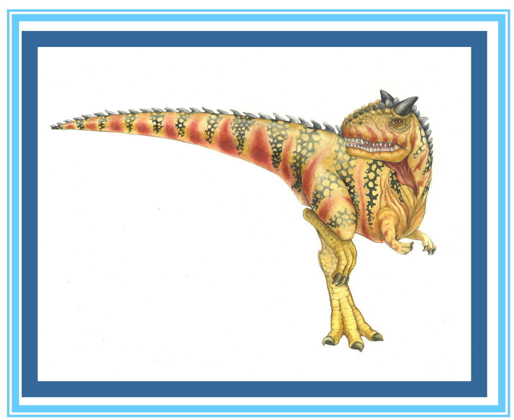
Section 6: Synchronization Tools



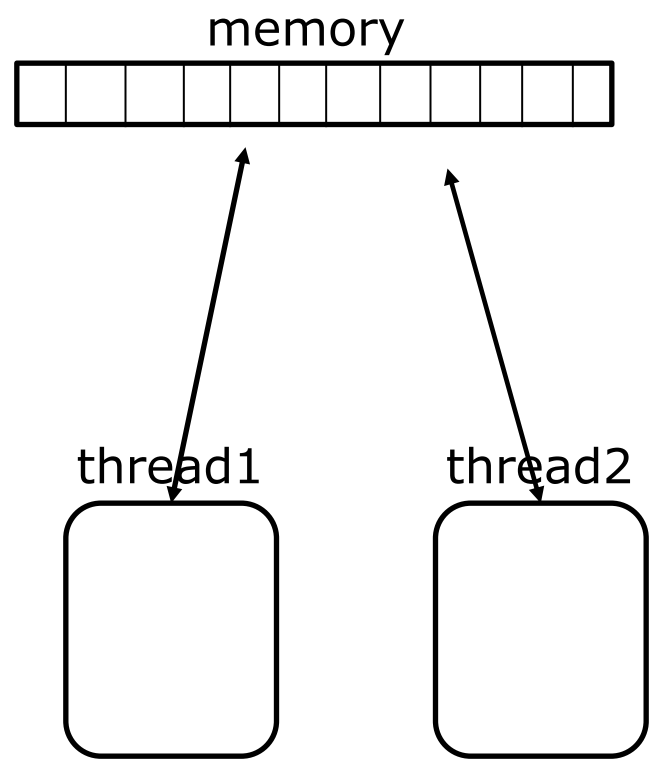
Processes/Threads Synchronization

- Synchronization problems such as the Critical-Section Problem
- Synchronization solutions:
  - Hardware instructions
  - Mutex locks
  - Binary semaphores
  - Counting semaphores
- Synchronization Examples

The data consistency problem

```
int sum = 0;
thread(k..l){
  int local-sum = 0;
  int temp;
  int i;
  for (i=k;i<= l;i++)
     local-sum  = local-sum + i;
  temp = sum;
--------------------------
  sum = temp + local-sum;
}

main(){
makethread1(thread(1..5));
makethread2(thread(6..10));
}
```



The value of sum depends on the schedule order of the two threads!

Objectives

- Introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- Present software and hardware synchronization primitives
- Describe different synchronization problems and solution approaches using synchronization primitives

Motivation

- When a parent process executes fork():
  1. The parent page table is updated to be copy on write
  2. The parent page table is copied to become the child page table
  3. A process control block is created for the child process
  4. A new row is added in the process table to store info about the new child process
- All these operations are done on behalf of the parent process
- At any time the parent process can be de-scheduled
  - The first 3 data structures are not shared among processes
  - However the fourth one is shared among all the processes and threads
- If the parent process is de-scheduled in the middle of updating the process table
  - ps –ef may return the entry for the child process with missing info (no big deal)
  - Or the same row is used to enter the info of 2 different processes (BIG DEAL)

Motivation

- One solution to the above problem is to make the kernel non-preemptive, i.e. processes running kernel code cannot be de-scheduled until kernel code is completed
- The Linux kernel is fully preemptive, unlike many other Unix like OSs
- In this case, the synchronization tools describe in this section are used to make sure shared data structure are updated in a consistent manner
- Preemptive kernels are much more complicated to design and program but they improve response time of user oriented processes as well as of high priority real time and system processes

Background/Motivation

- Cooperating processes have concurrent access to shared data:
  - Threads: data section
  - Processes: shared memory segment
- This sharing may result in data inconsistency (variables get values which cannot have occurred according to the algorithm)
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Example: thread

```
#define NTHREADS     4
#define ARRAYSIZE   1000000
#define ITERATIONS   ARRAYSIZE / NTHREADS
double  sum=0.0, a[ARRAYSIZE];
pthread_mutex_t sum_mutex;
void *do_work(void *tid) {
    int i, start, *mytid, end;
    double mysum=0.0;
    mytid = (int *) tid;
    start = (*mytid * ITERATIONS);
    end = start + ITERATIONS;
    for (i=start; i < end ; i++) {a[i] = i * 1.0;   mysum = mysum + a[i];   }
    pthread_mutex_lock (&sum_mutex);
    sum = sum + mysum;
    pthread_mutex_unlock (&sum_mutex);
    sleep(15);  pthread_exit(NULL);
}
```

