CSC 112: Computer Operating Systems
Lecture 3

Synchronization

Department of Computer Science,

Hofstra University

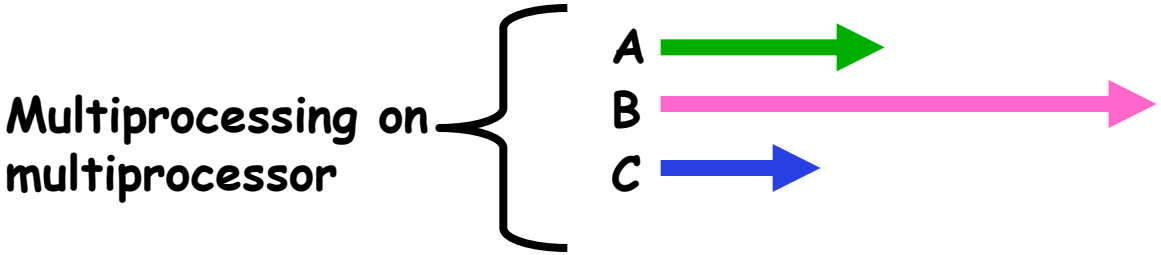
Acknowledgement: Lecture slides based on UC Berkeley CS 162: Operating Systems and System Programming
And NTNU course on Operating Systems

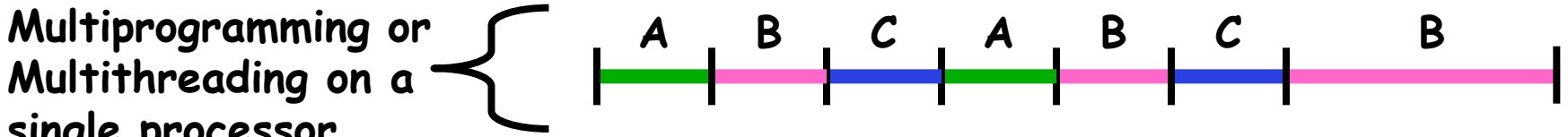
Outline

- Concurrency & Spinlocks
- Semaphores
- Monitors

Different Types of Concurrencies

- Multiprocessing → multiple CPUs running in parallel
- Multiprogramming → multiple processes scheduled on a single processor by time-sharing
- Multithreading → multiple threads per process scheduled on a single processor by time-sharing



