Bilkent University
Department of Computer Engineering
CS342 Operating Systems

Synchronization

Last Update: Mar 26, 2023

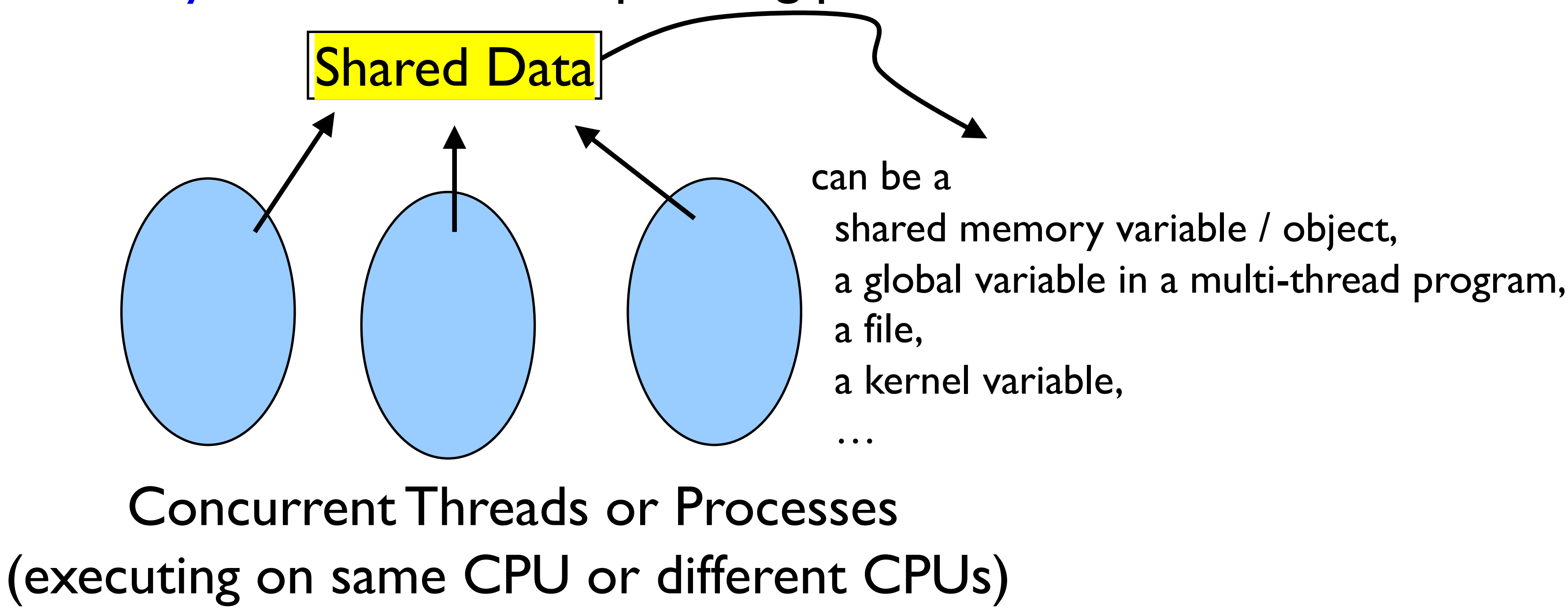
Objectives and Outline

Objectives and Outline

- The critical-section Problem
- Pure software solutions
- Synchronization hardware (TSL, SWAP instructions)
- Lock variables
- Condition variables
- Semaphores
- Monitors
- Classic problems of synchronization and their solutions

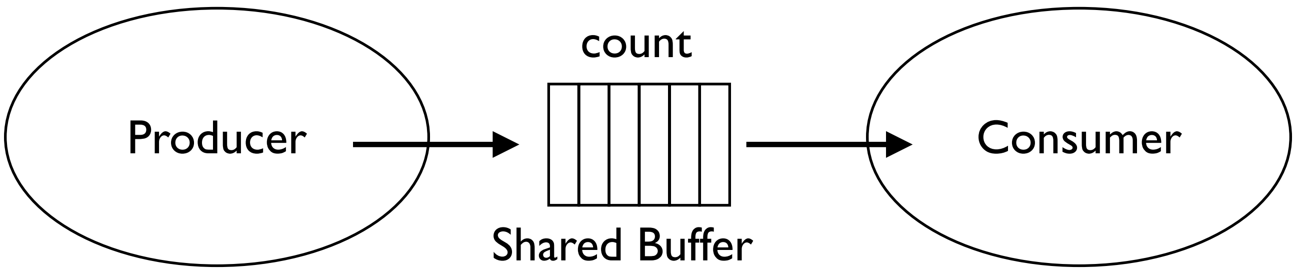
Background

- Concurrent access to shared data may result in data inconsistency.
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.



Producer-Consumer Problem Revisited

- Consider producer-consumer problem again.
- This time uses an integer count (shared) to keeping the number of full slots. Initialized to 0.
  - incremented by the producer after putting a new item
  - decremented by the consumer after retrieving an item



at most BUFFER_SIZE items

Producer and Consumer Code

Producer

```
while (true) {
        // produce an item
        item  = ….
        while (count == BUFFER_SIZE)
                ; // do nothing
        // add item
        buffer [in] = item;
        in = (in + 1) % BUFFER_SIZE;
        count++;
}
```

Consumer

```
while (true)  {
        while (count == 0)
                ; // do nothing
        // remove item
        item =  buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--;
        // consume item
}
```

Accessing shared data

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

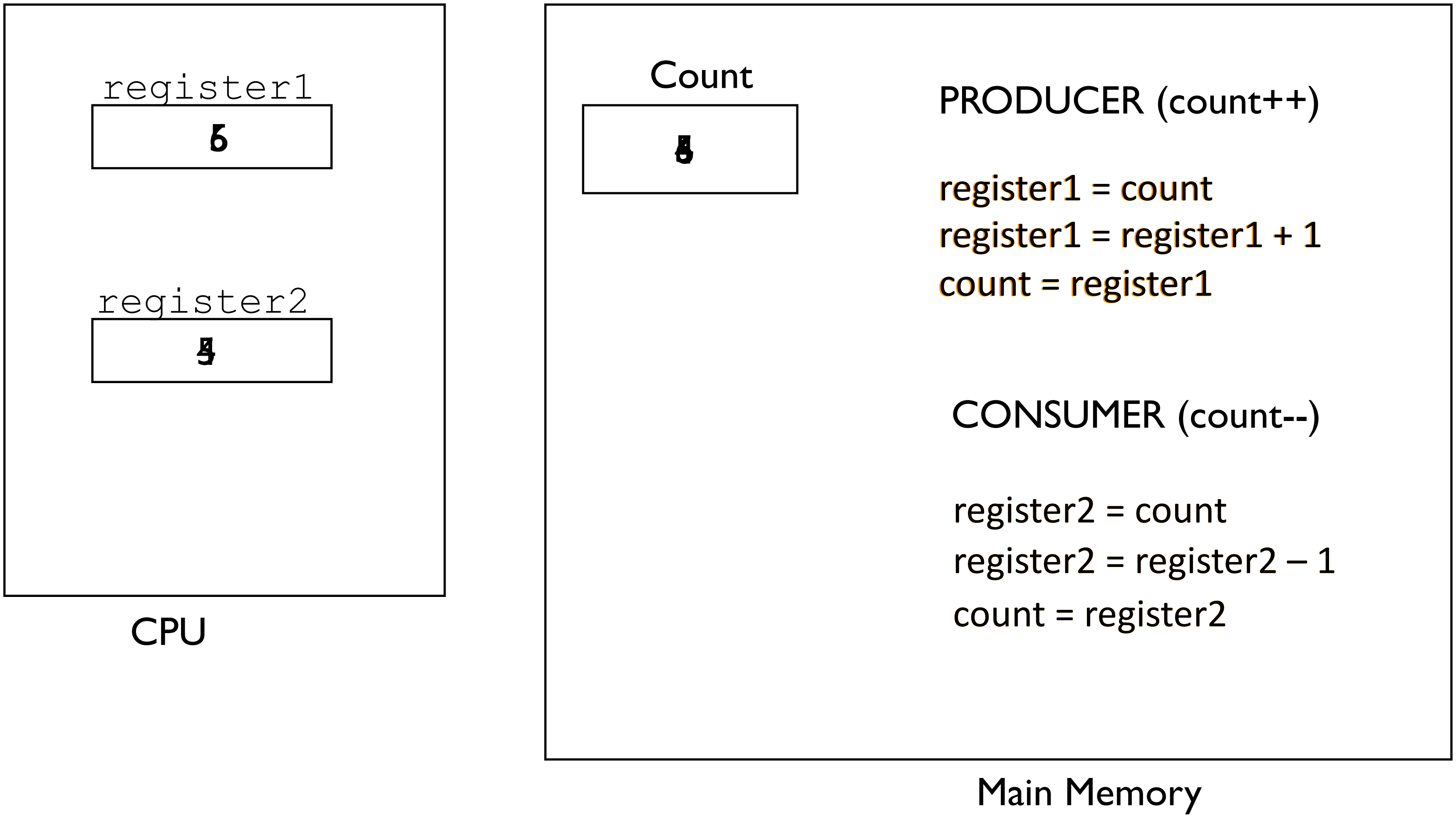
```
register2 = count
register2 = register2 - 1
count = register2
```

same physical register can be used, because registers are saved and reloaded while doing context switch

a possible problem: race condition

- Assume we have 5 items in the buffer
- Assume producer has just produced a new item (6 the item) and put it into buffer and is about to increment the count.
- Assume the consumer has just retrieved an item from buffer and is about to decrement the count.
- That means assume producer and consumer is now about to execute count++ and count-- statements.

