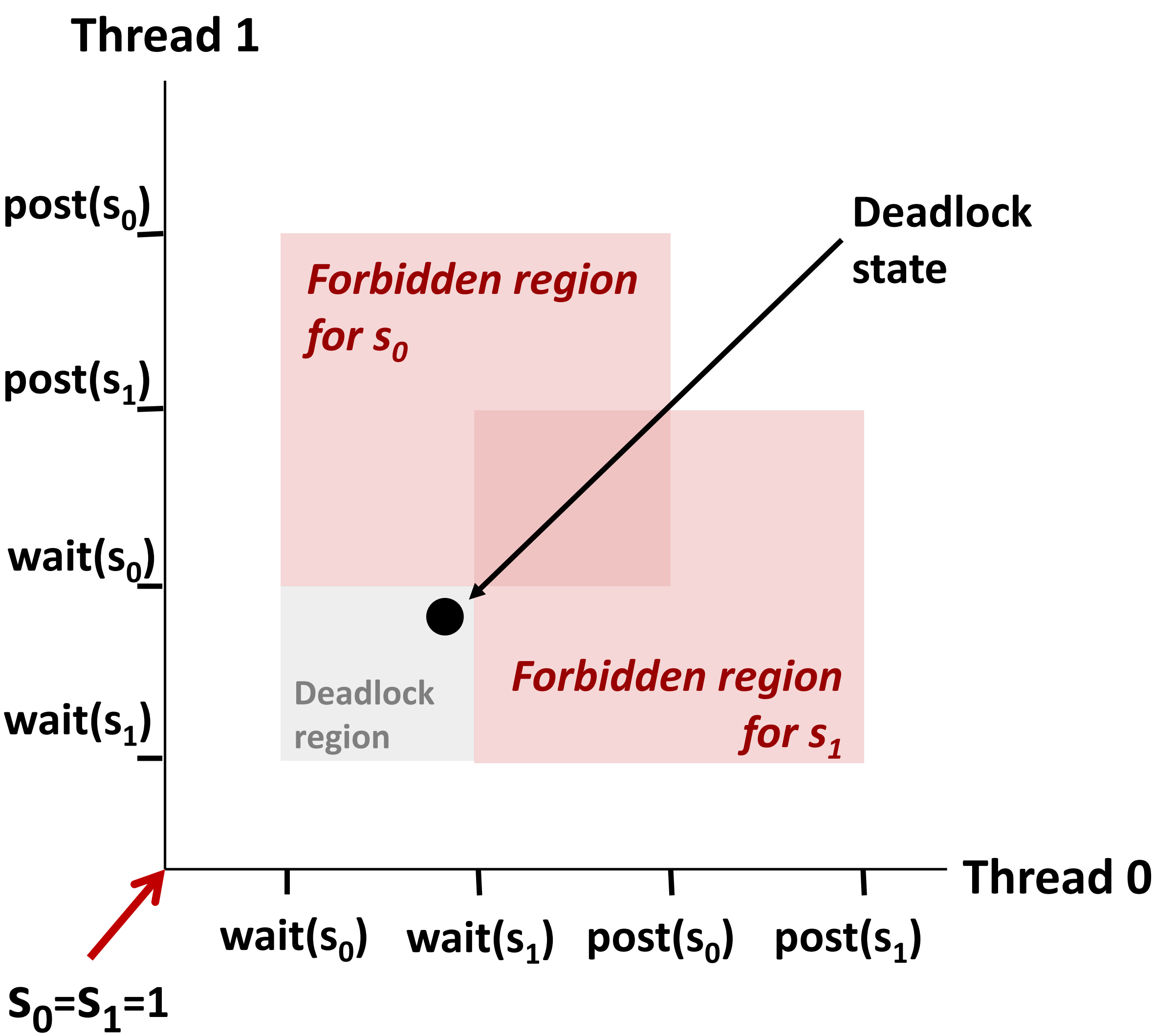
Tid[0]

wait(s_0);
wait(s_1);
cnt++;
post(s_0);
post(s_1);

