# Chapter 7:  Synchronization Examples

# Outline

- Explain the bounded-buffer synchronization problem

- Explain the readers-writers synchronization problem

- Explain and dining-philosophers synchronization problems

- Describe the tools used by Linux and Windows to solve synchronization problems.

- Illustrate how POSIX and Java can be used to solve process synchronization problems
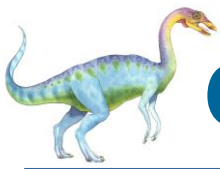
# Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes

  - Bounded-Buffer Problem

  - Readers and Writers Problem

  - Dining-Philosophers Problem

# Readers-Writers Problem

- A data set is shared among a number of concurrent processes
    - **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
- Several variations of how readers and writers are considered – all involve some form of priorities

# Readers-Writers Problem (Cont.)

- Shared Data

  - Data set

  - Semaphore `rw_mutex` initialized to 1

  - Semaphore `mutex` initialized to 1

  - Integer `read_count` initialized to 0

# Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
    down(rw_mutex);
        ...
        /* writing is performed */
        ...
    up(rw_mutex);
}
```

# Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true){
        down(mutex);
        read_count++;
        if (read_count == 1) /* first reader */
                down(rw_mutex);

        up(mutex);

        ...
        /* reading is performed */

        ...

        down(mutex);
        read_count--;
        if (read_count == 0) /* last reader */
                up(rw_mutex);

        up(mutex);

    }
```

# Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the "First reader-writer" problem.

- The "Second reader-writer" problem is a variation the first reader-writer problem that state:

  - Once a writer is ready to write, no "newly arrived reader" is allowed to read.

- Both the first and second may result in starvation, leading to even more variations

- Problem is solved on some systems by kernel providing reader-writer locks

# Dining-Philosophers Problem

- N philosophers' sit at a round table with a bowel of rice in the middle.



- They spend their lives alternating thinking and eating.

- They do not  interact with their neighbors.

- Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl

  - Need both to eat, then release both when done

- In the case of 5 philosophers, the shared data

    ▸ Bowl of rice (data set)

    ▸ Semaphore chopstick [5] initialized to 1

# Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher $i$ :

```
while (true){
    down (chopstick[i] );
    down (chopStick[ (i + 1) % 5] );

     /* eat for awhile */

    up (chopstick[i] );
    up (chopstick[ (i + 1) % 5] );

     /* think for awhile */

    }
```

- What is the problem with this algorithm?

# Solutie corecta cu semafoare

```
#define THINKING        0
#define HUNGRY          1
#define EATING          2
int state[N];
semaphore_t   mutex = {1};
semaphore_t   ph[N];   // toate initializate cu 0
void philosopher(int i) {
        while(true) {
                think();
                take_forks(i);
                eat();
                put_forks(i);
        }
}
```

# Solutie corecta cu semafoare

```
void take_forks(int i) {
        down(mutex);
        state[i] = HUNGRY;
        test(i);
        up(mutex);
        down(ph[i]);
}
void put_forks(int i) {
        down(mutex);
        state[i] = THINKING;
        test((i – 1) % N));
        test((i + 1) % N));
        up(mutex);
}
```

```
void test(int i) {
        if(state[i] == HUNGRY &&
            state[(i-1)%N] != EATING) &&
            state[(i+1)%N] != EATING)
        {
                    state[i] = EATING;
                    up(ph[i]);
        }
}
```

# Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers

{

    enum {THINKING; HUNGRY, EATING} state [5];

    condition self [5];


    void pickup (int i) {

          state[i] = HUNGRY;

          test(i);

          if (state[i] != EATING) self[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] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
                self[i].signal () ;
        }
}