Race Condition



Interleaved Execution sequence

- Interleaved execution, “count = 5” initially:

  P1: *producer executes* register1 = count {register1 = 5}
  P2: *producer executes* register1 = register1 + 1 {register1 = 6}
  C1: *consumer executes* register2 = count {register2 = 5}
  C2: *consumer executes* register2 = register2 - 1 {register2 = 4}
  P3: *producer executes* count = register1 {count = 6 }
  C3: *consumer executes* count = register2 {count = 4}

- At the end, count became 4. Should be 5.

Race condition

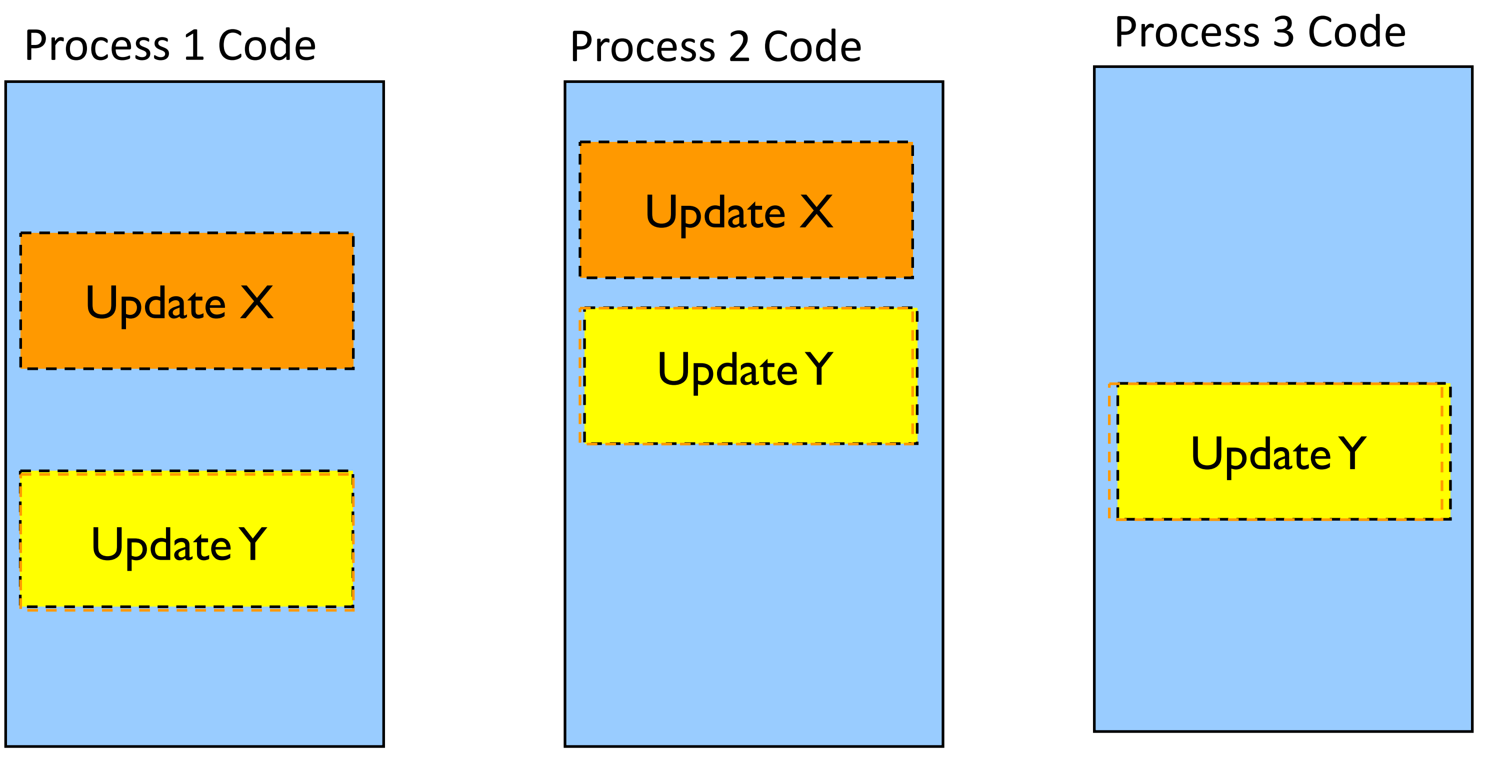
- Count value may be 4, 6, or 5 in various runs.
  - But it should be 5 (as a result of one increment, one decrement operation)
  - concurrent access to count causes data inconsistency: 4, 5, 6.
- Such situations are called race conditions: several processes access and manipulate the same data concurrently.
- We should develop programs that do not have race conditions.
  - Race conditions will cause incorrect operation. Additionally, they are hard to reproduce.

Race condition

- In previous example:
  - For consistent result (5), either count++ should be executed and finished first, or count-- should be executed and finished. Not interleaved.
- To avoid race conditions, we need to enforce non-interleaved access (atomic) to shared data.
- We need synchronization (coordination) of threads/processes while accessing shared data.

Programs and critical sections

The part of the program (process) that is accessing and using shared data is called its critical section (region).



Assuming X and Y are different shared data.

Program lifetime and its structure

- We should not allow more than one thread to be in their critical sections at the same time.
  - Critical sections should be executed one at a time.
- A thread may also be executing non-critical section code (remainder section). Concurrent execution of that part is allowed and may be desirable.

Program structure

- The general way to solve critical section problem:

```
do {

        critical section

        remainder section

} while (TRUE)
```

The general structure of a program

```
do {
        entry section

        critical section

        exit section

        remainder

} while (TRUE)
```

Entry section will allow only one process to enter and execute critical section code.

Solution to Critical-Section Problem

An ideal solution should have the following conditions satisfied:

1. Mutual Exclusion: If process P_i is executing in its critical section, then no other process can be executing in its critical section.
2. Progress: If there are processes wanting to enter critical section while there is nobody in the critical section, they should not be waited indefinitely to enter the critical section. // *no deadlock*
3. Bounded Waiting: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. // *no starvation of a process*

- Assume that each process executes at a nonzero speed.
- No assumption can be made concerning the relative speeds of the processes

Applications and Kernel

- Applications
  - Multi-process applications sharing a file or shared memory segment may face critical section problems.
  - Multi-threaded applications sharing global variables may also face critical section problems.
- Kernel
  - Similarly, kernel itself may face critical section problems. It has critical sections.

Kernel Critical Sections

- Execution of a kernel function x() may be interrupted by a hardware interrupt and interrupt handler h() may run. Care needed if x() and h() are accessing shared data.
- A process makes a system call say s1(). While s1() is running (kernel code), a context switch may cause another process run and that process may make a system call say s2() (in preemptive kernel) .
  - Care needed if s1() and s2(), kernel functions, access shared data.
- A note: When a process makes a system call, we say the process is running in kernel mode while the system call is being executed.
- Kernel is developed considering such cases.

Pure software solution
An example: Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE machine instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
  - int turn;
  - boolean flag[2];
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] == true implies that process P_i wants to enter the critical section.