Tid[1]

wait(s_1);
wait(s_0);
cnt++;
post(s_1);
post(s_0);

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

Unfortunate fact: deadlock is often non-deterministic (race)

Avoiding Deadlock

Acquire shared resources in same order

```
int main(int argc, char** argv)
{
    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);
    return 0;
}
```

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

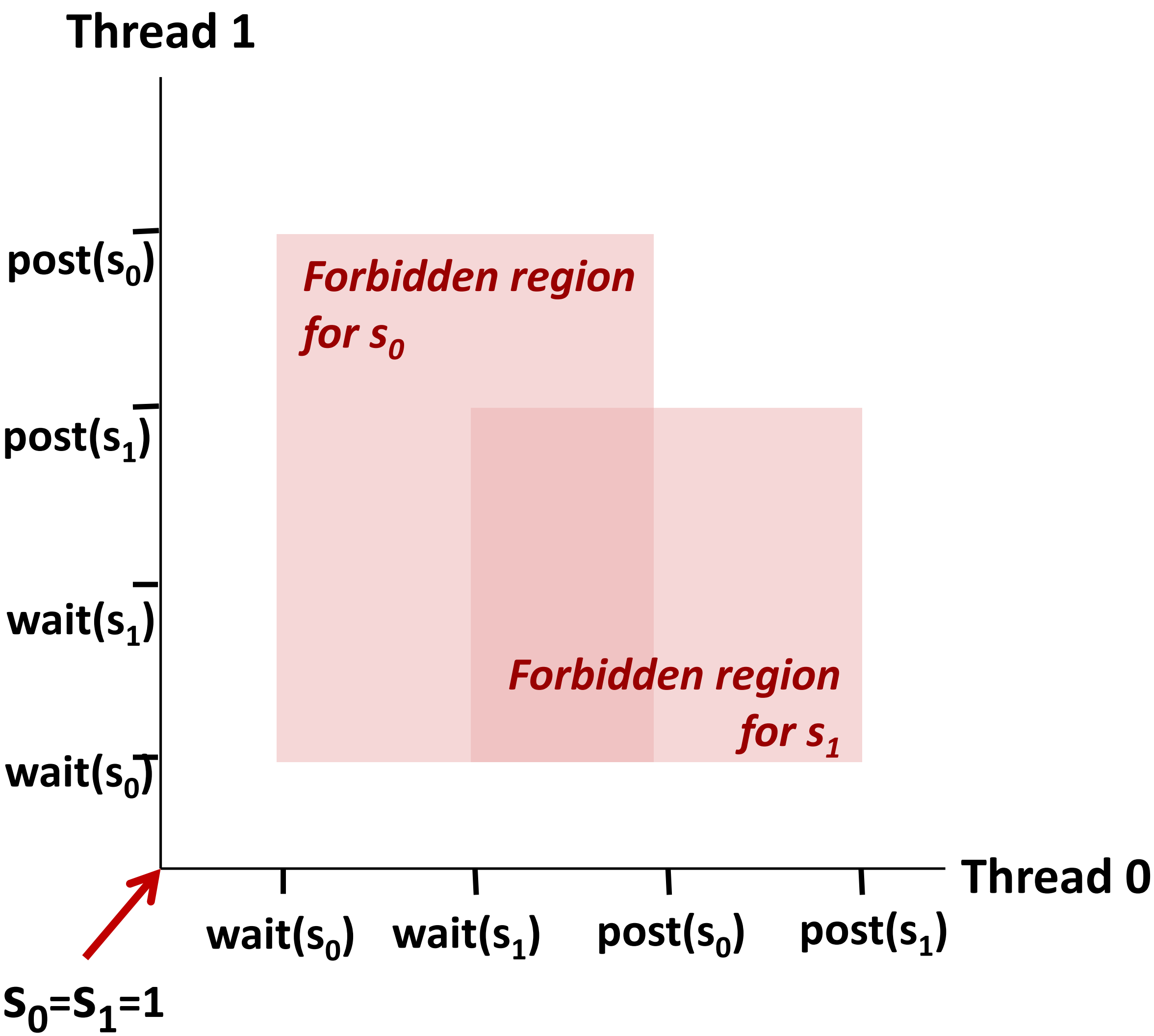
Tid[0]:

```
wait(s0);
wait(s1);
cnt++;
post(s0);
post(s1);
```

