7: Synchronization

Last Modified:
10/8/2002 9:37:06 PM

Last time

- Need for synchronization primitives
- Locks and building locks from HW primitives
- Semaphores and building semaphores from locks

Uses of Semaphores

- Mutual exclusion
  - Binary semaphores (wait/signal used just like lock/unlock)
  - "hold"
- Managing N copies of a resource
  - Counting semaphores
  - "enter"
- Anything else?
  - Another type of synchronization is to express ordering/scheduling constraints
  - "Don't allow x to proceed until after y"

Semaphores for expressing ordering

- Initialize semaphore value to 0
- Code:

P_i	P_j
⋮	⋮
A	*wait*
signal	*B*

- Execute *B* in P_j only after *A* executed in P_i
- *Note: If signal executes first, wait will find it is an signaled state (history!)*

Window's Events and UNIX Signals

- Window's Events
  - Synchronization objects used somewhat like semaphores when they are used for ordering/scheduling constraints
  - One process/thread can wait for an event to be signaled by another process/thread
- Recall: UNIX signals
  - Kill = send signal; Signal = catch signal
  - Many system defined but also signals left to user definition
  - Can be used for synchronization
    - Signal handler sets a flag
    - Main thread polls on the value of the flag
    - Busy wait though

Window's Events

- Create/destroy

```
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpsa, // security privileges (default = NULL)
    BOOL bManualReset,        // TRUE if event must be reset manually
    BOOL bInitialState,       // TRUE to create event in signaled state
    LPTSTR lpszEventName );   // name of event (may be NULL)
BOOL    CloseHandle( hObject );
```

- Wait

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

- Signal (all threads that wait on it receive)

```
BOOL SetEvent( HANDLE hEvent );  //signal on
BOOL ResetEvent( HANDLE hEvent ); //signal off
```

Generalize to Messaging

- Synchronization based on data transfer (atomic) across a channel
- In general, messages can be used to express ordering/scheduling constraints
  - Wait for message before do X
  - Send message = signal
- Direct extension to distributed systems

Problems with Semaphores

- There is no syntactic connection between the semaphore (or lock or event) and the shared data/resources it is protecting
  - Thus the "meaning" of the semaphore is defined by the programmer's use of it
    - Bad software engineering
      - Semaphores basically global variables accessed by all threads
    - Easy for programmers to make mistakes
- Also no separation between use for mutual exclusion and use for resource management and use for expressing ordering/scheduling constraints

Programming Language Support

- Add programming language support for synchronization
  - Declare a section of code to require mutually exclusive access (like Java's synchronized)
  - Associate the shared data itself with the locking automatically
- Monitor = programming language support to enforce synchronization
  - Mutual exclusion code added by the compiler!

Monitors

- A monitor is a software module that encapsulates:
  - Shared data structures
  - Procedures that operated on them
  - Synchronization required of processes that invoke these procedures
- Like a public/private data interface prevents access to private data members; Monitors prevent unsynchronized access to shared data structures

Example: bankAccount

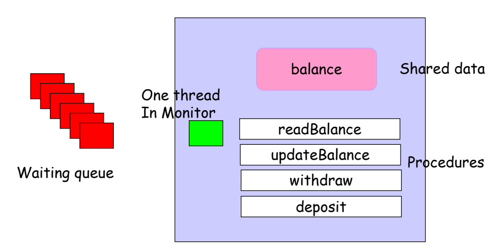
Locking added by the compiler!

```
Monitor bankAccount{
        int balance;

        int readBalance( ){return balance};
        void upateBalance(int newBalance){
                balance = newBalance;
        }
        int withdraw (int amount) {

                balance = balance - amount;
                return balance;
        }
        int deposit (int amount){
                balance = balance + amount;
                return balance;
        }
}
```


Monitor



Waiting Inside a Monitors

- What if you need to wait for an event within one of the procedures of a monitor?
- Monitors as we have seen to this point enforce mutual exclusion – what about the
- Introduce another synchronization object, the condition variable
- Within the monitor declare a condition variable:

  condition x;

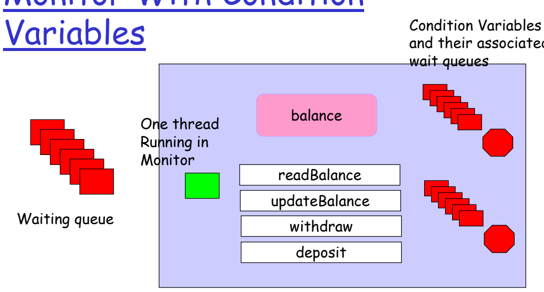
Wait and signal

- Condition variables, like semaphores, have the two operations have the two operations, wait and signal.
  - The operation **x.wait()** means that the process invoking this operation is suspended until another process invokes **x.signal();**
  - The operation wait allows another process to enter the monitor (or no one could ever call signal!)
  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect

Semaphores vs Condition Variables

- I'd like to be able to say that condition variables are just like semaphores but …
- With condition variables, if no process is suspended then the signal operation has no effect
- With semaphores, signal increments the value regardless of whether any process is waiting
- Semaphores have "history" (they remember signals) while condition variables have no history

Monitor With Condition Variables



Condition Variable Alone?