- The threads that execute the function “do_work” share the double “sum”
- Threads cannot be allowed to execute the instruction “sum = sum + mysum” is same time, otherwise the value of sum may be inconsistent
- The “mutex” synchronization primitive allows only one thread at a time to update sum

Producer-consumer problem

- This a prototypal example of the problems we address in this section:
  - A producer thread write data in an array “buffer[]” and a consumer thread read the data from an entry of “buffer[]”
  - Producer and consumer must synchronize, so for example the consumer does not try to consume an item when the buffer is empty
  - We can synchronize by having the two threads to share a synchronization variable, count.
  - Thus, count is incremented every time a new item is added in buffer and is decremented every time one item is removed from buffer

```
while (true) { /*Producer*/
    produce an item Item;
    while (count == buffer_size);
            //do nothing, buffer full
    buffer [in] = Item;
    in = (in + 1) % buffer_size;
    count++;
    }
```

```
while (true)  { /*Consumer*/
    while (count == 0);
        // do nothing, buffer empty
    Item =  buffer[out];
    out = (out + 1) % buffer_size;
    count--;
    consume the item Item;
    }
```

Problems with the shared var "count"

- The variable count "must be" a static/global variable shared by both threads
- Problems arise when the two threads try in same time to increase and decrease the value of count:
  - Suppose that the value of the variable count = 5
  - The producer and consumer execute the non-atomic statements count++ and count-- concurrently.
  - Then the new value of count could be 4, 5, or 6!

```
while (true) { /*Producer*/
    produce an item Item;
    while (count == buffer_size);
            //do nothing, buffer full
    buffer [in] = Item;
    in = (in + 1) % buffer_size;
    count++;
    }
```

```
while (true)  { /*Consumer*/
    while (count == 0);
        // do nothing, buffer empty
    Item =  buffer[out];
    out = (out + 1) % buffer_size;
    count--;
    consume the item Item;
    }
```

Imlementation of “count” update

- Instructions count++ and count– can be implemented in machine instructions such as:
  - count++

    register1 = count
    register1 = register1 + 1
    count = register1

  - count--

    register2 = count
    register2 = register2 - 1
    count = register2

Count update (cont)

The sequence of the concurrent executions of these instructions could be as below:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

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

```
register2 = count
register2 = register2 – 1
count = register2
```

Race Condition

- Variations in the outcome of the computation are caused by the scheduler which determines in which order threads modify the variable count
- This is called race condition because processes are seen as racing to access the shared variable

```
S0: producer execute register1 = count   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 = 6}
S2: consumer execute register2 = count   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 = 4}
S4: producer execute count = register1   {count = 6 }
S5: consumer execute count = register2   {count = 4}
```

Critical section

- Several threads access and manipulate the same data concurrently.
  - The outcome of the execution depends on the particular order in which the access is granted to sharing threads
- To guard against the race condition, we need to ensure that:
  - Only one thread at a time can access and modify shared variables
- This section of the code that can be executed by only one thread at a time is called a "critical section"

.

Critical section

- The critical sections in the producer-consumer problem above are the code sections where the shared variable "count" is updated

```
while (true) { /*Producer*/
    produce an item Item;
    while (count == buffer_size);
        //do nothing
    buffer [in] = Item;
    in = (in + 1) % buffer_size;
    count++; //critical section
    }
```

```
while (true)  { /*Consumer*/
    while (count == 0);
        // do nothing
    Item =  buffer[out];
    out = (out + 1) % buffer_size;
    count--;  //critical section
    consume the item Item;
}
```

- Note that although these two threads execute different functions, they still compete with each other to update the variable count
- The section of these functions that update count must not be executed in same time

Example: thread

```
#define NTHREADS     4
#define ARRAYSIZE   1000000
#define ITERATIONS   ARRAYSIZE / NTHREADS
double  sum=0.0, a[ARRAYSIZE];
pthread_mutex_t sum_mutex;
void *do_work(void *tid) {
    int i, start, *mytid, end;
    double mysum=0.0;
    mytid = (int *) tid;
    start = (*mytid * ITERATIONS);
    end = start + ITERATIONS;
    for (i=start; i < end ; i++) {a[i] = i * 1.0;   mysum = mysum + a[i];   }
    pthread_mutex_lock (&sum_mutex);
    sum = sum + mysum;
    pthread_mutex_unlock (&sum_mutex);
    sleep(15);  pthread_exit(NULL);
}
```

- In the above code example, the critical section is sum = sum + mysum;
- Entry in this critical section is controlled by the mutex synchronization primitive pthread_mutex_lock (&sum_mutex); that a thread must acquire in order to execute the code in the critical section

Critical section

- ***Critical section***:
  - Segments of code that updates shared variables
- Given *n* threads sharing same variables:
  - The code that access shared data in each thread is a critical section
  - When one process is executing in its critical section, no other process is allowed to execute in its critical section

```
do {
        entry section

                critical section

        exit section

                remainder section

} while (TRUE);
```

- Process requests permission to enter its critical section (in the entry section)
- Critical section may have an exit section
- Remainder of the code is the remainder section

Critical-Section Solution

A good solution to the critical section problem requires fairness as well as *exclusive access*.

