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

```
#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;
}
```

```
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!
- Note: threads in the same process share the same memory space, but have separate registers. So in both threads, [x9] refers to the same memory address at x9, but w8 refers to different registers in each thread.

counter++;

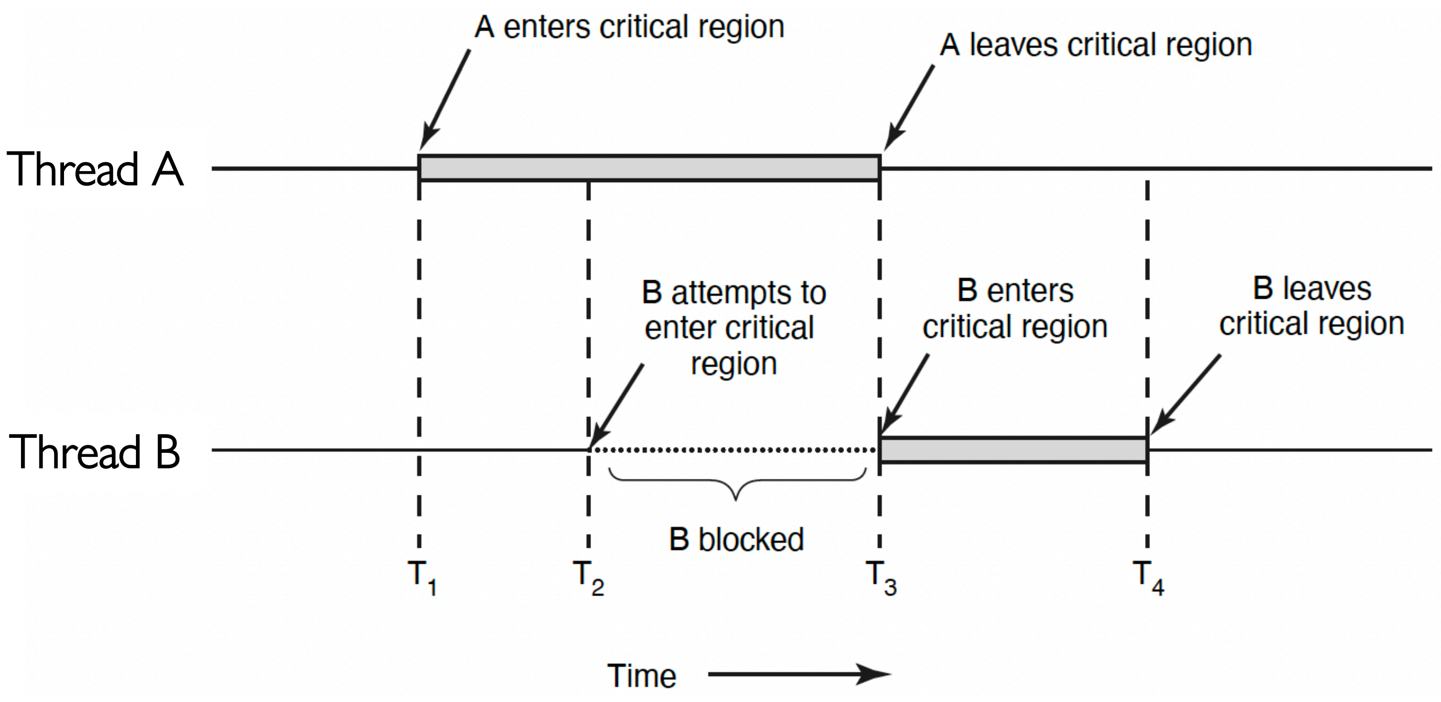
```
ldr w8, [x9]
add w8, w8, #0x1
str w8, [x9]
```
Thread 1