Algorithm for Process P_i

```
do {

        flag[i] = TRUE; // wants to enter CS
        turn = j;
        while (flag[j] && turn == j);   // busy wait      entry section
                  critical section
        flag[i] = FALSE;                                  exit section
                  remainder section

 } while (1)
```

Two processes executing concurrently

PROCESS **i** (0)

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j)
                ; // looping
        critical section.....
        flag[i] = FALSE;
        remainder section.....
}  while (1)
```

PROCESS **j** (1)

```
do {
        flag[j] = TRUE;
        turn = i;
        while (flag[i] && turn == i)
                ; // looping
        critical section.....
        flag[j] = FALSE;
        remainder section.....
} while (1)
```

shared variables $\left\{ \begin{array}{l} \text{flag[2]} \\ \text{turn} \end{array} \right.$

Synchronization Hardware

- We can use some hardware support (if available) for protecting critical section code.
  - 1) Disable interrupts?
    - Sometimes (only in kernel)
    - Not possible on multi-processors. (we should not disable interrupts on all processors)
  - 2) Special machine instructions (acting on lock variables)
    - TestAndSet (test and set lock – TSL instruction)
    - CompareAndSwap (CAS instruction)

Hardware support for synchronization
and
Locks

Solution to Critical-section Problem Using Locks

- Use of lock variables is a general and very common method for the solution of critical section problem (for mutual exclusion).
- A lock variable is shared (lock variable can simply be an integer variable with values 0 and 1). A process (thread) can be structured as follows:

```
do {
        acquire_lock (&lock)
        critical section
        release_lock (&lock)
        remainder section
} while (TRUE);
```

Only one process can acquire the lock. Others have to wait (or busy loop)

Locks can be implemented using special hardware instructions.

Locks

What happens if we use an integer as a lock variable without using special hardware instructions?

int lock = 0; // global variable (shared among threads)

Thread 1

```
while (lock == 1)
            ; // loop
lock =  1;
// critical section
lock = 0;
```

Thread 2

```
while (lock == 1)
            ; // loop
lock =  1;
// critical section
lock = 0;
```

above code is NOT a correct solution

Lock variable itself is source of race condition.

Synchronization Hardware

- Therefore we need to use special machine instructions that can do testing and setting atomically or something similar (like swapping).
- Some possible atomic (non-interruptable) machine instructions:
  - TestAndSet instruction (TSL):
    Test memory word and set its value to 1
  - CompareAndSwap instruction (CAS)

- Hardware ensures such an instruction is executed *atomically in a multi-processor environment* as well
  - one CPU at a time executes the instruction: it involves memory access; memory is shared.

TestAndSet Instruction

- is a machine/assembly instruction.

```
TestAndSet   REGISTER, LOCK;
```

  - Here we provide the definition of it using high-level language code.

Definition of TestAndSet Instruction

```
int  TestAndSet (int *target)
 {
          int rv = *target;
          *target = 1; // set to one - locked
          return rv:
}
```

Solution using TestAndSet

- To use it, we need to program in *assembly language*.

  We can use a shared integer variable, named lock for example. We initialize it 0.

```
do {
        while ( TestAndSet (&lock))        entry section
              ;   // do nothing

        //    critical section

        lock = 0;                          exit_section

        //    remainder section

} while (TRUE);
```

In assembly

```
entry_section:
        TestAndSet   REGISTER, LOCK;
        CMP REGISTER, #0   // cmp with 0
        JNE  entry_section;   // if not 0,  loop
        RET
```

entry section code

```
exit_section:
        move LOCK, #0
        RET
```

exit section code

```
main:
        ..
        call entry_section;
        execute critical region;
        call exit_section;
```

CompareAndSwap Instruction (CAS)

- Again a machine instruction
- It has three operands: value, *expected*, newvalue
- If value is equal to expected value, then swaps (value becomes newvalue). Old value returned.

Definition

```
int compare_and_swap (int *value,  int expected,  int new_value)
 {
            int temp = *value;
            if (*value == expected)
                        *value == new_value;
            return temp;    // old value returned
}
```

Solution using CompareAndSwap

- We need to program entry_section() in assembly

We use a shared int variable, named lock, which is initialized to 0 (unlocked).

```
             do {
entry_section      while (compare_and_swap(&lock, 0, 1) != 0)
                       ; /* do nothing */

                   /* critical section */

exit_section       lock = 0;

                   /* remainder section */
             } while (true);
```

Comments

- Use of TestAndSet and CompareAndSwap as explained provides *mutual exclusion*: 1st property satisfied
- *Progress* is also satisfied. (no deadlock).
- But, *Bounded waiting* property, 3rd property, may not be satisfied (starvation) in a uni-processor system.
  - A process X may be waiting (busy looping) in entry section, but we can have the other process Y going into the critical region repeatedly (no limit).
  - This will happen if context switches occur always while Y is in the critical section.
    - When X in CPU: it busy loops (tries to get lock)
    - When Y in CPU; it executes in critical section for a while, then leaves the critical section (releases lock), and then gets the lock again and enters critical section again. Then context switch occurs.

Bounded-waiting mutual exclusion with TestAndSet()

process (i) code

```
do {                                   entry section code
        waiting[i] = TRUE;  // assuming will wait
        key = 1;   // lock assumed to be locked
        while (waiting[i] && key)
                key = TestAndSet(&lock); // lock is 0, then key will be 0
        waiting[i] = FALSE;

        // critical section
                                                  exit section code
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])  // search for a process waiting
                j = (j + 1) % n;     // j was not interested
        if (j == i)          // no other process wants to enter CS
                lock = 0;  // set lock to 0
        else         // there is a process j that is waiting in while loop
                waiting[j] = FALSE; // process j will be in CS; lock still 1
        // remainder section
} while (TRUE);
```

Mutex locks

- We can put these entry and exit section codes into two functions: mutex_lock() and mutex_unlock(). These function may be put into a library or into kernel.
- Then we have a lock implementation. It is also called mutex lock (mutual exclusion lock)
- Applications will not be directly using HW instructions; instead lock implementation will use these HW instructions.
- Applications will just call mutex_lock() and mutex_unlock() functions.

Lock implementation (with busy waiting)

```
typedef struct __lock_t {
      int flag; // 0 or 1
} lock_t;
```

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

```
void mutex_lock (lock_t *lock)
{
      while ( TestAndSet (&lock->flag))
            ;   // loop - do nothing
}

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

Application using a lock

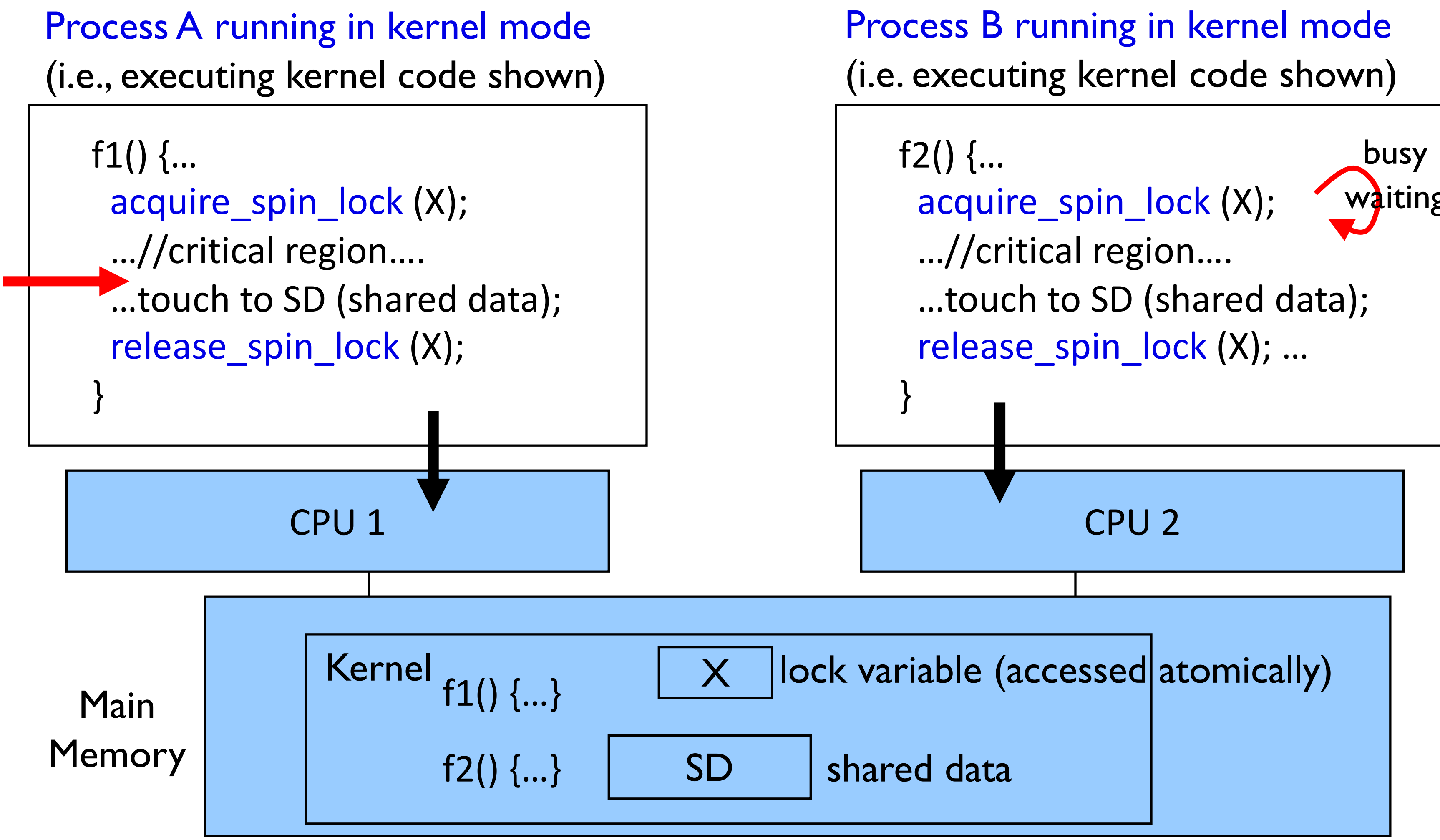
```
lock_t mutex; //define lock variable


init (&mutex);  // initialize

mutex_lock (&mutex);  // acquire lock
//critical section
mutex_unlock (&mutex);  // release lock
```

Terminology: during the time that a lock is held by a thread *X*, we say:
lock is acquired by X, or
held by X, or
belong to X, or
X has the lock, or
X got the lock, etc.