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
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

Hardware instructions

- Access control to critical sections can be implemented using machine code
- Computer systems provide different special hardware instructions that:
  - test and modify the content of a variable
  - or swap the contents of two variables

atomically, i.e. the execution of these hardware instructions cannot be interrupted

- We look at two hardware instructions:
  - **Test&Set** instruction: *test-and-modify* the content of a word
  - **Compare&Swap** instruction: *swap* the contents of two words atomically (uninterruptedly)

Test&set()

- The test&set instruction can be defined as follows

- ```
  boolean test&set (boolean *lock){
          boolean rv = *lock;  /* test */
          *lock = true;  /* set */
          return rv:
        }
  ```

- Fetch lock from main memory.
- Return the value of lock
- Store true into the memory address for lock
- While a test&set instruction is running, if another process/thread or CPU runs test&set, it will be served a hardware interrupt
  ```

Test&set(): application

- There is a global (shared) variable **lock**, a Boolean initialized to **false**
- The function below is executed by several threads
- Each thread try to get the lock, only one thread at a time is successful at capturing the lock

```
do {
        while (test&set(&lock))
         ; /* do nothing */
         Execute critical section
         lock = false; /*release lock*/
    }
```

- ```
  boolean test&set (boolean *lock){
          boolean rv = *lock;
          *lock = true;
          return rv:
         }
  ```
  ```

Compare&swap (cas): definition

- Definition

```
int cas(int *value, int expected, int new_value){
    int temp = *value;
    if (*value == expected)
         *value = new_value;
    return temp;
 }
```

- Interpretation:
  - Value is the state of the lock
  - (if value == expected) means “the lock is free (lock == 0)” then the calling process captures the lock by making value = 1
    - i.e. changing the state of the lock from free to not free
- All these steps occur without interruption, i.e. once cas() starts it will complete without been de-scheduled

cas(): application

- There is a global (shared) variable `lock`, an integer initialized to `0`
- The function below is executed by several threads
- Each thread try to get the lock, only one thread at a time is successful at capturing the lock

```
while (cas(&lock, 0, 1) != 0)
          ; /* do nothing */
     Execute critical section
     lock = 0; /* release the lock */
```

- **Always return the state of lock**
- **Compare lock with 0, if equal set the lock to 1**

```
int cas(int *value, int expected, int new_value){
  int temp = *value;
  if (*value == expected)
       *value = new_value;
  return temp;
}
```

Example: Atomic Adder

Application to obtain an atomic adder:

```
function add(int *p, int a) {
    value ← *p
    while (cas(p,value,value+a) != value)
        value ← *p
    }
}
```

```
int cas(int *value, int expected, int new_value){
  int temp = *value;
  if (*value == expected)
      *value = new_value;
  return temp;
 }
```

Example: Atomic Adder

Application to obtain an atomic adder:

```
function add(int *p, int a) {
    value ← *p
    while (cas(p,value,value+a) != value)
        value ← *p
    }
    return  value;
}
```

```
int cas(int *value, int expected, int new_value){
  int temp = *value;
  if (*value == expected)
       *value = new_value;
  return temp;
 }
```

Hardware Synchronization

- Solutions to the critical-section problem based on the atomic instructions test_and_set() and compare-and-swap() satisfy the mutual-exclusion condition
- Do not satisfy the bound-waiting condition
- Another test_and_set() instruction satisfies the conditions:
  - The method requires two data items to be shared between *n* processes:

```
Boolean waiting[n];
Boolean lock;
```

  - All initialized to false.

Bounded-waiting Mutual Exclusion with test_and_set()

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = FALSE;
    critical section
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
    remainder section } while (TRUE);
```

- *Process Pi execute this code*
- *Pi* enters its critical section only if either waiting[i] == false or key == false
- The first process to execute the TestAndSet() will find key == false. All others must wait.

Bounded-waiting Mutual Exclusion with test_and_set()

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
            key = TestAndSet(&lock);
        waiting[i] = FALSE;
       critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
              j = (j + 1) % n;
        if (j == i)
              lock = FALSE;
        else
             waiting[j] = FALSE;
      remainder section  } while (TRUE);
```

- Process P*i* find the next waiting process if any
- If no waiting process, release the lock
- Otherwise hold the lock, and *Pj* is granted to enter its critical section

Bounded-waiting Mutual Exclusion with TestAndSet()

- Mutual-exclusion condition:
  - When a process leaves its critical section, only one waiting[j] is set to false.
- Progress condition:
  - A process exiting the critical section either sets lock to false or sets waiting[j] to false
  - Both allow a process that is waiting to enter its critical section to proceed.
- Bounded-waiting condition:
  - When a process leaves its critical section, it scans the array waiting in the cyclic ordering `waiting`
  - Any waiting process will thus do so **within *n-1* turns**.

