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 < S >
    - means execute S atomically
  - await statements
    - allowed inside atomic actions
    - denoted by < await (B) S >
    - means atomically: wait for B to be true, then execute S
    - if no await (i.e., just < S >, we assume that B is “true”, i.e. < await (TRUE) S >

Example -- Readers/Writers (Review)

- ReadEnter( )

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

- ReadExit( )

  <nr-->

- WriteEnter( )

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

- WriteExit( )

  <nw-->

Advantages (Review)

- To avoid concern 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 with <await (B) S>

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:

  < await (B) S >

- 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 <await (B_1) S_1>

```
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 <S_1>

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++>

B_1 is (nw == 0)

- ReadExit()

  <nr-->

B_2 is (nw == 0 and nr == 0)

- WriteEnter()

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

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

```
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)
else if (nw == 0 and nr == 0 and dw > 0)
  dw--; V(w)
else
  V(e)
```

nw must be zero at this point in ReadEnter

```
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)
else if (nw == 0 and nr == 0 and dw > 0)
  dw--; V(w)
else
  V(e)
```

nw must be zero at this point in ReadEnter

```
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)
else if (nw == 0 and nr == 0 and dw > 0)
  dw--; V(w)
else
  V(e)
```

nw must be zero at this point in ReadEnter

nr cannot be zero at this point in ReadEnter

```
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)
```

nw cannot be negative, so != 0 replaced with > 0

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