Locks

- Such a lock is also called spin-lock, since it busy waiting.
- Use of spin-locks in uniprocessor systems is very inefficient. One process B will spin during the whole time quantum (*q* ms) if the lock is held by another process A.
- But, it can be useful for short critical sections in multi-processor systems.
- Kernel uses spin locks to protect short critical sections (a few instructions) on multi-processor systems.

Spin Locks


Process A running in kernel mode
(i.e., executing kernel code shown)
f1() {...
acquire_spin_lock (X);
...//critical region....
...touch to SD (shared data);
release_spin_lock (X);
}
Process B running in kernel mode
(i.e. executing kernel code shown)
f2() {...
acquire_spin_lock (X);
busy waiting
...//critical region....
...touch to SD (shared data);
release_spin_lock (X); ...
}
CPU 1
CPU 2
Main Memory
Kernel
f1() {...}
f2() {...}
X
lock variable (accessed atomically)
SD
shared data

Lock implementation without busy waiting

- It is possible to implement locks without busy waiting.
- There will be an associated waiting queue with the lock data type (lock object).
- A process that calls mutex_lock() will be waited (blocked) on the waiting queue of the lock (will not spin), if lock is not available.
- When a process calls mutex_unlock(), one of the waiting processes (if any) will be woken up and will be put to ready state; it will have the lock now. If nobody is waiting, lock is set to be available.
- We will not see the implementation here.

Condition Variables

Condition Variables

- Mutex variables (locks) can be used to solve the critical section problem (mutual exclusion problem).
- But there are other synchronization needs and problems.
  - For example, we may want to block (sleep, wait) a thread until some event/condition happens. How can we do that?
- Condition variables are for such cases.
- A condition variable (cv) is an object/variable (ADT) that can be used to cause a thread to sleep until a condition happens and another thread signals.
  - Internally, a cv has a waiting queue associated with it.
  - We can perform two operations on a cv: wait() and signal()

Condition Variables

- cv.wait() blocks (sleep/wait) the calling thread and adds it to the cv waiting-queue.
- cv.signal() wakes up (unblocks) one of the waiting threads (if any) and removes it from the cv waiting-queue.
- If there is no thread sleeping on cv, signal() has no effect (signal lost).
- cv.broadcast() wakes up all the waiting threads on cv (all are removed from waiting-queue of cv and set to be ready), if any.
- A condition variable is used together with a mutex lock.

Example 1

- POSIX Pthreads API provides mutex and condition variables.
- Consider a program with two threads. They will share a variable count.
  - One thread (thread2) will update the count.
  - We want the other thread (thread1) to wait until count becomes 100.

global variables

```
int count = 0; // shared state between threads
pthread_mutex_t lock;   // lock variable
pthread_cond_t   cv;    // condition variable
```

Example 1

cond_wait() body first releases the lock; then sleeps on the cv queue. When waken up (signaled), it is removed from cv queue. Then it tries to get the lock again (added to the lock queue). when it has the lock, cond_wait() returns.

// executed by thread 1

```
void * function1 (void *p)     // executed by thread 1
{
      // assume waited condition is "count == 100"
      pthread_mutex_lock (&lock);  // get the lock
      while (count < 100)
            pthread_cond_wait (&cv, &lock);

      // we are sure count is 100 now.
      // do something with count being 100 if you wish
      pthread_mutex_unlock(&lock);  // release the lock
}
```

condition (count >= 100) is a predicate that includes shared variables.

Example 1

```
// executed by thread 2
void *function2 (void *p) {
        pthread_mutex_lock (&lock); // get the lock
        while (count < 100)
                count += 1;  // changing shared data (count)
        // waited condition happened
        pthread_cond_singal (&cv);  // signal and continue
        // signaling thread still has the lock.
        // waken up thread is moved from cv wait-queue
        // to lock wait-queue.
        pthread_mutex_unlock(&lock)  // release the lock now
}
```

Example 2

- Assume we have a resource type that has 100 identical instances. Multiple threads, for example N threads, are running concurrently. Each thread x $(1 \leq x \leq N)$ will want to use k instances (k is randomly chosen) from time to time (in an endless loop). Write a program that will control access to the resource.

global – shared - variables

```
int rcount = 100; //resource count; shared
pthread_mutex_t lock; // protects rcount
pthread_cond_t   cv; // to enforce sleep
```

Example 2

a thread x will execute the code below.

```
...
int k // local variable of the thread - needed instances
while (1)  {
      k = generateRandomValue(1,100)// between 1 and 100
      allocate_resources (k);
      // use resources  - may take a while
      deallocate_resources (k);
}
```

Example 2

```
void allocate_resources (int n) // executed by thread x
{
        pthread_mutex_lock (&lock);
        while (rcount <  n)  // check resource availability
               pthread_cond_wait (&cv, &lock);  // wait
        rcount = rcount - n
        pthread_mutex_unlock(&lock);
}
```

- The function will block until the requested number of resource instances become available.
- Many threads may call the function simultaneously.

Example 2

```
void deallocate_resources (int n)  // executed by thread x
{
       pthread_mutex_lock (&lock);
       rcount = rcount + n;
       pthread_cond_broadcast (&cv); // wakeup all waiting
       pthread_mutex_unlock(&lock)
}
```

lock and condition variable in pthread_cond_wait()

```
while (rcount < n)      // rcount is shared variable
        pthread_cond_wait (&cv, &lock);
```

When a thread calls pthread_cond_wait(), it releases the lock and gets added to the cv queue and is blocked (sleeping). When it is waken up, it will wait for the lock to become available – will be added to the lock's waiting-queue. When the thread finally gets the lock, it will return from pthread_cond_wait() function.

Then it will loop and check the condition again while holding the lock. If the condition did not happen yet, the thread will call pthread_cond_wait() again. The call will cause the thread to release the lock and sleep on the condition variable again.

POSIX mutex and condition variables

Tips

- Always wait in a while loop for condition to be true (unless you are sure if statement is needed instead of while). This is safer and more modular.
- If there are multiple threads waiting, just one signal() may not wake up the correct thread in some cases. In those cases, use of broadcast() can simplify coding.
  - broadcast() wakes up all threads waiting on the condition variable. Each thread, one by one, will check the condition again, and if condition is not true, will wait on condition variable queue again.
- Have the lock while accessing shared variables.

Semaphores

Semaphore

- Synchronization tool that does not require busy waiting
  - Supported by OS or by a Library
- A semaphore S has an integer variable and a wait queue associated. It is a shared object (for example, can be a kernel object).
- Two standard operations modify S: wait() and signal()
    - Originally called P() and V()
    - Also called down() and up()
- Semaphores can only be accessed via these two atomic operations;
- Semaphores can be implemented in kernel and accessed by system calls.

Meaning (semantics) of operations

- wait (S):

  if S positive

  S-- and return

  else

  block here (until somebody wakes you up; then return)

- signal(S):

  if there is a process waiting

  wake it up and return

  else

  S++ and return

Comments

- Wait body and signal body have to be executed atomically: one process at a time. Hence the body of wait and signal are critical sections to be protected by the kernel.

Semaphores as general synchronization tool

- Binary semaphore: value can be 0 or 1. It *can be simpler to implement.*
  - Also known as non-busy-waiting mutex locks (that does not busy-wait, but sleep)
  - Binary semaphores provide mutual exclusion; can be used for the critical section problem.
- Counting semaphore: integer value can be any value >= 0
  - Can be used for other synchronization problems; for example, for resource allocation.
  - Can be implemented by using binary semaphores.

Semaphores usage: critical section problem

- A semaphore variable should be defined, initialized to 1.
- Will be shared by multiple processes (say N) processes.
- Each process i, $1 \leq i \leq N$, is structured as follows:

```
Semaphore Mutex = 1;   // define and initialize shared semaphore

    do {
        wait (mutex);
         // critical section code
        signal (mutex);
        // remainder section
     } while (1);
```

Semaphores usage: critical section problem

Process 0

```
do {
        wait (mutex);
          // Critical Section
        signal (mutex);
        // remainder section
} while (TRUE);
```

Process 1

```
do {
        wait (mutex);
        // Critical Section
        signal (mutex);
        // remainder section
} while (TRUE);
```

```
wait() {…}        signal() {…}
    Semaphore mutex;    //  initialized to 1
```

Kernel

Semaphore usage: other synchronization problems

P0	P1
… S1; ….	… S2; ….