```
ldr w8, [x9]
add w8, w8, #0x1
str w8, [x9]
```
Thread 2

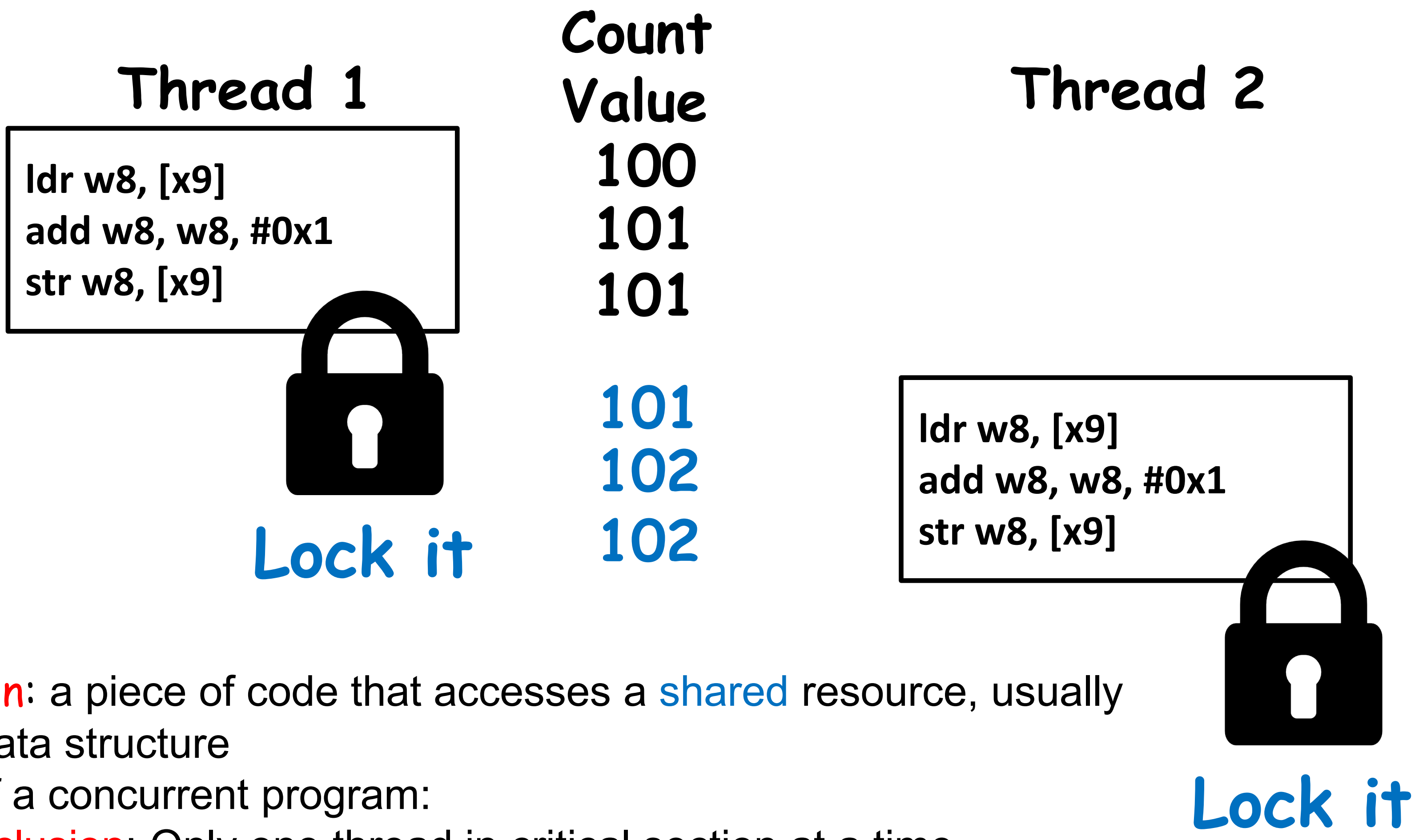
Thread 1	**counter**	**Thread 2**
ldr w8, [x9]	100	
add w8, w8, #0x1	101	
str w8, [x9]		
	100	ldr w8, [x9]
	101	add w8, w8, #0x1
	101	str w8, [x9]
str w8, [x9]	101	

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



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

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

- *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;
}
```