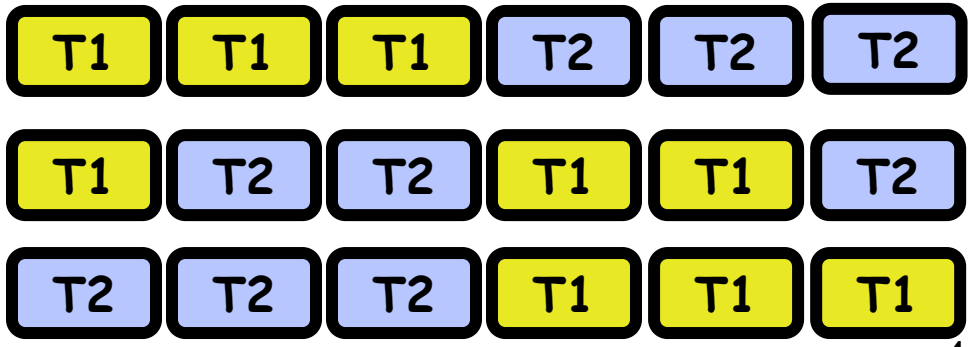


Concurrency

```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;
}
```

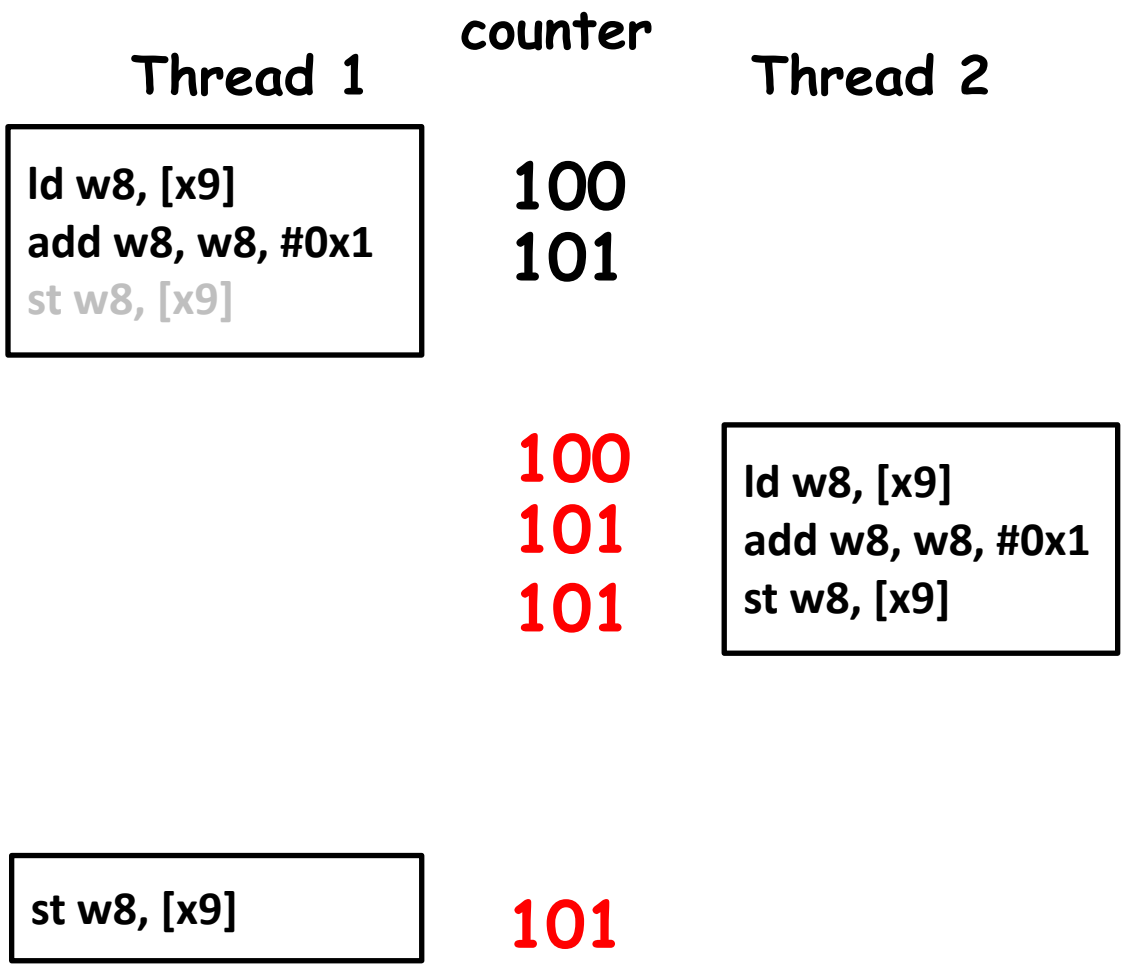
```c
int main(int argc, char *argv[])
{
        if (argc != 2){
                fprintf(stderr, "usage: threads <loops>\n");
                exit(1); }
        loops = atoi(argv[1]);
        pthread_t p1, p2;
        printf("Initial value : %d\n", counter);
        pthread_create(&p1, NULL, worker, NULL);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

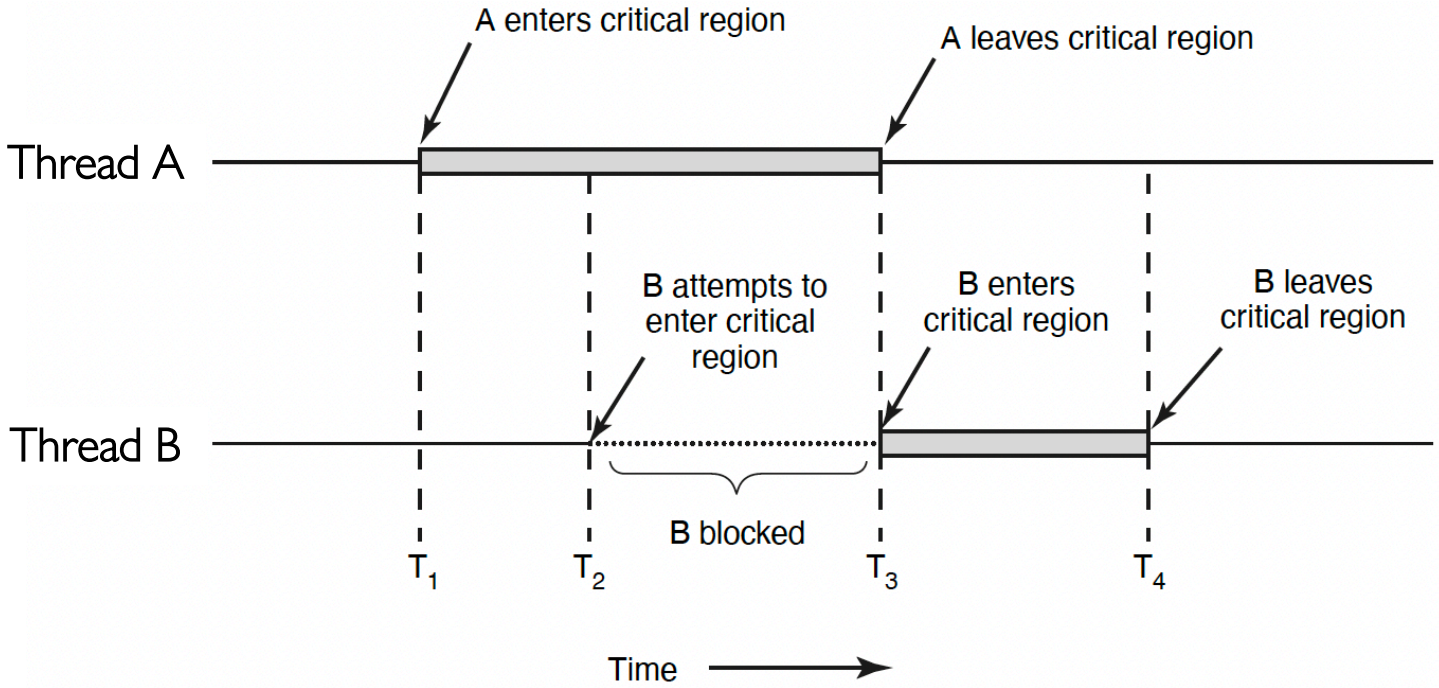
Thread 1

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

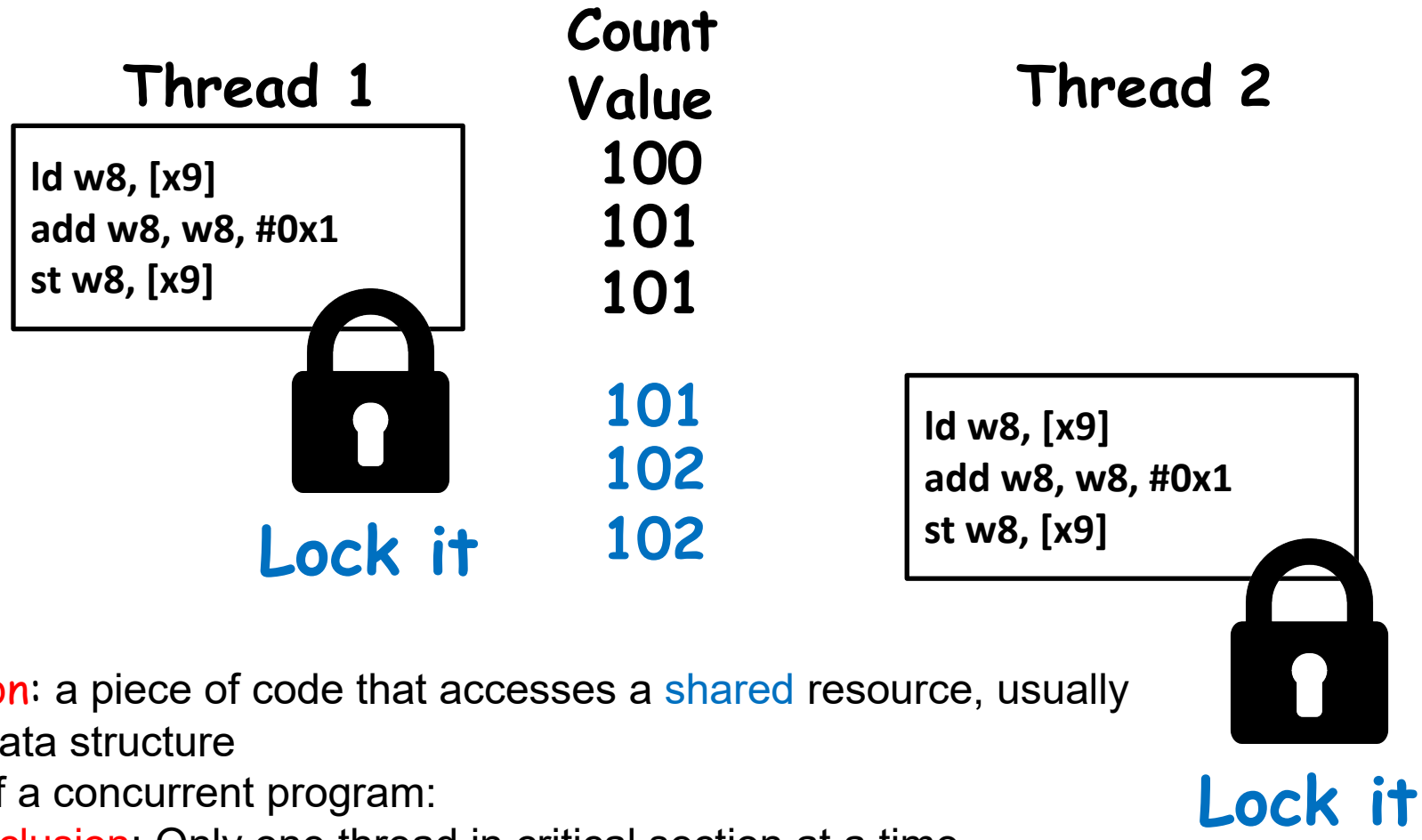
Thread 2

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	**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 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
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?**

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

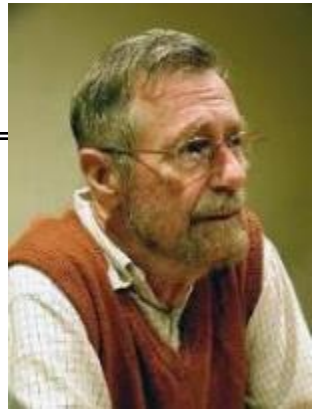
```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```
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
  - If the waiting time is long, then sleeping is more efficient since the thread gives up the CPU to other threads, but incurs system call (kernel) overhead to go to sleep and wake up; if waiting time is short, then spinlock may be more efficient since it does not involve the kernel.
  - Spinlock may cause starvation, e.g., if the waiting thread has higher priority than the signaler thread under fixed priority scheduling (but not under round-robin scheduling).