Software: Mutex Lock

- The hardware-based solutions are complicated and often not accessible to application programmers.
- Instead, operating systems provide higher-level software
- Tools to solve the critical-section problem such as mutex lock and semaphore.
- The simplest of these tools is the mutex lock

Mutex lock

- Threads must explicitly acquire the lock and then explicitly release the lock when exiting the critical section:
  - First **`acquire()`** a lock
  - Then **`release()`** the lock

```
do {
    acquire lock
        critical section
    release lock
}
```

- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually, these two functions are implemented via hardware atomic instructions such as compare&swap.

POSIX Mutex Locks

- A mutex lock is a Boolean variable. It can only be modified and used by specific functions
- Crea

```
#include <pthread.h>

pthread_mutex_t mutex;

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

- Acquiring

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

/* critical section */

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

Inconvenient of mutex lock

- The main disadvantage of mutex lock is that it requires, while a process is in its critical section, that any other process that tries to enter its critical section to loop continuously in calling for the lock
- This is called **busy waiting,** and a mutex lock is called a **spinlock** because the threads “spins” while waiting for the lock to become available.
- Busy waiting from spinning threads wastes CPU cycles that some other threads/process might be able to use productively.
- *Spinlocks do have an advantage, however, no context switch is required when a process must wait on a lock, a context switch may take considerable time*

Semaphores

- Semaphore is another tool to synchronize access to critical sections
- A semaphore *S* – is a special type of integer where, apart from initialization, only two other operations can modify the value of a variable of type semaphore: wait() and signal()
- wait() decrement by 1 the value of a semaphore
- signal() does the opposite, it increments by 1 the value of a semaphore
- Modifications to the integer variable in wait() and signal() are atomic operations
- Definitions of wait() and signal() are as followed:

```
wait (S) {
   while S <= 0;
     // no-op
    S--;
    }
```

```
signal (S) {
      S++;
     }
```

Types of semaphores

- Two types of semaphores:
  - **Binary semaphore** – integer value can range only between 0 and 1. Same as a mutex lock
  - **Counting semaphore** – integer value can range over an unrestricted domain

Binary Semaphore: Mutex

- Integer value can range only between 0 and 1;
- The code below can be executed by several threads

```
semaphore mutex;    //  initialized to 1

do {
    wait(mutex);
    Execute critical section
    signal(mutex);
 }
```

- When mutex = 0, the thread trying to execute this code will block as the value of a mutex cannot be negative

Usage of binary semaphore

- Synchronization using a binary semaphore:
  - Assume process P_1 with a statement S_1 and process P_2 with a statement S_2
  - It is required that S_2 is executed only after S_1 has completed
  - Using binary semaphore sync, initialized to 0,
    - when P_1 is ready it will execute S_1
    - P_2 will execute its statement S_2 only after P_1 has invoked signal(sync)
- P_1 P_2

```
S1;
signal(sync);
```

```
wait(sync);
S2;
```

Exercise with bin semaphores

Consider the following two processes that run concurrently

P_1	P_2

print(A);
print(B);
print(C);
	print(E);
	print(F);
	print(G);

Insert semaphores to satisfy the following properties. Don't forget to provide the initial values of the semaphores

(a) Print A before F

(b) Print F before C

- Sem1 = 0; sem2 = 0
- T1 T2
- Print(A) print(E)
- Signal(sem1) wait(sem1)
- Print(B) Print(F)
- Wait(sem2) signal(sem2)
- Print(C) Print(G)

Counting Semaphore

- Integer value can range over an unrestricted domain.
- Examples of counting semaphore declarations in Windows and Java:
  - Windows:

```
CreateSemaphore(
        NULL,          // default security attributes
        INIT_COUNT,  // initial value of the semaphore, 0 <= INIT_COUNT <= MAX_COUNT
        MAX_COUNT,  // maximum value of the semaphore
        SEM_NAME);         // name of the semaphore
```

  - Java:

```
Semaphore sem = new Semaphore(8);   //8 is the max value of the semaphore
```

Usage of counting semaphores

- Counting semaphores are used to manage computer resources or to allow more than one process/thread in same time in a critical section
- Managing resources:
  - The OS may put limits on the number of open files a process may have. This policy can be implement using a counting semaphore *cs* initialized to the number of allowed open files
    - Each time the process open a file, cs is decremented
    - Once *cs* = 0, the process is not allowed to open more files
    - Once the process close a file, *cs* is incremented
  - The OS may put limit on the number of frames a process may have in physical memory.
    - Each time a page fault occur, the counting semaphore *cs* is decremented
    - If a page fault occurs and *cs* = 0, then a replacement page has to be found among the page the process currently hold in main memory

Semaphore Implementation

- The basic implementation of semaphores we have just seen is also a busy waiting:
  - When wait(S) is called, the instruction busy wait until S > 0.

