

CSC 112: Computer Operating Systems

Lecture 3

Synchronization

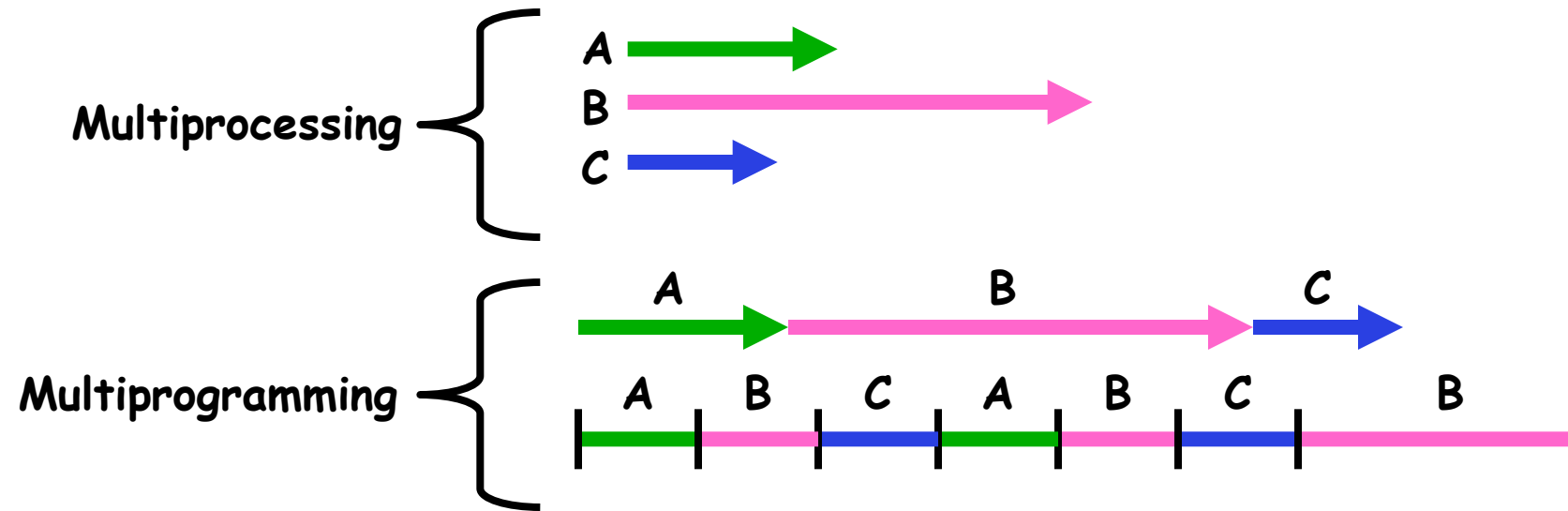
Department of Computer Science,
Hofstra University

Outline

- Concurrency & Spinlocks
- Semaphores
- Monitors

Different Types of Concurrency

- Multiprocessing → multiple CPUs running in parallel
- Multiprogramming → multiple processes
- Multithreading → multiple threads per process



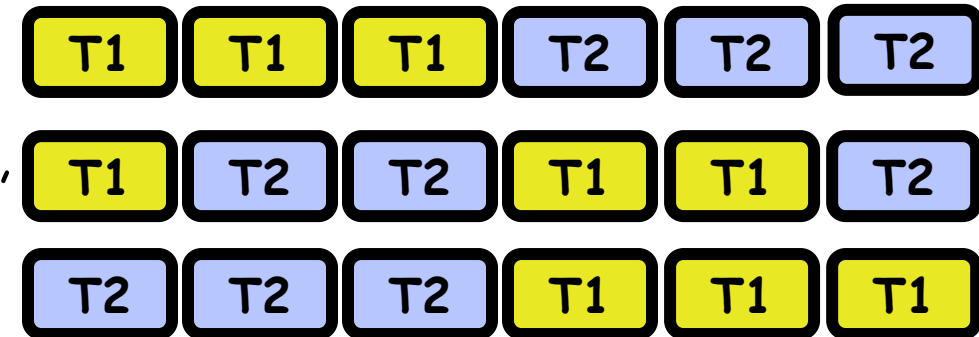
Concurrency

```
#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 non-deterministic interleavings of worker threads



Race Condition

- Incrementing **counter** has **3 instructions** in assembly code:
- **ld 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
- **st 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++;
```

```
ld w8, [x9]
add w8, w8, #0x1
st w8, [x9]
```

Thread 1

```
ld w8, [x9]
add w8, w8, #0x1
st w8, [x9]
```

Thread 2

st w8, [x9]

counter

Thread 1

```
ld w8, [x9]
add w8, w8, #0x1
st w8, [x9]
```

Thread 2

100
101

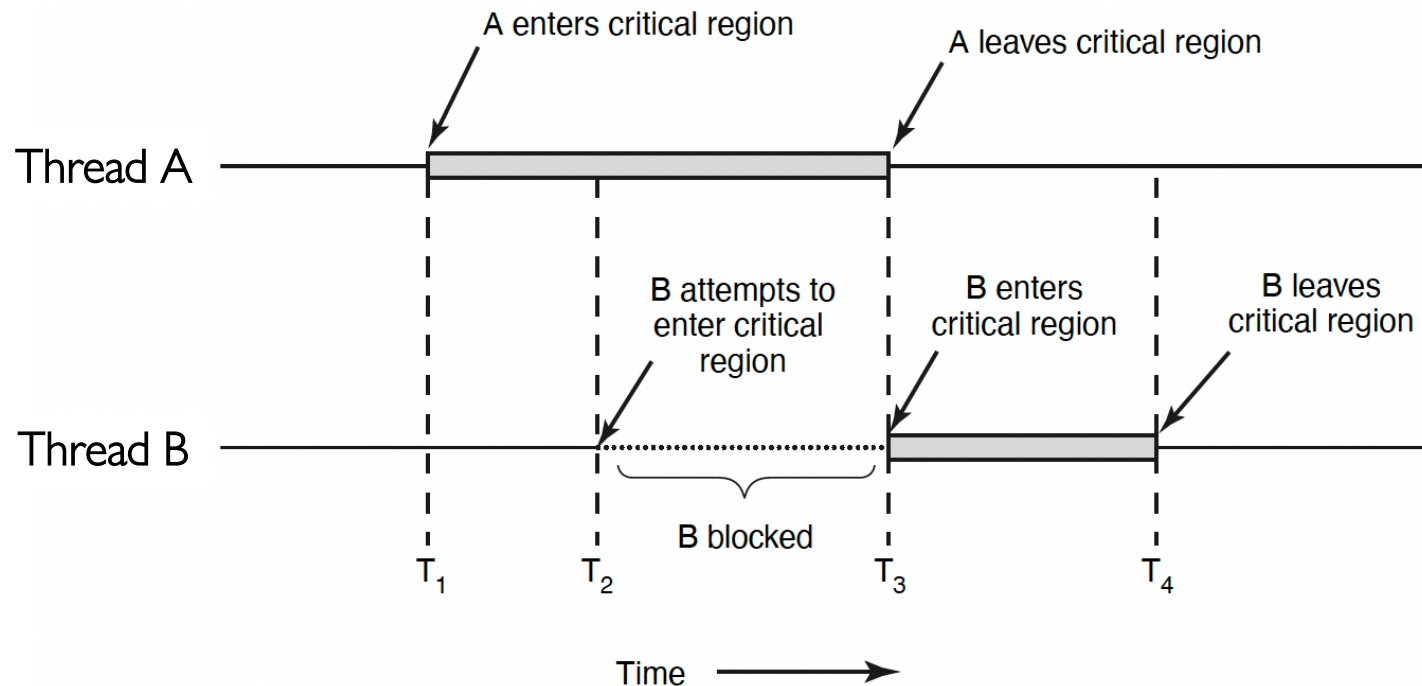
100
101
101

```
ld w8, [x9]
add w8, w8, #0x1
st w8, [x9]
```