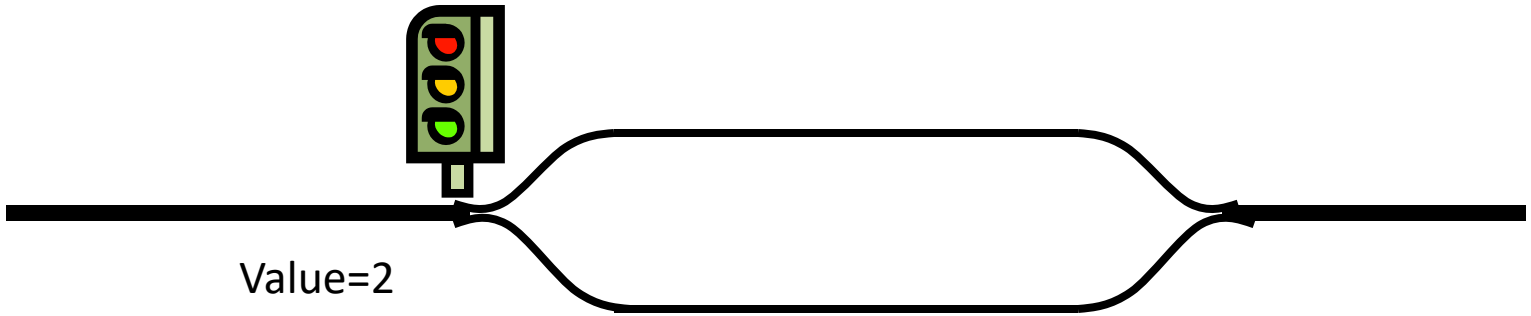
POSIX pthreads API

- 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
pthread_create	Create a new thread in the caller's address space
pthread_exit	Terminate the calling thread
pthread_join	Wait for a thread to terminate
pthread_mutex_lock	Lock a mutex
pthread_mutex_unlock	Unlock a mutex
sem_wait	Wait on a semaphore
sem_post	Signal or post on a semaphore
pthread_cond_wait	Wait on a condition variable
pthread_cond_signal	Wake up one thread waiting on a condition variable
pthread_cond_broadcast	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() {
  //Spin-wait while guard is
true
  while (TestAndSet(guard));
  if (value == 0) {
    put thread on wait queue;
    guard = 0;
    sleep();
  } else {
    value = value - 1;
    guard = 0;
  }
}
```

```
sem_post() {
  //Spin-wait while guard is
true
  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.
- Equivalently, pthread_mutex_t is designed specifically for mutual exclusion, meaning only one thread can hold the lock at a time. Only the thread that locks the mutex can unlock it, with strict ownership semantics.

