Mutual Exclusion using Monitors

Some programming languages, such as Concurrent Pascal, Modula-2 and Java provide mutual exclusion facilities called monitors.

They are similar to modules in languages that provide abstract data types in that:

- programmer defines a set of data types and procedures that can manipulate the data.
- procedures can be exported to other modules, which may import them.
- system invokes initialization routine before execution begins.

Monitors differ in that they support guard procedures. Java programmers can use the keyword synchronized to indicate methods of a class where only one method can execute at a time.

Guard procedures (synchronized methods) have the property that:

- only one process can execute a guard procedure at a time.
- When a process invokes a guard procedure, its execution is delayed until no other processes are executing a guard procedure (important)
Monitor Example with Java Class

Look at Java class where the keyword `synchronized` is used to indicate a “guard” procedure.

```java
public class Account {
    private int balance;

    public Account() {
        balance = 0; // initialize balance to zero
    }

    // use synchronized to prohibit concurrent access of balance
    public synchronized void Deposit(int deposit) {
        int newbalance; // local variable
        newbalance = balance + deposit;
        balance = newbalance;
    }

    public synchronized int GetBalance() {
        return balance; // return current balance
    }
}
```

Monitors are a higher-level, making parallel programming less-error prone than with semaphores.

Note, however, that they are implemented using a lower level facility provided by the hardware or operating system (such as semaphores).
Synchronization using Monitors

As described above, monitors solve the mutual exclusion problem. Monitors use conditions to solve the synchronization problem:

- new variable type called condition
- \texttt{wait(condition)} — blocks the current process until another process signals the condition
- \texttt{signal(condition)} — unblocks exactly one waiting process (does nothing if no processes are waiting)

Look at Fig. 2-27 as an example. Java provides \texttt{wait()}, \texttt{notify()}, and \texttt{notifyAll()}. However, Java only uses a single condition. Look at an example later.

Like semaphores, but no counters and do not accumulate signals. Must use own counters to keep track of states.

Problem:

- when does the blocked process continue?
- if immediately, we violate the invariant that only one process may execute a guard at any one time.
- if later, the condition being waiting on may no longer hold

Precise definitions vary in the literature. One solution:

- Suspend the signaling process.
- Process that issues a signal immediately exits the monitor. (Justification: most signals occur at end of guard anyway)
Other primitives: event counters, sequencers, path expressions

**Message Passing**

System calls for *direct* message passing between processes

\begin{verbatim}
send(destpid, &message)
\end{verbatim}

\begin{verbatim}
receive(srcpid, &message).
\end{verbatim}

srcpid can be **ANY** to receive from any destination.

Can also use indirect message passing where messages are sent to *mailboxes* or *ports*.

Design issues:

- buffering messages (mailbox) — allowed? how big?
- blocking or non-blocking operations. What to do if there is no buffer space on send. What to do if there is no message available on receive.
- Rendezvous? does the sender block until the receiver receives? Minix-style.
- fixed or variable sized messages
- synchronous vs. asynchronous reception. Only on receive or can a message handler be defined.

Look at Fig. 2-29.

**Barriers**

Multiple processes must synchronize before proceeding.

Look at Fig. 2-30.
**Summary**

Equivalence of primitives. Assignment is to build a message passing system on top of semaphores and shared memory.

Talked about:

- mutual exclusion—two activities competing for shared resource.
- synchronization—activity waiting on a condition (one process waiting on another’s completion).
- hybrid schemes—using both mutual exclusion and synchronization. Producer/Consumer problem with multiple producers and large buffer. Or complex locks using both spin locks and blocking if will wait too long.
- Methods—hardware techniques (interrupts) to operating system constructs (semaphores) to programming-language constructs (monitors).
- Facilities available—what language the operating system is written in, what facilities are offered by the operating system.

Want to avoid race conditions—timing dependent outcomes!