    initialization_code() {
      for (int i = 0; i < 5; i++)
      state[i] = THINKING;
     }
}
```

# Solution to Dining Philosophers (Cont.)

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

        DiningPhilosophers.pickup(i);

                /** EAT **/

        DiningPhilosophers.putdown(i);

- No deadlock, but starvation is possible

# POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.

- Named semaphores can be used by unrelated processes, unnamed cannot.

# POSIX Named Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name SEM.

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

# POSIX Unnamed Semaphores

- Creating an initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

# POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor, POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

# POSIX Condition Variables

- Thread waiting for the condition `a == b` to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
        pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 #define ITEMS          8
5 #define BUFFER_SIZE    4
6 int buffer[BUFFER_SIZE];
7
8 void *producer(void*);
9 void *consumer(void*);
10
11 int spaces, items, tail;
12 pthread_cond_t space, item;
13 pthread_mutex_t buffer_mutex;
14
15 int main()
16 {
17         pthread_t producer_thread, consumer_thread;
18         void *thread_return;
19         int result;
20
21         spaces = BUFFER_SIZE;
22         items = 0;
23         tail = 0;
24
25         pthread_mutex_init(&buffer_mutex, NULL);
26         pthread_cond_init(&space, NULL);
27         pthread_cond_init(&item, NULL);
28
29         if(pthread_create(&producer_thread, NULL, producer, NULL) ||
30            pthread_create(&consumer_thread, NULL, consumer, NULL))
31                 exit(1);
32
33         if(pthread_join(producer_thread, &thread_return))
34                 exit(1);
35         else
36                 printf("producer returns with %d\n", (int)thread_return);
37
38         if(pthread_join(consumer_thread, &thread_return))
39                 exit(1);
40         else
41                 printf("consumer returns with %d\n", (int)thread_return);
42         exit(0);
43 }
44
```

# Producator-consumator POSIX

```
 1 void *producer(void *arg)
 2 {
 3         int i;
 4
 5         for(i = 0; i < ITEMS; i++)
 6         {
 7                 pthread_mutex_lock(&buffer_mutex);
 8                 while(spaces == 0)
 9                         pthread_cond_wait(&space, &buffer_mutex);
10                 buffer[(tail + items) % BUFFER_SIZE] = i;
11                 items += 1;
12                 spaces -= 1;
13                 printf("producer puts %d\n", i);
14                 pthread_mutex_unlock(&buffer_mutex);
15                 pthread_cond_signal(&item);
16         }
17         pthread_exit(0);
18         return (void*)0;
19 }
20
21 void *consumer(void *arg)
22 {
23         int i, b;
24
25         for(i = 0; i < ITEMS; i++)
26         {
27                 pthread_mutex_lock(&buffer_mutex);
28                 while(items == 0)
29                         pthread_cond_wait(&item, &buffer_mutex);
30                 b = buffer[tail % BUFFER_SIZE];
31                 tail += 1;
32                 items -= 1;
33                 spaces += 1;
34                 printf("consumer gets %d\n", b);
35                 pthread_mutex_unlock(&buffer_mutex);
36                 pthread_cond_signal(&space);
37         }
38         pthread_exit(0);
39         return (void*)0;
40 }
```

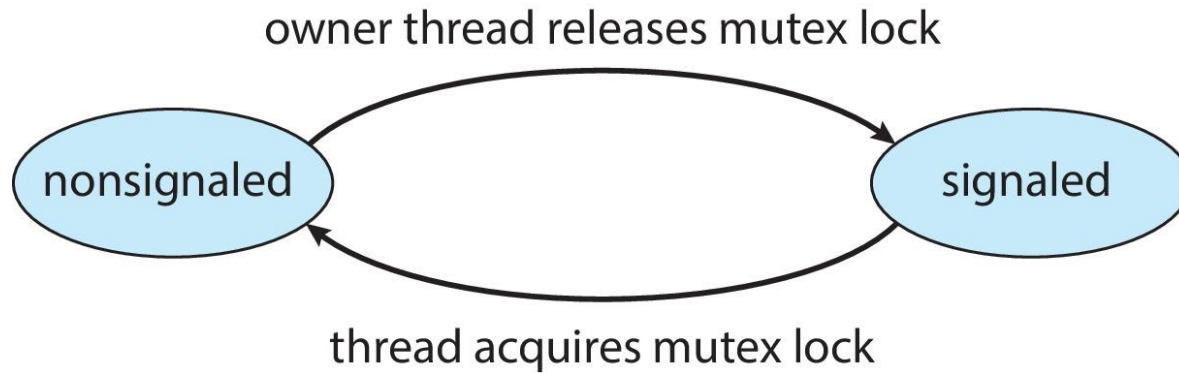# Kernel Synchronization - Windows

- Uses interrupt masks to protect access to global resources on uniprocessor systems

- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted

- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers

  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

# Kernel Synchronization - Windows

- Mutex dispatcher object

owner thread releases mutex lock

nonsignaled ⟷ signaled

thread acquires mutex lock

# Linux Synchronization

- Linux:

  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections

  - Version 2.6 and later, fully preemptive

- Linux provides:

  - Semaphores

  - Atomic integers

  - Spinlocks

  - Reader-writer versions of both

- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption

# Linux Synchronization

- Atomic variables

  `atomic_t` is the type for atomic integer

- Consider the variables

  `atomic_t counter;`
  `int value;`

| Atomic Operation | Effect |
|---|---|
| atomic_set(&counter,5); | counter = 5 |
| atomic_add(10,&counter); | counter = counter + 10 |
| atomic_sub(4,&counter); | counter = counter - 4 |
| atomic_inc(&counter); | counter = counter + 1 |
| value = atomic_read(&counter); | value = 12 |

# Sincronizare in Solaris

- lock-uri adaptive (adaptive locks), variabile de conditie, semafoare, reader-writer locks, turnstiles

- lock-uri adaptive

  - folosite pt sectiuni critice scurte (cateva sute de instructiuni)

  - comportament polimorf spinlock-semafor

  - daca un thread vrea acces la date protejate de un lock detinut de un alt thread care ruleaza pe alt CPU, lock-ul se comporta ca un spinlock

  - daca lock-ul e detinut de un thread care nu ruleaza momentan pe nici un procesor, threadul care incearca sa obtina lock-ul va fi pus in stare dormanta (sleep), i.e., lock-ul se comporta ca un semafor

  - pe un sistem uniprocesor, cand un thread incearca sa obtina un lock detinut de alt thread, comportamentul lock-ului e intotdeauna de tip semafor

# POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS

# POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

# Transactional Memory

- Consider a function update() that must be called atomically. One option is to use mutex locks:

```
void update ()
{
    acquire();

    /* modify shared data */

    release();
}
```

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically. A transaction can be completed by adding **atomic{S}** which ensure statements in **S** are executed atomically:

```
void update ()
{
    atomic {
        /* modify shared data */
    }
}
```

# Memorie tranzactionala

- software transactional memory, Herlihy & Moss 1993

- am vazut deja ex. LL/SC

- exploateaza ideea de tranzactii din baze de date (v. proprietati ACID) si controlul optimist al concurentei

- un thread termina modificarile operate pe o zona de memorie partajata fara a se coordona cu alte threaduri

- se inregistreaza fiecare operatie de citire/scriere intr-un log

- la finalul tranzactiei se incearca operatia de *commit*

  - daca reuseste (i.e., nici o alta tranzactie nu a apucat sa faca un commit reusit intre timp), tranzactia e validata si modificarile devin permanente

  - daca esueaza, tranzactia e abandonata (aborted) si se re-executa de la inceput pana reuseste (*transaction roll-back*)

- beneficiu major: grad crescut de concurenta, thread-urile nu au nevoie sa se sincronizeze prin lock-uri la accesul memoriei partajate

# Suport HW pt. memorie tranzactionala

- ex. Intel TSX (Transactional Synchronization Extension), extensie ISA x86, microarhitectura Haswell (2013)

- accesibila prin intermediul a doua interfete
  - HLE (Hardware Lock Elision) – backward compatibility
  - RTM (Restricted Transactional Memory)

- HLE adauga doua prefixuri de instr. XACQUIRE & XRELEASE
  - se pot folosi doar pt anumite instructiuni care trebuie prefixate explicit cu LOCK
  - permite executia optimista a sectiunii critice sarind scrierea lock-ului, a.i. acesta apare liber pt alte threaduri
  - o tranzactie esuata determina reluarea operatiei de la instructiunea prefixata cu XACQUIRE