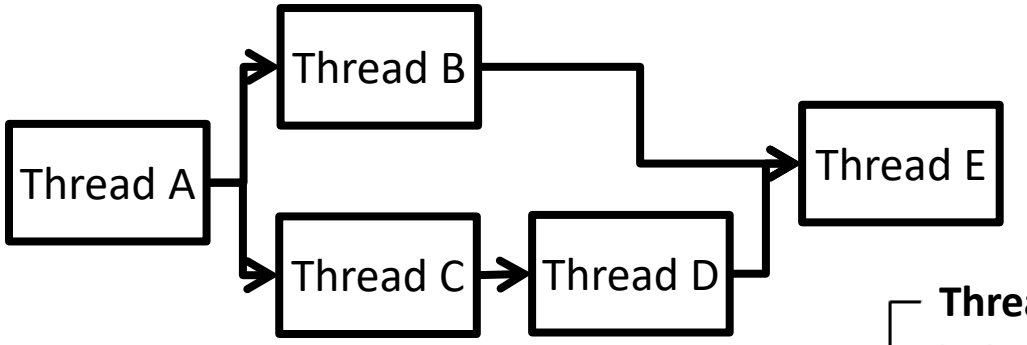
Scheduling Constraints (value >= 0)

- Called “Counting Semaphore”.
- Any thread can signal or release the semaphore, regardless of which thread acquired it. Can be used as signaling mechanisms, such as notifying other threads that a resource is available or an event has occurred.
- See next slide for an example.

```
//Mutual exclusion using binary semaphore
sem_t sem;
sem_init(&sem, 0, 1);  // Initialize to 1 for mutex-like behavior
sem_wait(&sem);
// Critical section
sem_post(&sem);


//Mutual exclusion using mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  // Initialize mutex to 1 (unlocked)
pthread_mutex_lock(&mutex);
//Critical section
pthread_mutex_unlock(&mutex);
```

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)



Thread A
start
process
signal (semA)
signal (semA)
finish

Thread B
start
wait (semA)
semA = 0
process
signal (semB)
finish

Thread C
start
wait (semA)
semA = 0
process
signal (semC)
finish

Thread D
start
wait (semC)
semC = 0
process
signal (semD)
finish

Thread E
start
wait (semB)
semB = 0
wait (semD)
semD = 0
process
finish

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

- This program ensures mutual exclusion between writers, and between the 1st reader and any writers, but not between multiple readers.
- 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 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;
```

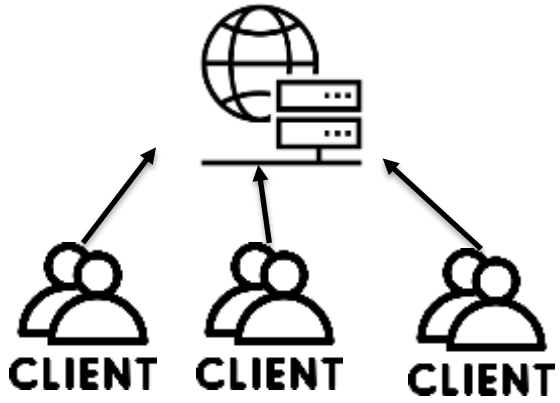
```
/* initialization.*/
mutex = 1;
wrt   = 1;
readcount = 0;
```

```
/* 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




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

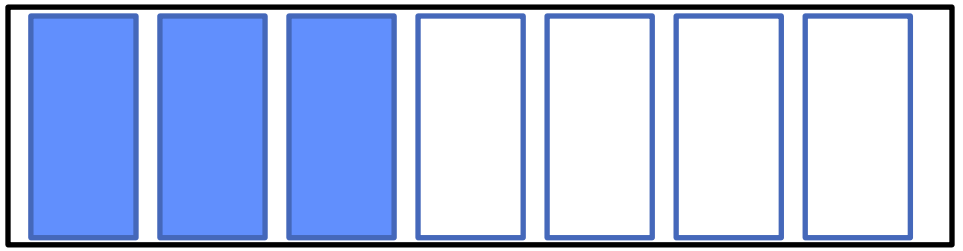
producer consumer

- 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



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

Bounded buffer
fullSlots==3, emptySlots==4

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 full slot

mutex protects integrity of the queue within critical sections

Indicates 1 more empty slot

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).

