CS 537 Notes, Section #8: 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.

Existing implementations of monitors are embedded in programming languages. Best existing implementations are the Java programming language from Sun and the Mesa language from Xerox.

There is one binary semaphore associated with each monitor, mutual exclusion is implicit: P on entry to any routine, V on exit. This synchronization is automatically done by the compiler (i.e., makes automatic calls to the OS), and the programmer does not seem them. They come for free when the programmer declares a module to be a **monitor**.

Monitors are a higher-level concept than P and V. They are easier and safer to use, but less flexible, at least in raw form as above.

Probably the best implementation is in the Mesa language, which extends the simple model above with several additions to increase the flexibility and efficiency.

Do an example: implement a producer/consumer pair.

The "classic" Hoare-style monitor (**using C++ style syntax**):

```
class QueueHandler {

private:
    static int BUFFSIZE = 200;
    int first;
    int last;
    int buff[BUFFSIZE];
    condition full;
    condition empty;

    int ModIncr(int v) {
        return (v+1)%BUFFSIZE;
    }

public:
    void QueueHandler (int);
    void AddToQueue (int);
    int RemoveFromQueue ();
};
```

```
void
QueueHandler::QueueHandler (int val)
{
    first = last = 0;
}

void
QueueHandler::AddToQueue (int val) {
{
    while (ModIncr(last) == first) {
        full.wait();
    }
    buff[last] = val;
    last = ModIncr(last);
    empty.notify();
}

int
QueueHandler::RemoveFromQueue ();
{
    while (first == last) {
        empty.wait();
    }
    int ret = buff[first];
    first = ModIncr(first);
    full.notify();
    return ret;
}
```

Java only allows one condition variable (implicit) per object. Here is the same solution in Java:

```
class QueueHandler {

    final static int BUFFSIZE = 200;
    private int first;
    private int last;
    private int buff[BUFFSIZE];


    private int ModIncr(int v) {
        return (v+1)%BUFFSIZE;
    }

    public QueueHandler (int val)
    {
        first = last = 0;
    }

    public synchronized void AddToQueue (int val) {
    {
        while (ModIncr(last) == first) {
            try { wait(); }
            catch (InterruptedException e) {}
        }
        buff[last] = val;
```

```
            last = ModIncr(last);
            notify();
        }

        public synchronized int RemoveFromQueue ();
        {
            while (first == last) {
                try { wait(); }
                catch (InterruptedException e) {}
            }
            int ret = buff[first];
            first = ModIncr(first);
            notify();
            return ret;
        }
```

Condition variables: things to wait on. Two types: (1) classic Hoare/Mesa condition variables and (2) Java condition variables.

Hoare/Mesa condition variables:

- `condition.wait()`: release monitor lock, put process to sleep. When process wakes up again, re-acquire monitor lock immediately.
- `condition.notify()`: wake up one process waiting on the condition variable (FIFO). If nobody waiting, do nothing.
- `condition.broadcast()`: wake up all processes waiting on the condition variable. If nobody waiting, do nothing.

Java condition variables:

- `wait()`: release monitor lock on current object; put thread to sleep.
- `notify()`: wake up one process waiting on the condition; this process will try to reacquire the monitor lock.
- `notifyall()`: wake up all processes waiting on the condition; each process will try to reacquire the monitor lock. (Of course, only one at a time will acquire the lock.)

Show how wait and notify solve the semaphore implementation problem. Mention that they can be used to implement *any* scheduling mechanism at all. How do wait and notify compare to P and V?

Do the readers and writers problem with monitors.

Summary:

- Was not present in very many languages, but extremely useful. Java made monitors *much* more popular and well known.
- Semaphores use a single structure for both exclusion and scheduling, monitors use different structures for each.
- A mechanism similar to wait/notify is used internally to Unix for scheduling OS processes.
- Monitors are **more** than just a synchronization mechanism. Basing an operating system on them is an important decision about the structure of the entire system.