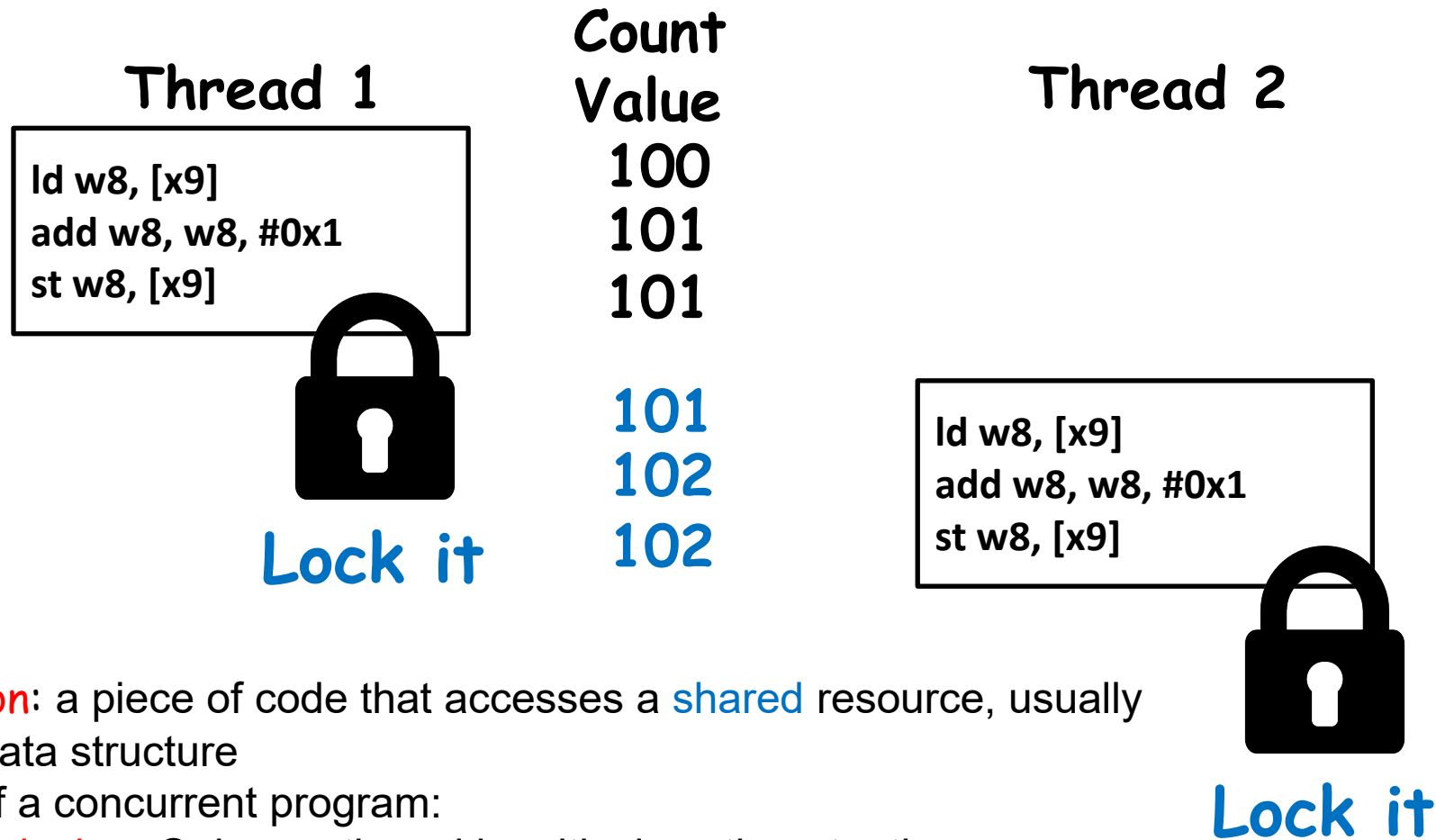
101

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



- **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!

```
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

call lock()

while (flag == 1)

interrupt: switch to Thread 2

flag = 1; // set flag to 1 (too!)

Thread 2

call lock()

while (flag == 1)

flag = 1;

interrupt: switch to Thread 1

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 it to 1 and enter the critical section, and all the other threads will see lock-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 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