- Could you use a condition variable concept outside of monitors?
- Yes, basically a semaphore without history
  - Couldn't do locking with it because no mutual exclusion on its own
  - Couldn't do resource management (counting semaphore) because no value/history
  - Could you use it for ordering/scheduling constraints? Yes but with different semantics

Condition Variables for ordering/scheduling

- Code:

P_i	P_j
⋮	⋮
A	*wait*
signal	*B*

- Execute *B* in P_j only after *A* executed in P_i
- *If signal first, it is lost; wait will block until next signal (no history!)*

Pseudo-Monitors

- Monitor = a lock (implied/added by compiler) for mutual exclusion PLUS zero or more condition variables to express ordering constraints
- What if we wanted to have monitor without programming language support?
  - Declare locks and then associate condition variables with a lock
  - If wait on the condition variable, then release the lock

Pthread's Condition Variables

- Create/destroy

int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_destroy (pthread_cond_t *cond);

- Wait

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mut);

- Timed Wait

int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mut, const struct timespec *abstime);

- Signal

int pthread_cond_signal (pthread_cond_t *cond);

- Broadcast

int pthread_cond_broadcast (pthread_cond_t *cond);

Example: Pseudo-monitors

```
pthread_mutex_t monitorLock;
pthread_cond_t conditionVar;

void pseudoMonitorProc(void)
{
   pthread_mutex_lock(&mutexLock);
   …..

   pthread_cond_wait(&conditionVar, &monitorLock);
   ….

   pthread_mutex_unlock(&mutexLock);
}
```


More about monitors

Monitor Invariants

- Can specify invariants that should hold whenever no thread is in the monitor
- Not checked by compiler
- More like a precondition to be respected by the programmer

Who first?

- If thread in Monitor calls x.signal waking another thread then who is running in the monitor now? (Can't both be running in the monitor!)
- Hoare monitors
  - Run awakened thread next; signaler blocks
- Mesa monitors
  - Waiter is made ready; signaler continues

Does it matter? Yes

- If waiter runs immediately, then clearly "condition" being signaled still holds
  - Signaler must restore any "monitor invariants" before signaling
- If waiter runs later, then when waiter finally enters monitor must recheck condition before executing
  - Signaler need not restore any "monitor invariants" before signaling upon exiting

Write different code as a result

- If waiter runs immediately then

```
if (condition not true)
    C.wait()
```

- If waiter runs later then

```
while (condition not true)
    C.wait()
```

- Conclusion?
  - Mesa style (waiter runs later) has fewer context switches and directly supports a broadcast primitive (I.e. c.signalAll)
  - While instead of if not a big price to pay

Semaphores vs Monitors

- If have one you can implement the other…

Implementing Semaphores With Monitors

```
Monitor semaphore {

   int value;
   conditionVariable_t waitQueue;

    void setValue(int value){
         value = newValue;
   }

   int getValue(){return value;}
```

```
void wait(){
          value--;
          while (value < 0){
               //Notice Mesa
   semantics

   condWait(&waitQueue);
          }


   }

void signal (){
   value++;
   condSignal(&waitQueue);
}


} //end monitor semaphore
```


Implementing Monitors with Semaphores

```
semaphore_t mutex, next;
int nextCount = 1;

Initialization code:

mutex.value = 1;
next.value = 0;



For each procedure P in Monitor,
implement P as

Wait (mutex);
unsynchronizedBodyOfP();
if (nextCount >0){
   signal(next);
}else {
   signal(mutex);
}
```

```
conditionVariable_t {
    int count;
    semaphore_t sem;
}
condWait (conditionVariable_t *x) {
    //one more waiting on this cond
    x->count = x_count++;
    //wake up someone
    if (nextCount > 0){
          signal(next);
    }else {
          signal (mutex);
    }
    wait(x->sem);
    x->count = x->count--;
}
condSignal(conditionVariable_t *x){
    //if no one waiting do nothing!
    if (x->count > 0){
          next_count = nextCount++;
          signal(x->sem);
          wait (next);
          nextCount--;
    }
}
```


Software Synchronization Primitives Summary

- Locks
  - Simple semantics, often close to HW primitives, often inefficient
  - Used to build other primitives
- Semaphores
  - More efficient
  - Simple primitives, surprisingly difficult to program correctly with
- Events/Messages
  - Simple model of synchronization via data sent over a channel
- Monitors
  - Language constructs that automate the locking
  - Easy to program with where supported and where model fits the task

Adaptive Locking in Solaris

- Adaptive mutexes
  - Multiprocessor system if can't get lock
    - And thread with lock is not running, then sleep
    - And thread with lock is running, spin wait
  - Uniprocessor if can't get lock
    - Immediately sleep (no hope for lock to be released while you are running)
- Programmers choose adaptive mutexes for short code segments and semaphores or condition variables for longer ones
- Blocked threads placed on separate queue for desired object
  - Thread to gain access next chosen by priority and priority inversion is implemented

Conclusion?

- Synchronization primitives all boil down to representing shared state (possibly large) with a small amount of shared state
- All need to be built on top of HW support
- Once have one kind, can usually get to other kinds
- Which one you use is a matter of programmatic simplicity (matching primitive to the problem) and taste

Next time

- Classic synchronization problems and their solutions
  - Bounded Buffer
  - Readers/Writers
  - Dining Philosophers