CS4410 Announcements

- Tutorial on Semaphores (Lects 9/17, 9/19) for students who attended Tape Conference. Any CS4410 student welcome to attend Sunday 9/29 3pm-4:30pm Hollister B14 (?)

- Assignment 2 hints
  * For problem 1:
    - Critical sections always terminate.
    - Non-critical sections (NCS) is not required to terminate and might not!

  * For problem 2:
    - Mutual exclusion protocols are expected to work for $N \geq 2$ processes.

- Looking ahead: Prelim 1 is Thurs 10/18 + 2 weeks likely review session 10/8 in class
Beyond Semaphores

Synchronization primitives
- Shared memory
- Semaphores
- Monitors vs. regions

Tension:
- Generality / expressiveness vs.
  - Usability / clarity
    - Avoiding errors!

Exploit:
- Syntax
- Compilers (\(=\) syntax checking)
- Run-time
foo: object

<table>
<thead>
<tr>
<th>Var</th>
<th>monitor vars</th>
</tr>
</thead>
</table>

allocate on heap

allocate on stack

op1: operation (p1, p2, ...)

<table>
<thead>
<tr>
<th>Var</th>
<th>locals</th>
</tr>
</thead>
</table>

| {}   | code   |

op2: operation (p1, p2, ...)

<table>
<thead>
<tr>
<th>Var</th>
<th>locals</th>
</tr>
</thead>
</table>

| {}   | code   |

...:

initialization:

schematic representation of monitor
Support for synchronization = mutex + Cond Synch

per operation

op. operation \((p_1, p_2, \ldots)\)

var locals

Require that \(B\) holds \((\ast)\) \(\text{assert} \, \text{locks} \, N\)...

end
BB:  Monitor
  var  buff [0..N] of portion
  sht:  0..N
  len:  0..N+1
  slots:  0..N+1

Insert:  operation (v: portion)
  need a slot!
  buff [sht+len] := v;
  len := len + 1;  slots := slots - 1

RMV:  operation (v: portion)
  need a portion!
  val := buff [sht];
  len := len - 1;  slots := slots + 1
  sht := sht + 1 mod N+1

begin
  sht := 0
  len := 0
  slots := 0
end
Condition variables

\[ \text{var } C : \text{ condition } \rightarrow \text{ associate } B_c \]

\[ \text{a. wait } \quad \text{(1) releases mutex block for Monitor} \]
\[ \text{(2) blocks process} \]

\[ \text{b. continue: exit monitor & unblock 1 process} \]
BB: \textbf{Monitor}

\begin{itemize}
  \item \textbf{var} buff [0..N] of porten
  \item stt: 0..N
  \item len: 0..N+1
  \item slots: 0..N+1
  \item space: Cond
  \item stuff: Cond
\end{itemize}

\textbf{Insert: Operation (w: porten)}
- need a slot! \[ \text{if slots} = 0 \text{ then space.out} \]
- buff [stt + len] := w
- len := len + 1
- slots := slots - 1
- stuff. continue

\textbf{RMV: Operation (w: porten)}
- need a porten! \[ \text{if len} = 0 \text{ then stuff. wait} \]
- val := buff [stt]
- len := len - 1
- slots := slots + 1
- stt := stt + 1 \mod N+1
- space. continue

\textbf{BEGIN}
- stt := 0
- len := 0
- slots := 0
\textbf{END BB}
Suppose signal operation does not cause output?

If \( A \) runs after \( B \) signal, then \( B \) \( \Rightarrow \) \( C \) \( \Rightarrow \) \( \lambda B c \) \( \Rightarrow \) \( \lambda A B c \) \( \Rightarrow \) \( \text{signal} \) \( \Rightarrow \) \( \text{signal} \)

Then \( B \) runs signal/urgent semantics is before new entry to monitor.
Implementation of signal/urgent semantics

- entry queue for monitor
- condition queue for each condition variable
- urgent queue for monitor

Monitor call:
- If monitor in use
  - then add thread to entry queue
  - else grant access
- e.wait: Put thread on queue for c
  - Invoke scheduler
- e.signal: Put thread on urgent queue
  - If condition queue for c not empty
    - then run head next
    - else invoke scheduler

Scheduler:
- Any thread on urgent queue?
  - Run it!
- Else any thread on entry queue?
  - Run it!
Implementation of signal/urgent semaphores with semaphores

\[ m: \text{monitor} \]
\[
\text{op: operation}(---)
\]
\[
\text{c.wait}
\]
\[
\text{c.signal}
\]
\[
\text{end}
\]

\[
\text{for mutex: semaphore init}(1)
\]
\[
\text{semc: semaphore init}(0)
\]
\[
\text{urgm: semaphore init}(0)
\]

\[
P(\text{mutex})
\]
\[
\begin{cases}
\text{c.count} := \text{c.count} + 1 \\
\text{if urgcount}_m > 0 \\
\quad \text{then urgcount}_m := \text{urgcount}_m + 1 \\
\quad \text{else} \quad V(\text{mutex}) \\
\end{cases}
\]
\[
P(\text{semc})
\]
\[
\begin{cases}
\text{urgcount}_m := \text{urgcount}_m + 1 \\
\text{if c.count > 0} \\
\quad \text{then} \quad \text{c.count} := \text{c.count} - 1 \\
\quad \text{else} \quad V(\text{semc}) \\
\quad V(\text{mutex})
\end{cases}
\]
\[
\begin{cases}
\text{if urgcount}_m > 0 \quad \text{then} \\
\quad V(\text{urgm}) \\
\text{else} \quad V(\text{mutex})
\end{cases}
\]
CS4410  Announcements 10/1