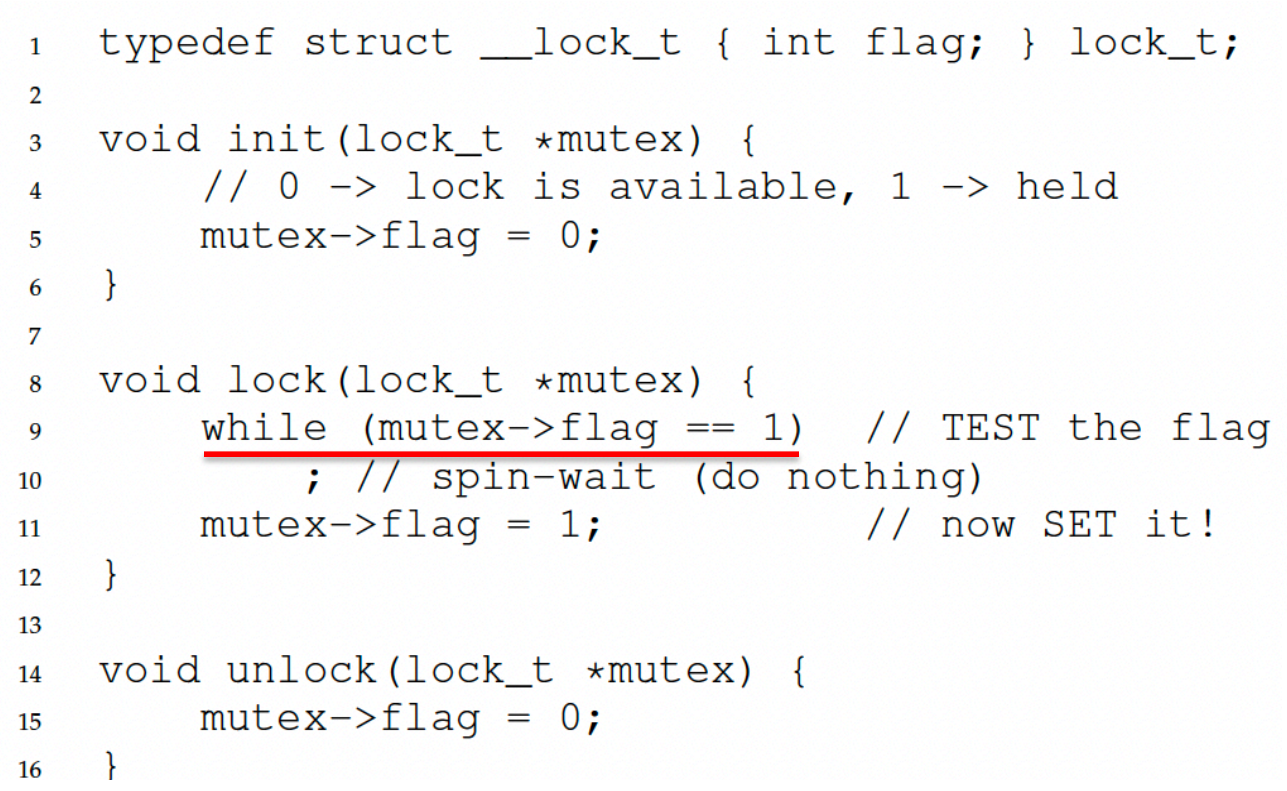
Locks: Disable Interrupts

- 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, as disabling interrupts on all processor cores requires inter-core messages and would be very time consuming

```
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!

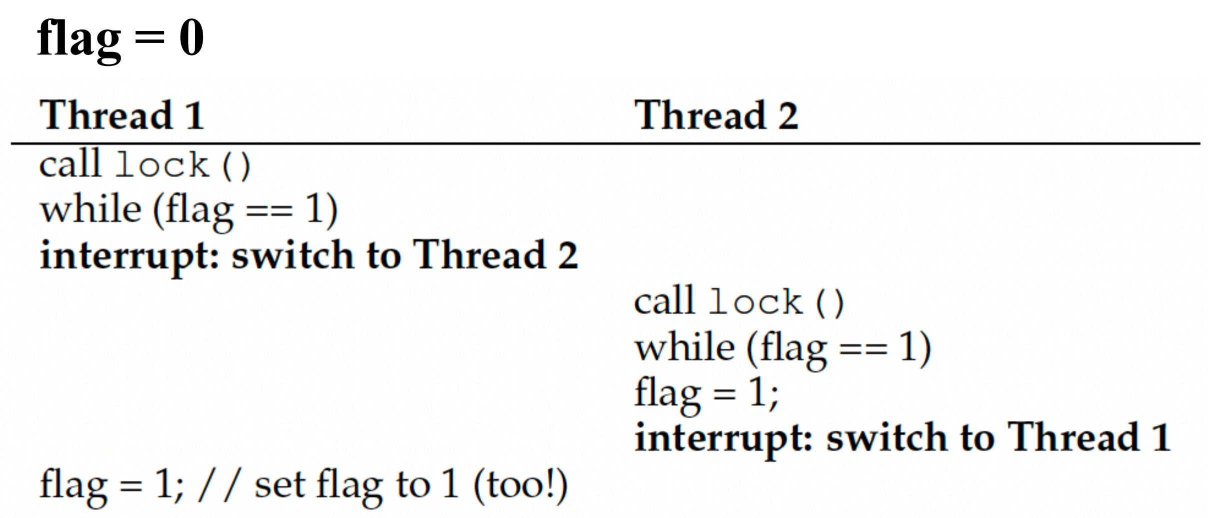


