6: Synchronization

Last Modified:
9/24/2002 9:54:32 AM

Concurrency is a good thing

- So far we have mostly been talking about constructs to enable concurrency
  - Multiple processes, inter-process communication
  - Multiple threads in a process
- Concurrency critical to using the hardware devices to full capacity
  - Always something that needs to be running on the CPU, using each device, etc.
- We don't want to restrict concurrency unless we absolutely have to

Restricting Concurrency

When might we *have* to restrict concurrency?

- Some resource so heavily utilized that no one is getting any benefit from their small piece
  - too many processes wanting to use the CPU (while (1) fork)
  - "thrashing"
  - Solution: Access control
- Two processes/threads we would like to execute concurrently are going to access the same data
  - One writing the data while the other is reading; two writing over top at the same time
  - Solution: Synchronization
  - Synchronization primitives enable SAFE concurrency

Correctness

- Two concurrent processes/threads must be able to execute correctly with *any* interleaving of their instructions
  - Scheduling is not under the control of the application writer
  - Note: instructions != line of code in high level programming language
- If two processes/threads are operating on completely independent data, then no problem
- If they share data, then application programmer may need to introduce synchronization primitives to safely coordinate their access to the shared data/resources
  - If shared data/resources are read only, then also no problem

Illustrate the problem

- Suppose we have multiple processes/threads sharing a database of bank account balances
- Consider the deposit and withdraw functions

```
int withdraw (int account, int amount) {
  balance = readBalance(account);
  balance = balance - amount;
  updateBalance(account, balance);
  return balance;
}
int deposit (int account, int amount) {
  balance = readBalance(account);
  balance = balance + amount;
  updateBalance(account, balance);
  return balance;
}
```

- What happens if multiple threads execute these functions for the same account at the same time?
  - Notice this is not read-only access

Example

- Balance starts at $500 and then two processes withdraw $100 at the same time
  - Two people at different ATMs; Update runs on the same back-end computer at the bank

```
int
withdraw (int account, int amount)
{
    balance = readBalance(account);
    balance = balance - amount;
    updateBalance(account, balance);
    return balance;
}
```

```
int
withdraw (int account, int amount)
{
    balance = readBalance(account);
    balance = balance - amount;
    updateBalance(account, balance);
    return balance;
}
```

- What could go wrong?
  - Different Interleavings => Different Final Balances !!!

$500 - $100 - $100 = $400

- If the second does readBalance before the second does writeBalance…….
- Two examples:

```
balance = readBalance(account);
```

```
balance = readBalance(account);
balance = balance - amount;
updateBalance(account, balance);
```

```
balance = balance - amount;
updateBalance(account, balance);
```

$500

$500

$400

```
balance = readBalance(account);
```

```
balance = readBalance(account);
```

```
balance = balance - amount;
updateBalance(account, balance);
```

```
balance = balance - amount;
updateBalance(account, balance);
```

- Before you get too happy, deposits can be lost just as easily!

Race condition

- When the correct output depends on the scheduling or relative timings of operations, you call that a race condition.
- Output is non-deterministic
- To prevent this we need mechanisms for controlling access to shared resources
  - Enforce determinism

Synchronization Required

- Synchronization required for all shared data structures like
  - Shared databases (like of account balances)
  - Global variables
  - Dynamically allocated structures (off the heap) like queues, lists, trees, etc.
  - OS data structures like the running queue, the process table, …
- What are not shared data structures?
  - Variables that are local to a procedure (on the stack)
  - Other bad things happen if try to share pointer to a variable that is local to a procedure

Critical Section Problem

- Model processes/threads as alternating between code that accesses shared data (critical section) and code that does not (remainder section)

```
do {
   ENTRY SECTION
        critical section
   EXIT SECTION
        remainder section
}
```

- ENTRY SECTION requests access to shared data ; EXIT SECTION notifies of completion of critical section

Solution to Critical Section Problem

- Mutual Exclusion
  - Only one process is allowed to be in its critical section at once
  - All other processes forced to wait on entry
  - When one process leaves, others may enter
- Progress
  - If process is in the critical section, it should not be able to stop another process from entering it
  - Decision of who will be next can't be delayed indefinitely
  - Can't just give one process access; can't deny access to everyone
- Bounded Waiting
  - After a process has made a request to enter its critical section, there should be a bound on the number of times other processes can enter their critical sections

Synchronization Primitives

- Synchronization Primitives are used to implement a solution to the critical section problem
- OS uses HW primitives (we've talked about these)
  - Disable Interrupts
  - HW Test and set
- OS exports primitives to user applications; User level can build more complex primitives from simpler OS primitives
  - Locks
  - Semaphores
  - Monitors
  - Messages