# Suport HW pt. memorie tranzactionala

- RTM adauga la ISA trei noi instructiuni

  - XBEGIN – marcheaza inceputul zonei de memorie tranzactionala

  - XEND - marcheaza sfarsitul zonei de memorie tranzactionala

  - XABORT – abandoneaza explicit o tranzactie

  - esecul tranzactiei redirecteaza executia catre codul specificat de instr. XBEGIN, iar codul de eroare e stocat in registrul EAX

- instructiunea XTEST permite testarea starii procesorului (este sau nu in mijlocul executiei unei regiuni de memorie tranzactionala)

# RTM elision pt. pthread_mutex_lock

```
void elided_lock_wrapper(lock) {
        if(_xbegin() == _XBEGIN_STARTED) { // start tranzactie
                if(lock  e liber)
                        return;    // executa SC in tranzactie
                _xabort(0xff);       // abandoneaza tranzactia
        }
        obtine lock
}
void elided_unlock_wrapper(lock) {
        if (lock e liber)
                _xend();            // comite tranzactia
        else
                elibereaza lock;
}
```

# Java Synchronization

- Java provides rich set of synchronization features:
  - ➢ Java monitors
  - ➢ Reentrant locks
  - ➢ Semaphores
  - ➢ Condition variables

# Java Monitors

- Every Java object has associated with it a single lock.

- If a method is declared as `synchronized`, a calling thread must own the lock for the object.

- If the lock is owned by another thread, the calling thread must wait for the lock until it is released.

- Locks are released when the owning thread exits the `synchronized` method.

```java
public class BoundedBuffer<E>
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private E[] buffer;

    public BoundedBuffer() {
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    /* Producers call this method */
    public synchronized void insert(E item) {
        /* See Figure 7.11 */
    }

    /* Consumers call this method */
    public synchronized E remove() {
        /* See Figure 7.11 */
    }
}
```
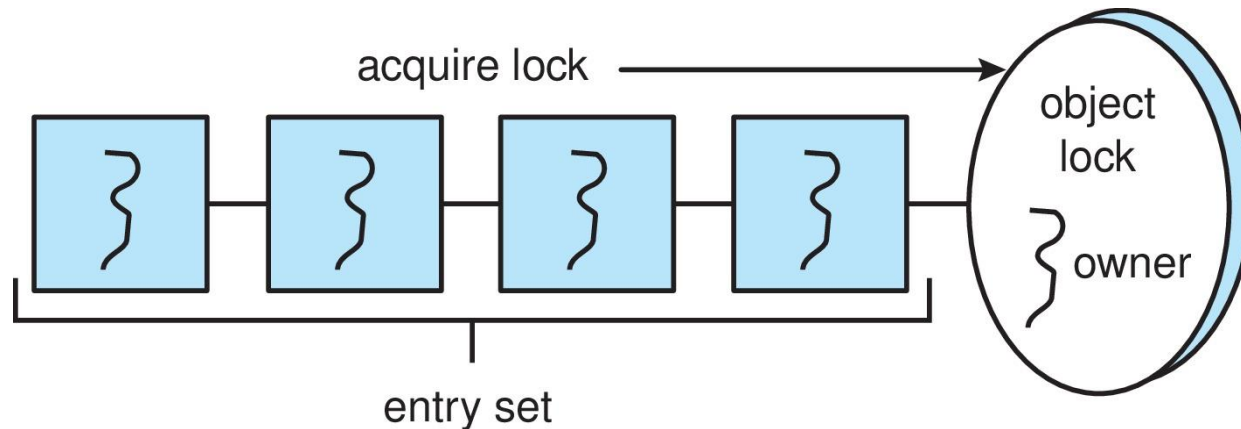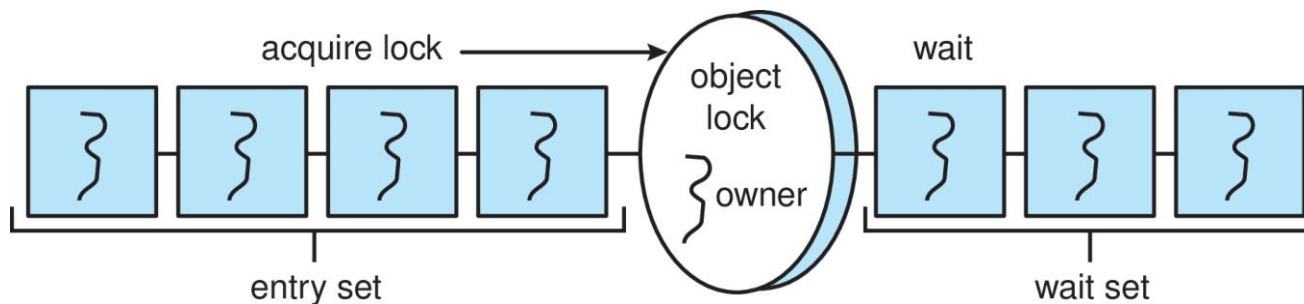
# Java Synchronization