```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```



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 **Test-and-Set** (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 spin-waits in the while loop and does not enter critical section
- If multiple threads call TAS when lock-flag==0, only one thread will see lock-flag==0 , set flag=1 and enter the critical section, and all the other threads will see flag==1 and spin-wait.

```
typedef struct __lock_t{
  int flag;
} lock_t;

int TestAndSet(int *old_ptr, int new){
  int old = *old_ptr; // fetch old value at old_ptr
  *old_ptr = new;     // store 'new' into old_ptr
  return old;          // return the old value
}

void lock(lock_t *lock){
  while (TestAndSet(&lock->flag, 1) == 1)
    ;  // spin-wait
}

void unlock(lock_t *lock){
  lock->flag = 0;
}
```

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

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

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

TAS vs. CAS

Feature	Test-and-Set (TAS)	Compare-and-Swap (CAS)
Operation	Sets a bit and returns its old value	Compares current value with expected value and swaps if equal
Parameters	Single memory location	Memory location, expected value, new value
Consensus Number	Limited to 2	Arbitrary number of processes
Use Cases	Simple spinlocks	Complex synchronization primitives like mutexes
Efficiency	Faster for simple locks	More versatile but computationally heavier

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;
}
```

Ticket Lock

- A ticket lock is a synchronization mechanism used in multithreaded programming to ensure that threads acquire a lock in the order they request it. It uses two counters:
  - tickets (or next_ticket): Tracks the next "ticket number" to be assigned to a thread requesting the lock.
  - turn: Tracks the "ticket number" of the thread currently holding the lock.
- Lock Acquisition (lock()):
  - A thread atomically increments the tickets counter (using fetch-and-add) and receives its "ticket number.“
  - The thread then spin-waits until its ticket number matches the turn counter, indicating it is its turn to enter the critical section.
- Lock Release (unlock()):
  - When a thread finishes its critical section, it increments the turn counter, signaling that the next thread in line can proceed.
  - This ensures that threads are served in a first-come, first-served (FCFS) manner, preventing starvation and ensuring fairness.

```
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(), A enters CS	1	0
B lock(), spin-waits	2	0
C lock(), spin-waits	3	0

	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(), A enters CS	1	0
B lock(), spin-waits	2	0
C lock(), spin-waits	3	0
A unlock(), B enters CS	3	1
A lock(), spin-waits	4	1

	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(), A enters CS	1	0
B lock(), spin-waits	2	0
C lock(), spin-waits	3	0
A unlock(), B enters CS	3	1
A lock(), spin-waits	4	1
B unlock(), C enters CS	4	2
C unlock(), A enters CS	4	3
A unlock()	4	4

	myturn
A	3
B	1
C	2

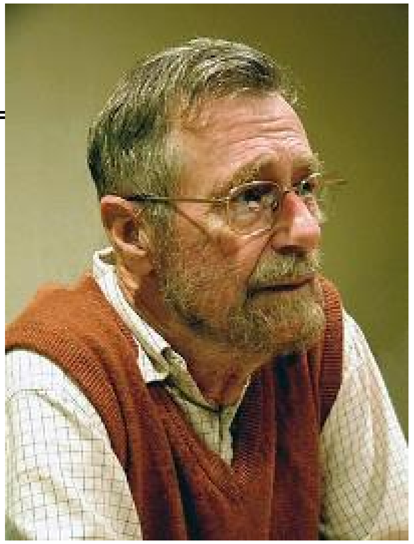
```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

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

Outlines

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

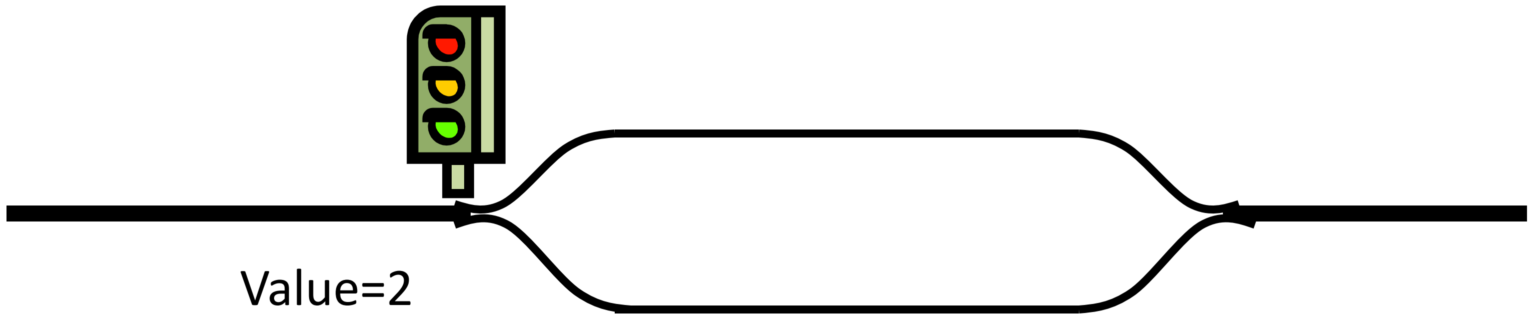
Semaphores



- Semaphores were proposed by a Dutch computer scientist Dijkstra in late 60s
- Definition: a Semaphore has a non-negative integer value and supports the following operations:
  - Set value when you initialize
  - `wait():` also called Down() or P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
  - `signal():` also called Up() or V(): an atomic operation that increments the semaphore by 1, waking up a waiting thread, if any

Semaphores Like Integers Except…

- Semaphores are like integers, except:
  - No negative values
  - Only operations allowed are wait() and signal() – cannot read or write value, except initialization
  - Operations must be atomic
    - » Two calls to wait() together can't decrement value below zero
    - » Thread going to sleep in wait() won't miss wakeup from signal() – even if both happen concurrently
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2, to allow two trains to enter the two tracks in the middle



Implementing Semaphores w/ test&set

Use TAS, but only busy-wait to atomically check `guard` value (very short waiting time)