Locks

- Object with two simple operations: lock and unlock
- Threads use pairs of lock/unlock
  - Lock before entering a critical section
  - Unlock upon exiting a critical section
  - If another thread in their critical section, then lock will not return until the lock can be acquired
  - Between lock and unlock, a thread "holds" the lock

Withdraw revisited

```
int
withdraw (int account, int amount)
{
    lock(whichLock(acccount));      ENTER CRITICAL SECTION

    balance = readBalance(account);
    balance = balance - amount;     ENTER CRITICAL SECTION
    updateBalance(account, balance);

    unlock(whichLock(account));     EXIT CRITICAL SECTION

    return balance;
}
```

- What would happen if the programmer
  - forgot lock? No exclusive access
  - Forgot unlock? deadlock
  - put it at the wrong place?
  - called lock or unlock in both places?
- Consider the locking granularity? One lock or one lock per account?
- Is it ok for return to be outside the critical section?

$500 - $100 - $100 = $300

```
lock (whichLock(account));
balance = readBalance(account);
```

```
lock (whichLock(account));          BLOCKS!
```

```
balance = balance - amount;
updateBalance(account, balance);
unlock (whichLock(account));
```

```
balance = readBalance(account);     UNTIL GREEN UNLOCKS
balance = balance - amount;
updateBalance(account, balance);
unlock (whichLock(account));
```


Implementing Locks

- Ok so now we see that all is well *if* we have these objects called locks
- How do we implement locks?
  - Recall: The implementation of lock has a critical section too (read lock; if lock free, write lock taken)
- Need help from hardware
  - Make basic lock primitive atomic
    - Atomic instructions like test-and-set or read-modify-write, compare-and-swap
  - Prevent context switches
    - Disable/enable interrupts

Disable/enable interrupts

- Recall how the OS can implement lock as disable interrupts and unlock as enable interrupts
- Problems
  - Insufficient on a multiprocessor because only disable interrupts on the single processor
  - Cannot be used safely at user-level -not even exposed to user-level through some system call!
    - Once interrupts are disabled, there is no way for the OS to regain control until the user level process/thread yields voluntarily (or requests some OS service)

Test-and-set

- Suppose the CPU provides an atomic test-and-set instruction with semantics much like this:

```
bool test_and_set( bool *flag){
   bool oldValue = *flag;
  *flag = true;
  return old;
}
```

- Without an instruction like this, use multiple instructions (not atomic)

  load $register $mem vs. test-and-set $register $mem

  store 1 $mem

Implementing a lock with test-and-set

```
struct lock_t {
   bool held = FALSE;
}

void lock( lock_t *l){
   while (test_and_set(lock->held)){};
}

void unlock( lock_t *l){
   lock->held = FALSE;
}
```

- When call lock function, if the lock is not held (by someone else) then will swap FALSE for TRUE atomically!!! Test_and_set will return FALSE jumping out of the while loop with the lock held
- When call lock function, if the lock is held (by someone else) then will frantically swap TRUE for TRUE many times until other person calls unlock

Spinlocks

- The type of lock we saw on the last slide is called a spinlock
  - If try to lock and find already locked then will spin waiting for the lock to be released
- Very wasteful of CPU time!
  - Thread spinning still uses its full share of the CPU cycles waiting – called busy waiting
  - During that time, thread holding the lock cannot make progress!
  - What if thread waiting has higher priority than the threads holding the lock!!

Other choices?

- OS can choose between spinlocks and disable/enable interrupts
- At user level are we stuck with wasteful spinlocks?
  - No – can build higher level synchronization primitives and objects that avoid the constant spinning
  - Examples: semaphores and monitors

Semaphores

- Recall: the lock object has one data member the boolean value, held
- The semaphore object has two data members: an integer value and a queue of waiting processes/threads

Wait and Signal

- Recall: Locks are manipulated through two operations: lock and unlock
- Semaphores are manipulated through two operations: wait and signal
- Wait operation (like lock)
  - Decrements the semaphore's integer value and blocks the thread calling wait until the semaphore is available
  - Also called P() after the Dutch word, proberen, to test
- Signal operation (like unlock)
  - Increments the semaphore's integer value and if threads are blocked waiting, allow one to "enter" the semphore
  - Also called V() after the Dutch word, verhogen, to increment
- Why Dutch? Semaphores invented by Edgar Dykstra for the THE OS (strict layers) in 1968

Implementing a semaphore

```
struct semaphore_t {
   int value;
   queue waitingQueue;
}
void wait( semaphore_t *s){
   s->value--;
   if (s->value < 0){
    add self to s->waitingQueue
    block
   }
}
void signal( semaphore_t *s){
   s->value++;
   if (s->value <=0) {
        P =remove process from s->waitingQueue
        wakeup (P)
}
```

Whats wrong with this?