```
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;
}

```

```

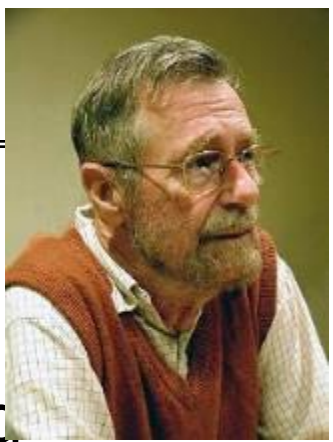
void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}

```

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**

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:
 - **sem_t sem** or **semaphore sem**: Declare a semaphore
 - **sem_init(&sem, 0, N)**: Initialize the semaphore with an initial value of 1, shared among threads (indicated by the middle 0)
 - **sem_wait(&sem)**: also called down() or P(), an atomic operation that decrements it by 1 if non-zero. If the semaphore is equal to 0, go to sleep waiting to be signaled by another thread
 - **sem_post(&sem)**: also called signal(), up() or V(), an atomic operation that increments it by 1, and wakes up a waiting/sleeping thread, if any
- Semaphores are also called sleeping locks, since the waiting thread goes to sleep instead of spin-waiting

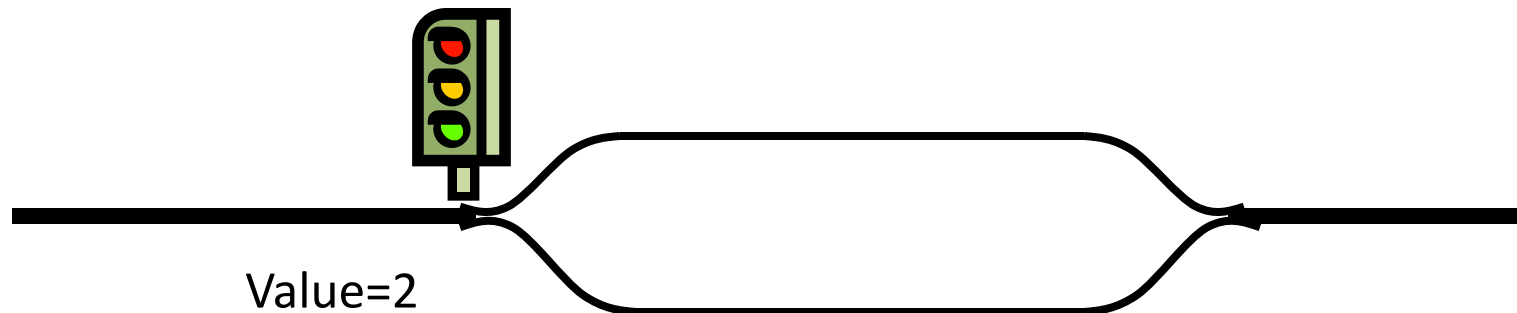
POSIX pthreads API

- A Portable Operating System Interface (POSIX) library (IEEE 1003.1c), written in C language
- In this lecture, we sometimes use some simpler notations for brevity, e.g.,
- `sem_init(&sem, 0, N)`
 - written as: semaphore `sem=N`;
- `sem_wait(&sem)`
 - written as `sem.wait()`
- `sem_post(&sem)`
 - written as `sem.signal()`

API	Functionality
<code>pthread_create</code>	Create a new thread in the caller's address space
<code>pthread_exit</code>	Terminate the calling thread
<code>pthread_join</code>	Wait for a thread to terminate
<code>pthread_mutex_lock</code>	Lock a mutex
<code>pthread_mutex_unlock</code>	Unlock a mutex
<code>sem_wait</code>	Wait on a semaphore
<code>sem_post</code>	Signal or post on a semaphore
<code>pthread_cond_wait</code>	Wait on a condition variable
<code>pthread_cond_signal</code>	Wake up one thread waiting on a condition variable
<code>pthread_cond_broadcast</code>	Wake up all threads waiting on a condition variable

Semaphores Like Integers Except...

- Semaphores are like integers, except:
 - No negative values
 - Only operations allowed are `sem_wait()` and `sem_post()` – cannot read or write value, except initialization
 - Operations must be atomic
 - » Two calls to `sem_wait()` together can't decrement value below zero
 - » A thread going to sleep in `sem_wait()` won't miss wakeup from `sem_post()` – 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 with TestAndSet

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

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

```
sem_wait() {  
    //While guard is true, spin-  
    wait  
    while (TestAndSet(guard));  
    if (value == 0) {  
        guard = 0;  
        put thread on wait queue;  
        sleep();  
    } else {  
        value = value - 1;  
        guard = 0;  
    }  
}
```

```
sem_post() {  
    //While guard is true, spin-  
    wait  
    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 or 1)


- Called “Binary Semaphore” or “mutex”. Can be used for mutual exclusion as a lock
- Example: sem is initialized to 1. The first thread that calls `sem_wait()` decrements sem to 0 and enters the critical section: other threads will be blocked when they see `sem==0`. When the first thread calls `sem_post()` to increment sem to 1, one of the waiting threads will wake up, decrement sem to 0 and enter the critical section.

```
sem_init(&sem, 0, 1):  
sem_wait(&sem);  
    //Critical section  
sem_post(&sem);
```

Scheduling Constraints (value ≥ 0)

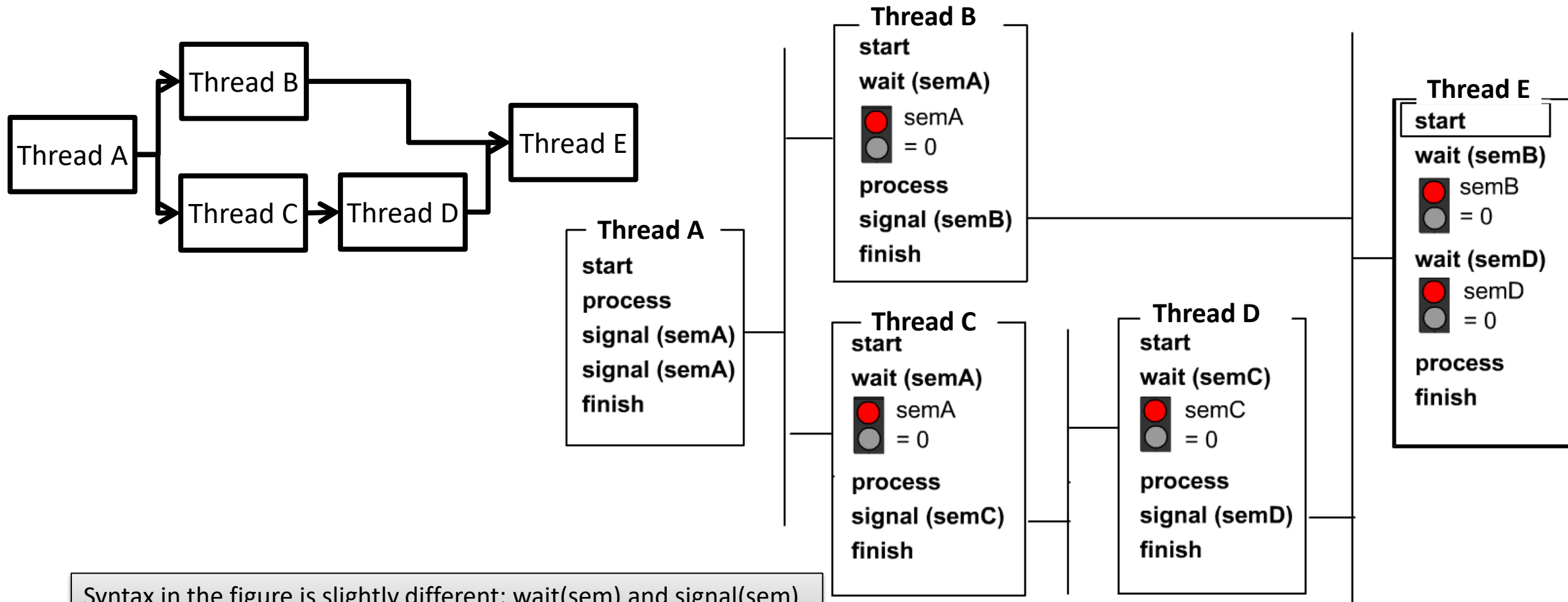
- Called “Counting Semaphore”
- Example: sem is initialized to 0. thread 1 calls `sem_wait()` in `ThreadJoin()` and is blocked; when another thread 2 calls `sem_post()` in `ThreadFinish()` to increment sem to 1, thread 1 will wake up, decrement sem to 0, and continue.

```
sem_init(&sem, 0, 0):  
ThreadJoin() {  
    sem_wait(&sem);  
}  
ThreadFinish() {  
    sem_post(&sem);  
}
```



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 in the figure is slightly different: wait(sem) and signal(sem) instead of sem_wait(&sem) and sem_post(&sem).