```
int guard = 0;
int value = 0;

wait() {
   // Short busy-wait time
   while (TestAndSet(guard));
   if (value == 0) {
      guard = 0;
      put thread on wait queue;
      sleep();
   } else {
      value = value - 1;
      guard = 0;
   }
}
```

```
signal() {
   // Short busy-wait time
   while (TestAndSet(guard));
   if any thread in wait queue {
      take thread off wait queue;
      place on ready queue;
   } else {
      value = value + 1;
   }
   guard = 0;
}
```

Two Uses of Semaphores

Mutual Exclusion (value = 0 or1)

- Also called “Binary Semaphore” or “mutex”.
- Can be used for mutual exclusion as a lock:

```
sem.wait();
  //Critical section
sem.signal();
```

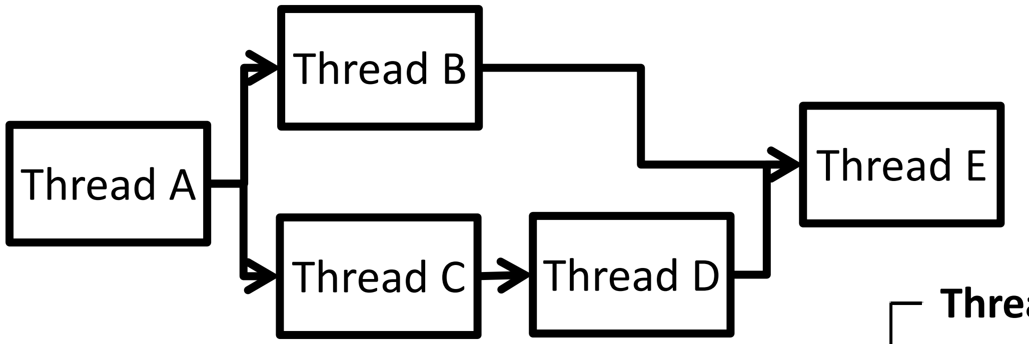
Scheduling Constraints (value >= 0)

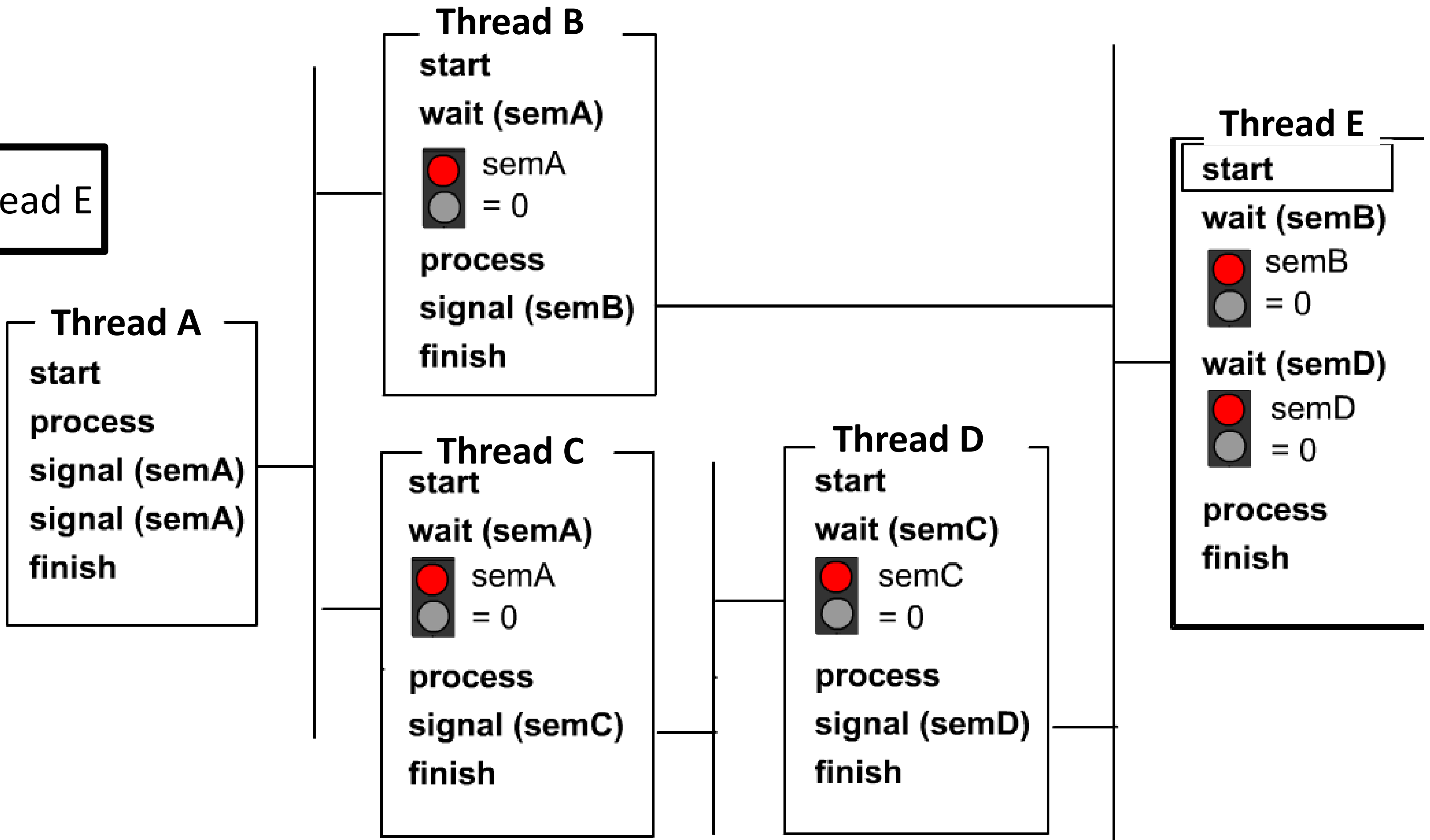
- Allow thread 1 to wait for a signal from thread 2
  - thread 2 wakes up thread 1 when a given event occurs