Implementing a semaphore with a lock

```
struct semaphore_t {
   int value;
   queue waitingQueue;
   lock_t l;
}
```

```
void wait( semaphore_t *s){
   lock(&s->l);
   s->value--;
   if (s->value < 0){
    add self to s->waitingQueue
    unlock(&s->l);
    block
   }
   unlock(&s->l);
}
void signal( semaphore_t *s){
   lock(&s->l);
   s->value++;
   if (s->value <=0) {
        P =remove process from s->waitingQueue
        wakeup (P)
    } else {
        unlock(&s-l);
   }
}
```


Avoiding busy-waiting?

- Threads block on the queue associated with the semaphore instead of busy waiting
- Busy waiting is not gone completely
  - When accessing the semaphore's critical section, thread holds the semaphore's lock and another process that tries to call wait or signal at the same time will busy wait
- Semaphore's critical section is normally much smaller than the critical section it is protecting so busy waiting is greatly minimized

Semaphore's value

- When value > 0, semaphore is "open"
  - Thread calling wait will continue (after decrementing value)
- When value <= 0, semaphore is "closed"
  - Thread calling wait will decrement value and block
- When value is negative, it tells how many threads are waiting on the semaphore
- What would a positive value say?

Binary vs Counting Semaphores

- Binary semaphore
  - Semaphore's value initialized to 1
  - Used to guarantee exclusive access to shared resource (functionally like a lock but without the busy waiting)
- Counting semaphore
  - Semaphore's value initialized to N >0
  - Used to control access to a resource with N interchangeable units available (Ex. N processors, N pianos, N copies of a book,…)
  - Allow threads to enter semaphore as long as sufficient resources are available

Pthread's Locks (Mutex)

- Create/destroy

```
int pthread_mutex_init (pthread_mutex_t *mut, const pthread_mutexattr_t
    *attr);
int pthread_mutex_destroy (pthread_mutex_t *mut);
```

- Lock

```
int pthread_mutex_lock (pthread_mutex_t *mut);
```

- Non-blocking Lock

```
int pthread_mutex_trylock (pthread_mutex_t *mut);
```

- Unlock

```
int pthread_mutex_unlock (pthread_mutex_t *mut);
```


Semaphores

- Not part of pthreads per se
  - #include <semaphore.h>
  - Support for use with pthreads varies (sometime if one thread blocks whole process does!)
- Create/destroy

```
int sem_init (sem_t *sem, int sharedBetweenProcesses , int initalValue);
Int sem_destory(sem_t *sem)
```

- Wait

```
int sem_wait (sem_t *sem)
int sem_trywait(sem_t * sem)
```

- Signal

```
int sem_post(sem_t *sem);
```

- Get value

```
int sem_getvalue(sem_t *, int * value);
```


Window's Locks (Mutex)

- Create/destroy

```
HANDLE CreateMutex(
        LPSECURITY_ATTRIBUTES lpsa,  // optional security attributes
        BOOL bInitialOwner          // TRUE if creator wants ownership
        LPTSTR lpszMutexName )      // object's name
BOOL    CloseHandle( hObject );
```

- Lock

```
DWORD WaitForSingleObject(
    HANDLE hObject,        // object to wait for
    DWORD dwMilliseconds );
```

- Unlock

```
BOOL ReleaseMutex(
        HANDLE hMutex );
```


Window's Locks (CriticalSection)

- Create/Destroy

```
VOID InitializeCriticalSection( LPCRITICAL_SECTION lpcs );
VOID DeleteCriticalSection( LPCRITICAL_SECTION lpcs );
```

- Lock

```
VOID EnterCriticalSection( LPCRITICAL_SECTION lpcs );
```

- Unlock

```
VOID LeaveCriticalSection( LPCRITICAL_SECTION lpcs );
```

Window's Semaphores

- Create

```
HANDLE CreateSemaphore(
        LPSECURITY_ATTRIBUTES lpsa,  // optional security attributes
        LONG lInitialCount,          // initial count (usually 0)
        LONG lMaxCount,              // maximum count (limits # of threads)
        LPTSTR lpszSemName );        // name of the                (may be NULL)
BOOL    CloseHandle( hObject );
```

- Lock

```
DWORD WaitForSingleObject(
    HANDLE hObject,        // object to wait for
    DWORD dwMilliseconds );
```

- Unlock

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore,
    LONG lRelease,         // amount to increment counter on release
                           // (usually 1)
    LPLONG lplPrevious );  // variable to receive the previous count
```


Sharing Window's Synchronization Objects

- Threads in the same process can share handle through a global variable
- Critical sections can only be used within the same process
  - Much faster though
- Handles to mutexes and semaphores can be shared across processes
  - One process creates another and the child inherits the handle (must specifically mark handle for inheritance)
  - Unrelated processes can share through DuplicateHandle function or OpenMutex or OpenSemaphore (based on knowledge of the name – like a shared file name)

Next time

- Other synchronization primitives
- Using synchronization primitives to solve some classic synchronization problems