First half

Process synchronization

Homework Questions

Q: Regarding Q3, do we have to store arguments in separate mem

- Can have global variables.

Q: Can we have helper functions

- Yes you can have helper functions

File system get bytes into this mem

Prog (binary) -> prog.c

homework

Stack

fixed addr (3/4k block)

homework.c

Stacks come into play only in Q4.
Byte by byte retrieval is better fgte(); returns integer value

Last lecture overview

- Mutex - At times we don't want 2 threads access the same variable. Want to do some atomic operations.

In order to meet this is using Mutex lock:
- disable interrupts ? low-level way to
- spin lock provide mutual exclusion.

In practice these mechanisms are building blocks for larger synchronizing methods.
mutex object
A single level mutex build out of lower level components

spinlock
lock
owner
ptr to thread
queue
queue for threads

impl lock()
spinlock in.lock
if owner == null
    owner = me
else
    add me to queue
    temp = owner
    spinlock in.lock
    if temp != me
        sleep;

unlock()
spinlock in.lock
if q is empty
    owner = null
else
    owner = pop(queue)
    wake owner
    spin, unlock(in.lock)

Never access shared variables, outside the threads.
Each thread takes turn in accessing the thread
Classic Synchronisation Problems.

**Bounded Buffer**

- **writer()**: put() -> buffer -> get() [read()]
- If buffer is full in write case, block until there is free space.
- If buffer is empty, wait/block until some one writes.
- In case of single writer and reader, we can use circular buffer.
  - **read_ptr** - points to the tail
  - **write_ptr** - points to the head
  - Hence neither reader or writer points to the same data at any time.

```c
int head, tail
item buf[N]
put(): while (head - 1) mod N = tail
    do nothing
    writer[head] = item
    head = (head + 1) mod N
```

For multiple readers and writers, this is not appropriate.

**Semaphore**

A semaphore is an object like mutex provided by language environment.

- It has a value
- and functions: wait(), p(), lock(), down, signal(), v(), unlock(), up
Semaphore $S = \text{new Sem}(3) \Rightarrow \text{will allow } 3 \text{ of accesses without blocking.}$

Mutex is a special case of semaphore where $n = 1$.

This is a perfect fit for a bounded buffer problem.

Bounded Buffer problem using Semaphore

Mutex $m$

Semaphore spaces($N$), values(0)

Get: \begin{align*}
\text{wait (values)} \\
\text{lock (m)} \\
\text{val = buf [tail]} \\
\text{tail = tail + 1} \\
\text{unlock (m)} \\
\text{signal (spaces)}
\end{align*}

Put: \begin{align*}
\text{wait (spaces)} \\
\text{lock (m)} \\
\text{but [head] = val} \\
\text{head = (head+1) mod N} \\
\text{unlock (m)} \\
\text{signal (values)}
\end{align*}

values = 0, spaces = 3

get() \begin{align*}
\text{wait (values)} \\
\text{2 \rightarrow wait (spaces)} \\
\text{signal (spaces)} \\
\text{2 \leftarrow signal (values)} \\
\text{3 \rightarrow available space/ values.}
\end{align*}

N = 3

Spinlock is an OS const, or a mutex

Mutex are being set within a thread, but in Semaphores, there is explicit messaging of
Deadlocks

\[
\text{thr1} \xrightarrow{\text{waiting}} \text{thr2} \\
\text{lock(A)} \xleftarrow{\text{waiting}} \text{lock(B)} \\
\text{lock(B)} \quad \text{lock(A)}
\]

\[
\text{unlock(B)} \\
\text{unlock(A)}
\]

Lock ranking

Take all the locks that you are using and put an order to them. If you are acquired a lock and trying to acquire a lock at lower level, this action is not allowed.

Reader/Writer problem

Solution to this is often provided as a library function:

\[
\text{read - lock/unlock} \\
\text{write - lock/unlock}
\]

If the resource is in reading state and a thread requests for a write, it has to be blocked.