- Example: a thread calls ThreadJoin to wait for another thread to finish:

```
sem = 0;
ThreadJoin{
   sem.wait();
}
ThreadFinish{
   sem.signal();
}
```

Using Semaphores for Scheduling

- Consider 5 threads A, B, C, D, E. They must execute based on the partial ordering below, regardless of the ordering of process start (e.g., if E starts before B and D finishes, it will be blocked waiting for B and D to finish before it can execute)

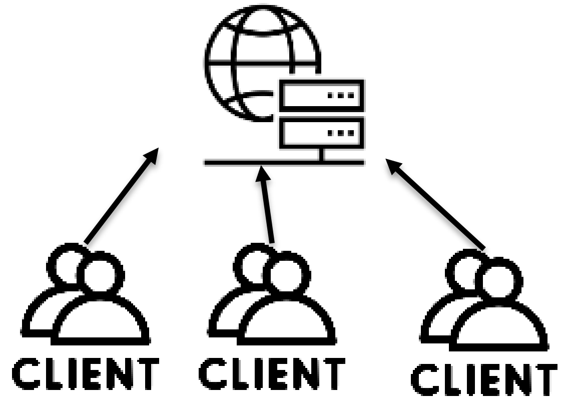


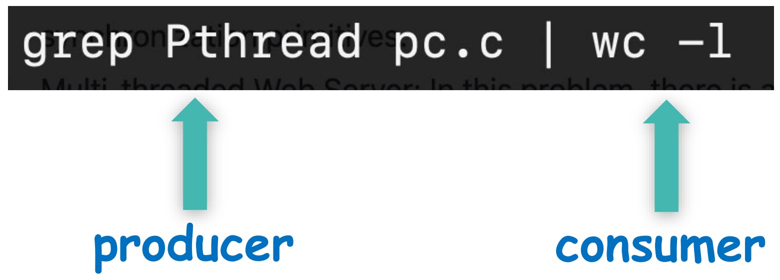


Syntax here is slightly different: wait(sem) and signal(sem) instead of sem.wait() and sem.signal().

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
  - **Linux Pipes**
  - **Coke vending machine**
    - » Producer can put limited number of cokes in machine
    - » Consumer can't take cokes out if machine is empty



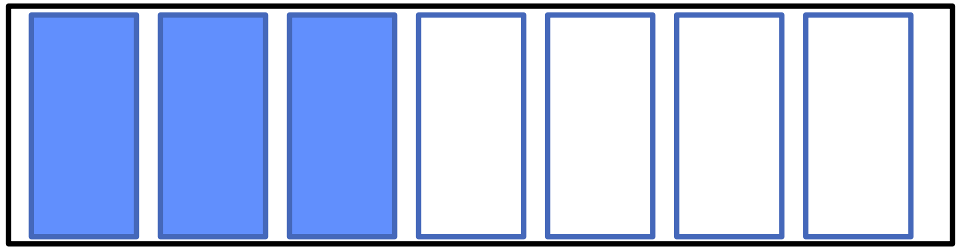