Tid[1]:

```
wait(s1);
wait(s0);
cnt++;
post(s1);
post(s0);
```

Avoided Deadlock in Progress Graph



No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

Today

- **Using semaphores to schedule shared resources**
  - Readers-writers problem
- **Other concurrency issues**
  - **Races**
  - **Deadlocks**
  - **Thread safety**
  - Interactions between threads and signal handling

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 call thread-unsafe functions

Thread-Unsafe Functions (Class 1)

- **Failing to protect shared variables**
  - Fix: Use *wait* and post semaphore operations (or mutex)
  - 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

```
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 part of 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-Safe Random Number Generator

- **glibc implementation**

Interface	Attribute	Value
rand(), rand_r(), srand()	Thread safety	MT-Safe

```
long int
__random (void)
{
  int32_t retval;

  __libc_lock_lock (lock);

  (void) __random_r (&unsafe_state, &retval);

  __libc_lock_unlock (lock);

  return retval;
}
```

```
void
__srandom (unsigned int x)
{
  __libc_lock_lock (lock);
  (void) __srandom_r (x, &unsafe_state);
  __libc_lock_unlock (lock);
}
```

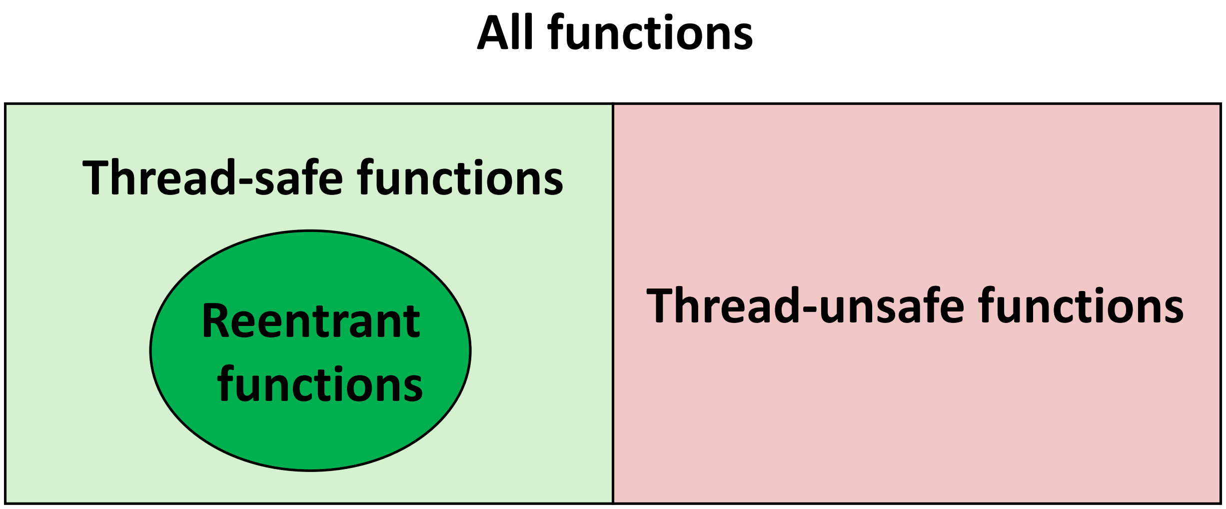
https://github.com/lattera/glibc/blob/master/stdlib/random.c

Thread-Unsafe Functions (Class 3)

- **Calling thread-unsafe functions**
  - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - Fix: Modify the function so that it only calls 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
