Synchronization Mechanisms in Nachos

Sorin Manolache

sorma@ida.liu.se

Outline

• Nachos
• Resources
• Setting up the lab
• Synchronization mechanisms
Nachos

- Not Another Completely Heuristic Operating System
- It is a program that emulates an operating system for educational purposes
- Allows for the inspection and modification of the internals of an OS

Resources

- Main resources are listed on the course web page
  http://www.ida.liu.se/~TTIT61/info/litteratur.sv.shtml
- “A Roadmap through Nachos” is a must
Setting up the Lab

~$ gzip -dc /sw/nachos/nachida.tar.gz | tar xf -

The default compiler does not work. Therefore load the old one:

~$ module unload prog/gcc/2.95.2
~$ module load prog/gcc/2.8.1

In the first assignment you will work with the files in the code/threads subdirectory.

Mutual Exclusion (1)

Look in Silberschatz and Galvin, Chapter VI.

UnboundedBuffer b;

b.elems[b.crtPos] = getThreadId(); /* 1 */
b.crtPos++; /* 2 */

| Before execution | A, B | A, B? |
|------------------|--|--|---|
Mutual Exclusion (2)

UnboundedBuffer b;

/* Ensure by some mechanism that no switching of threads may occur while executing the next two instructions */
/* Enter the critical section */
b.elems[b.crtPos] = getThreadId(); /* 1 */
b.crtPos++;
/* 2 */
/* until here */
/* Leave the critical section */

Semaphores

• Semaphores
  – Have an internal counter indicating the number of occupants of the critical section
  – One can operate on the counter only by means of the operations P and V
  – P: if the counter is 0 block, else decrement and return
  – V: increment the counter, if the counter was 0, unblock one thread that was blocked on this semaphore
  – A semaphore’s counter is initialized upon creation with the maximum number of threads that are allowed in the critical section at the same time
  – Look at the implementation in code/threads/synch.cc
Locks

- Locks
  - Simpler than semaphores
  - Have two operations, `Acquire()` and `Release()`
  - `Acquire`: If the lock is not taken, take it and return, else block
  - `Release`: Release the lock. Make sure somehow that one of the blocked threads can somehow proceed.
  - You will have to implement it. Edit the file `code/threads/synch.cc`
  - Hint: Binary semaphore maybe?

Monitors

- Imagine that the buffer in our example is bounded
- In order to operate on it, we need exclusive access, so we protect it with an exclusion mechanism (lock)
- Once “inside”, we could get in the situation that the buffer is full and we cannot append to it, or that the buffer is empty and we cannot extract elements from it
- Because we are “inside” nobody may enter and change the condition we blocked on (nobody can extract if it was full, or append if it was empty) and then we would deadlock
- Therefore, we have to exit the critical section and to queue again at its entrance, giving a chance to some other thread to change the unfortunate condition
• Naive implementation:

```c
int done = 0;
do {
    lock.Acquire();
    if (successful())
        done = 1;
    lock.Release();
} while (!done);
```

- It's not efficient. Why should we try to enter again if it could be the case that nobody changed the condition?
- We would like to enter only when we would have a good chance to be successful
- The solution is to wait in a special queue, a queue dedicated to the condition we are waiting to be true

---

### Monitors

- Lock
- Condition variables
- Critical section
- Monitor
Condition Variables

- Two operations, \texttt{Wait(Lock *)} and \texttt{Signal(Lock *)}
- \texttt{Wait} releases the lock got as an argument (the thread exits the monitor the condition variable belongs to) and blocks (\texttt{Wait} is called when the condition was not true) until signalled, then re-acquires the lock (re-enters the monitor)
- \texttt{Signal} just wakes up one of the threads waiting on the condition. \texttt{Signal} is called by the thread that changes a condition.
- Additionally, there is the \texttt{Broadcast} operation, that wakes up all threads waiting on a condition
- You will have to implement these operations. Edit \texttt{code/threads/synch.cc}

Bounded Buffer

- Implement a synchronized bounded buffer class, with two methods, \texttt{Put} and \texttt{Get}
- Write two threads, a producer and a consumer. They will continuously attempt to append/remove elements to/from the bounded buffer.
- Spawn several of these threads
- Get inspiration from \texttt{code/threads/synchlist.*} and \texttt{code/threads/threadtest.cc}
Summary

- Implementation of the unimplemented methods of Lock and Condition
- Implementation of the BoundedBuffer class
- Demo