```
grep Pthread pc.c | wc -l
```

Producer/Consumer Problem

- Correctness Constraints:
  - When buffer is full, producer must wait
  - When buffer is empty, consumer must wait
  - Only one thread can manipulate buffer at a time (mutual exclusion)
- Use a separate semaphore for each constraint
  - Semaphore fullSlots; // consumer's constraint
  - Semaphore emptySlots;// producer's constraint
  - Semaphore mutex;   // mutual exclusion

Producer writes data items to buffer



Bounded buffer
fullSlots==3, emptySlots==4

Consumer reads and removes data items from buffer (destructive read)

Solution to P/C Problem

```
semaphore fullSlots = 0; //Initially, no data item
semaphore emptySlots = bufSize;//Initially, all slots empty
semaphore mutex = 1;

Producer(item) {
    emptySlots.wait();//Wait until empty slots available
    mutex.wait();
    Enqueue(item);
    mutex.signal();
    fullSlots.signal();
}
Consumer() {
    fullSlots.wait();//Wait until full slots available
    mutex.wait();
    item = Dequeue();
    mutex.signal();
    emptySlots.signal();
    return item;
}
```

fullSlots signals 1 more data item

mutex protects integrity of the queue within critical sections

emptySlots signals 1 more empty slot

Discussion about Solution

Decrease # of empty slots

Increase # of occupied slots

- Two semaphores
  - Producer does: emptyBuffer.wait(), fullBuffer.signal()
  - Consumer does: fullBuffer.wait(), emptyBuffer.signal()

Decrease # of occupied slots

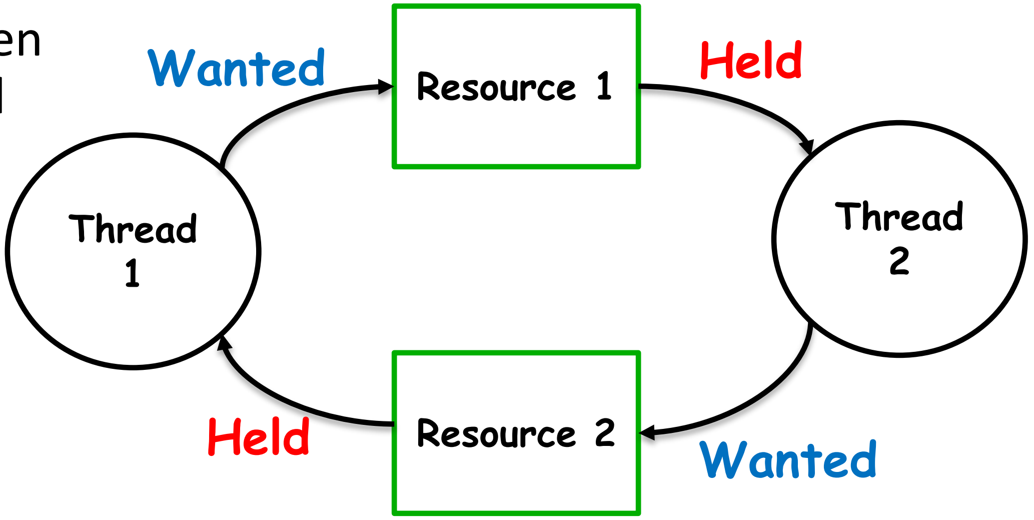
Increase # of empty slots

- Can we put mutex.wait()/signal() outside of emptySlots and fullSlots wait()/signal()?
- No! This may cause deadlock if producer calls mutex.wait(), blocks waiting for Consumer to call mutex.signal. Consumer calls fullSlots.wait(), blocks waiting for Producer to call fullSlots.signal().