- Tutorial on this week's lectures (10/1, 10/3)
  Sunday 10/6  3pm-4:30pm
  Hollister B14

- Looking ahead: Prelim 1 is Thurs 10/10
  +1 week

  Likely review session 10/8 in class
Beyond Semaphores (con’t)

M: Monitor

\[ \text{var} \quad \text{... monitor vars...} \]
\[ C : \text{condition} \quad \in \mathbb{B} \]

op: operation \((p_1, p_2, \ldots)\)

\[ \text{var} \quad \text{loads} \]

\[ \text{end} \]

\[ \vdots \]

C. wait: thread suspended

C. continue: thread exits monitor & thread blocked on C

US

C. signal: thread suspends on "urgent" queue & thread blocked on C

\[ \text{run} \]
Acut: monitor

var ant: integer
incr: condition

\begin{align*}
\text{deposit: } & \text{ operate } (n: \text{ integer}) \\
\text{ant} & := \text{ant} + n - \text{incr. signal} \\
\text{end} & \\
\text{withdraw: } & \text{ operate } (w: \text{ integer}) \\
\text{while} \ \text{ant} < w & \text{ do incr. wait end} \\
\text{ant} & := w \\
\text{end} & \\
\text{end Acut} & 
\end{align*}
My notify
Suppose signal operation does not cause yield.
If (B) runs next...
\[ c \text{. wait} \]
\[ c \text{. notify} \]
Note: Actually, I need not hold because after notify because no process gets control at that point!
Implementation of signal-agent semantics

- entry queue for monitor
- condition queue for each condition variable
- urgent queue for monitor

Monitor call: If monitor in use
  - then add thread to entry queue
  - else grant access

C.wait: Put thread on queue for condition C
  - invoke scheduler

C.notify: 1 process on condition queue for C made runnable
  - continue executing in monitor

C.notifyAll: all processes on condition queue for C made runnable
  - continue executing in monitor

Scheduler: Pick some runnable thread
  - run it at most one process executes in monitor
Summary of signal regimes

C. continue: thread exits monitor

C. signal: thread suspends on "urgent" queue

C. notify: thread continues to exec in monitor

All cause thread suspended on C. wait to obtain monitor lock eventually

C. continue  \( \rightarrow \) immediately

C. signal  \( \rightarrow \) eventually

C. notify  \( \rightarrow \) eventually

Compare with P & V!
Use of notify is tricky.
Two processes can acquire when if instead of while.

```
monitor:
  var locked : boolean init false
  Q : condition

  Acquire: operation
            if locked then Q.wait
            while locked do Q.wait end
            locked := true
  end

  Release: operation
            locked := false
            Q.notify
  end

end monitor
```
Cautions when using monitors

I Nested locking

1. Call m.op

2. Call m_op

I would have to hold prior to call if we wanted to relax mutex for m (Bad idea!)

3. C.wait
Priority Inversion

A: high prio
B: med prio
C: low prio

C: lock mutex M
C yields processor to B
B: runs long computation

B: yields processor to A
A: attempts to lock mutex M
   (blocked due to C)
   • A yields processor to B
B: continues long computation
Deconstructing Monitors: (Critical) Regions

Yet another synchronization pinata.

(Good ref: Birrell, "An Introduction to Programming with Threads")

Monitor: \{ mutual exclusion (for ops) \}
        \{ cond synch (within ops) \}
        \{ selective mutes (due to visibility rules for vars) \}

- locks define regions

Syntax of a region:

```plaintext
var m : lock

region m do
    ...
end
```
Condition variables allow release of locks (wait, notify, notifyAll)

```java
var m: lock
space: condition with m
stuff: condition with m

region m do
while N-len = 0 do space.wait end
buf[(shl+len) mod N] := val
len := len + 1
stuff.signal
end

region m do
while len = 0 do stuff.wait end
val := buf[shl]
shl := (shl+1) mod N
len := len - 1
space.signal
end
```
associate sets of variables with each lock.

allows arbitrary fine-grain grouping of variables

. Problems if each var is associated with multiple locks.
Message Passing

send m to dest
receive m from source

Various design decisions
How to specify destination & source of msg for send & for receive

1. Direct naming: sender names receiver
   receiver names sender
   \(O(N^2)\) channels

   Problem - Processes need to know each other's names

2. Asymmetric direct naming: sender names receiver
   receiver names nobody
Synchrony

- **Blocky** ("synchronous"): causes primitive to delay until some event
- **Non-blocky** ("asynchronous"): primitive continues

**Blocky Send:**

Client:

- Send x,y,z to C1
- Receive val

Server:

- Receive a,b,c
- Send res to
Buffer capacity

How many sent but not received msgs allowed

0 - capacity
N - capacity

send as V?
receive as P.