CSC 112: Computer Operating Systems
Lecture 3

Synchronization

Department of Computer Science,

Hofstra University

Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming
And NTNU course on Operating Systems

Outline

- Concurrency & Locks
- Spinlocks

Concurrency

```c
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"

int counter = 0;
int loops;
void *worker(void *arg) {
        int i;
        for (i = 0; i < loops;i++)
        {counter++; }
        return NULL;
}
```

```c
int main(int argc, char *argv[])
{
        if (argc != 2){
                fprintf(stderr, "usage: threads <loops>\n");
                exit(1); }
        loops = atoi(argv[1]);
        pthread_t p1, p2;
        printf("Initial value : %d\n", counter);
        pthread_create(&p1, NULL, worker, NULL);
        pthread_create(&p2, NULL, worker, NULL);
        pthread_join(p1, NULL);
        pthread_join(p2, NULL);
        printf("Final value : %d\n", counter);
        return 0;
}
```

This concurrent program has a race condition, and may produce different final values of counter for different runs, depending on different interleavings of worker threads

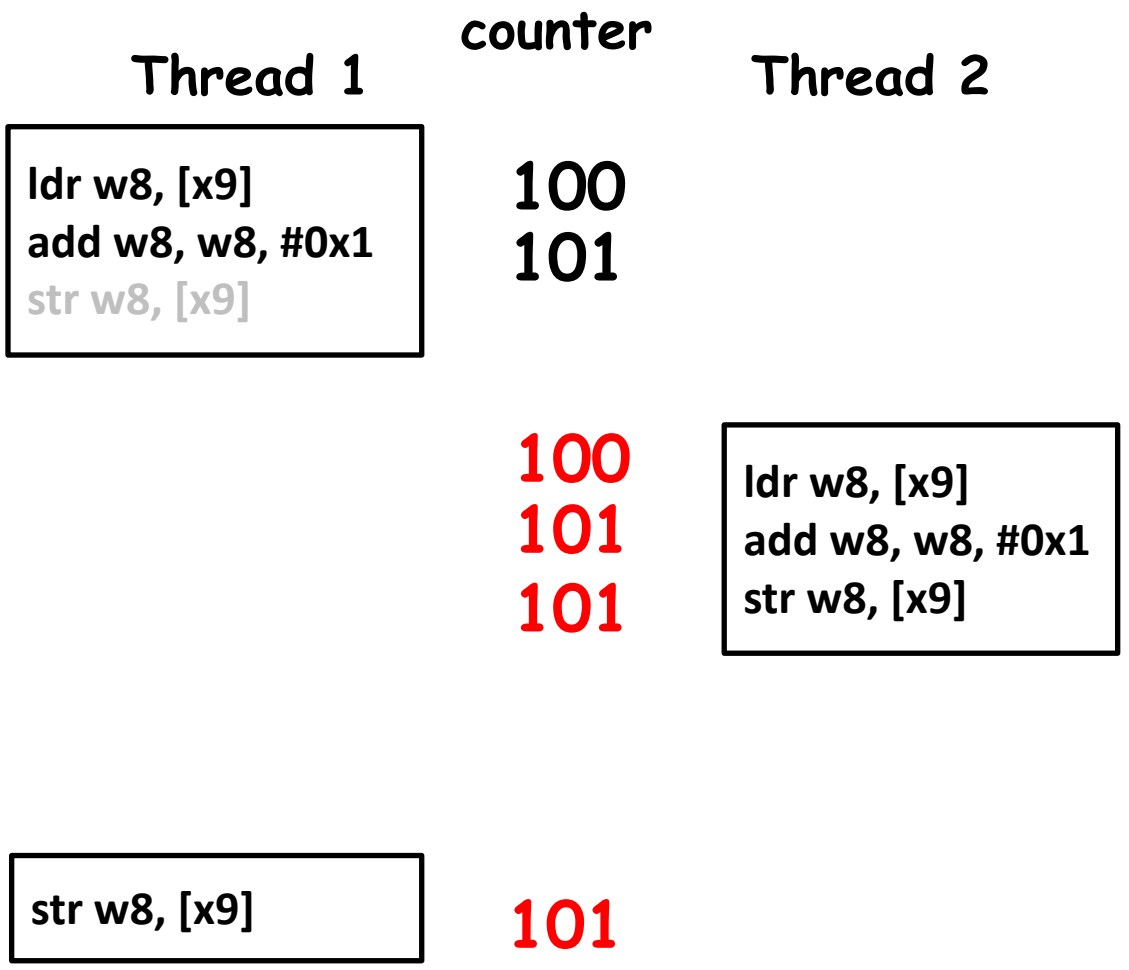
Race Condition

- Incrementing **counter** has 3 instructions in assembly code:
- ldr w8, [x9]: Read the value of counter at memory address x9 into register w8
- add w8, w8, #0x1: increment the value of register w8 by 1
- str w8, [x9]: write the new value of counter in register w8 to memory address x9
- When both threads read the same value of counter before writing to it, counter is incremented only by 1 instead of by 2!

counter++;