Assume we definitely want to have statement S1 executed before statement S2.

semaphore x = 0; // initialized to 0

solution:

P0	P1
… S1; signal (x); ….	… wait (x); S2; ….

Uses of Semaphore: synchronization

Buffer is an array of BUF_SIZE Cells (at most BUF_SIZE items can be put)

Producer

```
do {
        // produce item
        …
        put item into buffer
        ..
        signal (Full_Cells);

} while (TRUE);
```

Consumer

```
do {
        wait (Full_Cells);
        ….
        remove item from buffer
        ..
        …
} while (TRUE);
```

wait() {…}   signal() {…}

Kernel

Semaphore Full_Cells = 0;   // initialized to 0

Semaphore usage: resource allocation

- Assume we have a *resource* that has 5 *identical instances*. A process will need *one instance from time to time*. We can allow at most 5 processes to use the resource concurrently. Other processes that want to use the resource need to wait.
- Solution: one of the processes creates and initializes a semaphore to 5. Each process has to be coded as below.

```
Semaphore  x = 5;
```

```
wait (x);
…
….use one instance
of the resource…
…
signal (x);
```

Semaphore Implementation

- Semaphore data structure can be defined as below.

```
typedef struct {
      int value;                    // semaphore value
      struct process *waitlist; // semaphore wait queue
} semaphore;
```

- With each semaphore there is an associated wait queue.
  - The processes waiting for the semaphore are waited here.

Semaphore Implementation

Implementation sketch of wait:

short critical
section

```
wait (semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->waitlist;
                block() // kernel blocking the process;  context-switch happens
        }
}
```

implementation sketch of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->waitlist;
                wakeup (the process);

        }
}
```

Kernel Implementing wait and signal

- The wait and signal operations must be atomic.
  - The integer value, wait-queue updated.
- Multiple processes can make calls to wait() and signal() simultaneously
- But no two processes can execute wait() and signal() critical sections at the same time. Short critical sections.
- Kernel can guarantee this by:
  - disabling interrupts in a single CPU system
  - use of spin-locks in a multi-processor system

POSIX Semaphores

Unnamed Semaphores

```
sem_t S;

sem_init (&S, 0, 1);

sem_wait(&S); // wait operation
....
sem_post(&S); // signal operation

sem_destroy (&S);
```

Named Semaphores

```
sem_t * Sp;
char * sname = "semname1";

Sp = sem_open(sname,
        O_CREAT, 0666,   1);

sem_wait(Sp);   // wait operation
...
sem_signal(Sp);   // signal operation

sem_close(Sp);
sem_unlink (sname);
```

(you can learn more info using **man sem_overview** at Linux command line)

Potential problems with semaphores

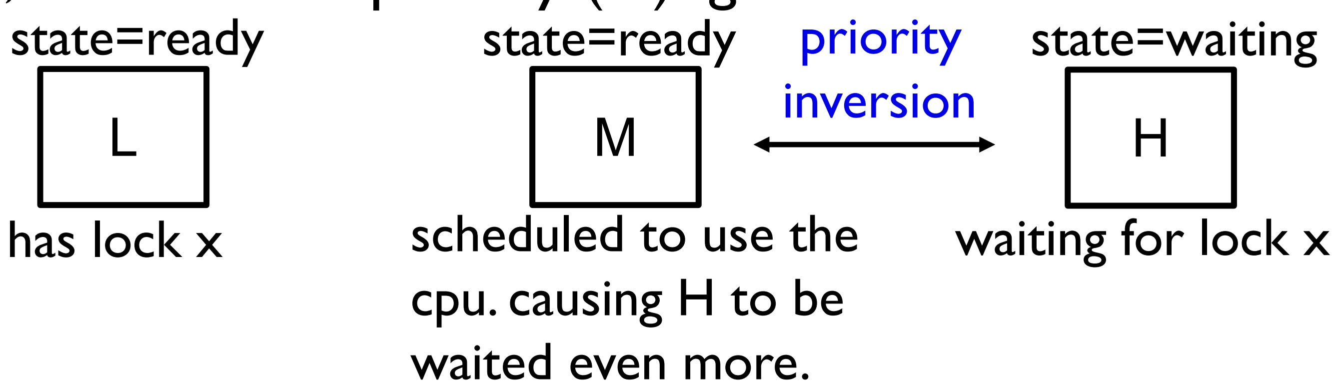
- Deadlocks: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Example: Let S and Q be two semaphores initialized to 1.

```
Sem S=1;
Sem Q=1;
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
….	….
signal (S);	signal (Q);
signal (Q);	signal (S);

Potential problems with semaphores

- Starvation: A process may never be removed from the semaphore queue in which it is suspended.
- Priority Inversion: Scheduling problem when lower-priority process (L) holds a lock needed by a higher-priority process (H), and medium priority (M) gets scheduled.



> Solution: process holding the lock *inherits* the priority of the process waiting for the lock. Then, L can run soon and release lock, so H can run earlier than M.

Potential problems with semaphores

Incorrect use of semaphore operations:

- signal (mutex) …. wait (mutex)
- wait (mutex) … wait (mutex)
- Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

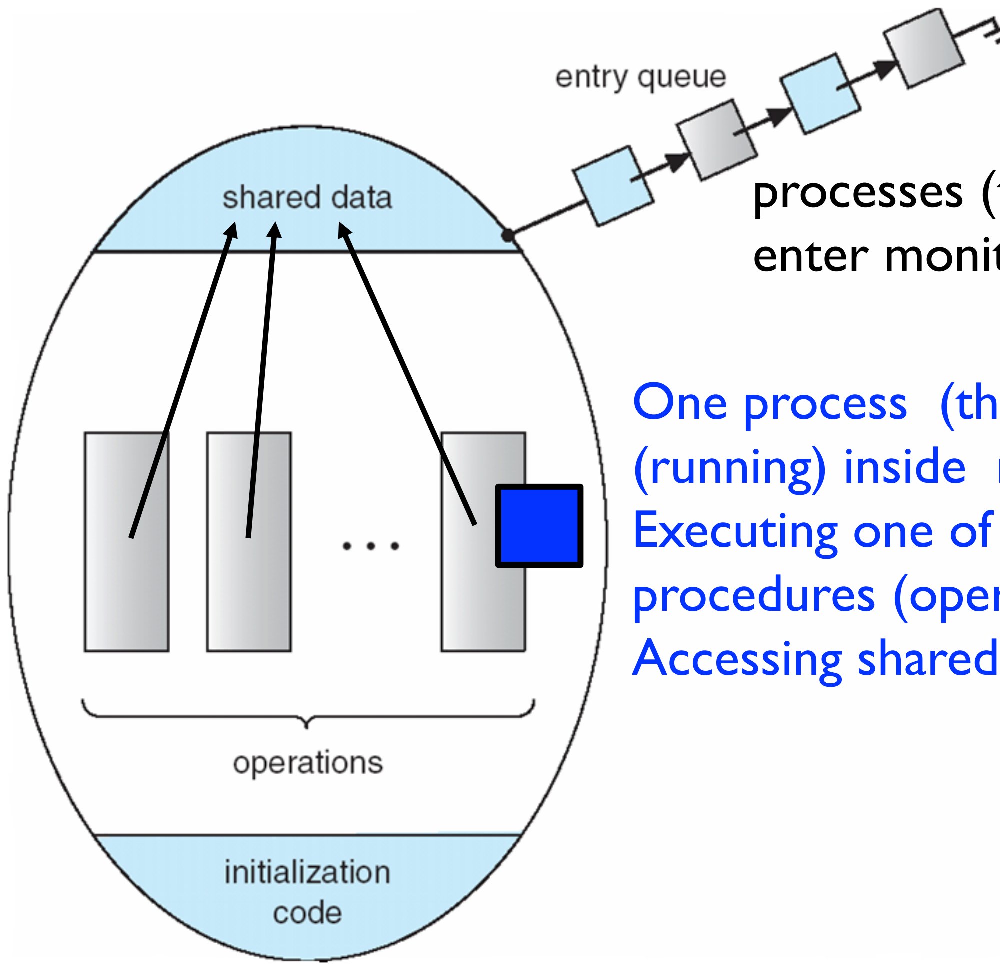
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for thread (process) synchronization (programming language construct).
- Only one thread (process) may be active (executing) within the monitor at a time.
- Shared variables/data/objects are put and accessed in monitor. Monitor controls access. No race condition then.
- Other names: thread-safe class, thread-safe object, or thread-safe module.
- You define a monitor like you define a class.
- Suitable for object-oriented programming.