```
wait (S) {
   while S <= 0;  // no-op
   S--;}
```

Semaphore Implementation without busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list

```
typedef struct {
   int value;
   struct process *list;
} semaphore;
```

Implementation with no busy waiting

Two operations:

block – place the process invoking the operation on the appropriate waiting queue

wakeup – remove one of the processes in the waiting queue and place it in the ready queue

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      add this process to
S->list;
      block();
   }
}
```

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

Blocking Semaphores

- In busy waiting semaphore implementations, the value of a semaphore cannot be negative
- In the waiting queue implementation, semaphore values can be negative:
  - indicate the number of processes waiting on that semaphore
- List of waiting processes can be implemented as FIFO queue

Two applications of semaphores

- We look at two synchronization problems with solutions based on semaphores
  - Bounded-Buffer (producer-consumer) Problem
  - Readers and Writers Problem

Producer-consumer problem

```
while (true) { /*Producer*/
      produce an item Item;
      while (count == buffer_size);
                //do nothing, buffer full
      buffer [in] = Item;
      in = (in + 1) % buffer_size;
      count++;
      }
```

```
while (true)  { /*Consumer*/
      while (count == 0);
            // do nothing, buffer empty
      Item =  buffer[out];
      out = (out + 1) % buffer_size;
      count--;
      consume the item Item;
      }
```

Bounded-Buffer Problem

- Same problem as the producer-consumer problem beginning of this section. Called bounded-buffer because the buffer is of finite size
- The previous solution used a variable “count”, however this variable could have been updated incorrectly by competing threads
- Here introduce a solution with semaphores, no “count” variable, it is an example of how counting semaphores can be used

- *The producer and consumer threads share the following data structures*
  - A buffer of size n
  - Binary semaphore **`mutex`** initialized to the value 1
  - Counting semaphore **`full`** initialized to the value 0
  - Counting semaphore **`empty`** initialized to the value n

- Updating the buffer (adding or removing an item) is now a critical section protected by the *mutex* binary semaphore, *mutex* is initialized to the 1.

Bounded Buffer Problem (Cont.)

- Code for the producer thread:

```
while (true) {
   produce an item
   wait(empty);
   wait(mutex);
   add item in the buffer  /*critical section */
   signal(mutex);
   signal(full);
}
```

- Semaphore *empty* initialized to *n* count the number of available entries in the buffer. If *empty* = 0, the producer thread is blocked.
- Semaphore *mutex,* initialized to 1, controls access to the critical section. If *mutex* = 0, the producer thread is blocked.
- Semaphore *full* initialized to 0 count the number of items in the buffer. signal(full) increments this semaphore, indicating one more item has been added in the buffer

Bounded Buffer Problem (Cont.)

- Code for the consumer thread:

```
while (true) {
      wait(full);
      wait(mutex);
      remove an item from the buffer /* critical section */
      signal(mutex);
      signal(empty);
   }
```

- If there is no item in the buffer, the counting semaphore *full* = 0, thus the consumer thread block.
- If *mutex* = 0, meaning the producer is adding an item in the buffer, the consumer thread block.
- signal(empty) decrements the value of the counting semaphore *empty,* indicating there is one less item in the buffer

Running the threads

```
while (true) { /*producer*/
    produce an item
    wait(empty);
    wait(mutex);
    add item in the buffer
    signal(mutex);
    signal(full);
 }
```

```
while (true) {/*consumer*/
    wait(full);
    wait(mutex);
    remove an item from
        the buffer
    signal(mutex);
    signal(empty);
  }
```

Readers-Writers Problem

- A database is to be shared among several concurrent processes.
- Some processes only read the database (*readers*), others seek to update (read and write) the database (*writers*)
- There is no problems for two or more readers accessing the database simultaneously
- If a writer update the database while other processes access it (readers or writers), the requests may return incoherent results
- To avoid incoherence, it is required that writers have exclusive access to the database
- This synchronization problem is referred to as the **readers–writers problem**. It has been used to test nearly every new synchronization primitive.
- Here we solve this synchronization problem using semaphores

Readers-Writers Problem

- A database 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

Readers-Writers Problem (Cont.)

- Shared Data
  - Database
  - Binary semaphore *rw_mutex* initialized to 1
  - Binary semaphore *mutex* initialized to 1
  - int *read_count* initialized to 0
  - Counting semaphore *readers* initialized to *n*
- *mutex* and *rw_mutex* are binary semaphores while *read_count* is an integer variable
- The read_count variable record the number of reader processes currently accessing the database

Readers-Writers Problem (Cont.)

A writer process

```
while (true) {
     wait(rw_mutex);
          ...
     /* writing is performed */
          ...
     signal(rw_mutex);
   }
```

While(true) {
Wait(readers);

A reader process

```
while (true){
   wait(mutex);
   read_count++;
   if (read_count == 1)
       /* first reader */
      wait(rw_mutex);
   signal(mutex);
   ...
   /* reading is performed */
   ...
   wait(mutex);
   read count--;
   if (read_count == 0)
      /* last reader */
      signal(rw_mutex);
   signal(mutex);
 }
```

Semaphores Ubuntu

```
#include etc

sem_t mutex; //semaphore declaration


    sem_wait(&mutex);


    //critical section

    sem_post(&mutex); //equivalent to signal
}


int main()
{
    sem_init(&mutex, 0, 1);
sem_init(sem_t *sem, int pshared, unsigned int value);
```