Readers/Writers Problem

- We have two classes of concurrent processes:
 - Writers: they change data, so only one writer can be active
 - Readers: these only read data, thus multiple readers can be active, as long as there is no active writer
- Shared Resource Conflict:
 - Multiple readers can safely access the resource at the same time, but if any writer is modifying the resource, no other process (either reader or writer) should access it. This ensures data consistency.
- Readers vs. Writers Priority:
 - If a reader is already accessing the resource, additional readers are allowed to enter immediately. A writer, however, must wait until all readers have finished. Consequently, readers are favoured over writers, which can lead to writer starvation if new readers keep arriving.

Readers/Writers Problem Solution

- A semaphore named `mutex` is used to ensure mutual exclusion when readers update a shared counter called `readcount`, which tracks the number of active readers. Another semaphore named `wrt` is used to control access to the shared resource. It is acquired by writers and by the first reader.
- First Reader Behavior: If the reader finds that it is the first one to enter (i.e., `readcount` increments from 0 to 1), it then calls `sem_wait(&wrt)` to acquire the lock `wrt`. This prevents any writer from entering the critical section while at least one reader is present.
- Last Reader Behavior: If the reader finds that it has been the last to exit (i.e., `readcount` becomes 0), it calls `sem_post(&wrt)` to allow a writer (if any are waiting) to acquire the lock `wrt` and enter the critical section.
- Writer Behavior: A writer begins by calling `sem_wait(&wrt)` to acquire the lock `wrt` and enter the critical section to write data. Since a writer must have exclusive access, it will block until `wrt` is available—that is, until no reader holds it (because the first reader acquired it) and no other writer is active. Upon exiting the critical section, it calls `sem_post(&wrt)` to allow waiting readers or writers to continue.
- Readers-Preference and Its Consequences: Because the first reader blocks any writer until all readers have exited, if new readers continuously arrive, a writer may starve. This readers-preference model is efficient for systems primarily performing read operations but might cause fairness issues when writes are necessary.

```
/* shared memory */  
semaphore mutex;  
semaphore wrt;  
int readcount;
```

```
/* initialization.*/  
mutex = 1;  
wrt = 1;  
readcount = 0;
```

```
/* writer */  
sem_wait(&wrt);  
  
... critical section  
to write data ...  
  
sem_post(&wrt);
```

```
/* reader */  
sem_wait(&mutex);  
readcount++;  
if(readcount==1)  
    sem_wait(&wrt);  
sem_post(&mutex);  
  
... read data ...  
  
sem_wait(&mutex);  
readcount--;  
if(readcount==0)  
    sem_post(&wrt);  
sem_post(&mutex);
```

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

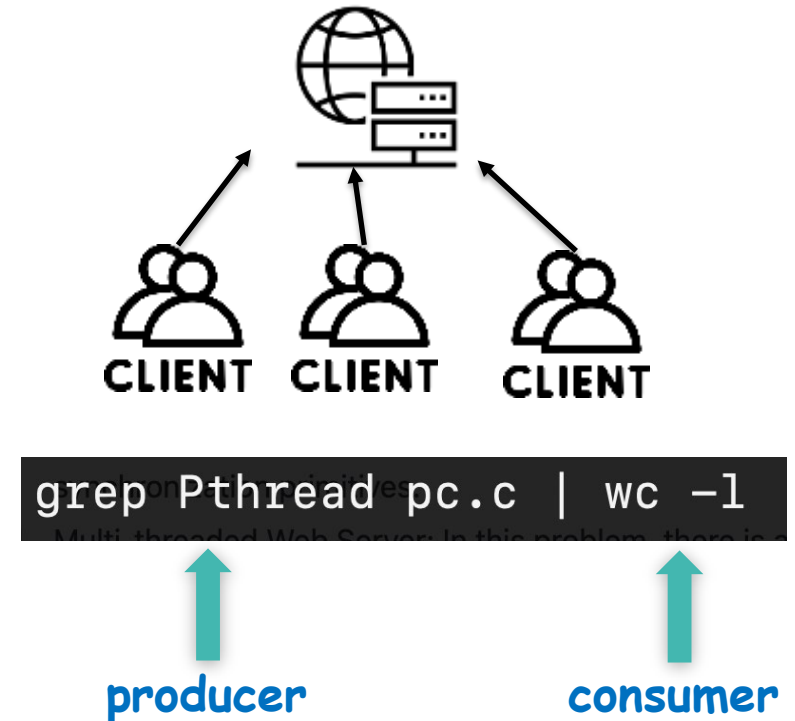
- **Linux Pipes**

- **Coke vending machine**

- » Producer can put limited number of cokes in machine
- » Consumer can't take cokes out if machine is empty

- Different from Readers/Writers problem

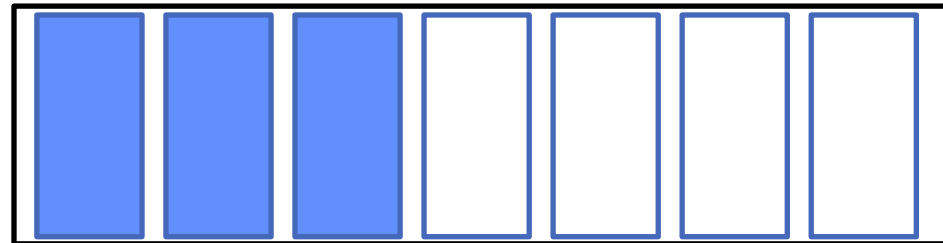
- There is a queue of items
- Consumer performs destructive read: reading an item removes it from the queue



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)

Full Solution to Bounded Buffer (coke machine)



```
semaphore fullSlots=0; //Initially, no full slots  
semaphore fullSlots=bufSize; //Initially, all slots empty
```

```
semaphore mutex=1;
```

```
Producer(item) {  
    sem_wait(&emptySlots); //Wait until emptySlots non-zero  
    sem_wait(&mutex);  
    enqueue(item);  
    sem_post(&mutex);  
    sem_post(&fullSlots);  
}
```

```
Consumer() {  
    sem_wait(&fullSlots); //Wait until fullSlots non-zero  
    sem_wait(&mutex);  
    item = dequeue();  
    sem_post(&mutex);  
    sem_post(&emptySlots);  
    return item;  
}
```

Indicates 1
more empty
slot

Indicates 1 more full slot

mutex protects
integrity of the
queue within
critical sections

emptySlots==0: Producer waits; fullSlots ==0: Consumer waits.
fullSlots>0 && emptySlots>0: Producer and Consumer can enqueue/dequeue items.
concurrently (within critical section protected by mutex).

