Operating System (OS)
CS232

Concurrency: Mutual Exclusion, Locks

Dr. Muhammad Mobeen Movania
Dr Muhammad Saeed

Outlines

- Concurrency & Race Condition (recap)
- Producer Consumer Problem
  - Mutual exclusion
  - Synchronization
- Semaphore
  - Counting and binary
- Mutual Exclusion
  - Mutex
  - Race Condition – solution
- Lock based Concurrent Data Structures

Concurrency & Race Condition

```
long *balance;//Shared Memory amoung multiple processes
```

```
void credit(int arg){
   int amount = arg;
   printf("Credit : balance = balance + %d\n",ar
   for(long i=0;i<5000000;i++){
        *balance = *balance + amount;
   }
}
```

```
void debit(int arg){
   int amount = arg;
   printf("Dedit : balance = balance - %d\n",arg
   for(long i=0;i<5000000;i++){
        *balance = *balance - amount;
    }
}
```

```
int main(){
//IPC - Shared Memory technique is used to store variable balance
   key_t key = ftok("sm_bal", 65);
   int shm_id=shmget(key, 8, IPC_CREAT | 0666);
   balance = (long*)shmat(shm_id, NULL, 0);
   *balance=0;   //initializing balance
```

```
int cpid = fork();
   if (cpid == 0){
       credit(1);
       shmdt(balance);
       exit(0);
   }
```

```
else{
    debit(1);
    waitpid(cpid,NULL,0);
    //debit(1);
    printf("Value of balance is: %ld\n", *balance);
    shmdt(balance);
    shmctl(shm_id, IPC_RMID, NULL);
    return 0;
}
```

```
Dedit : balance = balance - 1
Credit : balance = balance + 1
Value of balance is: -3762270
```

Concurrency & Race Condition

```
long balance = 0;//Shared, Global Variable
```

```
void * credit(void * arg){
    int amount = (int) arg;
    printf("Credit : balance = balance + %d\n",(int)arg);
    for(long i=0;i<5000000;i++){
        balance = balance + amount;
    }
    pthread_exit(NULL);
}
```

```
void * debit(void * arg){
    int amount = (int) arg;
   printf("Dedit : balance = balance - %d\n",(int)arg);
    for(long i=0;i<5000000;i++){
        balance = balance - amount;
    }
    pthread_exit(NULL);
```