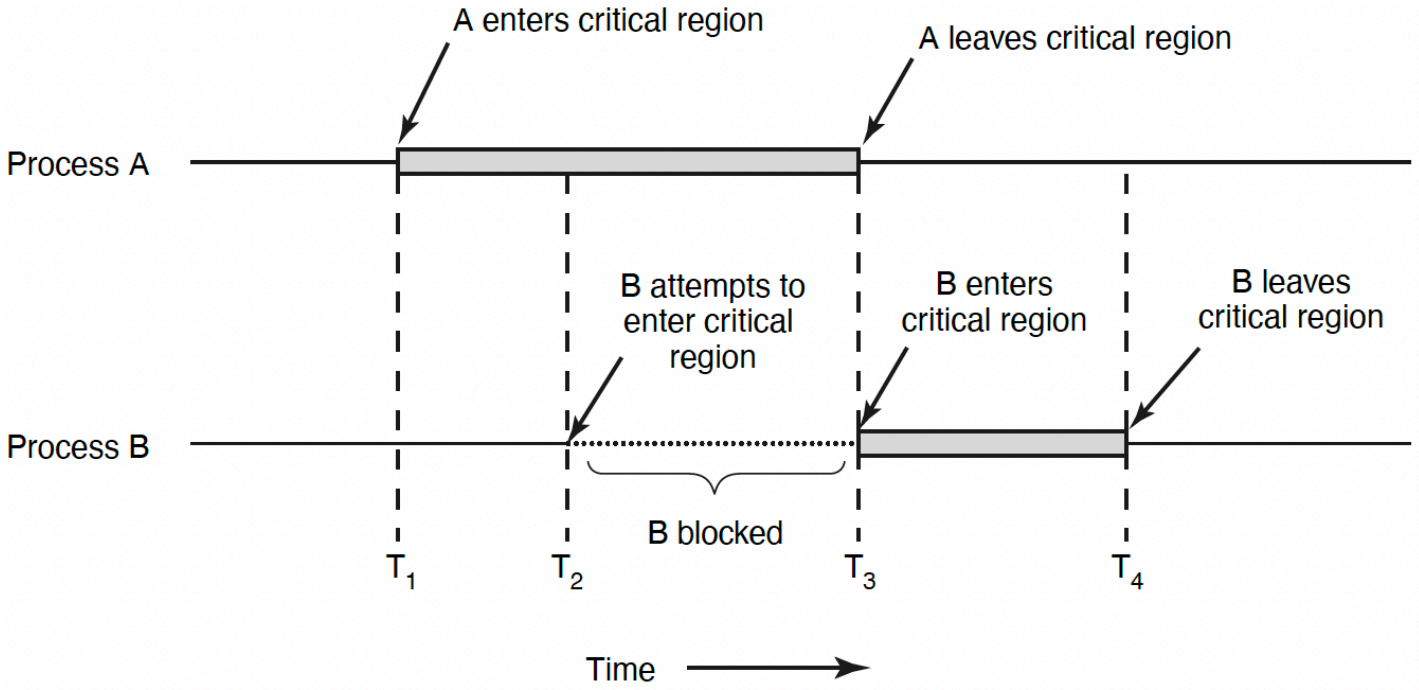
Thread 1	Thread 2

ldr w8, [x9]
add w8, w8, #0x1
str w8, [x9]
	ldr w8, [x9]
	add w8, w8, #0x1
	str w8, [x9]

