Homework Notes

Debugging

gdb homework

gdb sing

<segmentation error>

By offsets, &s, or s, address, etc...

gdb br "backtrace" (only works if didn't completely crash)

Use &e in microcode.

Using vector debug:

```
```

![Diagram of vector debug process]
Monitor = User-defined class
    + implicit monitor on all methods
    + condition variable

operations:  wait(c)
             signal(c)
             broadcast(c)

and monitor M
condition C

A() \&
\text{wait}(c)
\text{lock}
\text{unlock}
\text{signal}
B() \&
\text{signal}(c)
\text{lock}
C() \&
\text{broadcast}(c)

Condition vars are basically for synchronization.

And provide framework I implement later yourself.
Use own vars to determine when to sleep/wake up.

As an exercise, implement a semaphore as a monitor.

monitor S:
    int count, condition C
    
    sem_wait:
        if count > 0
            \text{wait}(c)
            count--
        sem_signal:
            count++
            \text{signal}(c)

Note: Don't need explicit mutex to protect, because
it's implicit.
Bounded String Buffer

```c
put(val, len)

jet(len)

"some text"

jet(5) → "some"
```

Very simple with somephores. Store straight bounded of monitors:

2) simple bounded buffer problem

```c
monitor bounded buffer
int count

put: if count < max
    queue buffer
    add item to buf
    cond_t signal (reader)

jet: if count > 0
    cond_t remove
    count --
    cond_t signal (writer)
```

3) variable-length buffer

```c
while (max - count < len)
    cond_t wait (writer)

add to buffer:
    cond_t count = count + len

signal (reader)
```

Monitors in practice

Most high-level threadings implementations use monitors.

```java
new java.util.concurrent.ReentrantLock
```

```java
new java.lang.Object
```

POSIX threads

```java
new java.lang.concurrent.locks.ReentrantLock
```

Not:
We only get 1 implicit condition in this Java version.
**Semaphore**

- fixed type
- counting logic
- release 1 thread at a time
- boolean signal/wait

**Monitor**

- user-defined
- user-defined logic
- signal/broadcast
- no history (signal/wait), could have many signals per wait

Note that these are always ways of working around these restrictions. But seems to use the more powerful method.

Another synch method: 'Rendezvous'

rendezvous R:

\[ \text{rendezvous} \ R; \]

\[ \text{val} = \text{R.m} \text{eet}(x) \]

\[ \text{condition} \ C \]

\[ \text{if} \ \text{waiting} \ = \ 0 \]

\[ \text{waiting} = 1 \]

\[ \text{setc} = \text{val} \]

\[ \text{wait}(\cdot) \]

\[ \text{return} \ 2 \text{val} \]

\[ \text{else} \]

\[ \text{setc} = \text{val} \]

\[ \text{signal} \ C \]

\[ \text{return} \ 1 \text{val} \]

\[ \text{writing} = 0 \]

\[ \text{meet}(x) \]

\[ \text{rendezvous} \ R; \]

\[ \text{val} = \text{R.m} \text{eet}(x) \]

\[ \text{condition} \ C \]

\[ \text{if} \ \text{waiting} \ = \ 0 \]

\[ \text{waiting} = 1 \]

\[ \text{setc} = \text{val} \]

\[ \text{wait}(\cdot) \]

\[ \text{return} \ 2 \text{val} \]

\[ \text{else} \]

\[ \text{setc} = \text{val} \]

\[ \text{signal} \ C \]

\[ \text{return} \ 1 \text{val} \]

\[ \text{writing} = 0 \]
Simpl. rendezvous using semaphores:

Semaphore s(c)

int waiting, h1, h2, h3

wait(s);

if (!waiting)

waiting = 1
h1 = h2 = h3 = 1

if (h1) dump = 2*h1
unlock(h1)
return dump

else

2*h2 <= c

signal(h2)

h2 = h3

unlock(h2)

return h2