Discussions

- 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

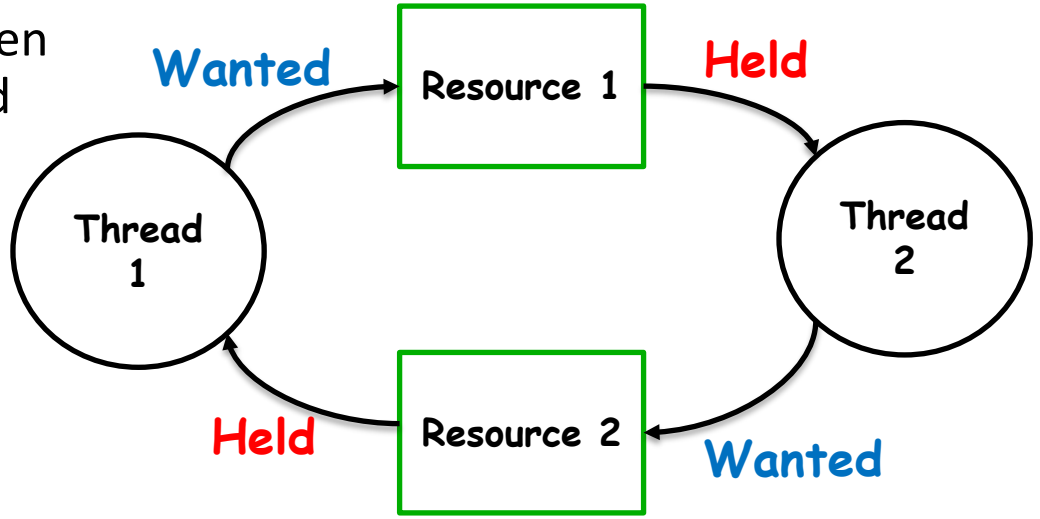
Increase # of empty slots

- 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!

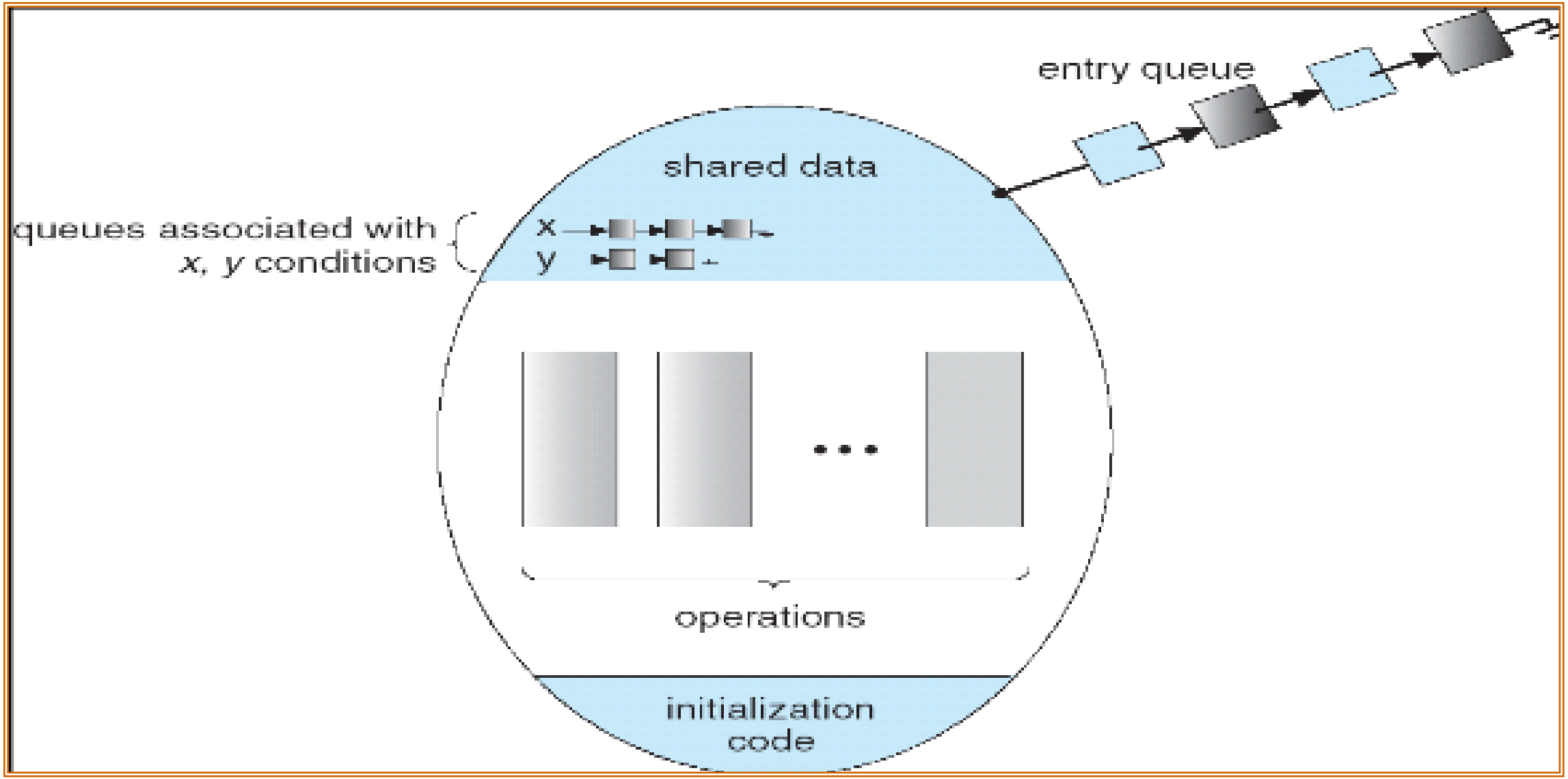
Monitors

- Semaphores are dual purpose, used for both mutex and scheduling constraints
- Monitors provide a higher-level abstraction that 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 (if no waiter thread, then the signal is lost/has no effect)
  - pthread_cond_broadcast(&cond): Broadcast(): Wake up all waiters (if no waiter thread, then the signal is lost/has no effect)

Monitor with Condition Variables (CV)