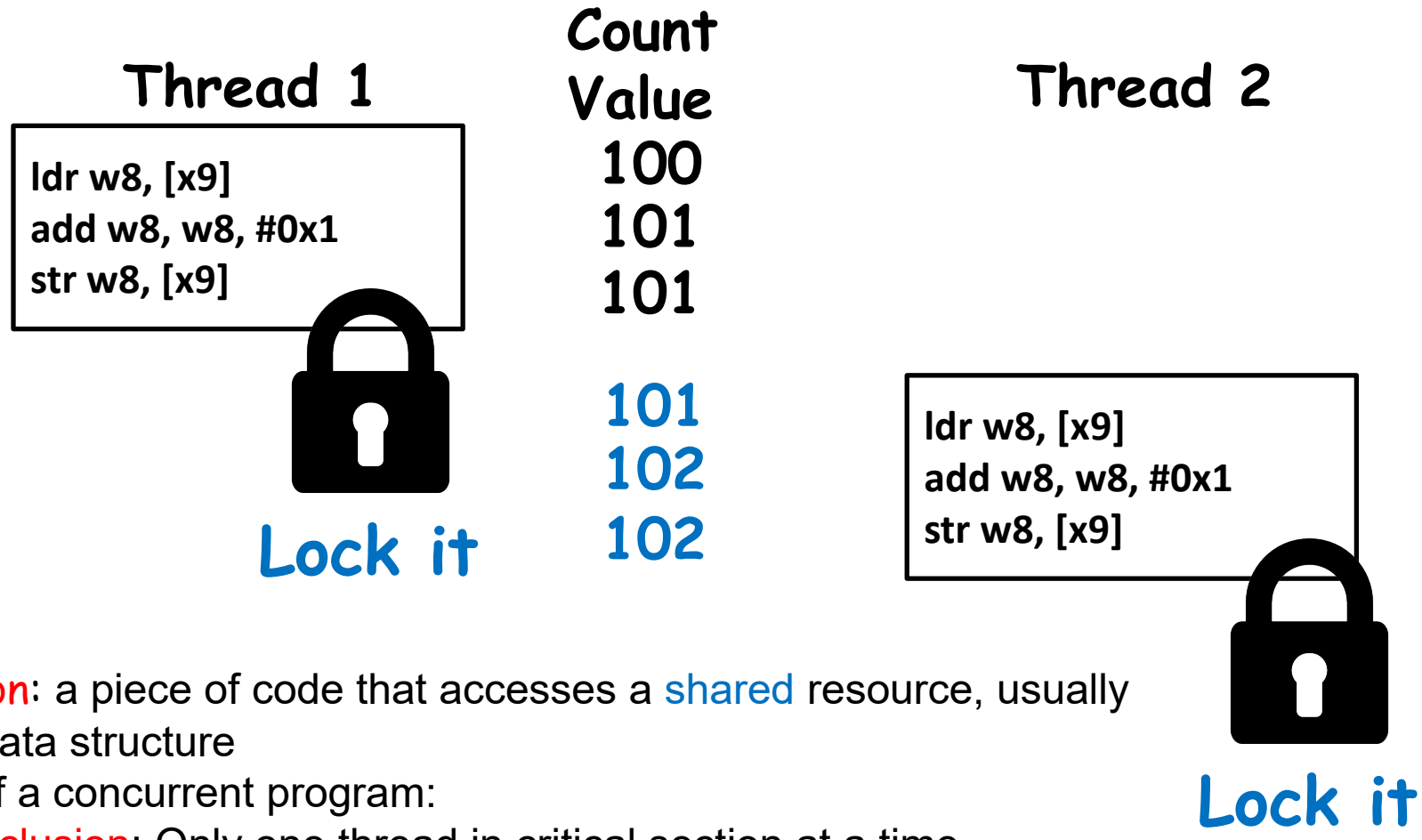


Race Condition & Critical Section

- **Race condition**:
  - Multiple threads of execution update shared data variables, and final results depend on the execution order
  - Race condition leads to non-deterministic results: different results even for the same inputs
- To prevent race condition, a **critical section** should be used to protect shared data variables
  - A critical section is executed atomically
  - Mutual exclusion (mutex) ensures that when one thread is executing in its critical section, no other thread is allowed to execute in that critical section



Lock to Protect a Critical Section



- Critical section: a piece of code that accesses a shared resource, usually a variable or data structure
- Correctness of a concurrent program:
  - Mutual exclusion: Only one thread in critical section at a time
  - Progress (deadlock-free): If several simultaneous requests, must allow one to proceed
  - Bounded (starvation-free): Must eventually allow each waiting thread to enter

Locks

- A **lock** is a **variable**
- **Objective:** Provide mutual exclusion (mutex)
- Two states
  - Available or free
  - Locked or held
- lock(): tries to acquire the lock
- unlock(): releases the lock that was previously acquired

```
lock_t mutex
void *worker(void *arg) {
        int i;
        for (i = 0; i < loops;i++)  {
                lock(&mutex);
                counter++;
                unlock(&mutex)}
        return NULL;
}
```

Locks: Disable Interrupts

- An early solution: disable interrupts for critical sections
- Problems:
  - System becomes irresponsive if interrupts are disabled for a long time
  - Does not work on multiprocessors

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Locks: Loads/Stores

- This implementation does not ensure mutual exclusion, since both threads may grab the lock:
- After Thread 1 reads flag==0 and exits the while loop, it is preempted/interrupted by Thread 2, which also reads flag==0 and exits the while loop. Then both threads set flag=1 and enter the critical section.
- Root cause: Lock is not an atomic operation!

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1)  // TEST the flag
10         ; // spin-wait (do nothing)
11     mutex->flag = 1;          // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

flag = 0

Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Locks: Test-And-Set

- How to provide mutual exclusion for locks?
  - **Get help from hardware!**
- CPUs provide special hardware instructions to help achieve mutual exclusion
  - The **TestAndSet** (TAS) instruction tests and modifies the content of a memory word **atomically**
- Locking with TAS: TAS fetches the old value of lock->flag into variable old, sets lock->flag to 1, then return variable old, all in one atomic operation
  - If lock-flag==0, then lock() sets it to 1 and returns old=0, so the thread exits the while loop and enters critical section
  - If lock-flag==1, then lock() returns old=1, so the thread spins in the while loop and does not enter critical section

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  int TestAndSet(int *old_ptr, int new) {
6      int old = *old_ptr; // fetch old value at old_ptr
7      *old_ptr = new;     // store 'new' into old_ptr
8      return old;         // return the old value
9  }
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Locks: Compare-And-Swap

- Another hardware primitive: **compare-and-swap (CAS)**
- Locking with CAS: CAS fetches the old value of lock-flag into variable original, compares original with expected (0), and if they are equal (lock-flag==0), sets lock->flag to 1, then return variable original, all in one atomic operation
  - If lock-flag==0, then lock() sets it to 1 and returns original=0, so the thread exits the while loop and enters critical section
  - If lock-flag==1, then lock() returns original=1, so the thread spins in the while loop and does not enter critical section
- Compared to TAS, assignment *ptr=new is not executed if lock-flag==1, hence slightly less overhead

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}

void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

Locks: Busy Waiting

```
void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}
```

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

- Both TAS and CAS are spinlocks based on **busy waiting**
  - A thread is stuck in a while loop endlessly checking lock->flag if the lock is held by others
- Goals achieved?
  - **Mutual exclusion (Yes!)**
  - **Fairness (NO!!)**
  - **Performance (NO!!)**

Ticket Lock

- Basic spinlocks are **not fair** and may cause **starvation**
- Ticket lock uses hardware primitive **fetch-and-add** to guarantee fairness
- **Lock:**
  - Use fetch-and-add on the ticket value
  - The return value is the thread's "turn" value
- **Unlock:**
  - Increment the turn

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

```
void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}
```

Initial value tickets=0 turn=0

	Ticket	Turn
A lock()	1	0
B lock()	2	0
C lock()	3	0
A unlock()		
A lock()		
B unlock()		
C unlock()		
A unlock()		

	myturn
A	0
B	1
C	2

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```
void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}
```

Initial value tickets=0 turn=0

	Ticket	Turn
A lock()	1	0
B lock()	2	0
C lock()	3	0
A unlock()	3	1
A lock()	4	1
B unlock()		
C unlock()		
A unlock()		

	myturn
A	3
B	1
C	2

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```
void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