```
Producer(item) {
    mutex.wait();
    emptySlots.wait();
    Enqueue(item);
    fullSlots.signal();
    mutex.signal();
 }
 Consumer() {
    mutex.wait();
    fullSlots.wait();
    item = Dequeue();
    emptySlots.signal();
    mutex.signal();
    return item;
 }
```

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
  - Only one thread at a time can use a given resource
- 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; can be released only voluntarily by each holder
- Circular wait
  - There exists a circle of threads such that each holds one or more resources that are being requested by next thread in the circle



Not a perfect analogy, just a fun image!

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)

- Semaphores are dual purpose: They are used for both mutex and scheduling constraints
- 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;
}
```

Parent

wait

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

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.

Recap

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

```
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
    }
}
```

```
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
    }
}
```

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**

Summary

- Locks: provide **mutual exclusion**
- Spinlocks:
  - Test-And-Set()
  - Compare-and-swap()
- 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:

```
t1(){
    int x;
    x = y + z;
}
```

t2:

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

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:

```
sem s = 1;
t1(){
   int x;
   s.wait();
   x = y + z;
  s.signal();
}
```

t2:

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

Quiz: Semaphores

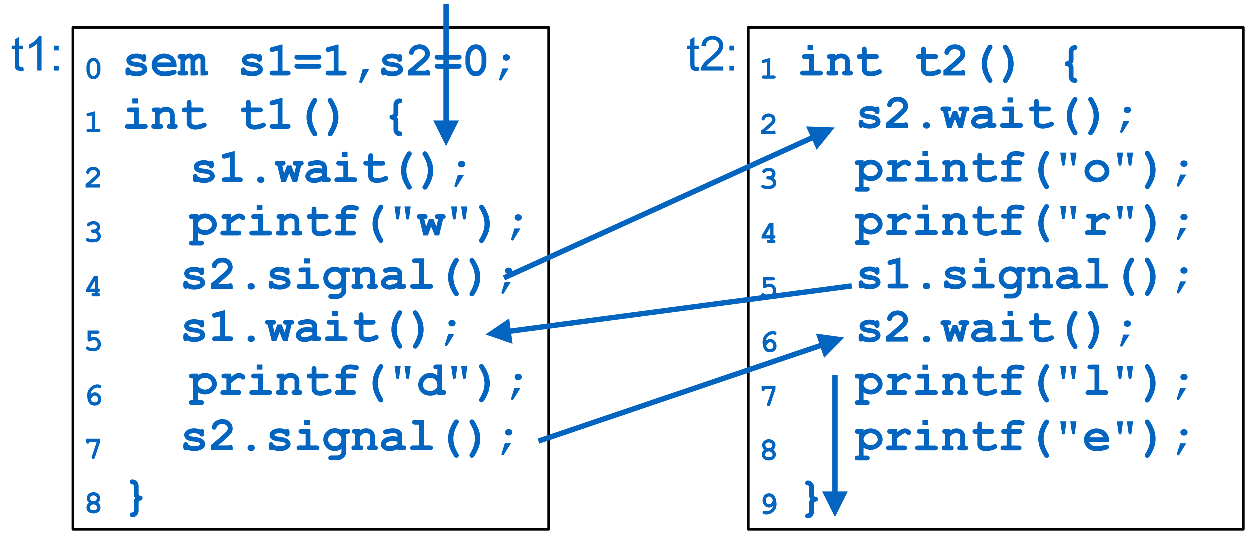
t1:

```
int t1() {
    printf("w");
    printf("d");
}
```

t2:

```
int t2() {
    printf("o");
    printf("r");
    printf("l");
    printf("e");
}
```

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



t1:

```
sem s1=1,s2=0;
int t1() {
    s1.wait();
    printf("w");
    s2.signal();
    s1.wait();
    printf("d");
    s2.signal();
}
```

t2:

```
int t2() {
    s2.wait();
    printf("o");
    printf("r");
    s1.signal();
    s2.wait();
    printf("l");
    printf("e");
}
```

- 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

```
int t1() {
   while(1) {
      printf("A");
      s_c.signal();
      s_a.wait();
   }
}
```

```
int t2() {
   while(1) {
      printf("B");
      s_c.signal();
      s_b.wait();
   }
}
```

```
int t3() {
   while(1) {
      s_c.wait();
      s_c.wait();
      printf("C");
      s_a.signal();
      s_b.signal();
   }
}
```

```
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 }
```