Semaphores: Potential Problems

- Semaphores are effective, but they can be used incorrectly by a programmer which will result in synchronization errors difficult to detect.
- These errors happen only when some specific execution sequences take place
  - Here is a normal sequence of execution

```
wait(mutex);
    critical section
signal(mutex);
    remainder
```

  - Suppose a programmer swaps signal() and wait()

```
signal(mutex);
    critical section
wait(mutex);
    remainder
```

Deadlock

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Could happen in semaphores with the waiting queue
- Let S and Q be two semaphores initialized to 1

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

Deadlock again

- Suppose a programmer replaces signal() with wait()

```
wait (mutex);
    critical section
wait (mutex);
    remainder
```

- A deadlock will occur even when there is a single process as process will not be freed from the wait() operation

Monitors

- A monitor is a class in an object-oriented language
- A class of type monitor is such that a thread that execute a method in the class has exclusive access to the data in the class
- A monitor has also condition variables that can de-schedule a thread inside a monitor until conditions are satisfied.
- Only one thread is active inside a monitor at a time
- Note that C++ and Python don't have monitors, Java does

Schematic view of a Monitor

monitor monitor-name

{

// shared variable declarations

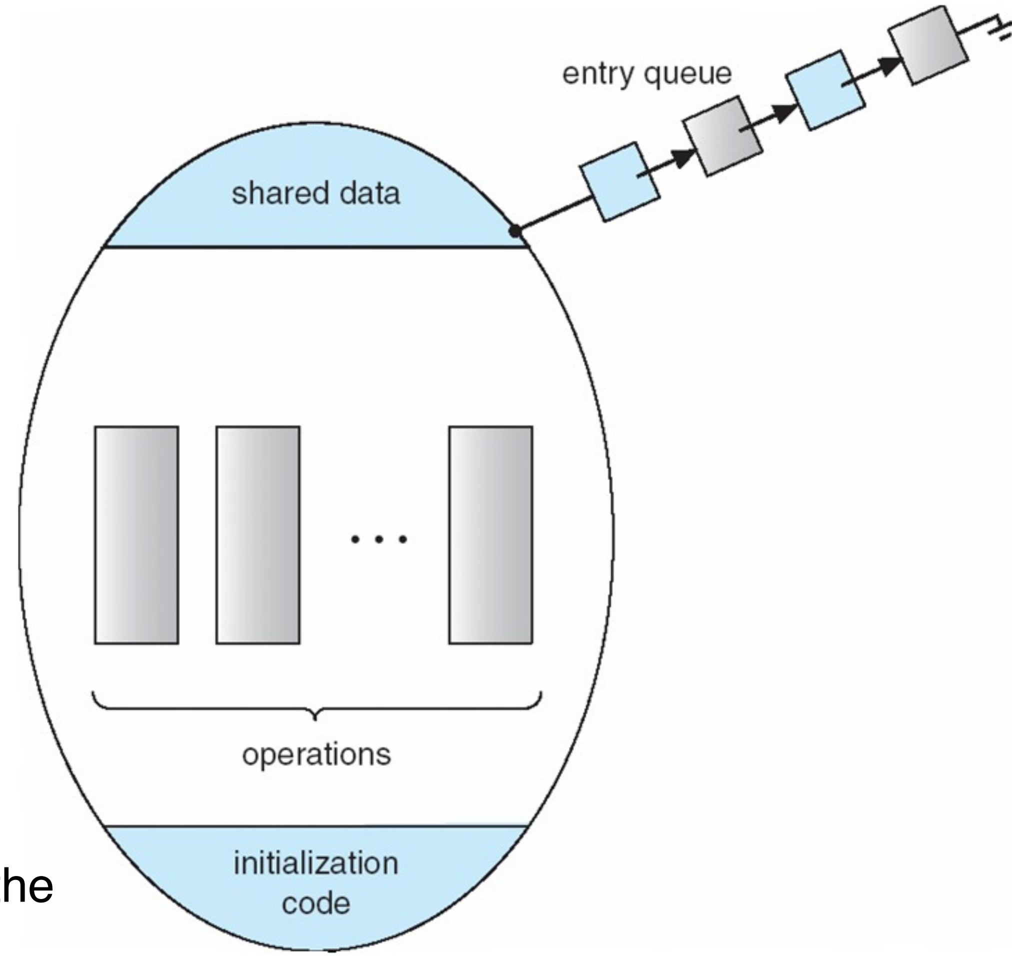
method M1 (…) { …. }

…

method Mn (…) {……}

Initialization code (….) { … }

}



The entry queue is a list of threads seeking to execute a method in the monitor

Monitor Implementation Using Semaphores

- Monitors can be implemented using semaphores

- Each method ***M*** is replaced by

```
wait(mutex);
    …
  body of M;
    …
signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Definition of a monitor class

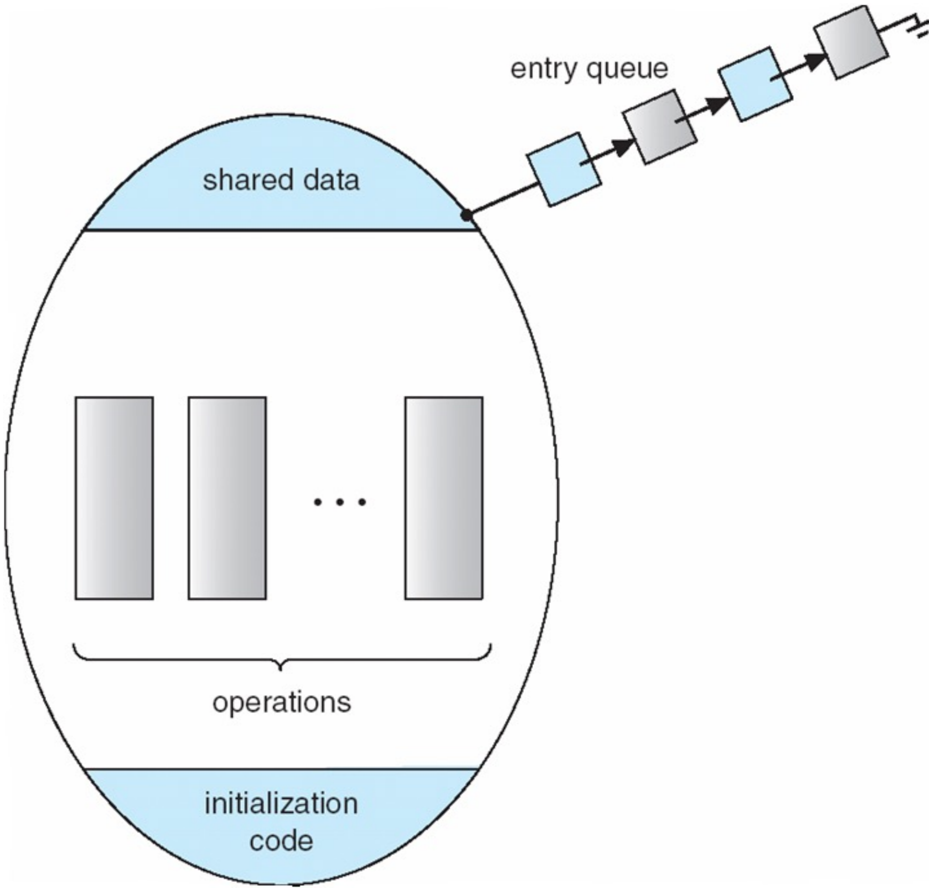
```
monitor class Account {
  private int balance = 0;
  public method boolean withdraw(int amount)
    if balance < amount return false;
    else
      balance := balance – amount; return true;
  public method deposit(int amount)
     balance := balance + amount;
```

Implementation of account