- A thread that tries to acquire an unavailable lock is placed in the object's **entry set**:

# Java Synchronization

- Similarly, each object also has a **wait set**.
- When a thread calls `wait()`:
  1. It releases the lock for the object
  2. The state of the thread is set to blocked
  3. The thread is placed in the wait set for the object

# Java Synchronization

- A thread typically calls wait() when it is waiting for a condition to become true.

- How does a thread get notified?

- When a thread calls `notify()`:

  1. An arbitrary thread T is selected from the wait set
  2. T is moved from the wait set to the entry set
  3. Set the state of T from blocked to runnable.

- T can now compete for the lock to check if the condition it was waiting for is now true.

# Bounded Buffer – Java Synchronization

```java
/* Producers call this method */
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException ie) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    count++;

    notify();
}
```

```
/* Consumers call this method */
public synchronized E remove() {
   E item;

   while (count == 0) {
      try {
         wait();
      }
      catch (InterruptedException ie) { }
   }

   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   count--;

   notify();

   return item;
}
```

# Java Reentrant Locks

- Similar to mutex locks

- The **finally** clause ensures the lock will be released in case an exception occurs in the **try** block.

```
Lock key = new ReentrantLock();

key.lock();
try {
   /* critical section */
}
finally {
   key.unlock();
}
```

# Java Semaphores

- Constructor:

```
Semaphore(int value);
```

- Usage:

```java
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   /* critical section */
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

# Java Condition Variables

- Condition variables are associated with an **ReentrantLock**.

- Creating a condition variable using **newCondition()** method of **ReentrantLock**:

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

- A thread waits by calling the **await()** method, and signals by calling the **signal()** method.

# Java Condition Variables

- Example:

- Five threads numbered 0 .. 4

- Shared variable `turn` indicating which thread's turn it is.

- Thread calls `doWork()` when it wishes to do some work. (But it may only do work if it is their turn.

- If not their turn, wait

- If their turn, do some work for awhile ….…

- When completed, notify the thread whose turn is next.

- Necessary data structures:

```java
Lock lock = new ReentrantLock();
Condition[] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
    condVars[i] = lock.newCondition();
```

# Java Condition Variables

```java
/* threadNumber is the thread that wishes to do some work */
public void doWork(int threadNumber)
{
    lock.lock();

    try {
        /**
         * If it's not my turn, then wait
         * until I'm signaled.
         */
        if (threadNumber != turn)
            condVars[threadNumber].await();

        /**
         * Do some work for awhile ...
         */

        /**
         * Now signal to the next thread.
         */
        turn = (turn + 1) % 5;
        condVars[turn].signal();
    }
    catch (InterruptedException ie) { }
    finally {
        lock.unlock();
    }
}
```

# Alternative Approaches

- Transactional Memory

- OpenMP

- Functional Programming Languages

# OpenMP

- OpenMP is a set of compiler directives and API that support parallel progamming.

```
void update(int value)
{
        #pragma omp critical
        {
                count += value
        }
}
```

- The code contained within the **#pragma omp critical** directive is treated as a critical section and performed atomically.

# Functional Programming Languages

- Functional programming languages offer a different paradigm than procedural languages in that they do not maintain state.

- Variables are treated as immutable and cannot change state once they have been assigned a value.

- There is increasing interest in functional languages such as Erlang and Scala for their approach in handling data races.

# End of Chapter 7