Monitors

```
monitor monitor-name
{
        // shared variable declarations (shared data - object)

        procedure P1 (…) { …//using shared data… }
                …
        procedure Pn (…) {…// using shared data…}

        initialization code ( ….) { …//initialize shared data… }
                …
}
```

Schematic view of a Monitor



processes (threads) waiting to enter monitor

One process (thread) active (running) inside monitor. Executing one of the monitor procedures (operations). Accessing shared data.

Critical section (CS) problem easily solved. Put CS inside monitor as a procedure.

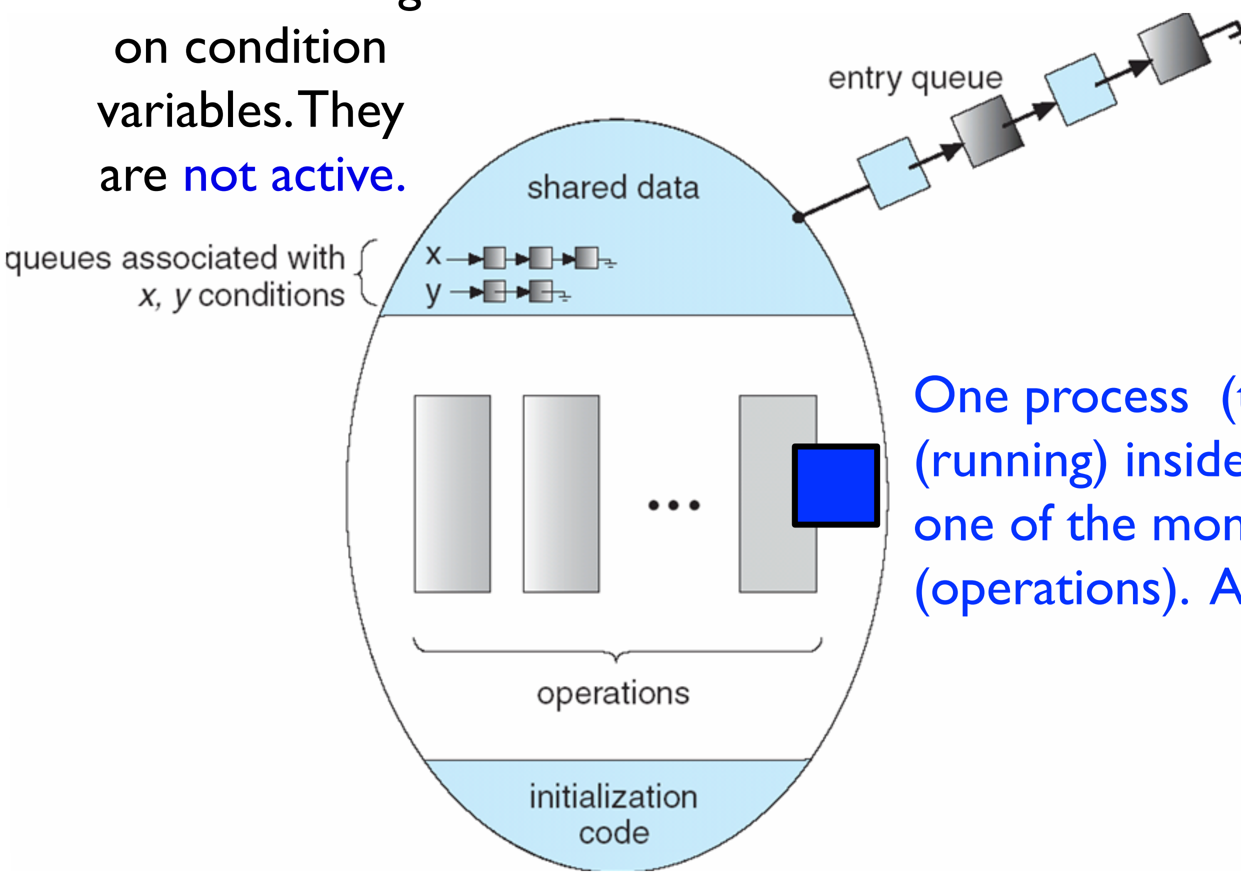
Condition variables in monitors

- Solving *other synchronization issues* requires additional support.
  - Therefore monitor construct also supports condition variables.
- condition x, y;   // defining condition variables
- Two operations on a condition variable:
  - x.wait () – a thread that invokes the operation is suspended (i.e., blocked, sleeping, waited).
  - x.signal () – resumes (wakes up) one of the threads (if any) that is sleeping on x.

Monitor with condition variables

Threads waiting on condition variables. They are not active.

Threads waiting to enter the monitor. They are not active.



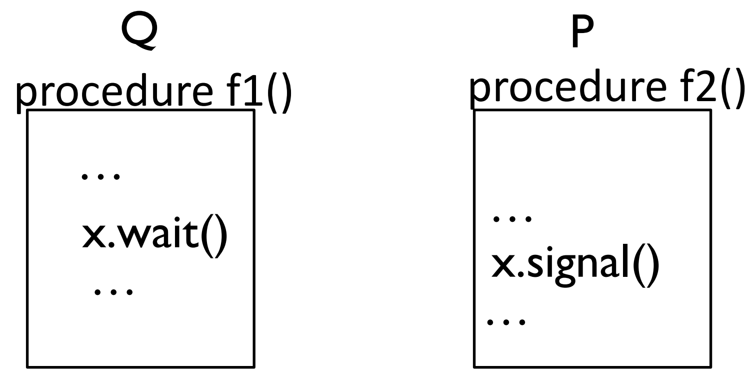
One process (thread) active (running) inside monitor. Executing one of the monitor procedures (operations). Accessing shared data.

Condition Variables versus Semaphores

- Condition variables and semaphores are different.
  - A condition variable does not count: have no associated integer.
  - A signal on a condition variable x is lost (not saved for future use) if there is no thread waiting (blocked) on the condition variable x.
  - The wait() operation on a condition variable x will always cause the caller to block.
  - The signal() operation on a condition variable will wake up a sleeping thread on the condition variable, if any. It has no effect if there is nobody sleeping.

What happens when a thread signals?

- If a thread P signals (a waiting thread Q), there is danger of two threads being in the monitor:
  - Signaling thread (P).
  - Waiting thread (Q).



- Two possibilities after signaling:
  1-Signal and Wait: Q will be active in monitor (Hoare semantics)
  2-Signal and Continue: P will be active in monitor (Mesa semantics). This semantics (Mesa) is *most commonly used now!*
- If possible, we can put signal() to the end of the procedure.

Monitor and condition variables: example 1

- Assume we have a resource to be accessed by many threads.
- Assume we have a total of 5 instances of the resource.
- This implies 5 threads can use the resource simultaneously.

- We want to implement a monitor that will implement two functions: request() and release() that can be called by a thread before and after using a resource.

Monitor and condition variables: example 1

```
monitor AllocateMon
{
        int count = 5;  // we initialize count to 5.
        condition c;

        void request () {
                while (count == 0)
                        c.wait();
                count--;
        }

        void release () {
                count++;
                c.signal();
        }
}
```

A thread (or process) will be coded like below:

```
AllocateMon MA;
// resource allocation monitor
….
MA.request();

// ….use the resource …

MA.release();
…
```

Monitor and condition variables: example 2

- Assume this time a thread may request multiple instances (k instances) in one request: request (k)
  - Total number of instances can be, for example, 100.
  - Then k should be <= 100 in request (k).
- If a thread can get all *k instances*, the request() call will return and the thread will use the resource instances.
  - Otherwise, the thread will be waited (sleeping) in request() call.
- Develop the monitor for controlling access to the resource having many instances.

Monitor and condition variables: example 2

```
monitor AllocateMon2 {
        count = 100;  // count of available instances (initially 100).
        condition c;   // to sleep on when required
        request (n)   // request n instances
        {
                while (count < n)
                        c.wait();
                count = count - n;
                return;
        }

        release (n)   // release n instances
        {
                count = count + n;
                c.broadcast();
        }
}
```

Thread Code

```
.

int k = …
M.request(k);

// access k instances
// do something

M.release (k);
..
..
```

POSIX pthreads
mutex and condition variables

POSIX Pthreads synchronization tools: mutex and condition variables

- POSIX Pthreads API is OS-independent (Linux, Solaris, FreeBSD, etc.)
- It provides:
  - mutex locks
  - condition variables
  - semaphores (named and unnamed)

- Mutex locks and conditions variables together can be used like monitors.
- Non-portable extensions include:
  - read-write locks
  - spin locks

Use of mutex/cond-vars instead of monitor construct

- There is no monitor in C.
- Hence, Pthreads condition variables in a C program is not accessed inside a monitor.
- Therefore condition variables are used in combination with mutex locks.