```
class Account {
  private lock myLock; private int balance = 0;
  public method boolean withdraw(int amount)
    myLock.acquire()
    try {
      if balance < amount {
        return false
      } else {
        balance := balance - amount
        return true
      }
    } finally {
      myLock.release()
    }
  public method deposit(int amount)
    myLock.acquire()
    try {
      balance := balance + amount
    } finally {
      myLock.release()
    }
}
```

Mutual Exclusion with Monitors

- If a thread calls a monitor method, it must acquire the lock
- Success to acquire the lock happen only if no other thread is active in the monitor
- If a thread is active in the monitor, the calling thread will be suspended until the currently executing thread releases the lock and exits the monitor
- Monitors reduce the risk caused by programmer mistakes



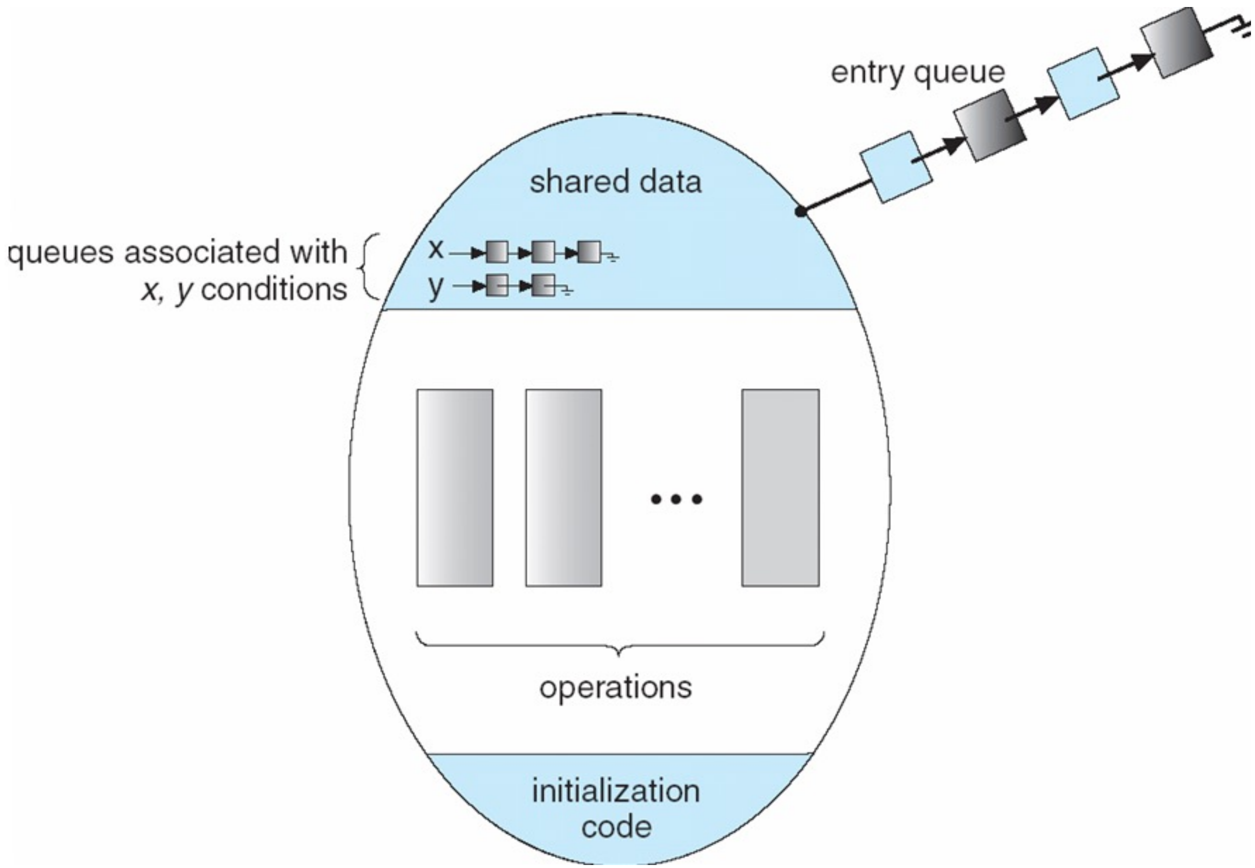
Condition Variables

- Monitors provide an easy way to achieve mutual exclusion, but this is not enough
- Also need a way for threads already in a monitor to block (give up exclusive access of the monitor) when they cannot proceed
- Blocking is implemented using ***condition*** variables

Condition Variables

- Programmers defines one or more condition variables.
- condition x, y;
- Only two operations can be executed on a condition variable:
  - x.wait () – a thread invoking the operation is suspended.
  - x.signal () – resumes one of the threads (if any) that has invoked x.wait ()

Monitor with Condition Variables



Condition Variables: Example

- condition full;
  - full.wait () – thread that invokes the operation is suspended.
  - full.signal () – resumes one of threads (if any) that invoked full.wait ()

Full Implementation of Monitors

- Variables

```
semaphore mutex;  // (binary, initially  = 1)
semaphore next;   // (binary, initially  = 0)
int next_count = 0; // number of processes waiting
                       inside the monitor
```

- Each method ***P*** will be replaced by

```
wait(mutex);
     …
   body of P;
     …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Implementation – Condition Variables

- For each condition variable ***x***, we have:

```
semaphore x_sem; //(binary, initially = 0)
int x_count = 0;//# of threads blocked on
condition variable x
```

- The operation `x.wait()` can be implemented as:

```
x_count++;
if (next_count > 0)
   signal(next);//thread in sem exec
else
   signal(mutex);// a new thread in sem
exec
wait(x_sem);  //thread block on x
x_count--;
```

Implementation (Cont.)

- The operation **`x.signal()`** can be implemented as:

```
        if (x_count > 0) {
           next_count++;
           signal(x_sem);
           wait(next); //thread block itself because 2 threads cannot execute in same time
           next_count--;
        }
```

Condition Variables: Example

- For instance, in the producer-consumer problem, we should block a producer thread **when the buffer is full** or the consumer thread **when the buffer is empty**
  - When the producer finds the buffer full, it does a wait() on condition variable “full”
    - This action causes the calling thread to block, and allows another thread that had been previously prohibited from entering the monitor to enter now
  - When the consumer finds the buffer empty, it does a wait() on condition variable “empty”
    - This action causes a thread that was blocked in the monitor to resume activity

Monitor class for producer/consumer

```
monitor ProducerConsumer
    condition full, empty;
    int count = 0;

  method add();
    if (count == N) wait(full);          // if buffer is full, block
    put_item(item);                      // put item in buffer
    count = count + 1;                   // increment count of full slots
    if (count == 1)  signal(empty);  // if buffer was empty, wake consumer

  method remove();
    if (count == 0)  wait(empty);        // if buffer is empty, block
    remove_item(item);                   // remove item from buffer
    count = count - 1;                   // decrement count of full slots
    if (count == N-1)  signal(full);   // if buffer was full, wake producer
end monitor;
```

Threads for producer/consumer

```
Thread 1
Producer();
   while (TRUE)
  {
     make_item(item);              // make a new item
     ProducerConsumer.add;           // call add method in monitor
  }

Thread 2
Consumer();
   while (TRUE)
   {
     ProducerConsumer.remove;  // call method remove in monitor
     consume_item;                   // consume an item
  }
```

End of Section 6