    - Example: rand_r



Thread-Safe Library Functions

- **All functions in the Standard C Library (at the back of your K&R text) are thread-safe**
  - Examples: **`malloc`**, **`free`**, **`printf`**, **`scanf`**
- **Most Unix system calls are thread-safe, with a few exceptions**
  - "man page" provides the information

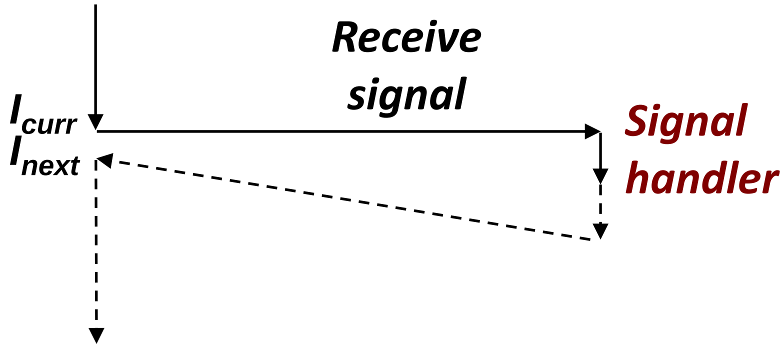
Interface	Attribute	Value
asctime()	Thread safety	MT-Unsafe race:asctime locale
asctime_r()	Thread safety	MT-Safe locale
ctime()	Thread safety	MT-Unsafe race:tmbuf race:asctime env locale
ctime_r(), gm-time_r(), lo-caltime_r(), mktime()	Thread safety	MT-Safe env locale
gmtime(), lo-caltime()	Thread safety	MT-Unsafe race:tmbuf env locale

Interface	Attribute	Value
strtok()	Thread safety	MT-Unsafe race:strtok
strtok_r()	Thread safety	MT-Safe

Today

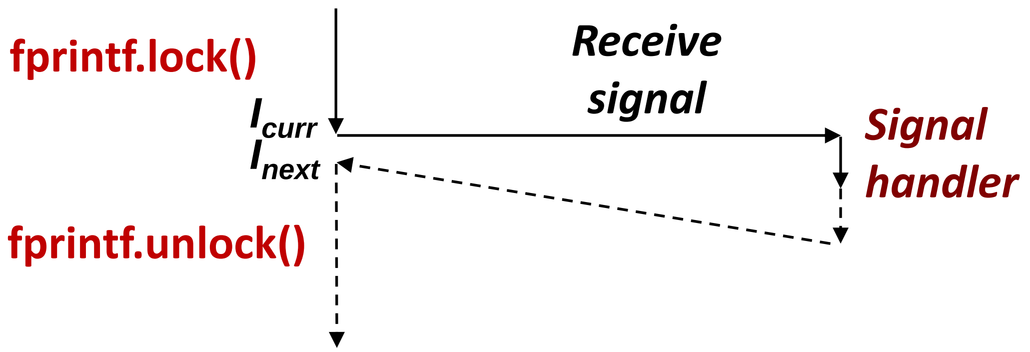
- **Using semaphores to schedule shared resources**
  - Readers-writers problem
- **Other concurrency issues**
  - **Races**
  - **Deadlocks**
  - **Thread safety**
  - **Interactions between threads and signal handling**

Signal Handling Review



- **Action**
  - Signal can occur at any point in program execution
    - Unless signal is blocked
  - Signal handler runs within same thread
  - Must run to completion and then return to regular program execution

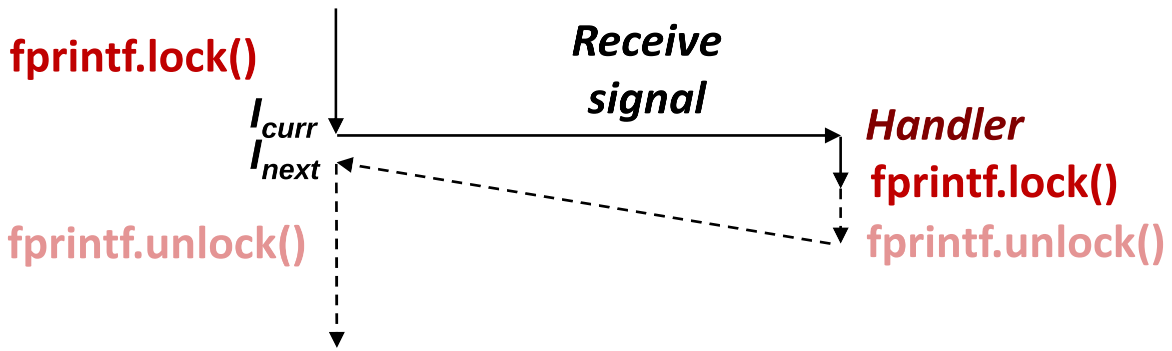
Threads / Signals Interactions



- **Many library functions use "locks" for thread safety**
  - Because they have hidden shared state