```
void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}
```

Initial value tickets=0 turn=0

	Ticket	Turn
A lock()	1	0
B lock()	2	0
C lock()	3	0
A unlock()	3	1
A lock()	4	1
B unlock()	4	2
C unlock()	4	3
A unlock()	4	4

	myturn
A	3
B	1
C	2

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```
void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Spinlocks: Performance

- Can be fast:
  - Locks held for a short time
  - Pros: No context switch
- But also, can be slow:
  - Locks held for a long time
  - Cons: Spinwaiting is wasteful

Spinlocks: yield()

- Busy waiting with spinlocks is wasteful of CPU time
- Instead of spinning, just **give up** the CPU to anther process/thread

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Spinlocks: Locks with Queue

- Locks with queue
  - Sleep and put thread on a queue instead of spinning

- This can guarantee that **starvation** does not happen as long as all threads will relinquish locks

- An example from Solaris
  - **Park()**: put the calling thread to sleep
  - **Unpark(ThreadID)**: to walk up a specific thread with ThreadID

Spinlocks: Locks with Queue

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}
```

```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}
```

```
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```

Spinlocks: Two Phase locks

- Lock released **quickly** ▸ Spin-wait
- Lock released **slowly** ▸ Sleep/block

- **Two phase lock** is a hybrid approach that combines both spin-wait and sleep/block
  - **First phase**: the lock spins for a while
  - **Second phase**: if the lock is not acquired in the first phase, put the calling thread to sleep.

- **Adaptive Mutexes**: If a thread has a **locked adaptive mutex** and the process/thread **holding the adaptive mutex** is running, the adaptive mutex executes busy waiting. Otherwise, it just **blocks the thread**.

Summary

- Locks: Provide a **mutual exclusion**
- Spinlocks:
  - Test-And-Set()
  - Compare-and-swap()
  - Yield()
  - Locks with Queue
  - Two phase locks
  - Adaptive mutex

Outlines

- Semaphore
- Semaphore operations
- Binary Semaphore
- Semaphore for ordering
- Semaphore for P/C
- Deadlock
- Semaphore for P/C without deadlock

Recap

- Producer/Consumer Problem
- Condition Variable for Producer/Consumer Problem
  - **Two CVs** and **while loop**

```c
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&mutex); //p1
        while (count == MAX_ITEMS) //p2
            cond_wait(&empty, &mutex); //p3
        put(i); //p4
        cond_signal(&fill); //p5
        mutex_unlock(&mutex); //p6
    }
}
```

```c
void *consumer(void *arg) {
    while(1) {
        mutex_lock(&mutex); //c1
        while (count == 0)  //c2
            cond_wait(&fill, &mutex); //c3
        int tmp = get(); //c4
        cond_signal(&empty); //c5
        mutex_unlock(&mutex); //c6
        printf("%d\n", tmp); //c7
    }
}
```

Semaphores

- An object with **an integer value**
- Introduced by **E. W. Dijkstra**
- Manipulate with two routines
  - sem_wait(s) P(s) down(s)
  - sem_post(s) V(s) up(s)
- Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied

Semaphore Operations

- **Initialization**

- **Wait** or **test**
  - Decrements sem value by 1. Waits if value of sem is <0

- **Signal** or **post**:
  - Increments sem value by 1. Then wake a single waiter if exists

```
sem_init(sem_t *s, int initval) {
    s->value = initval;
}
```

```
int sem_wait(sem_t *s) {
    s->value--;
}
```

```
int sem_post(sem_t *s) {
    s->value++;
}
```

Binary Semaphore

- **Binary Semaphore** is a **mutex lock**

```
void init(lock_t *lock){
    sem_init(lock, 1)
}

void acquire(lock_t *lock) {
    sem_wait(lock);
}

void release(lock_t *lock) {
    sem_post(lock);
}
```

Possible execution trace of two threads using a binary semaphore

Val	Thread 0	State	Thread 1	State
1		Run		Ready
1	call `sem_wait()`	Run		Ready
0	`sem_wait()` returns	Run		Ready
0	`(crit sect begin)`	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	call `sem_wait()`	Run
-1		Ready	`decr sem`	Run
-1		Ready	`(sem<0)→sleep`	Sleep
-1		Run	*Switch→T0*	Sleep
-1	`(crit sect end)`	Run		Sleep
-1	call `sem_post()`	Run		Sleep
0	`incr sem`	Run		Sleep
0	`wake(T1)`	Run		Ready
0	`sem_post()` returns	Run		Ready
0	*Interrupt; Switch→T1*	Ready		Run
0		Ready	`sem_wait()` returns	Run
0		Ready	`(crit sect)`	Run
0		Ready	call `sem_post()`	Run
1		Ready	`sem_post()` returns	Run

Semaphore for Ordering

- Parent call thr_join() and waits for child to complete by calling thr_exit():

```
sem_init(s, 0);

void thr_join() {
    sem_wait(s);
}
```

```
void thr_exit() {
    sem_post(s);
}
```

- Equivalent implementation using a Condition variable (discussed later):

```
void thr_join() {
    Pthread_mutex_lock(&m);          // w
    while (done == 0)                // x
        Pthread_cond_wait(&c, &m);   // y
    Pthread_mutex_unlock(&m);        // z
}
```

```
void thr_exit() {
    Pthread_mutex_lock(&m);      // a
    done = 1;                    // b
    Pthread_cond_signal(&c);     // c
    Pthread_mutex_unlock(&m);    // d
}
```

Semaphore for Ordering

Sem=-1 here

- Case 1: parent runs first

	Val	Parent	State	Child	State
	0	`create(Child)`	Run	*(Child exists, can run)*	Ready
	0	call `sem_wait()`	Run		Ready
Sem=-1	-1	`decr sem`	Run		Ready
	-1	`(sem<0)`→`sleep`	Sleep		Ready
	-1	*Switch*→*Child*	Sleep	`child runs`	Run
	-1		Sleep	call `sem_post()`	Run
Sem=0	0		Sleep	`inc sem`	Run
	0		Ready	`wake(Parent)`	Run
	0		Ready	`sem_post()` returns	Run
	0		Ready	*Interrupt*→*Parent*	Ready
	0	`sem_wait()` returns	Run		Ready

Semaphore for Ordering

- Case 2: child runs first

	Val	Parent	State	Child	State
	0	`create(Child)`	Run	*(Child exists; can run)*	Ready
	0	*Interrupt→Child*	Ready	`child runs`	Run
	0		Ready	call `sem_post()`	Run
Sem=1	1		Ready	`inc sem`	Run
	1		Ready	`wake(nobody)`	Run
	1		Ready	`sem_post()` returns	Run
	1	`parent runs`	Run	*Interrupt→Parent*	Ready
	1	call `sem_wait()`	Run		Ready
Sem=0	0	`decrement sem`	Run		Ready
	0	`(sem≥0)→awake`	Run		Ready
	0	`sem_wait()` returns	Run		Ready

Semaphore for P/C

- Shared data:

```
sem_t empty;    producer
sem_t full;     consumer
```

- Initialization:

```
sem_init(&empty, MAX);
sem_init(&full, 0);
```

Semaphore for P/C

MAX=1

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);          // Line P1
        put(i);                    // Line P2
        sem_post(&full);           // Line P3
    }
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);           // Line C1
        tmp = get();               // Line C2
        sem_post(&empty);          // Line C3
        printf("%d\n", tmp);
    }
}
```

Semaphore for P/C

MAX > 1

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);          // Line P1
        put(i);                    // Line P2
        sem_post(&full);           // Line P3
    }
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);           // Line C1
        tmp = get();               // Line C2
        sem_post(&empty);          // Line C3
        printf("%d\n", tmp);
    }
}
```

Race condition

Semaphore for P/C

2 producers

Empty = 9

MAX=10

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 2

```
void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
}
```

```
void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
}
```

Semaphore for P/C

2 producers

Empty = 8

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 1

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}
```

Producer 2

Interrupted

```
void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
}
```

```
void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
}
```

When Producer 2 thread preempts/interrupts Producer 1 thread just after it the line buffer[fill]=value, there is a race condition on the global variable fill, so fill is incremented only by 1 instead of by 2.

Semaphore for P/C – Mutex

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // Line P0 (NEW LINE)
        sem_wait(&empty);        // Line P1
        put(i);                  // Line P2
        sem_post(&full);         // Line P3
        sem_post(&mutex);        // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);        // Line C0 (NEW LINE)
        sem_wait(&full);         // Line C1
        int tmp = get();         // Line C2
        sem_post(&empty);        // Line C3
        sem_post(&mutex);        // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

Solution: add a mutex lock

Semaphore for P/C – Mutex

mutex = 1
full = 0
empty = 10

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Semaphore for P/C – Mutex

mutex = 0
full = 0
empty = 10

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

Ready
Wait for lock

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Running
Hold lock
Wait for full

Deadlock occurred!

Semaphore for P/C – Mutex

mutex = 0
full = 0
empty = 10

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&empty);
        put(i);
        sem_post(&full);
        sem_post(&mutex);
    }
}
```

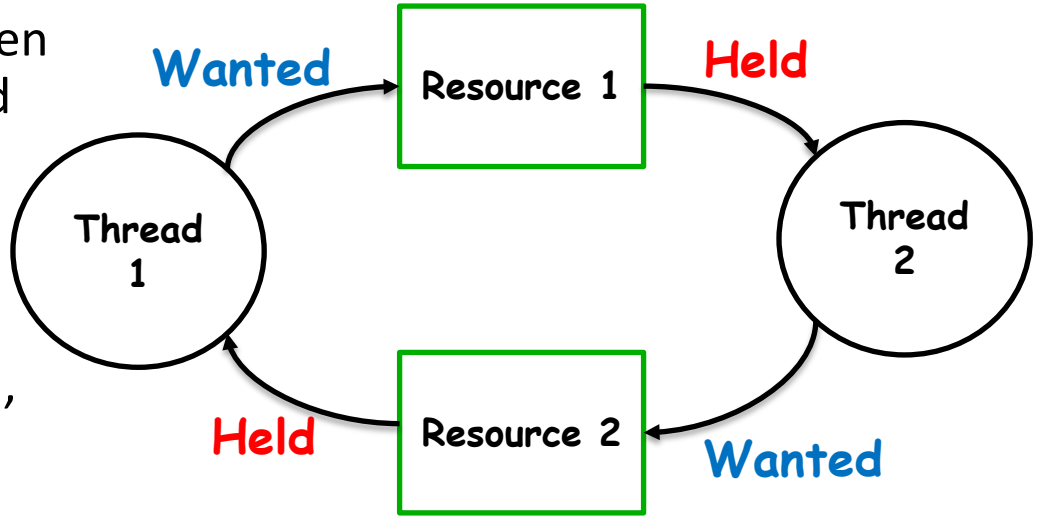
Running
Wait for lock

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex);
        sem_wait(&full);
        int tmp = get();
        sem_post(&empty);
        sem_post(&mutex);
        printf("%d\n", tmp);
    }
}
```

Block
Hold lock
Wait for full

Deadlock

- Definition: A set of threads are said to be in a deadlock state when every thread in the set is waiting for an event that can be caused only by another thread in the set
- Conditions for Deadlock
- Mutual exclusion
  - Threads claim exclusive control of resources that require e.g., a thread grabs a lock
- Hold-and-wait
  - Threads hold resources allocated to them while waiting for additional resources
- No preemption
  - Resources cannot be forcibly removed from threads that are holding them
- Circular wait
  - There exists a circular chain of threads such that each holds one or more resources that are being requests by next thread in chain



Not a perfect analogy, just a fun image!