Pthreads

- Pthreads API provides wait(), signal(), and broadcast() operations on a condition variable.
- Mesa semantics is used.
  - The process calling signal() or broadcast() operation has the lock and can continue after signal() or broadcast().
  - Woken up process (or processes) is (are) added to the lock queue.

Usage Pattern

- When there is *shared state (data)* to be accessed by multiple threads, encapsulate it into a shared data structure (abstract data type):
  - Define a data structure with methods and shared data (variables).
  - Define a lock variable associated with the data structure (defined in the structure).
  - Define one or more condition variables if needed (defined in the structure).

Usage Pattern

```
//shared state example
int x;      // shared
lockt_t lock;
condition_t  c1, c2;

foo1() {
        lock (&lock);
        .....
        ....../access-modify x....
        while (...)
                cv_wait(&c1, &lock);

        ....
        unlock(&lock);
}
```

cv_wait() body first releases the lock; then sleeps on cv queue. when waken up; tries acquiring lock again. when it has lock cv_wait() returns.

```
foo2() {
        lock (&lock);
        ....
        ....../access-modify x....
        ....
        c1.signal();
        ....
        unlock (&lock)
}

foo3() {
        lock (&lock);
        ....../access-modify x....
        unlock (&lock);
}
```

Usage Pattern

- When a method will access shared data, make it first acquire the lock in the beginning and release the lock at the end.
- Wait on a condition variable when needed.
- Signal or broadcast on a condition variable when needed.

Usage Pattern

Shared Data (variables)

lock: lock variable
cv: condition variable
(one or more CVs)

```
method1(...) {
	acquire (&lock)
	...
	//access shared data
	...
	while (! condition)
		wait (&cv, &lock);

	release(&lock);
	return;
}
```

```
method2(...) {
	acquire (&lock)
	...
	//access shared data
	...
	// change the shared data to
	// make expected
	// condition true
	signal(&cv)
	// or broadcast (&cv);
	...
	release(&lock);
	return;
}
```

condition is an expression including shared variables

Classical Problems of Synchronization

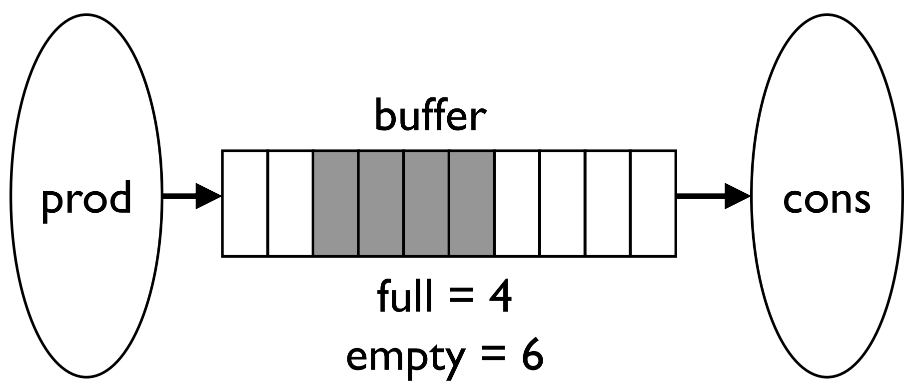
Classical Problems of Synchronization

- Bounded-Buffer Problem (producer/consumer problem)
- Readers and Writers Problem
- Dining-Philosophers Problem

- They can be used to test a new tool for synchronization: how well the tool is solving the problems.

Bounded Buffer Problem (producer-consumer)

- N buffer entries, each can hold one item
- buffer is a circular array
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N.



Bounded Buffer Problem Solution with semaphores

The structure of the producer process

```
do  {
            //   produce an item

             wait (empty);
             wait (mutex);

             //  add the item to the  buffer

             signal (mutex);
             signal (full);

} while (TRUE);
```

The structure of the consumer process

```
do {
             wait (full);
             wait (mutex);

            //  remove an item from
            //  the buffer

            signal (mutex);
            signal (empty);

            //  consume the retrieved item
} while (TRUE);
```

Bounded Buffer Problem
Solution with mutex lock and condition variables

```
// shared -global- variables
item buffer[N];
int in, out;
int count;
pthread_mutex_t lock;
pthread_cond_t empty, full;
```

// empty: a buffer slot is empty

// full : a buffer slot is full

```
initialization code (in main thread)
in = 0;
out = 0;
count = 0;
pthread_mutex_init (&lock, NULL);
pthread_cond_init (&empty, NULL);
pthread_cond_init (&full, NULL);
```

Bounded Buffer Problem
Solution with mutex lock and condition variables

```
producer thread code
while (1) {
        // produce new item
        pthread_mutex_lock (&lock);
        while (count == N)
                pthread_cond_wait (&empty, &lock);
        buffer[in] = item;
        in = (in + 1) % N;
        count++;
        pthread_cond_signal (&full);
        pthread_mutex_unlock (&lock)
}
```

Bounded Buffer Problem
Solution with mutex lock and condition variables

consumer thread code

```
while (1) {
        pthread_mutex_lock (&lock);
        while (count == 0)
                pthread_cond_wait (&full, &lock);
        item = buffer[out];
        out = (out + 1) % N;
        count--;
        pthread_cond_signal (&empty);
        pthread_mutex_unlock (&lock);
        // consume the retrieved item
}
```

Bounded Buffer Problem
Solution with Monitors (pseudo-code)

```
monitor ProducerConsumer {
        item buffer[N];
        int in = 0;
        int out = 0;
        integer count = 0;
        condition empty, full;  // you could give different names
    void insert(item x) {
            while (count == N);
                    empty.wait();
            buffer[in] = x;
            in = (in + 1) % N;
            count++;
            full.signal();
            return;
    }
    item remove() {
            while (count == 0);
                    full.wait();
            x = buffer[out];
            out = (out + 1) % N;
            count--;
            empty.signal();
            return (x)
    }
} // end of monitor definition
```

Bounded Buffer Problem
Solution with Monitors (pseudo-code)

ProducerConsumer PCmon; *// PCmon is a monitor object*

<u>producer thread code</u>

```
while (1) {
        // produce an item
        item = ...
        PCmon.insert (item);
}
```

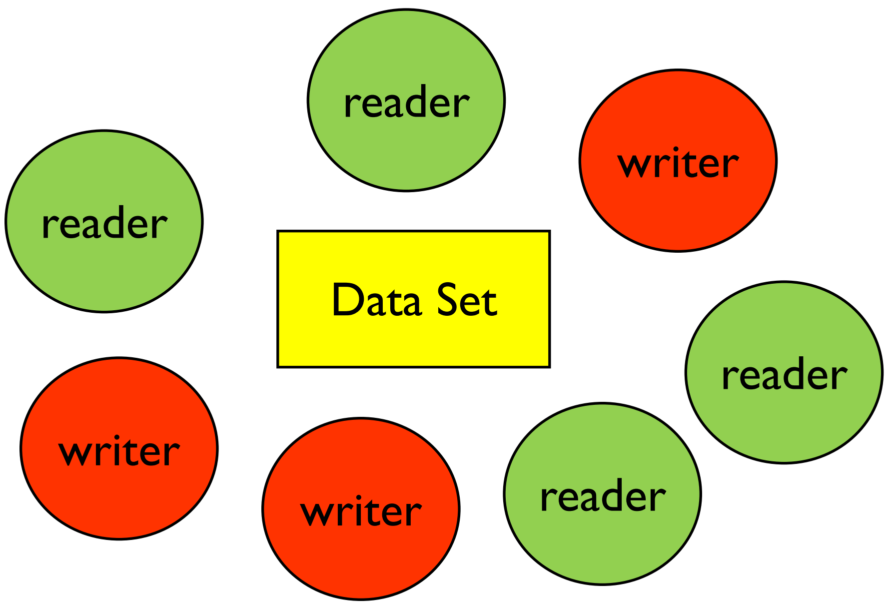
<u>consumer thread code</u>

```
while (1) {
        item = PCmon.remove ();
        // consume the item
}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes (or threads)
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write

- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time



Readers-Writers Problem

- Shared Data
  - Data set (Database)
  - integer readcount initialized to 0
    - Number of readers reading the data at the moment
  - Semaphore mutex initialized to 1
    - Protects the readcount variable
      (multiple readers may try to modify it)
  - Semaphore ds_lock initialized to 1
    - Protects the Data set
      (either writer or reader(s) should access Data set at a time)

Readers-Writers Problem

The structure of a writer process

```
do {
        wait (ds_lock) ;
        //    writing is performed
        signal (ds_lock) ;
} while (TRUE);
```

The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (ds_lock) ;
        signal (mutex);

        // reading is performed

        wait (mutex) ;
        readcount -- ;
        if (readcount  == 0)
                signal (ds_lock) ;
        signal (mutex) ;
} while (TRUE);
```

Dining-Philosophers Problem (DP)

5 philosophers around a table

- There are 5 philosophers sitting around a table.
- In a loop, they think and they want to eat from time to time.
  - When a philosopher wants to eat, he needs two forks.
  - He can only eat with 2 forks (one fork is not enough).
  - When a philosopher has 2 forks, he can eat.
  - Then, after eating, he will release the forks and will start thinking again.
- Forks are resources.
- Philosophers are processes.
- Develop the code for such a program using 5 processes or threads running concurrently.