  - malloc
    - Free lists
  - fprintf, printf, puts
    - So that outputs from multiple threads don't interleave

- **Q. What would happen if the signal handler call these library functions?**

Bad Thread / Signal Interactions



- **What if:**
  - Signal received while library function holds lock
  - Handler calls same (or related) library function
- **Deadlock!**
  - The signal handler can return only if the lock is acquired
  - The lock would be released only if the signal handler returns

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
  - Read carefully with threads!

Thread safe Vs. Async signal safe

- **Thread safe**
  - A function X is thread safe if X does not have race conditions when invoked by multiple threads simultaneously
  - e.g., thread-safe ensures the safety when the function X is invoked twice individually by two different threads

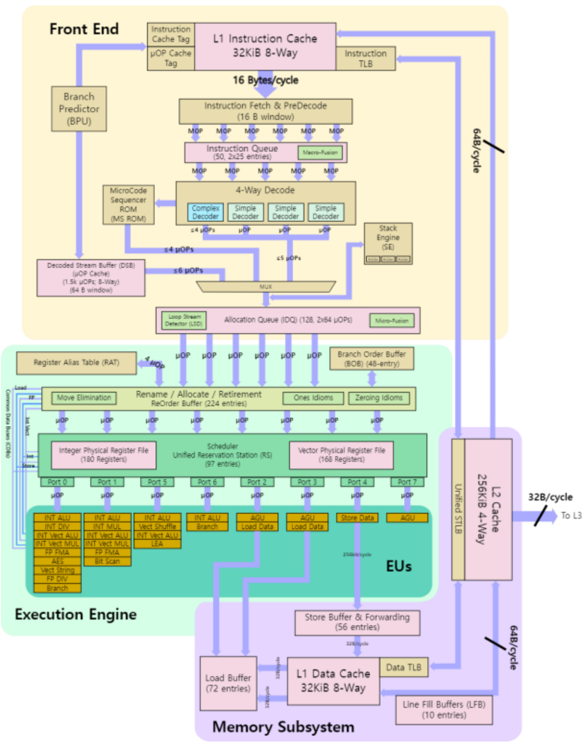
- **Async-signal safe generally implies thread safe**
  - The opposite does not hold
  - e.g., Async-signal safe ensures the safety when the function X is invoked twice recursively by the same thread

- **Check more**
  - https://en.wikipedia.org/wiki/Thread_safety
  - https://en.wikipedia.org/wiki/Reentrancy_(computing)

Memory Consistency Models

- **Multi-processors reorder memory operations in unintuitive, scary ways**
  - Mostly for optimizing performances

- **You may observe very strange behaviors due to the memory reordering ☹**



Multithreaded Programs

Initially A = B = 0

Thread 1

