

Designing Concurrent Programs

- It's hard
 - where to start?
 - translation from pseudocode not always clear
 - tricky race conditions
 - seems to be ad hoc
- What's needed
 - systematic ways to write concurrent programs

Systematic Concurrent Program Design (Review)

- Concepts:
 - atomic actions
 - denoted by $\langle S \rangle$
 - means execute S atomically
 - await statements
 - allowed inside atomic actions
 - denoted by $\langle \text{await } (B) S \rangle$
 - means atomically: wait for B to be true, then execute S
 - if no await (i.e., just $\langle S \rangle$, we assume that B is “true”, i.e. $\langle \text{await } (\text{TRUE}) S \rangle$

Example -- Readers/Writers (Review)

- ReadEnter()
 <await (nw == 0) nr++>
- ReadExit()
 <nr-->
- WriteEnter()
 <await (nw == 0 and nr == 0) nw++>
- WriteExit()
 <nw-->

Advantages (Review)

- Whenever “worried” about race conditions
 - just put code inside an atomic action
- Don’t need to worry about ensuring threads are eligible to proceed past an await
 - this is done automatically

How to implement atomic actions and await statements?

- Use one single entry semaphore, e , for the whole program -- initialized to 1
- Consider each atomic action of form:
 $\langle \text{await } (B) \ S \ \rangle$
- Associate with each a counter db , a blocking semaphore b ; both initialized to 0
 - semaphore b will block threads when B is false
 - counter db will keep track of number of threads delayed on semaphore b

Translating $\langle \text{await } (B_1) S_1 \rangle$

```
P(e)           // gain mutual exclusion
if (!B1) {      // if B1 false, better block
    db1++;      // increase counter
    V(e);        // release mutual exclusion
    P(b1)       // block
}
S1           // now we execute S1
SIGNAL         // maybe others can wake up
```

Translating $\langle S_1 \rangle$

| | |
|----------------|----------------------------------|
| P(e) | // gain mutual exclusion |
| S ₁ | // now we execute S ₁ |
| SIGNAL | // maybe others can wake up |

What's SIGNAL?

- Suppose there are n different guards in atomic actions in the program
- Then, SIGNAL is:

if B_1 and $db_1 > 0$

db_1-- ; $V(b_1)$

else if B_2 and $db_2 > 0$

db_2-- ; $V(b_2)$


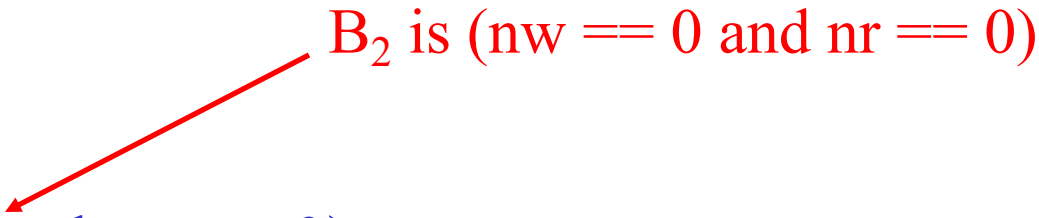
else if

else if B_n and $db_n > 0$

db_n-- ; $V(b_n)$

else $V(e)$

Example -- Readers/Writers

- ReadEnter()
 <await (nw == 0) nr++>
 B₁ is (nw == 0)
- ReadExit()
 <nr-->
- WriteEnter()
 <await (nw == 0 and nr == 0) nw++>
 B₂ is (nw == 0 and nr == 0)
- WriteExit()
 <nw-->

SIGNAL for Readers/Writers

```
if (nw == 0 and dr > 0)
```

```
    dr--; V(r)
```

```
else if (nw == 0 and nr == 0 and dw == 0)
```

```
    dw--; V(w)
```

```
else
```

```
    V(e)
```

Translating ReadEnter()

$\langle \text{await } (nw == 0) \text{ nr}++ \rangle$

P(e)

if (!(nw == 0)) {

 dr++;

 V(e)

 P(r);

}

nr++;

SIGNAL

SIGNAL can be often be optimized

- May be the case that
 - Some guards can (1) not possibly be true or (2) are always true
 - e.g., in Readers/Writers, SIGNAL can be optimized in each of the four functions

SIGNAL for Readers/Writers

```
if (nw == 0 and dr > 0)
```

```
    dr--; V(r)
```

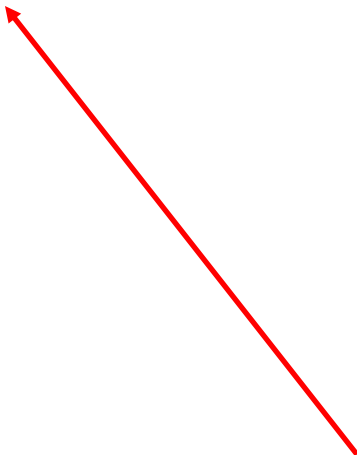
```
else if (nw == 0 and nr == 0 and dw == 0)
```

```
    dw--; V(w)
```

```
else
```

```
    V(e)
```

```
P(e)
if (!(nw == 0)) {
    dr++;
    V(e)
    P(r);
}
nr++;
SIGNAL
```



SIGNAL for Readers/Writers

Optimized for ReadEnter()

if (nw == 0 and dr > 0)

dr--; V(r)

nw must be zero at this point in ReadEnter

else if (nw == 0 and nr == 0 and dw == 0)

dw--; V(w)

else

V(e)

```
P(e)
if (!(nw == 0)) {
    dr++;
    V(e)
    P(r);
}
nr++;
SIGNAL
```

SIGNAL for Readers/Writers

Optimized for ReadEnter()

if (~~nw == 0~~ and dr > 0)

dr--; V(r)

nw must be zero at this point in ReadEnter

else if (nw == 0 and nr == 0 and dw == 0)

dw--; V(w)

else

V(e)

```
P(e)
if (!(nw == 0)) {
    dr++;
    V(e)
    P(r);
}
nr++;
SIGNAL
```

SIGNAL for Readers/Writers

Optimized for ReadEnter()

if (~~nw == 0~~ and dr > 0)

dr--; V(r)

nw must be zero at this point in ReadEnter

else if (nw == 0 and nr == 0 and dw == 0)

dw--; V(w)

nr cannot be zero at this point in ReadEnter

else

V(e)

```
P(e)
if (!(nw == 0)) {
    dr++;
    V(e)
    P(r);
}
nr++;
SIGNAL
```


SIGNAL for Readers/Writers

Optimized for ReadEnter()

~~if (nw == 0 and dr > 0)~~

~~dr--; V(r)~~

nw must be zero at this point in ReadEnter

~~else if (nw == 0 and nr == 0 and dw == 0)~~

~~dw--; V(w)~~

nr cannot be zero at this point in ReadEnter

~~else~~

~~V(e)~~

```
P(e)
if (!(nw == 0)) {
    dr++;
    V(e)
    P(r);
}
nr++;
SIGNAL
```

Final ReadEnter()

<await (nw == 0) nr++>

P(e)

if (!(nw == 0)) {

dr++;

V(e)

P(r);

}

nr++;

if (dr > 0)

dr--; V(r)

else

V(e)

← SIGNAL

This is the code that appears in ReadEnter in semrw.pdf

Practice: ReadExit(), WriteEnter()

Atomic Actions become “Passing the Baton” solution

- Advantages:
 - methodical
 - compiler could make transformation
 - passing the baton solutions are easy to modify to achieve different goals (e.g., who has preference, fairness, etc.)
- Disadvantages
 - solution overly general
 - can optimize by hand, but difficult for compiler