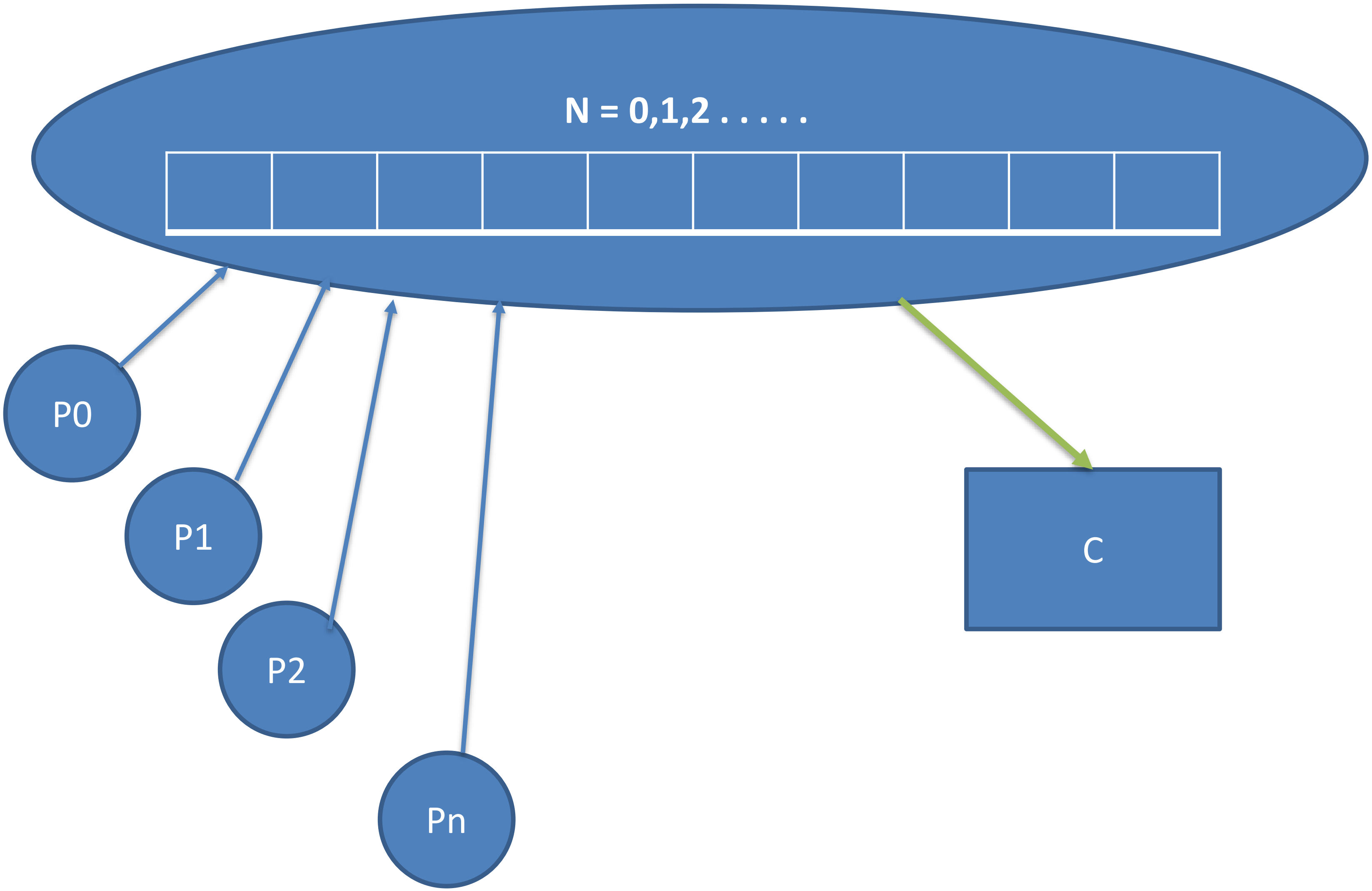
```
int main(){
    pthread_t t1, t2;
    pthread_create(&t1, NULL, credit,(void *)5);
    pthread_create(&t2, NULL, debit,(void *)5);

    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    printf("Value of balance is :%ld\n", balance);
    return 0;
}
```

```
Dedit : balance = balance - 5
Credit : balance = balance + 5
Value of balance is :-417255
```

Larger Context

- Client/Server Applications
- Peer to Peer Applications
- Producer / Consumer Problem
- Readers / Writers Problem

Produce/ Consumer Problem



Solution

- Requirements
  - Mutual Exclusion
  - Synchronization
- Semaphore
- Programming Techniques
  - Mutex & Condition Variables
  - Semaphore

Semaphore (counting)

```
struct semaphore {
      int count;
      queueType queue;
};
void semWait(semaphore s)
{
  s.count--;
  if (s.count < 0) {
   /* place this process in s.queue */;
   /* block this process */;
  }
}
void semSignal(semaphore s)
{
  s.count++;
  if (s.count <= 0) {
   /* remove a process P from s.queue */;
   /* place process P on ready list */;
  }
}
```

Semaphore (binary)

```
struct binary_semaphore {
        enum {zero, one} value;
        queueType queue;
};
```

```
void semWaitB(binary_semaphore s)
{
   if (s.value == one)
         s.value = zero;
   else {
      /* place this process in s.queue
      /* block this process */;
   }
}
```

```
void semSignalB(semaphore s)
{
   if (s.queue is empty())
         s.value = one;
   else {
      /* remove a process P from s.queue
      /* place process P on ready list */
   }
}
```