Mutex without Deadlock

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);          // Line P1
        sem_wait(&mutex);          // Line P1.5 (MUTEX HERE)
        put(i);                    // Line P2
        sem_post(&mutex);          // Line P2.5 (AND HERE)
        sem_post(&full);           // Line P3
    }
}
```

Lock just around the critical section

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);           // Line C1
        sem_wait(&mutex);          // Line C1.5 (MUTEX HERE)
        int tmp = get();           // Line C2
        sem_post(&mutex);          // Line C2.5 (AND HERE)
        sem_post(&empty);          // Line C3
        printf("%d\n", tmp);
    }
}
```

Summary

- Semaphores -> locks + condition variables
  - Binary semaphores -> locks
  - Semaphore for ordering
- Semaphore for P/C

Recap

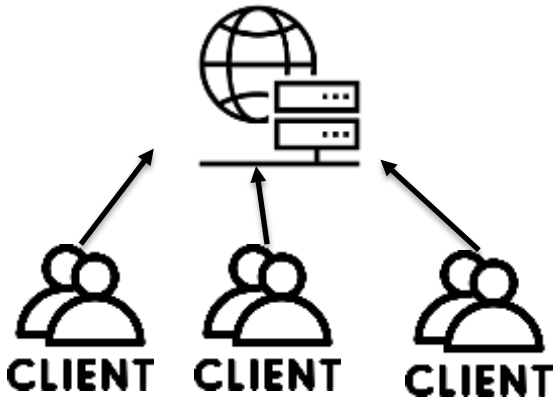
- Locks --- **mutual execution**
  - Only one thread must execute critical section
- Hardware support – **atomical execution**
  - Test-and-set and compare-and-swap
- Busy-waiting --- **spinlock**
- Metrics to evaluate locks:
  - Correctness: mutual execution
  - Fairness: no starvation
  - Performance: no high cost to acquire and release a lock
- Ticket locks --- **No starvation**
- Queue locks --- **No starvation** and **predictable order**

Condition Variables

- Producer/Consumer problem
- Ordering execution
- Condition variable

Producer/Consumer Problem

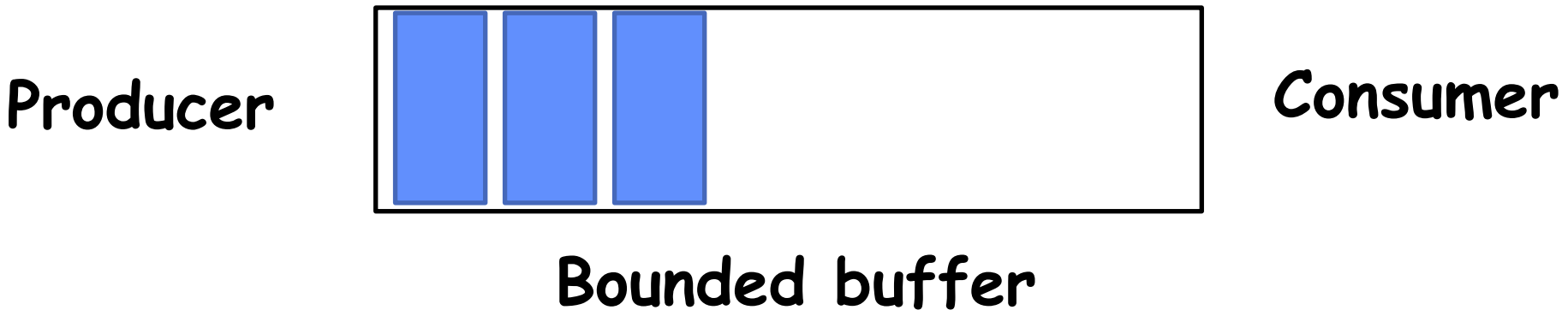
- A classical synchronization problem, also called the **bounded-buffer problem**
- A buffer has a **bounded size**
- Examples of Producer/Consumer Problems:
  - **Web servers:**
    - » Producer puts requests in a queue
    - » Consumers picks requests from the queue to process
  - **Unix Pipes**




```
grep Pthread pc.c | wc -l
```

producer consumer

Producer/Consumer Problem



- When buffers are **full**, producer must **wait**
- When buffers are **empty**, consumer must **wait**

- **Synchronization between producers**
- **Synchronization between producers and consumers**
- Producing/Consuming to/from buffer require **locking**

P/C Problems With Only Locks

```
volatile int empty_slot = 5;
volatile int filled_slot = 0;
struct lock_t empty_cnt_lock;
struct lock_t filled_cnt_lock;
```

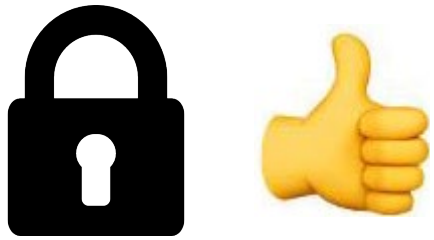
```
void producer(void) {
    int new_msg;
    while(1) {
        new_msg = produce_new();
        lock(&empty_cnt_lock);
        while (empty_slot == 0) {
            unlock(&empty_cnt_lock);
            lock(&empty_cnt_lock);
        }
        empty_slot--;
        unlock(&empty_cnt_lock);
        buffer_add(new_msg);
        lock(&filled_cnt_lock);
        filled_slot++;
        unlock(&filled_cnt_lock);
    }
}
```

Busy looping

```
void consumer(void) {
    int msg;
    while(1) {
        lock(&filled_cnt_lock);
        while(filled_slot == 0) {
            unlock(&filled_cnt_lock);
            lock(&filled_cnt_lock);
        }
        filled_slot--;
        unlock(&filled_cnt_lock);
        msg = buffer_remove();
        lock(&empty_cnt_lock);
        empty_slot++;
        unlock(&empty_cnt_lock);
    }
}
```

More Than Mutual Exclusion

- **Mutual exclusion**
  - No two threads access a critical section at the same time
  - Thread A and B don't run at the same time
  - **Locks**



- **Condition**
  - A thread wishes to check whether a condition is true before execution
  - Thread B runs after thread A completes
  - **Condition variables and semaphores**

Condition Variables (CV)

- Condition variables are **synchronization primitives**:
  - **A queue of waiting threads**
  - A thread waits for a condition to be true and can put itself into the queue
  - Other thread can wake up one or more waiting threads
  - Used with mutex to prevent race conditions

- Two functions:
  - wait(CV…): put itself on the waiting queue

```
pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

  - Signal(CV,…): Send signal to CV when it is done

```
pthread_cond_signal(pthread_cond_t *cond);
```

Condition Variables (CV)

- A parent waits for the child by calling thr_join(); the child signals completion by calling thr_exit(). We need to implement thr_join() and thr_exit() with CV.