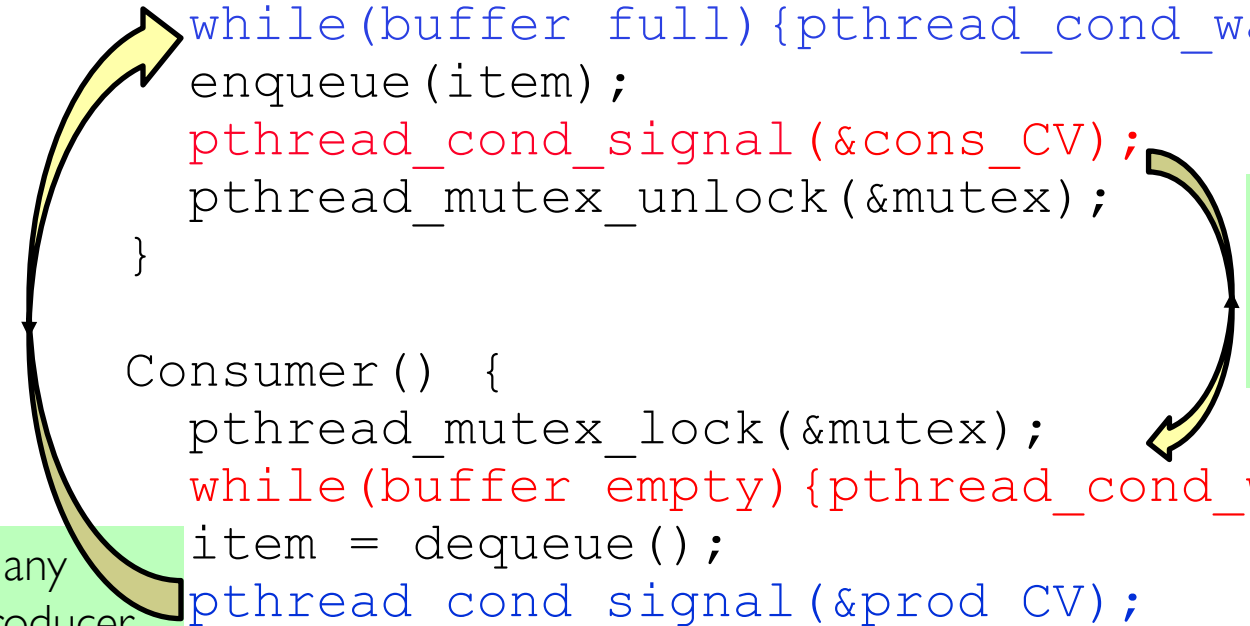
CV Common Usage Pattern

```
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
```



Wake up any waiting consumer thread blocked on an empty buffer (if any)

Wake up any waiting producer thread blocked on a full buffer (if any)

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

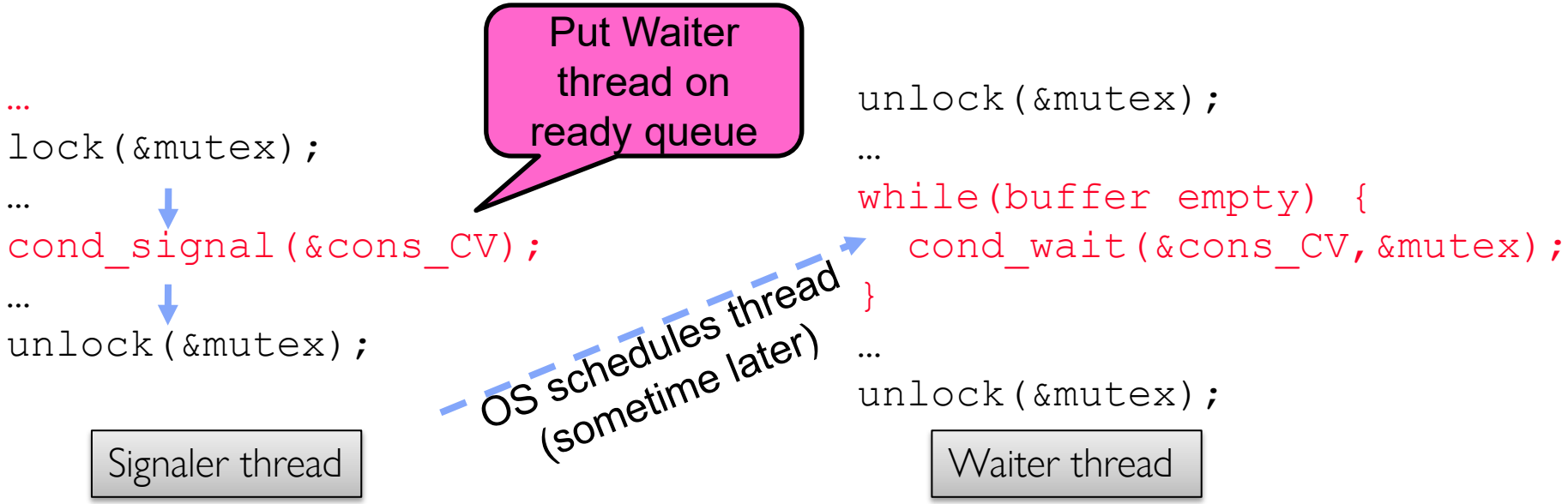
- Consider the dequeue code in Consumer thread:

```
while(buffer empty){ //Why not if(buffer empty) here
   cond_wait(&cons_CV, &mutex);
}
item = dequeue(&queue);// Get next item
```

- Why do we use a while loop to check the Boolean flag “buffer empty”?
  - Most OSes use Mesa-style monitor (named after Xerox-Park Mesa Operating System), where the waiter thread may start to run some time after Signaler thread calls cond_signal()

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”).