Producer / Consumer - Solution

```
/* program producerconsumer */
int n;
binary_semaphore s = 1, delay = 0;
void producer()
{
     while (true) {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)  {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Producer / Consumer - Solution

```
/* program  producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
     while (true) {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true) {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.11 A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

Mutual Exclusion

- To protect shared resources from race condition and data inconsistency. (Balance = Balance +/- Amount)

Thread 1	Thread 2	Shared Data X
A = X		100
	B = X	100
	B = B + 300	100
A = A + 200		100
X = A		300
	X = B	400

Thread - Issues

- When we have a number of threads accessing a shared resource
  - Multiple threads can enter critical section simultaneously
  - Critical section may be interrupted in the middle
- How can we make the critical section mutually exclusive!
  - i.e. only one thread can enter a critical section at any given instant
  - This would have the effect of the critical section being atomic!
- Solution
  - Locks

Mutex

- Library - #include <pthread.h>

- Structure
  - pthread_mutex_t

- Fuctions
  - pthread_mutex_init (mutex , attr)
  - pthread_mutex_destroy (mutex)

  - pthread_mutexattr_init (attr)
  - pthread_mutexattr_destroy (attr)

Locking and Unlocking Mutex

- Functions
  - pthread_mutex_lock (mutex)
  - pthread_mutex_unlock (mutex)
  - pthread_mutex_trylock (mutex)

Race Condition - Solution

```c
//For MUTEX LOCK()/UNLOCK()
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
//Shared Vairable
long balance = 0;//Shared, Global Variable
```

```c
void * credit(void * arg){
   int amount = (int) arg;
   printf("Credit : balance = balance + %d\n",(int)arg);
   for(long i=0;i<5000000;i++){
         pthread_mutex_lock(&mut);
         balance = balance + amount;
        pthread_mutex_unlock(&mut);
   }
   pthread_exit(NULL);
}
```

```c
void * debit(void * arg){
   int amount = (int) arg;
  printf("Dedit : balance = balance - %d\n",(int)arg);
   for(long i=0;i<5000000;i++){
         pthread_mutex_lock(&mut);
         balance = balance - amount;
        pthread_mutex_unlock(&mut);
    }
   pthread_exit(NULL);
}
```

```c
int main(){
   pthread_t t1, t2;
   pthread_create(&t1, NULL, credit,(void *)5);
   pthread_create(&t2, NULL, debit,(void *)5);

   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   printf("Value of balance is :%ld\n", balance);
   return 0;
}
```

```
Dedit : balance = balance - 5
Credit : balance = balance + 5
Value of balance is :0
```

Reading Assignment

- Types of locks and locking strategies
- Their pros and cons
- Two phase and Three phase locking

Locks

- Locks are constructs provided by the OS (or libraries)
- Locks can be ***acquired*** and ***released***
- With OS (and hardware) support, it is ensured that only ***one*** thread can acquire a given lock at any given time!
- We use locks to protect our critical sections in multiple threads

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

Locks (2)

- A lock is a variable which can be in any of two states:
  - Available (or free, or unlocked)
  - Acquired (or held, or locked)

- When a thread calls lock() on a particular lock variable
  - if the lock is in free state, it acquires the lock, and lock() function returns
  - If the lock is in held state, it will block and the lock() function will not return until it has acquired the lock

- A thread can free an acquired lock by calling unlock() on it.