Discussion about Solution

- Two semaphores

- Producer does: `sem_wait(&emptySlots)`, `sem_post(&fullSlots)`
- Consumer does: `sem_wait(&fullSlots)`, `sem_post(&emptySlots)`

Decrease # of
empty slots

Increase # of
occupied slots

Decrease # of
occupied slots

Increase # of
empty slots

- Can we put `sem_wait()/sem_post()` for mutex outside of `sem_wait()/sem_post()` for `emptySlots` and `fullSlots`?
- No! This may cause deadlock. Suppose the queue is initially empty. Producer enters the critical section, calls `sem_wait(&emptySlots)` and is blocked waiting for Consumer to put items into the queue; Consumer calls `sem_wait(&mutex)` and is blocked waiting to enter the critical section. But Producer will never exit the critical section and call `sem_post(&mutex)` to wake up Consumer!
- Similar deadlock situation when the queue is full, Consumer is blocked on `sem_wait(&fullSlots)` and Producer is blocked on `sem_wait(&mutex)`.

//Incorrect code

```
Producer(item) {  
    sem_wait(&mutex);  
    sem_wait(&emptySlots);  
    enqueue(item);  
    sem_post(&fullSlots);  
    sem_post(&mutex);  
}
```

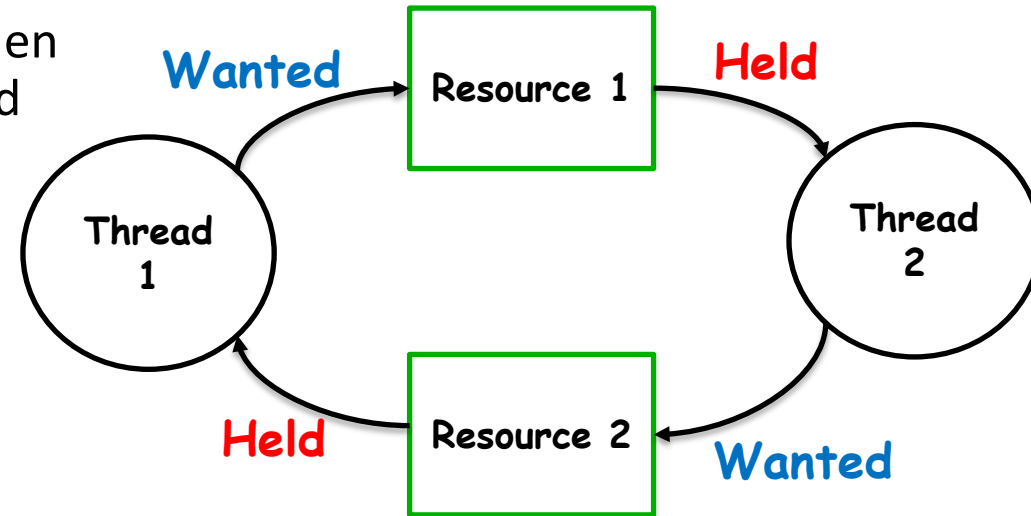


```
Consumer(item) {  
    sem_wait(&mutex);  
    sem_wait(&fullSlots);  
    enqueue(item);  
    sem_post(&emptySlots);  
    sem_post(&mutex);  
}
```



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!

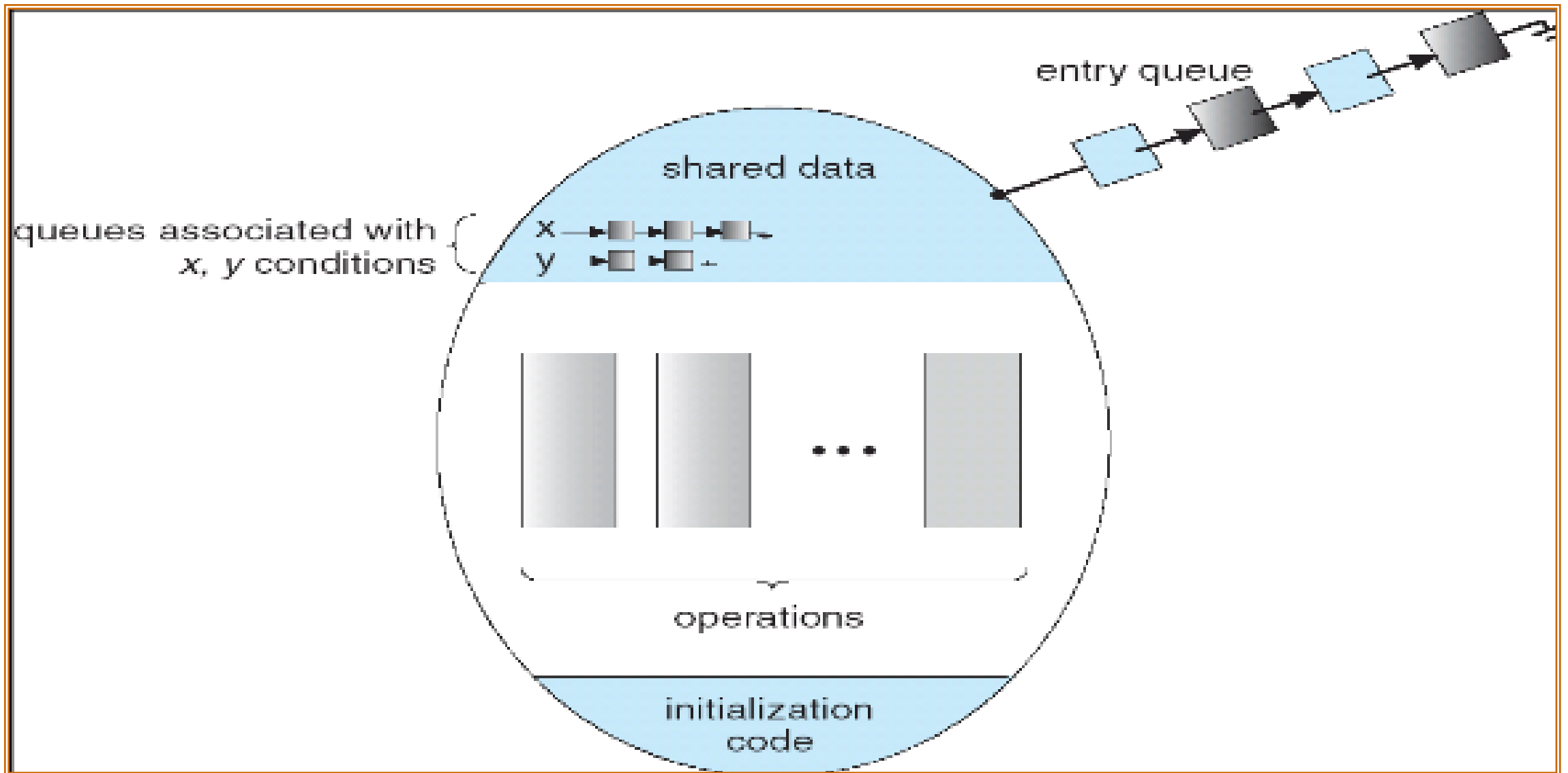
Semaphores are good but...Monitors are better!