```
A = 1
if (B == 0)
   print "Hello";
```

Thread 2

```
B = 1
if (A == 0)
   print "World";
```

Q. What can be printed?

- **“Hello”?**
- **“World”?**
- **“Hello World”?**
- **“World Hello”?**
- **Nothing?**

Multithreaded Programs

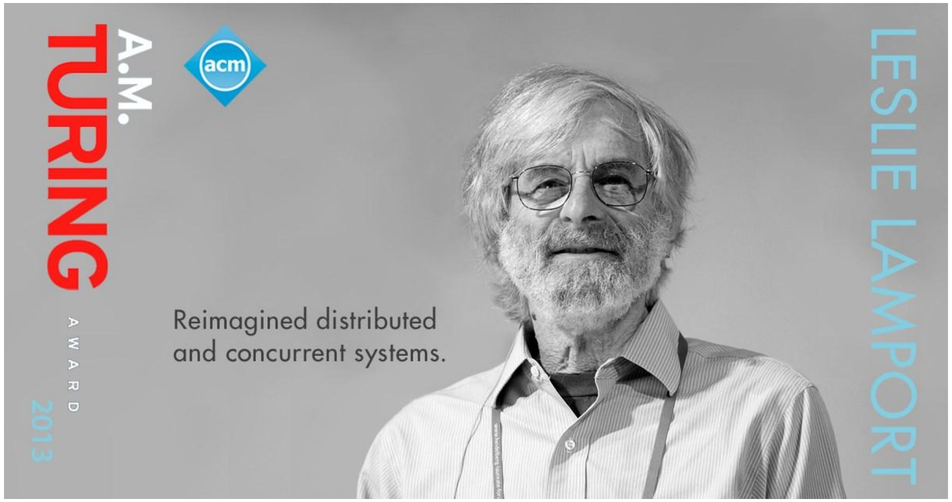
Initially A = B = 0

Thread 1	**Thread 2**
A = 1	B = 1
r0 = B	r1 = A
if (r0 == 0)	if (r1 == 0)
print “Hello”;	print “World”;

Let’s clarify each thread loads using registers, r0 and r1

Sequential Consistency

- **Two invariants**
  - All operations executed in some sequential order
  - Each thread's operations happen in program order
- **Sequential consistency is the strongest memory model**
  - It allows the fewest reorderings/strange behaviors…



Sequential Consistency

Initially A = B = 0

Thread 1

```
A = 1
r0 = B
if (r0 == 0)
    print "Hello";
```

Thread 2

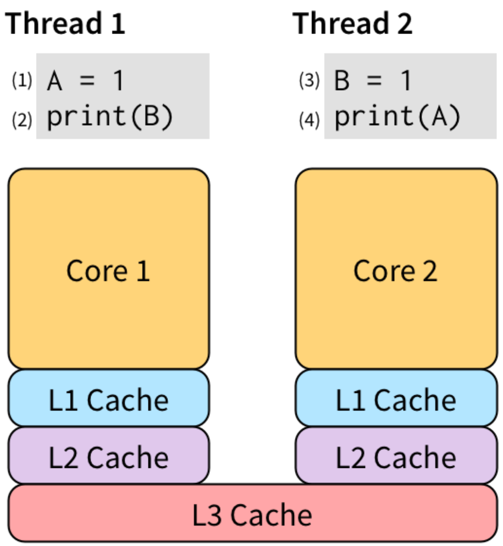
```
B = 1
r1 = A
if (r1 == 0)
    print "World";
```

Following the sequential consistency:

- **"Hello"**
- **"World"**

Memory Consistency Models

- **A memory consistency model defines the permitted reorderings of memory operations during execution**
- **It is a contract between hardware and software: the hardware will only mess with your memory operations in these ways**
- **Why sequential consistency?**
  - Agrees with programmer's intuition



- **Why not sequential consistency?**
  - Horribly slow to guarantee in hardware
    - Coherence guarantee: all writes *to the same location* are seen in the same order by every thread
  - You can reorder the memory operations, so why not?

Memory Consistency Models

- **Total Store Ordering (TSO)**
  - Sequential consistency + store buffers
  - x86 specifies TSO as its memory models
  - Going back to the example:
    - “Hello World” and “World Hello” are also possible

- **Weak Ordering**
  - Sequential consistency + store buffers + load buffers
  - Almost everything can be reordered…
  - ARM specifies this memory models