Lock (3)

- Two approaches to locking
  - **Coarse-grained locking:** Having one big lock used when any critical section is accessed
  - **Fine-grained locking:** Having different data and data structures with different locks
- Pthread locks (mutexes)

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper; exits on failure
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

Evaluating Locks

- Goals:
  - Correctness - Achieve mutual exclusion
  - Fairness – Give chance to each waiting thread
  - Performance - Be efficient
    - Single thread single processor
    - Multiple threads single processor
    - Multiple threads multiple processors

Lock Implementation-Disable Interrupt

- First solution
  - Disable interrupts
  - CPU has special instructions for this
  - Works for single processor systems
- Issues
  - Turning off interrupts is a privileged operation
  - If interrupts are off, useful interrupts can be lost

```
void lock() {
    DisableInterrupts();
}
void unlock() {
    EnableInterrupts();
}
```

Lock Implementation – Use Flag

- Second solution
  - Use a variable (flag) to communicate b/w threads

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Lock Implementation – Use Flag

- Second Solution
  - Issues
    - Performance (spin waiting)
    - Correctness? No Mutual Exclusion

Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Figure 28.2: **Trace: No Mutual Exclusion**

Lock Implementation - (Use H/W Support)

- Accessing a shared flag might be interrupted hence mutual exclusion may not be possible
- Third solution – Use hardware support
  - test-and-set (atomic exchange) instruction
  - It provide an atomic instruction

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Lock Implementation - (Use H/W Support)

- Test-and-set spin lock

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

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

- Issues
  - Needs a preemptive scheduler on a single processor otherwise a thread may never relinquish the CPU

Evaluating spin locks

- Correctness
  - Yes it provides mutual exclusion
- Fairness
  - No, spin locks don’t provide fairness guarantee (a thread spinning may lock the CPU causing waiting threads to starve
- Performance
  - Worse performance if the thread holding the lock is preempted within a critical section
    - All other threads will wait spin wasting CPU cycles

Lock Implementation – (Use H/W Support)

- compare-and-swap instruction

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

Lock Implementation – (Use H/W Support)

- load-linked and store-conditional instruction
- Both instructions operate in tandem

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

Lock Implementation – (Use H/W Support)

- Store-conditional succeeds **only** if no intervening store has happened to that address since the last load-linked!!

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
    }
}

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

Lock Implementation – (Use H/W Support)

- fetch-and-add instruction
- Used to implement ticket lock

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

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

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

Figure 28.7: **Ticket Locks**

Lock Implementation – (Use H/W Support)

- All lock implementations using hardware support spin wait wasting CPU cycles
- We can avoid spin wait by process calling **yield** function to voluntarily give the CPU time to other waiting thread
- Requires support of the operating system, yield is a system call

Lock Implementation – Using yield

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Locks Implementation – Use Queues

- Use Queues

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}
```

```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```

- Sleep instead of spinning, if lock is held.
- Park() and unpark() support provided by Solaris for sleep

- *guard* is a spin lock around *flag* and *wait queues*
- some spinning is done but only for the time we access the flag & queue

- Flag is not set to 0 on unpark()!

Futexes

```
void mutex_lock (int *mutex) {
  int v;
  /* Bit 31 was clear, we got the mutex (the fastpath) */
  if (atomic_bit_test_set (mutex, 31) == 0)
    return;
  atomic_increment (mutex);
  while (1) {
      if (atomic_bit_test_set (mutex, 31) == 0) {
          atomic_decrement (mutex);
          return;
      }
      /* We have to waitFirst make sure the futex value
         we are monitoring is truly negative (locked). */
      v = *mutex;
      if (v >= 0)
        continue;
      futex_wait (mutex, v);
  }
}
```

Futexes

- MSB is the status
- All other bits keep count

```
void mutex_unlock (int *mutex) {
  /* Adding 0x80000000 to counter results in 0 if and
     only if there are not other interested threads */
  if (atomic_add_zero (mutex, 0x80000000))
    return;

  /* There are other threads waiting for this mutex,
     wake one of them up.  */
  futex_wake (mutex);
}
```

Two-phased locks

- If the lock his held, spin for a while to see if it’s acquirable in near future.
- If not, then go to sleep.

Summary

- We saw what are the different mechanisms of locks and how are they implemented
- We saw the role that hardware support provides in lock implementation
- We saw the different metrics on which a lock implementation is evaluated (correctness, fairness and performance)
- We also saw how spin locks waste CPU cycles and so alternate OS supported functions like yield and sleep are used to improved performance