- Semaphores are dual purpose, used for both mutex and scheduling constraints
- Monitors provide a higher-level abstraction that naturally encapsulates shared state and condition variables.
- **Monitor**: a **mutex lock** and one or more **condition variables** for managing concurrent access to shared data
 - A paradigm for concurrent programming
 - Use lock for mutual exclusion and condition variables for scheduling constraints. (Must hold lock when doing condition operations!)
 - Java supports monitors natively

Monitor with Condition Variables (CV)

- **thread_mutex_t mutex**: a mutex lock
 - Provides mutual exclusion to critical section
 - Acquire before entering, release upon exiting critical section
- **pthread_cond_t cond**: one or more condition variables:
 - For each condition variable, a queue of threads may be **waiting for it to be signaled *inside* the critical section**.
 - » Key idea: allow threads to wait on a condition variable (sleeping) inside the critical section, since the mutex lock is released (implicitly) when a thread goes to sleep
 - » Contrast with semaphores: cannot wait on a semaphore inside critical section, otherwise it leads to a deadlock since mutex lock is still held
 - There may be an entry queue of threads waiting on the lock *outside* of the critical section
- Condition operations:
 - **pthread_cond_wait(&cond, &mutex)**: it releases the mutex lock temporarily and enters the monitor's wait queue to go to sleep. This allows other threads to acquire the lock and proceed with their tasks. When the waiting/sleeping thread is signaled, it re-acquires the lock before resuming execution.
 - **pthread_cond_signal(&cond)**: Wake up one waiter, if any
 - **pthread_cond_broadcast(&cond)**: Broadcast(): Wake up all waiters

Monitor with Condition Variables (CV)



CV Common Usage Pattern

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
bool flag; //Initialization value is application-specific, hence omitted here
// Signaler thread
Signaler(){
    pthread_mutex_lock(&mutex);
    update_flag();
    //Either signal 1 thread, or broadcast to all threads, but not both
    pthread_cond_signal(&cond);
    //pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
}

// Waiter thread
Waiter(){
    pthread_mutex_lock(&mutex);
    //Thread goes to sleep during waiting
    while (!flag){pthread_cond_wait(&cond, &mutex);}
    // Process data
    pthread_mutex_unlock(&mutex);
}
```

P/C Problem with Condition Variable

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t prod_CV = PTHREAD_COND_INITIALIZER;  
pthread_cond_t cons_CV = PTHREAD_COND_INITIALIZER;
```

```
Producer(item) {  
    pthread_mutex_lock(&mutex);  
    while(buffer full) {pthread_cond_wait(&prod_CV, &mutex);}  
    enqueue(item);  
    pthread_cond_signal(&cons_CV);  
    pthread_mutex_unlock(&mutex);  
}
```

```
Consumer() {  
    pthread_mutex_lock(&mutex);  
    while(buffer empty) {pthread_cond_wait(&cons_CV, &mutex);}  
    item = dequeue();  
    pthread_cond_signal(&prod_CV);  
    pthread_mutex_unlock(&mutex);  
    return item  
}
```

This program has the same behavior as [previous program using semaphores](#).
(Code for updating buffer status and setting Boolean flags “buffer full” or “buffer empty” are omitted)

While vs. if for Checking Boolean flag

- Need to be careful about precise definition of signal and wait. Consider the dequeue code in Consumer thread:

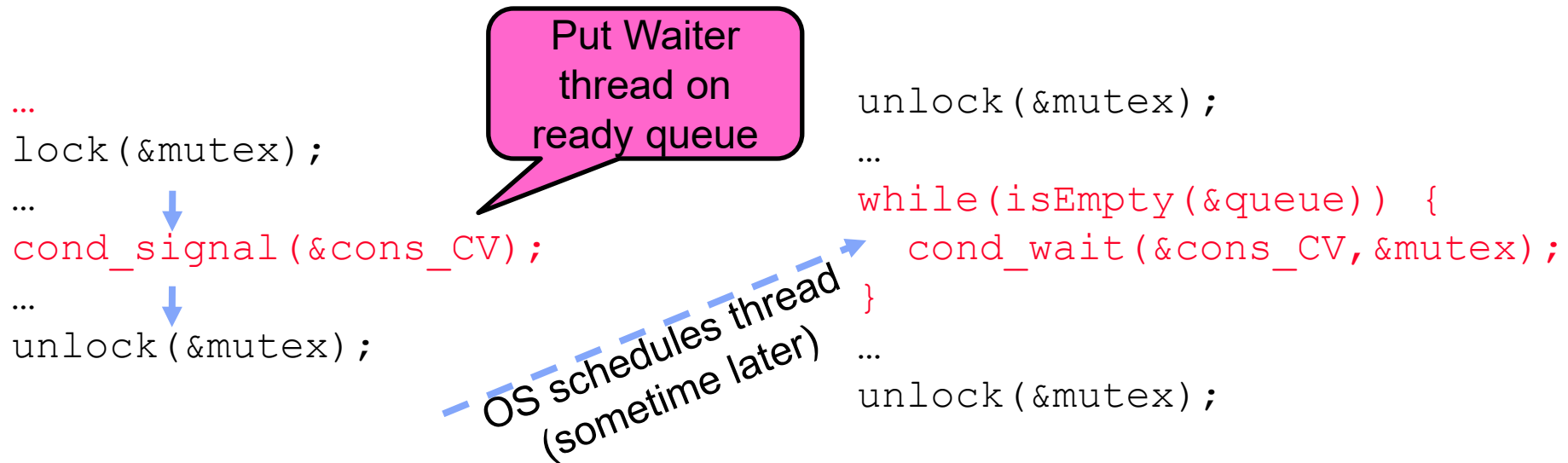
```
while (isEmpty(&queue)) {  
    cond_wait(&cons_CV, &mutex);  
}  
item = dequeue(&queue); // Get next item
```

–Why not using if(buffer empty) instead of while(buffer empty)?

- Answer: Most OSes use Mesa-style monitor (named after Xerox-Park Mesa Operating System)

Mesa monitors

- Inside `cond_wait()`, Waiter thread releases the mutex lock temporarily and enters the monitor's wait queue to go to sleep. This allows Signaler thread to acquire the mutex lock and proceed with its task.
- When Signaler thread calls `cond_signal()` to signal Waiter thread, Waiter thread is put on the ready queue (not woken up immediately). Signaler thread continues execution and releases the mutex lock. When Waiter thread gets to run on the CPU when OS actually schedules it, it re-acquires the mutex lock, exits `cond_wait()`, enters the critical section, and finally releases the mutex lock.



- Waiter thread must use a while loop to re-check condition upon wakeup
 - Another thread may be scheduled before Waiter thread gets to run, and "sneak in" to modify the state (e.g., empty the queue), so the condition may be false again (called "spurious wakeups").

Thread Join with Condition Variables

- 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;  
}
```

Child

signal

Incorrect: CV with Only Lock

```
//Declare mutex m and condition
c
//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 OK.

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			

- Declarations of mutex m and condition c omitted.
- 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.

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.
 - `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.
- Similar to Boolean flags “buffer full” and “buffer empty” in P/C problem.

