8 Synchronization with Monitors

Monitors are a high-level data abstraction tool combining three features:

- Shared data.
- Operations on the data.
- Synchronization, scheduling.

They are especially convenient for synchronization involving lots of state. Compare monitors to modules and abstract data types.

Monitors are embedded in some concurrent programming languages. Java has monitors. In the style of C, a queue manipulation monitor might look like:

```
monitor QueueHandler;
    struct {
        int add, remove, buffer[200];
    } queue;

    void AddToQueue(int val)
    { – add val to end of queue – }

    int RemoveFromQueue()
    { – remove value from queue, return it – }

endmonitor;
```

There is one binary semaphore associated with each monitor, mutual exclusion is implicit: P on entry to any routine, V on exit.

Monitors are a higher-level concept than P and V. They are easier and safer to use.

Monitors need more facilities than just mutual exclusion. Need some way to wait.

- Busy-wait inside monitor?
- Put process to sleep inside monitor?

Condition variables: things to wait on.

- Wait(condition): release monitor lock, put process to sleep. When process wakes up again, re-acquire monitor lock immediately.
- Signal(condition): wake up one process waiting on the condition variable (FIFO). If nobody waiting, do nothing (no history).
- Broadcast(condition): wake up all processes waiting on the condition variable. If nobody waiting, do nothing.

There are several different variations on the wait/signal mechanism. They vary in terms of who gets the monitor lock after a signal. Our scheme is called "Mesa semantics" (signal-and-continue):

- On signal, signaller keeps monitor lock.
- Awakened process waits for monitor lock with no special priority (a new process could get in before it). This means that the thing you were waiting for could have come and gone: must check again and be prepared to sleep again if someone else took it.

Be careful when comparing to Monitors with "Hoare semantics" (signal-and-wait). It will make a difference in solving problems.

Show how wait and signal solve the queue implementation problem. They can be used to implement *any* scheduling mechanism at all. How do wait and signal compare to P and V? (No history)

Readers and writers problem with monitors: each synchronization operation gets encapsulated in a monitored procedure: *checkRead*, *checkWrite*, *doneRead*, *doneWrite*. Use conditions *OKToRead*, *OKToWrite*.

```
checkRead()
{
    while ((AW+WW) > 0) {
        WR += 1;
        wait(OKToRead);
        WR -= 1;
    }
    AR += 1;
}

doneRead()
{
    AR -= 1;
    if (AR==0 & WW>0) {
        signal(OKToWrite);
    }
}



checkWrite()
{
    while ((AW+AR) > 0) {
        WW += 1;
        wait(OKToWrite);
        WW -= 1;
    }
    AW += 1;
}

doneWrite()
{
    AW -= 1;
    if (WW > 0) {
        signal(OKToWrite);
    } else {
        broadcast(OKToRead);
    }
}
```

Why are **while**s needed above?

Could all of the *signal*s be *broadcast*s?

Result of this monitor example: have used one synchronization primitive (monitors) to build a more powerful higher-level synchronization primitive.

Summary:

- Probably the best implementation is in the Mesa language, which extends the simple model above with several additions to increase the flexibility and efficiency.
- Not present in very many languages, but still useful; Java has monitors; can simulate in languages like C (see Nachos locks and condition variables).
- Semaphores use a single structure for both exclusion and scheduling, monitors use different structures for each.
- Monitors enforce a style of programming where complex synchronization code doesn't get mixed with other code: it is separated and put in monitors.
- A mechanism similar to wait/signal is used internally in Unix for scheduling OS processes.