```
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}
```

Parent

wait

signal

Child

Incorrect: CV with Only Lock

```
//Child
void thr_exit(){
  Pthread_mutex_lock(&m);//A
  Pthread_cond_signal(&c);//B
  Pthread_mutex_unlock(&m);}//C
//Parent
void thr_join(){
  Pthread_mutex_lock(&m);//X
  Pthread_cond_wait(&c, &m);//Y
  Pthread_mutex_unlock(&m);}//Z
```

Scenario 1: Parent calls thr_join() first. Works correctly.

Parent	X	Y				Z
Child			A	B	C	

Scenario 2: Child calls thr_exit() first. Parent blocks forever!

Parent				X	Y	
Child	A	B	C			

- Child thr_exit() function:
  - Line A: Child thread locks the mutex (pthread_mutex_lock(&m)).
  - Line B: It signals the condition variable (pthread_cond_signal(&c)) to notify the parent that it has completed.
  - Line C: It then unlocks the mutex (pthread_mutex_unlock(&m)).
- Parent thr_join() function:
  - Line X: Parent thread locks the mutex (pthread_mutex_lock(&m)).
  - Line Y: It waits on the condition variable (pthread_cond_wait(&c, &m)). This releases the mutex and puts the parent to sleep until it is signaled.
  - Line Z: Once signaled, it reacquires the mutex and then unlocks it (pthread_mutex_unlock(&m)).
- The program assumes that the parent will always call thr_join() (and thus wait on the condition variable) before the child calls thr_exit() to signal. If this ordering is not guaranteed, there is a race condition:
  - If the child calls thr_exit() before the parent starts waiting on pthread_cond_wait, the signal (pthread_cond_signal) may be missed because condition variables do not queue signals if no thread is waiting at that moment. As a result, the parent could block indefinitely on pthread_cond_wait.

Incorrect: CV with Flag & Lock

```
//Child
bool child_done = false; //Shared state
void thr_exit(){
  pthread_mutex_lock(&m);
  child_done = true; //Set flag
  pthread_cond_signal(&c); //Signal parent
  pthread_mutex_unlock(&m);
}
//Parent
void thr_join(){
  pthread_mutex_lock(&m);
  if(!child_done){ //Check flag
    pthread_cond_wait(&c, &m);//Wait only if
needed
  }
  pthread_mutex_unlock(&m);
}
```

- Add a Boolean flag child_done:
  - The child_done flag ensures that even if pthread_cond_signal occurs before pthread_cond_wait, the parent will not block indefinitely because it will detect that child_done is already set.
- Spurious wakeup problem:
  - The program assumes that once the parent thread is awakened, the condition child_done is guaranteed to be true. But it is possible for child_done to be false.
  - The parent thread may be awakened by the system without any signal from the child thread (i.e., pthread_cond_signal has not been called), since system-level events such as interrupts, signals, or other system-specific mechanisms may cause a thread to wake up prematurely. This is called spurious wakeups.

Correct: CV with Flag & Lock

```
//Child
bool child_done = false; //Shared state
void thr_exit(){
  pthread_mutex_lock(&m);
  child_done = true; //Set flag
  pthread_cond_signal(&c); //Signal parent
  pthread_mutex_unlock(&m);
}
//Parent
void thr_join(){
  pthread_mutex_lock(&m);
  while(!child_done){ //Check flag
    pthread_cond_wait(&c, &m);//Wait only if
needed
  }
  pthread_mutex_unlock(&m);
}
```

- Adding a Boolean flag child_done and using while(!child_done) to check the flag, makes the program robust and avoids race conditions.
  - The child_done flag ensures that even if pthread_cond_signal occurs before pthread_cond_wait, the parent will not block indefinitely because it will detect that child_done is already set.
  - The use of a while loop around pthread_cond_wait ensures correctness in case of spurious wakeups.

Summary

- Producer/consumer problem
- Condition variable
- Properly use CVs
  - Have a state
  - Use mutex to ensure no race condition
  - Recheck the state (while)

Quiz: Race Conditions

Consider the two threads each executing t1 and t2. Values of shared variables y and z are initialized to 0.

t1:

```
1 t1(){
2     int x;
3     x = y + z;
4 }
```

t2:

```
1 t2(){
2     y = 1;
3     z = 2;
4 }
```

Q. Give all possible final values for x and the corresponding order of execution of instructions in t1 and t2.

1) t1 runs to the end first; then t2 runs to the end: x = 0+0 = 0

2) t2 to line 2; then t1 to the end; then t2 to the end: x = 1+0 = 1

3) t2 to the end; then t1 to the end: x = 1+2 = 3

Are there other possibilities giving additional values?

Quiz: Race Conditions

- Addition operation x=y+z consist of multiple machine instructions in assembly language:
  - A. fetch operand y into register r1
  - B. fetch operand z into register r2
  - C. add r1 + r2, store result in r3
  - D. store r3 in memory location of x
- If a task switch to t2 occurs between machine instructions A and B; then t2 runs to completion before switching back to t1, then:
  - y is read as 0 (t2 didn't set y yet)
  - z is read as 2 (t2 sets z before execution instruction B of add. in t1)
  - the sum is then x = 0 + 2 = 2

t1:

```
1 t1() {
2     int x;
3     x = y + z;
4 }
```

t2:

```
1 t2() {
2     y = 1;
3     z = 2;
4 }
```

Quiz: Race Conditions

Q. Give a solution using semaphores and wait/signal operations.

Solution: we protect the addition x = y + z within a *critical section, using* a binary semaphore (mutex). This code guarantees that x can *never* have the value 1 or 2, possible values are x = 0, 3 (Line "int x" can be outside or inside the critical section with no difference.)

t1:

```
0 sem s = 1;
1 t1(){
2    int x;
3    s.wait();
4    x = y + z;
5    s.signal();
6 }
```

t2:

```
1 t2(){
2    s.wait();
3    y = 1;
4    z = 2;
5    s.signal();
6 }
```

Quiz: Semaphores

t1:

```
1 int t1() {
2    printf("w");
3    printf("d");
4 }
```

t2:

```
1 int t2() {
2    printf("o");
3    printf("r");
4    printf("l");
5    printf("e");
6 }
```

Q. Use semaphores and insert wait/signal calls into the two threads so that only “wordle” is printed.

t1:

```
0 sem s1=1,s2=0;
1 int t1() {
2    s1.wait();
3    printf("w");
4    s2.signal();
5    s1.wait();
6    printf("d");
7    s2.signal();
8 }
```

t2:

```
1 int t2() {
2    s2.wait();
3    printf("o");
4    printf("r");
5    s1.signal();
6    s2.wait();
7    printf("l");
8    printf("e");
9 }
```

- t1 has to run first to print "w", so s1 should be initialized to 1.
- t2 has to wait until the "w" has been printed by t1, then it is woken up by t1 calling s2.signal(), so s2 should be initialized to 0.

Quiz: Semaphores II

```
1 int t1() {
2   while(1) {
3     printf("A");
4     s_c.signal();
5     s_a.wait();
6   }
7 }
```

```
1 int t2() {
2   while(1) {
3     printf("B");
4     s_c.signal();
5     s_b.wait();
6   }
7 }
```

```
1 int t3() {
2   while(1) {
3     s_c.wait();
4     s_c.wait();
5     printf("C");
6     s_a.signal();
7     s_b.signal();
8   }
9 }
```

```
semaphore s_a=0, s_b=0, s_c=0;
```

Q. Which strings can be output when running the 3 threads in parallel?

- Either t1 or t2 could start first, so the first letter can be A or B
- Then both t1 and t2 signal s_c, only after both have signalled s_c, t3 can start and print C
- t3 signals s_a and s_b, which start in arbitrary order again
- Accordingly, the output is a regular expression ((AB|BA)C)+
  - Print A or B in arbitrary order, then print C, then the process repeats

Quiz: Deadlocks

```
int x=0, y=0, z=0;
semaphore lock1=1, lock2=1;
```

```
1 int t1() {
2     z = z + 2;
3     lock1.wait();
4     x = x + 2;
5     lock2.wait();
6     lock1.signal();
7     y = y + 2;
8     lock2.signal();
9 }
```

```
1 int t2() {
2     lock2.wait();
3     y = y + 1;
4     lock1.wait();
5     x = x + 1;
6     lock1.signal();
7     lock2.signal();
8     z = z + 1;
9 }
```

a. Executing the threads in parallel could result in a deadlock. Why?

- **t1 runs first until line 4 (so lock1=0, lock2=1) ↯ switch to t2**
- **t2 starts and runs until line 3 (so lock1=0, lock2=0) ↯ back to t1**
- **t1 waits for lock2 in line 5 ↯ switch to t2, waits for lock1 in line 4**
- **This results in a *mutual waiting condition* which is not resolved**

Note that this deadlock does not occur in all execution/task switch orders!

Quiz: Deadlocks

```
int x=0, y=0, z=0;
semaphore lock1=1, lock2=1;
```

```
1 int t1() {
2     z = z + 2;
3     lock1.wait();
4     x = x + 2;
5     lock2.wait();
6     lock1.signal();
7     y = y + 2;
8     lock2.signal();
9 }
```

```
1 int t2() {
2     lock2.wait();
3     y = y + 1;
4     lock1.wait();
5     x = x + 1;
6     lock1.signal();
7     lock2.signal();
8     z = z + 1;
9 }
```

Q. Executing the threads in parallel could result in a deadlock. Why?

- t2 runs first until line 2 (so lock2=0, lock1=1); switch to t1
- t1 starts and runs until line 3 (so lock1=0, lock2=0); back to t2
- t2 waits for lock2 in line 4; switch to t1, waits for lock1 in line 5

Note: There are other possible interleavings, as long as each thread grabs one lock and requests the other. You can remove all other statements and only leave the lock wait() instructions and get into this deadlock.)

Quiz: Deadlocks

- Q. What are the possible values of x, y and z in the deadlock state?
- t1 runs until Line 5 lock2.wait() and t2 runs until Line 4 lock1.wait(), so x = 2, y = 1, z = 2
- Q. What are the possible values of x, y and z if the program finishes successfully without a deadlock?
- t1 runs first to the end, then t2 (or vice versa): x=3, y=3, z=3
- In t1, lock1.signal() sets lock1=1, lock2.signal() sets lock2=1, this exiting the critical sections protected by lock1 and lock2.
- Since Line 2 of t1 "z=z+2", and Line 8 of t2 "z=z+1" are not protected within a critical section, a thread switch may occur in the middle of each line, e.g.,
  - t2 Line 8 reads z=0; before z is written back; switch to t1 Line 2, run t1 to the end; switch to t2 Line 8, write back z=0+1=1.
  - Or, t1 Line 2 reads z=0; before z is written back; switch to t2 Line 2, run t2 to the end; switch to t1 Line 2, write back z=0+2=2.
- Note: to prevent deadlocks, every thread should acquire locks in the same order, e.g. both acquire lock1 before lock2, or both acquire lock2 before lock1

```
int x=0, y=0, z=0;
semaphore lock1=1, lock2=1;
```

```
1 int t1() {
2    z = z + 2;
3    lock1.wait();
4    x = x + 2;
5    lock2.wait();
6    lock1.signal();
7    y = y + 2;
8    lock2.signal();
9 }
```

```
1 int t2() {
2    lock2.wait();
3    y = y + 1;
4    lock1.wait();
5    x = x + 1;
6    lock1.signal();
7    lock2.signal();
8    z = z + 1;
9 }
```