The Thread Join Problem

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

Parent

Child

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

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

Thread Join with Semaphore

- Semaphore sem acts as the synchronization flag.
- Works correctly regardless of whether parent or child executes first:
  - If child finishes first: sem_post(&sem) increases sem to 1, subsequent sem_wait(&sem) decrements it and parent thread continues immediately
  - If parent waits first: sem_wait(&sem) blocks until child's sem_post(&sem) wakes it up
- No Race Condition:
  - Unlike condition variables, semaphores maintain state; No need for additional flags or mutex protection.

```
semaphore sem = 0;

//Child
void thr_exit(){
  sem_post(&sem); //Signal parent
}

//Parent
void thr_join(){
  sem_wait(&sem);  // Wait for child
}
```

Thread Join with Condition Variable

- Boolean flag done is a state variable to track whether the child thread has completed. It ensures that even if pthread_cond_signal occurs before pthread_cond_wait, the parent will not block indefinitely because it will detect that done is already set. While loop around pthread_cond_wait ensures correctness in case of spurious wakeups.
- Condition variables don't preserve state like semaphores do, so we need explicit mutex protection, and a shared boolean flag to track completion status of child. (similar to Boolean flags “buffer full” and “buffer empty” in P/C problem.)

```
pthread_mutex_t m =
PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
bool done = false;
//Child
void thr_exit() {
    pthread_mutex_lock(&m);
    done = true;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m);
}
//Parent
void thr_join() {
    pthread_mutex_lock(&m);
    while (!done) {//Check if child has
finished
        pthread_cond_wait(&c, &m);
    }
    pthread_mutex_unlock(&m);
}
```

Thread Join with Condition Variables: Incorrect

- If we remove Boolean flag done, then the program is incorrect. If child calls thr_exit() before parent calls thr_join(), the signal will be lost because condition variables don't maintain state, and Parent will wait forever.
  - In contrast, semaphores maintain state; No need for additional flags or mutex protection.

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			

```
pthread_mutex_t m =
PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c =
PTHREAD_COND_INITIALIZER;
//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
```

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 his left and right forks (one at a time) to eat.
- Needs both forks to eat, then releases both when done eating.
- 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.

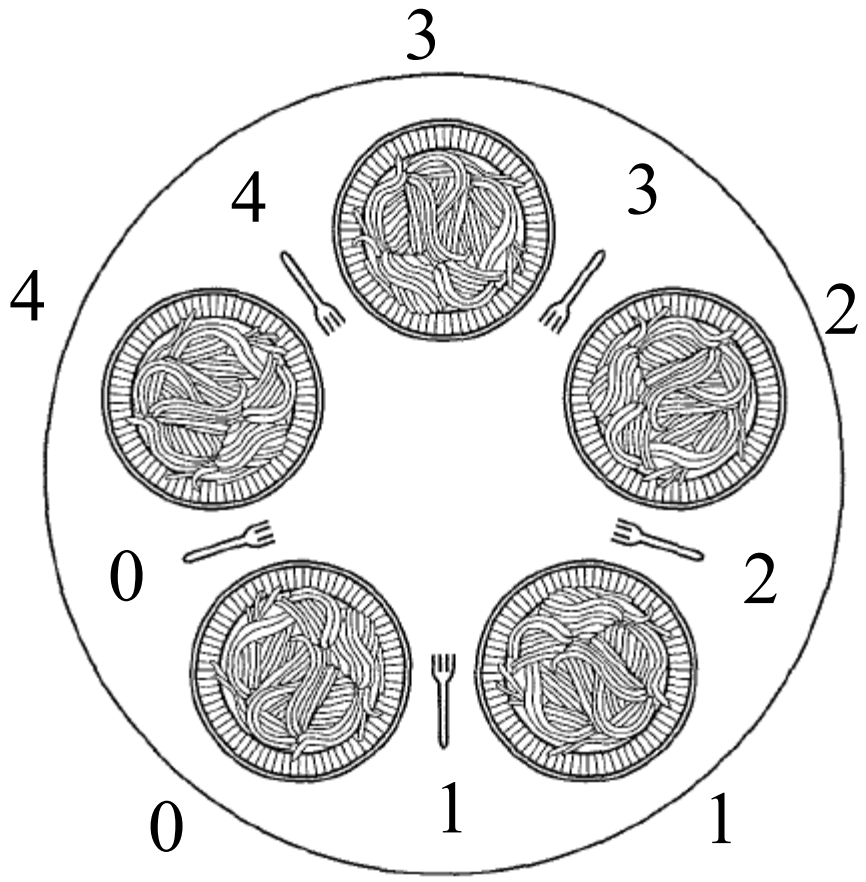


Figure 2-44. Lunch time in the Philosophy Department.

Semaphore-based Solution: Deadlock

- 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.
- **Deadlock situation:** 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 its neighbor), resulting in a deadlock, circular wait where none can proceed.

```
#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 0: Global Lock

- We use the global semaphore mutex to ensure that only one philosopher can pick up forks and eat at any one time.
- This solution works, but is very inefficient, since only one philosopher can be eating at one time; it should be possible for two philosophers to eat at the same time, since there are 5 forks.

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

void philosopher(int i) {
  while (true) {
    think();
    sem_wait(&mutex); // Pick up both forks in
one atomic operation
    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
    sem_post(&mutex);
  }
}
```

Semaphore-based Solution: Deadlock

- One solution is to let each philosopher pick up (and put down) both left and right forks within a critical section, protected by the global semaphore mutex.
- **Deadlock situation:** This solution is flawed because it can lead to deadlock, similar to the deadlock situation in the P/C problem.
- Philosopher A gets mutex, is blocked trying to get a fork (left or right); meanwhile, his neighbor Philosopher B (who has both forks) finishes eating and tries to put down forks. But B is blocked trying to get mutex (which A holds). Now we have a circular wait condition: A holds mutex, waiting for fork; B holds fork, waiting for mutex.
  - Philosopher B does not have to be A's direct neighbor. There may be a chain of philosophers starting from A, each holding his left fork, and B may be the last one's neighbor.