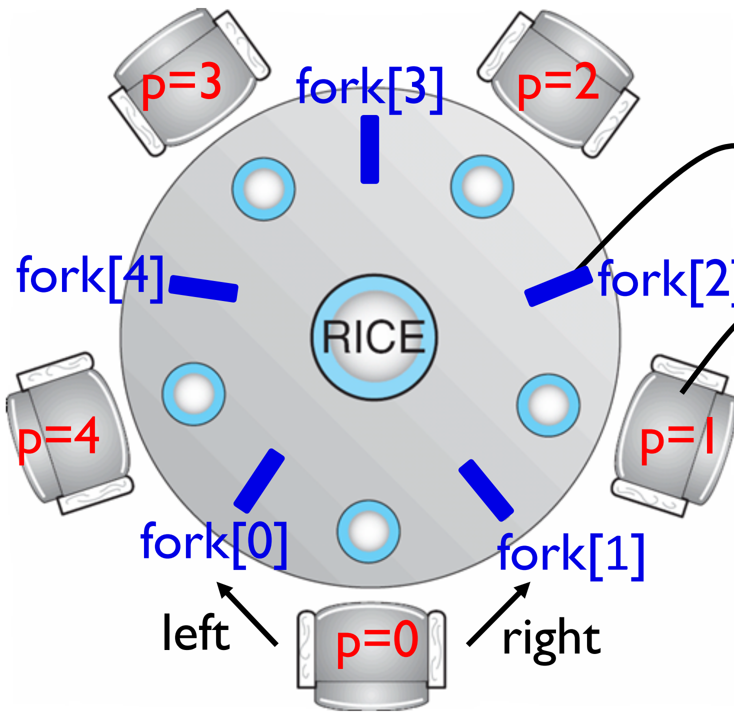
Dining-Philosophers Problem (DP)

5 philosophers around a table

5 philosophers (processes); 5 forks (resources); numbered as 0, 1, 2, 3, 4

Assume: to eat, a philosopher i needs two forks (*left* fork, *right* fork)

for p=4:
left=4
right=(4+1)%5=0



a resource

a process (philosopher)

for p=i:
left(i) = i;
right(i) = (i+1)%5

A fork can be used by one philosopher at a time.

Dining-Philosophers Problem

- Is not a real problem; it is a resource allocation problem.
- If we can solve this, we can also solve similar *real* resource allocation problems.
- For an ideal solution:
  - We want to have concurrency: two philosophers that are not sitting next to each other should be able to eat concurrently.
  - We don't want deadlock: waiting for each other indefinitely.
  - We don't want starvation: no philosopher waits forever.

DP semaphore solution but has deadlocks!

Semaphore **forks [5]** *initialized to 1. // shared among processes*

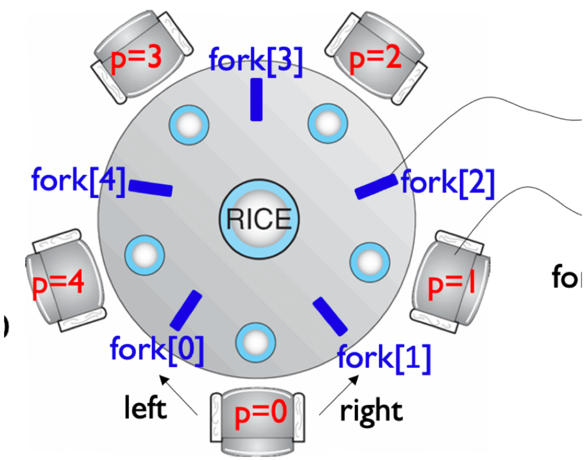
process (p) i code

```
do  {
            wait ( fork[i] ); // left fork
            wait ( fork[(i + 1) % 5] ); // right fork

             //  eat

            signal ( fork[i] ); // left fork
            signal ( fork[(i + 1) % 5] ); // right fork

           //  think
} while (TRUE);
```

This solution provides concurrency, but may result in deadlock and starvation. Hence it is not a good solution.

DP semaphore solution without deadlocks

Semaphore **forks [5]** *initialized to* 1. *Shared among processes.*



process (i) code

```
do  {
          if (i != 4) {  // i is 0, 1,2, or 3
                    wait ( fork[i] );  // left
                    wait ( fork[(i + 1) % 5] );  // right
          } else {  // i  is 4
                    wait ( fork[(i + 1) % 5] );  // right
                    wait ( fork[i] ); // left
          }
          //  eat
          signal ( fork[i] );  // left
          signal ( fork[(i + 1) % 5]); // right
          //  think
} while (TRUE);
```

We break the symetry

Monitor Solution to Dining Philosophers without deadlocks

```
monitor DiningPhilosophers {
    enum { THINKING;
            HUNGRY, EATING) state [5] ; // shared
    condition  cond [5]; // shared

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            cond[i].wait;
    }
    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
       test((i + 4) % 5)
       test((i + 1) % 5);
    }

    void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
             (state[(i + 1) % 5] != EATING) &&
             (state[i] == HUNGRY)) {
                state[i] = EATING ;
                cond[i].signal ();
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
} /* end of monitor */
```

Monitor Solution to Dining Philosophers without deadlocks

- Each philosopher invokes the operations pickup() and putdown() in the following sequence:

philosopher (p) i

```
…
DiningPhilosophers DP;   // DP: monitor object
….
while (1)
            THINK…
            // wants to eat
            DP.pickup (i);  // may block caller
             ....
            eat for a while  // uses resources – 2 forks
             ..
            DP.putdown (i);
            // finished with eeating
            THINK…
}
```

DP: another semaphore solution without deadlocks

```
// shared definitions
enum { THINKING, HUNGRY, EATING) state [5] ;
Semaphore mutex = 1;
Semaphore Sem[5]; // semaphore array; all initialized to 0.

pickup(int i) {
        wait (&mutex);
        state[i] = HUNGRY;
        test(i);
        signal (&mutex);
        wait (&Sem[i]);
}

putdown(int i) {
        wait (&mutex);
        state[i] = THINKING;
        test((i+4)%5);  // may wake up
        test ((i+ 1)%5);  // may wake up
        signal (&mutex);
}

test (int i) {
        if ((state[i] == HUNGRY) &&
                (state[(i+4)%5] != EATING) && (state[(i+ 1)%5] != EATING)) {
                state[i] = EATING;
                signal (&Sem[i]);
        }
}
```

DP: another semaphore solution without deadlocks

```
void philosopher (int i)
{
        while (1) {
                // THINK
                ....
                pickup (i);
                // Eat
                ....
                putdown (i)
        }
}
```

Each philosopher (process or thread) will execute this code.

References

- Operating System Concepts, Silberschatz et al., Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum et al.
- Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau et al.
- Operating Systems: Principals and Practice, T. Anderseon et al.

Additional material (option)

Implementing non-busy waiting locks

```
typedef struct __lock_t {
        int flag;        // this is the actual lock value (0: unlocked; 1: locked).
        int guard;       // spin lock to protect flag
        queue_t *q;   // threads (ids) will wait in this queue to get the lock
}


void lock_init (lock_t *m) {
        m->flag -= 0;
        m->guard = 0;
        queue_init (m->q);  // initially queue is empty
}
```

Implementing non-busy waiting locks

```
void lock (lock_t *m) {
        while (TestAndSet(&m->guard, 1) == 1)    // get spin lock
                ;
        if (m->flag == 0)
                m->flag = 1;                     // we have lock now
                m->guard = 0;                    // release spin lock
        } else {
                queue_add (m->q, gettid());  // add your id to queue
                m->guard = 0;                    // release spin lock
                park();   // sleep (kernel will change state to “waiting”).
        }
}
```

Implementing non-busy waiting locks

```
void unlock (lock_t *m) {
        while (TestAndSet(&m->guard, 1) == 1) // get the spin lock
                ;
        if (queue_empty(m->q))   // check if there is somebody in the queue
                m->flag = 0;  // release lock (there is no one)
        else
                unpark ( queue_remove (m->q)) //wake up one thread from queue
                                               // that has the lock now
        m->guard = 0;   // release spin-lock
}
```

Implementing non-busy waiting locks

- We assumed there are two system calls that thread library can call:
  - park(): put the the calling thread into waiting state (no longer runnable)
  - unpark(tid): make the thread (with id=tid) runnable (added to ready queue – so that it can run)
- This implementation (locks without busy-waiting – i.e., with a queue) uses internally spin lock to protect short critical regions internally.