Dinning Philosophers

- N philosophers sit at a round table.
- They spend their lives alternating thinking and eating.
- They do not communicate with their neighbors.
- Each philosophers occasionally tries to pick up 2 forks (one at a time) to eat
- Needs both forks to eat, then releases both when done eating

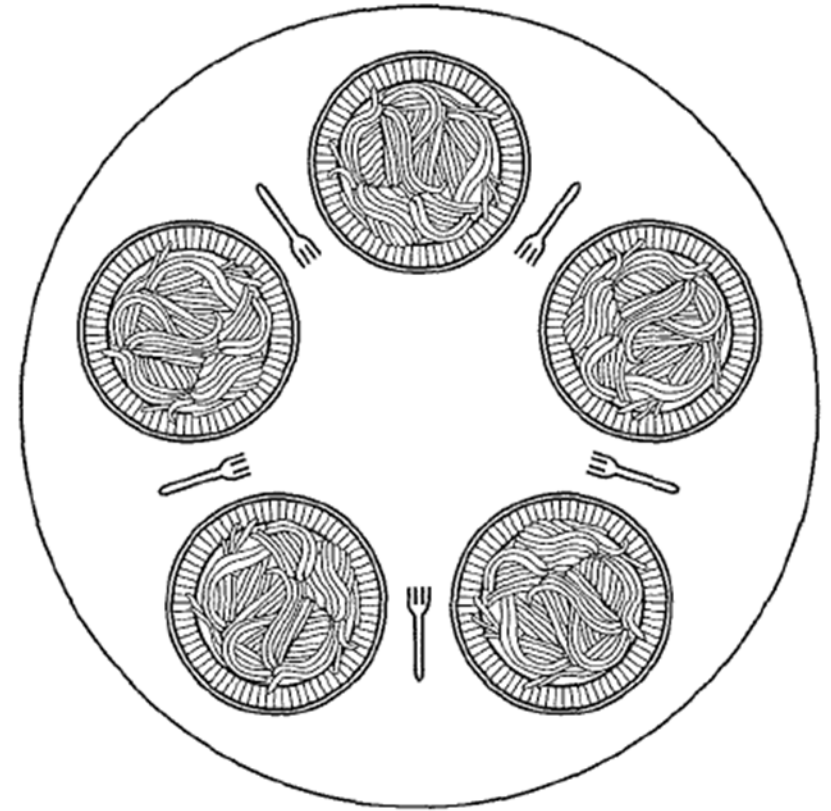
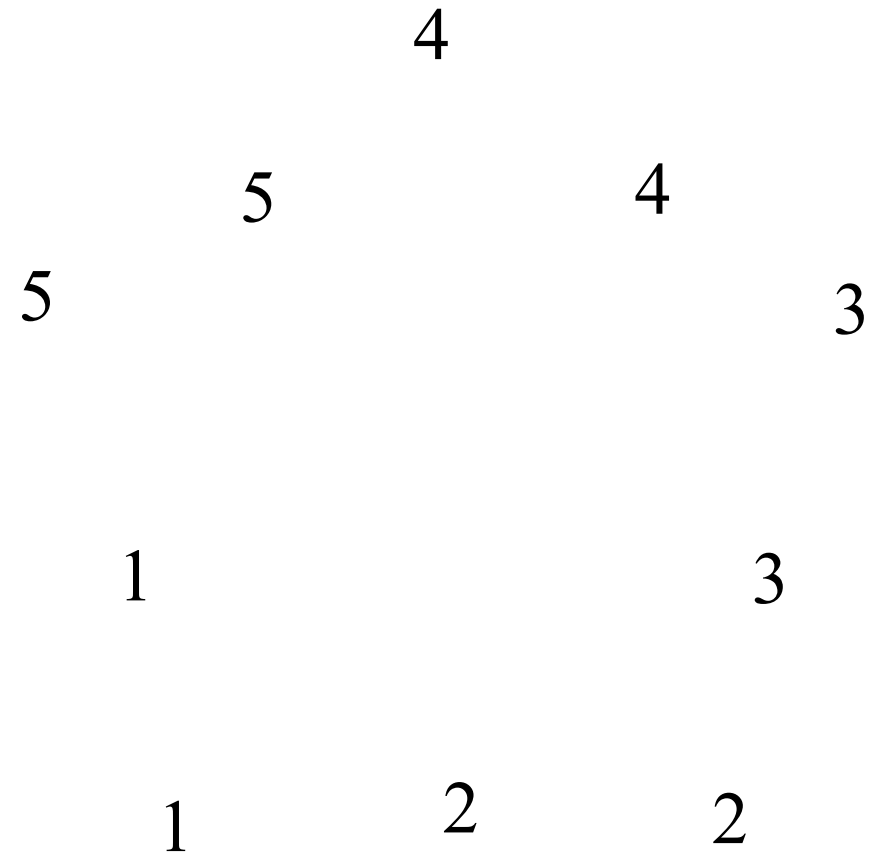


Figure 2-44. Lunch time in the Philosophy Department.

Banker's algo applied to Dining Philosophers cont'

- Model each fork as a separate resource, since each philosopher can only pick up his left and right forks.
- Suppose we have 5 philosophers numbered 1-5, and 5 forks numbered 1-5; philosopher i has left fork numbered i , and right fork $(i+1)\%5$.



Semaphore-based Solution: Incorrect

- Each fork (or chopstick) is modeled as a binary semaphore that is initially set to 1, meaning it is available. When a philosopher wants to eat, they perform a wait (or P) operation to pick up a fork and a signal (or V) operation to release it afterward. This basic model is often subject to deadlock if every philosopher simultaneously picks up one fork.
- This solution is flawed because it can lead to deadlock. In the provided code, each philosopher first executes a blocking wait to pick up the left fork and then tries to pick up the right fork. If all philosophers adopt this pattern simultaneously, every philosopher may pick up their left fork and then block waiting for the right fork (which is held by the neighbor), resulting in a circular wait where none can proceed.

```
semaphore room = 4;
semaphore fork[5] = {1, 1, 1, 1, 1};

void philosopher(int i) {
    while (true) {
        think();
        sem_wait(&room); // Limit number of
philosophers simultaneously hungry to 4
        sem_wait(&fork[i]); // Pick up left fork
        sem_wait(&fork[(i + 1) % 5]); // Pick up
right fork
        eat();
        sem_post(&fork[i]); // Put down left fork
        sem_post(&fork[(i + 1) % 5]); // Put down
right fork
        sem_post(&room); // Leave the room
    }
}
```

Semaphore-based Solution I

- One solution is to introduce an additional “room” semaphore that limits the number of philosophers permitted to start eating concurrently. For example, if there are 4 philosophers, initializing room to 4 guarantees that at least one philosopher can acquire both forks, thus breaking the circular wait condition.

```
#define N 5 // Number of philosophers and forks
semaphore fork[N] = {1, 1, 1, 1, 1};

void philosopher(int i) {
    while (true) {
        think();
        sem_wait(&fork[i]); // Pick up left fork
        sem_wait(&fork[(i + 1) % N]); // Pick up
right fork
        eat();
        sem_post(&fork[i]); // Put down left fork
        sem_post(&fork[(i + 1) % N]); // Put down
right fork
    }
}
```