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

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

Semaphore-based Solution I

- The solution is to let each philosopher pick up, but not put down, both left and right forks within a critical section, protected by the global semaphore pickup_mutex.
- **No deadlock:** If philosopher i is in the critical section protected by mutex, blocked in sem_wait() waiting for any fork (left or right), the neighbor who is holding the requested fork and eating can freely put down both forks without blocking, thus allowing philosopher i to pick up both forks.
- **Poor efficiency:** If philosopher i is in the critical section protected by pickup_mutex, waiting for any fork (left or right), then no other philosopher can enter the critical section before some other philosopher finishes eating and puts down his forks to let philosopher i exit the critical section and start eating, even for a philosopher with both left and right forks free. This is unnecessary blocking that reduces concurrency.

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


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

Semaphore-based Solution II

- 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 N=5 philosophers, room is initialized to N-1=4. With 5 forks and at most 4 philosophers competing, at least one philosopher can always get both forks

```
#define N 5  // Number of philosophers and forks
semaphore room = N-1;
semaphore fork[N] = {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) % N]); // Pick up right fork
    eat();
    sem_post(&fork[i]); // Put down left fork
    sem_post(&fork[(i + 1) % N]); // Put down right fork
    sem_post(&room); // Leave the room
  }
}
```

Semaphore-based Solution III

- Another option is to adjust the order in which resources are requested (for instance, having one philosopher, the (N-1)-th philosopher, pick up his right fork first while all the others pick up the left fork first), which disrupts the cycle that could lead to deadlock.
- This method forces each philosopher to pick up lower-numbered fork before higher-numbered fork (modulo N), i.e., assign a total order to the resources, and establish the convention that all resources will be requested in the same order. Here the order of forks is 0, 1, 2, …, N-1. Philosopher 0 picks up left fork 0 before right fork 1; Philosopher 2 picks up left fork 1 before right fork 2;…; Philosopher N-1 picks up right fork 0 before left fork N-1.
- A variant is to let even-numbered philosophers pick up right fork first.
- The order of acquiring resources (forks) must be controlled to prevent deadlock, but the release order doesn't matter. So this program can be simplified to let all philosophers put down left fork first.

```
#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) {//One of the philosophers
    //or if (i % 2 == 0){// Even numbered philosophers
    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) {
    //or if (i % 2 == 0){// Even numbered philosophers
    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
  }
}
```

Semaphore-based Solution IV

- A semaphore 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), which sets their state to HUNGRY and calls test(i) to check if any of its two neighbors are eating.
- If both adjacent philosophers are not eating, philosopher i's state is changed to EATING, and calls sem_post(&self[i]) to increment self[i] by 1, and it next calls sem_wait(&self[i]) to decrement self[i] by 1, and start eating. Otherwise, philosopher i's state stays to be HUNGRY, and it is blocked on sem_wait(&self[i]).
- 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 semaphore variables. This structure prevents the circular waiting condition that leads to deadlock.
- Since state is explicitly maintained in an array state[N], we need mutex protection in both pickup() and putdown() methods.

```
#define N 5  // Number of philosophers and forks
enum { THINKING, HUNGRY, EATING } state[N];
mutex_t mutex = 1;
semaphore self[N] = {0, 0, 0, 0, 0}; // Semaphore for each philosopher
// Initialize to 0
void philosopher(int i) {
    while (true) {
        think();
        pickup(i);
        eat();
        putdown(i);
    }
}
void pickup(int i) {
    mutex_lock(&mutex);
    state[i] = HUNGRY;
    test(i);
    mutex_unlock(&mutex);
    sem_wait(&self[i]); //Block if forks weren't acquired
}
void putdown(int i) {
    mutex_lock(&mutex);
    state[i] = THINKING;
    test((i + 4) % N);  // Test left neighbor
    test((i + 1) % N);  // Test right neighbor
    mutex_unlock(&mutex);
}
void test(int i) {
    if (state[i] == HUNGRY &&
        state[(i + 4) % N] != EATING &&
        state[(i + 1) % N] != EATING) {
        state[i] = EATING;
        sem_post(&self[i]);
    }
}
```

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 of its two neighbors are eating.
- If both adjacent philosophers are not eating, philosopher i's state is changed to EATING (cond_signal(&self[i]) has no effect and the signal is lost since no other philosopher is waiting on self[i]); otherwise, philosopher i waits on condition variable self[i].
- 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.
- Note that cond_signal(&self[i]) in test(i) has no effect during pickup(), but it is used to wake up waiting hungry philosophers during putdown(i).

```
#define N 5  // Number of philosophers and forks
enum { THINKING, HUNGRY, EATING } state[N];
mutex_t mutex = 1;  // Monitor's mutex
condition self[N];  // Condition variable for each philosopher
void philosopher(int i) {
    while (true) {
        think();
        pickup(i);
        eat();
        putdown(i);
    }
}
void pickup(int i) {
    mutex_lock(&mutex);
    state[i] = HUNGRY;
    test(i);
    while (state[i] != EATING)
      cond_wait(&self[i], &mutex); //Block if forks weren't acquired
    mutex_unlock(&mutex);
}
void putdown(int i) {
    mutex_lock(&mutex);
    state[i] = THINKING;
    test((i + 4) % N);  // Test left neighbor
    test((i + 1) % N);  // Test right neighbor
    mutex_unlock(&mutex);
}
void test(int i) {
    if (state[i] == HUNGRY &&
        state[(i + 4) % N] != EATING &&
        state[(i + 1) % N] != EATING) {
        state[i] = EATING;
        cond_signal(&self[i]);
    }
}
```

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)