Implementing locking method

mutex m, int count, Sem wlock(1)

\[
\text{RL: lock(m)} \\
\text{if count = 0} \\
\text{wait (wlock)} \\
\text{counter ++} \\
\text{unlock(m)}
\]

\[
\text{RUL: lock(m)} \\
\text{count ++} \\
\text{if count = 0} \\
\text{signal (wlock)} \\
\text{unlock(m)}
\]

RL: wait (wlock)

\[
\text{WUL: signal (wlock)}
\]

\[
\text{don't sleep under a mutex}
\]

\[
\text{WUL: signal (wlock)}
\]
1) don't read shared variables outside of
   a mutex. Use a local copy instead.
2) Rant locks to avoid deadlock
3) don't sleep holding a mutex

Second half

Questions about HW:
Debugging: gdb homework
  gdb> run q1
  <segmentation error> gives addr where it crashed
  gdb> bt = back-trace cmd

How vector table works.

initialize vector table
with addr of print function

Monitors
Both mutex and semaphore are specific object.
Monitor is a user-defined class with some addition
properties, which are:

- implicit mutex on all methods
- condition variables - similar to mutex
- semaphore that has following operations:
  wait(c), signal(c), broadcast(c)
Wait blocks on conditional variable.
monitor M, condition C
A & wait(c) B & signal(c)
Wait atomically releases the mutex

09/23/09
Srividya Mandalee
Difference between a condition in Monitor and Semaphore is that, Semaphore saves the value throughout. But a condition variable doesn't if there is a signal() when no thread is waiting there is no effect.

Implementing Semaphore as monitor.

Monitor $S$:

```c
int count, condition C

sem_wait:  
  if count < 1
    wait(C)
  count --;
```

```
sem_signal:
  count ++;
  signal(C);
```

If $Sem = 2$:

- Sem_wait
  - 1
  - 0
  - Sem_signal
    - 1

Bound buffer, with bounded string buffer:

```c
put(val, len)  
get(len)  
```

Simple bounded buffer as a Monitor:

- Monitor bounded buffer
  - int count
  - queue buffer
  - condition readers
  - writes
put: if count = max
    wait (writers)
    add item to buff
    count ++
    signal (readers)

get: if count = 0
    wait (readers)
    remove
    count --
    signal (writers)

Variable length bounded buffer problem

put: while (max - count < len)
    wait (writers)
    add to buffer
    count = count + len
    signal (readers)

Monitors in practice (POSIX thread Monitors)

Both Java and Posix use Monitor Mechanism
In Posix: pthread_cond_t
monitor M
    condition C
    method C
        signal (c)
        wait (c)

Also java.util.concurrent.ReentrantLock
    behaves exactly like above.
Monitors in Java

Class M

- implicit condition ⇒ it is the object itself
- synchronized method
  - `wait()`
  - `notify()` ⇒ `signal()`
  - `notifyAll()` ⇒ `broadcast()`

Monitors in Java do not have multiple conditions.

Semaphore

- It is a fixed primitive type
- Has counting logic
- Release 1 thread at a time
  (With extra variables we can make semaphore behave like monitor but is complicated)
- Balanced signal/wait
  (It remembers the count)

Monitor

- User-defined type
- An augmented user-defined type
- User-defined logic
- Signal/broadcast

No history (signal or wait)
(There could be more signals than waits)
Rendezvous

Characteristics
Rendezvous R: has a single method
val = R.meet(x)

\[
\begin{array}{c}
\text{th1} \\
\text{meet}(x) \\
\text{return secondval}
\end{array}
\begin{array}{c}
\text{th2} \\
\text{meet}(y) \\
2nd.val = 3 \\
\text{signal c} \\
\text{return 10}
\end{array}
\]

\[
\begin{array}{c}
\text{y} \\
\leftarrow x
\end{array}
\]

It arranges for a meeting and as part of meeting the threads exchange d messages.

Implementing Rendezvous as Monitor

condition C
int waiting, int waiting
int firstval, second val

meet(x)
if waiting = 0
  waiting = 1
  firstval = x
  wait(c)
  return secondval
else
  second value = x
  signal c
  return firstval
  waiting = 0
Rendezvous using Semaphores

mutex m
int waiting, 1st_val, 2nd_val

meet(x):
    lock(m)
    if (!waiting)
        waiting = 1
        1st_val = x
        wait(S)
        temp = 2nd_val
    unlock(m)
    return temp
    else
        2nd_val = x
        signal(S)
        temp = 1st_val
    unlock(m)
    return temp