Monitor Concept

Attempts to overcome semaphore's awkwardness

Originally, a *language* idea

A monitor is:

- Shared data
- Set of **monitor procedures** that are the ONLY ONES allowed to access the data
- A lock (for implementing "monitor invariant" property)
- Language syntax for encapsulating all the above
- Two operations **wait**, **signal** (also sometimes a third: **broadcast**)

Monitor Invariant

Monitor invariant behavioral constraint: no more than one monitor procedure will "run" at any time

What *run* actually means is tricky—see below

Monitor invariant enforced by (1) compiler emitting code that uses lock to make each monitor procedure be a critical section, and by (2) *exiting the monitor correctly*

Example

At most one of `foo`, `bar` may run at any moment:

```
begin monitor;

    int var1;   // monitor data
    int var2;

    void foo {
        // statements accessing var1 and/or var2
    }

    void bar {
        // statements accessing var1 and/or var2
    }

end monitor;
```

Implementation uses a lock that is acquired/dropped before/after execution of either `foo` or `bar`

Monitor Operations

Wait similar to semaphore P,
Signal/Broadcast similar to semaphore V

- Wait—atomically releases lock then waits on condition. When thread re-awakens (due to some broadcast/signal), thread requests lock. Wait returns only after lock has been re-acquired. (This allows thread to enter monitor.)

- Broadcast—enables ALL waiting threads to run. When operation returns, lock is *still held*.

- Signal—an optimization of broadcast; enables ONE waiting thread to run (thread is picked according to some unspecified policy). When operation returns, lock is *still held*.

Example Implementations

Wait:

```
                    // these two are atomic so
drop lock           // that thread certain to be
wait on condition   // waiting if lock is dropped
    ...
<get scheduled>     // wait ends
    ...
get lock            // may block
return
```

Signal:

```
/* lock is held at this point */
<indicate to scheduler that some waiter may run>
return  /* lock still held when signal returns */
```

Monitor Entry/Exit

Block to get lock at beginning of each monitor procedure

Drop lock at end of each monitor procedure

Lock get/drop is IMPLICIT if monitor written in language that supports the concept (e.g., Java)

Lock get/drop is EXPLICIT if monitor written in language that does not support the concept (e.g., C)

Example

One possible execution involving 2 threads:

```
0. Thread A is in monitor
[thread A holds lock]
1. Thread A calls WAIT for some condition
[lock dropped, thread A waits for condition]
2. Thread B gets lock, enters monitor
[thread B holds lock, thread A waiting]
3. Thread B establishes condition that A is waiting for
[thread B holds lock]
4. Thread B executes SIGNAL on condition
[thread B holds lock]
5. Thread A is scheduled
[thread B holds lock]
6. Thread A, still in WAIT, tries to get lock & blocks
[thread B holds lock]
7. Thread B is scheduled
[thread B holds lock]
8. Thread B exits monitor, drops lock
[lock now unheld]
9. Thread A gets lock
[thread A holds lock]
10. Thread A returns from WAIT
[thread A holds lock]
```

Example Use

Structure of typical monitor procedure:

```
<implicit: get monitor lock>

if (conditionA NOT established)
    WAIT(conditionA);
    ...
establish condition B
    ...
SIGNAL(conditionB);

<implicit: drop monitor lock>

return;
```

Monitor Example: Producer/Consumer, I

```
begin monitor;
   /* condition variables */
   condition items;
   condition spaces;

   /* also: buffer variables */
```

Monitor Example: Producer/Consumer, II

```
void produce() {
   /* hidden action: get monitor lock */
   if (no space)
      WAIT(spaces);
   <produce>
   SIGNAL(items);
   /* hidden action: drop monitor lock */
   return;
}

void consume() {
   /* hidden action: get monitor lock */
   if (no items)
      WAIT(items);
   <consume>
   SIGNAL(spaces);
   /* hidden action: drop monitor lock */
   return;
}

end monitor;
```


Recall: Semaphore Solution for Producer/Consumer

Recall meaning & placement of the P and V operations:

```
/* producer */
<if (no space)                              // P(spaces)
     then wait until space available>
P(mutex);
<produce>
V(mutex);
<signal to threads waiting for items:  // V(items)
              there is another item>


/* consumer */
<if (no items)                              // P(items)
     then wait until items available>
P(mutex);
<consume>
V(mutex);
<signal to threads waiting for space:  // V(spaces)
              there is more space>
```


Monitor Example: Producer/Consumer, III

Monitor invariant—implemented by hidden lock—ensures critical section for `<produce>` and `<consume>`

Hidden get/drop lock actions performed by code automatically produced by compiler

Compiler knows what a monitor is because monitor is a language-level concept

Monitor Example: Producer/Consumer, IV

1. Monitor invariant (an application-INdependent condition) eliminates need for mutex semaphore

2. Application-dependent condition (i.e., buffer must never overflow/underflow) is enforced by placement of **wait** and **signal** similar to semaphore P/V

Compare monitor and semaphore solutions on Page 11 – logically, they are very similar

Monitor Example: Producer/Consumer, V

Q: Lock for monitor invariant is OUTSIDE wait/signal — when studying semaphores we saw this could cause deadlock; does monitor solution have the same problem?

A: No – because monitor WAIT operation drops then re-acquires lock!