Semaphore-based Solution II

- Another option is to adjust the order in which resources are requested (for instance, having one philosopher, the (N-1)-th philosopher, pick up the right fork first while the others pick up the left fork first), which disrupts the cycle that could lead to deadlock.

```
#define N 5 // Number of philosophers and forks
semaphore fork[N] = {1, 1, 1, 1, 1};

void philosopher(int i) {
    while (true) {
        think();
        if (id == N - 1) {
            sem_wait(&fork[i+1]); // Pick up right fork
            sem_wait(&fork[(i) % N]); // Pick up left fork
        }
        else {
            sem_wait(&fork[i]); // Pick up left fork
            sem_wait(&fork[(i + 1) % N]); // Pick up right fork
        }
        eat();
        if (id == N - 1) {
            sem_post(&fork[i+1]); // Put down right fork
            sem_post(&fork[(i) % N]); // Put down left fork
        }
        else {
            sem_post(&fork[i]); // Put down left fork
            sem_post(&fork[(i + 1) % N]); // Put down right fork
        }
    }
}
```


Monitor-based Solution

- A monitor `self[i]` is created for each philosopher `i`.
- Each philosopher can be in any one of three states (THINKING, HUNGRY, or EATING). All philosophers have initial state of THINKING.
- When philosopher `i` becomes hungry, he calls `pickup(i)` inside the monitor, which sets their state to HUNGRY and calls `test(i)` to check if any neighbor is eating.
- If both adjacent philosophers are not eating, philosopher `i`'s state is changed to EATING; otherwise, the philosopher waits on a condition variable.
- Upon finishing eating, philosopher `i` calls `putdown(i)`, updates their state to THINKING, and then tests if adjacent philosophers can now eat by signaling their condition variables. This structure prevents the circular waiting condition that leads to deadlock.
- This solution requires that both forks are available before the philosopher can begin eating, thus naturally avoiding deadlock. Although monitors simplify mutual exclusion by bundling synchronization within a single construct, they may still allow for starvation if a waiting condition is triggered arbitrarily by the signal mechanism.

```
#define N 5 // Number of philosophers and forks
enum { THINKING, HUNGRY, EATING } state[N];
//All state[i] are initialized to THINKING
mutex_t m;
condition self[N];

void pickup(int i) {
    mutex_lock(&m);
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
        wait(&self[i], &m); //Wait until the philosopher can eat
    mutex_unlock(&m);
}

void putdown(int i) {
    mutex_lock(&m);
    state[i] = THINKING;
    test((i + 4) % N); // Test left neighbor
    test((i + 1) % N); // Test right neighbor
    mutex_unlock(&m);
}

void test(int i) {
    if (state[i] == HUNGRY &&
        state[(i + 4) % N] != EATING &&
        state[(i + 1) % N] != EATING) {
        state[i] = EATING;
        signal(&self[i]); // Allow the philosopher to eat
    }
}
```

Semaphores vs. Monitors

- **Semaphores**: Like integers with restricted interface
 - Initialize value to any non-negative value
 - Two operations:
 - » **sem_wait()**: Wait/sleep if zero; decrement when becomes non-zero
 - » **sem_post()**: also called signal(). Increment and wake up a waiting/sleeping thread (if one exists)
 - Use a separate semaphore for each constraint
- **Monitors**: A mutex lock plus one or more condition variables
 - Always acquire lock before accessing shared data
 - Use condition variables to wait inside critical section
 - Three operations: **wait()**, **signal()**, and **broadcast()**
 - » Wait if necessary (inside a while loop to check a Boolean flag)
 - » Signal (or broadcast) when something is changed to wake up one waiting thread (or all waiting threads)

Quiz: Race Conditions

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

```
int y=0, z=0;
```

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

```
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 $x = 0 + 2 = 2$

```
int y=0, z=0;
```

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

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

Quiz: Race Conditions

Q. Give a solution using semaphores.

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. We use a slightly different notation of `s.wait()/s.signal()` to denote `sem_wait(&s)` and `sem_post(&s)`).

```
int y=0, z=0;
semaphore s = 1;
```

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

```
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.

semaphore 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 }
```

```
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

- The following three functions of a program f1(), f2(), f3() run in separate threads each and print some prime numbers. All three threads are ready to run at the same time. Use synchronization using the semaphores S1, S2 and S3 and wait/signal operations on the semaphores to ensure that the program outputs the prime numbers in increasing order (2, 3, 5, 7, 11, 13).

```
Semaphore S1=0;
Semaphore S2=0;
Semaphore S3=0;
f1() {
    printf("3");
    printf("5");
}
f2() {
    printf("2");
    printf("13");
}
f3() {
    printf("7");
    printf("11");
}
```

Quiz: Semaphores

- Solution 1 (left): With initial values of all semaphores = 0, only f2 can run, prints 2, signals S1 and then waits for S2. S1.signal() starts f1, which was waiting for S1 and can now print 3 and 5 and then signal S3. S3.signal() now starts f3, which prints 7 and 11 and signals S2. This returns execution to f2, which can then finally print 13.
- Solution 2(right): s2 has initial value 1, so f2 calls S2.wait() and runs first. The rest of the same as Solution 1. You can see that initializing s2=0 has the same effect as initializing s2=1 and let f2 call S2.wait() first. So Solution 1 is better with one less call to wait().

```
semaphore S1=0;
semaphore S2=0;
semaphore S3=0;
f1 () {
    S1.wait();
    printf("3");
    printf("5");
    S3.signal();
}
f2 () {
    printf("2");
    S1.signal();
    S2.wait();
    printf("13");
}
f3 () {
    S3.wait();
    printf("7");
    printf("11");
    S2.signal();
}
```

```
semaphore S1=0;
semaphore S2=1;
semaphore S3=0;
f1 () {
    S1.wait();
    printf("3");
    printf("5");
    S3.signal();
}
f2 () {
    S2.wait();
    printf("2");
    S1.signal();
    S2.wait();
    printf("13");
}
f3 () {
    S3.wait();
    printf("7");
    printf("11");
    S2.signal();
}
```


Quiz: Semaphores III

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

```
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 }
```

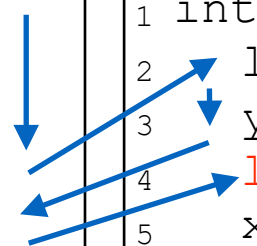
Q. Which strings can be output when running the 3 threads in parallel?

- 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

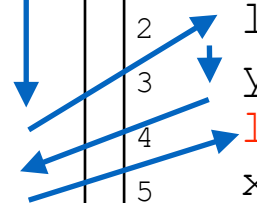
Quiz: Deadlocks

```
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 *circular waiting condition* which is not resolved

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

```
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. 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