Concurrency and Mutual Exclusion

Readings: Chapter 5

Principles / Assumptions

- Interleaving and overlapping
  - Can be viewed as examples of concurrent processing
  - Both present the same problems
- Uniprocessor - the relative speed of execution of processes cannot be predicted
- Depends on,
  - Activities of other processes
  - The way the OS handles interrupts
  - Scheduling policies of the OS

Race Condition

- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
- The “loser” of the race is the process that updates last and will determine the final value of the variable

Design and management issues raised by the existence of concurrency:
- The OS must:
  - Be able to keep track of various processes
  - Allocate and de-allocate resources for each active process
  - Protect the data and physical resources of each process against unintended interference by other processes
  - The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes

Operating System Concerns
 Degree of Awareness | Relationship | Influence that One Process Has on the Other | Potential Control Problems |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processes unaware of each other</td>
<td>Competition</td>
<td>Results of one process independent of the action of others</td>
<td>Mutual exclusion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timing of process may be affected</td>
<td>Deadlock (renewable resource)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Starvation</td>
</tr>
<tr>
<td>Processes indirectly aware of each other (e.g., shared object)</td>
<td>Cooperation by sharing</td>
<td>Results of one process may depend on information obtained from others</td>
<td>Mutual exclusion</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timing of process may be affected</td>
<td>Deadlock (renewable resource)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Starvation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Data coherence</td>
</tr>
<tr>
<td>Processes directly aware of each other (have communication primitives available to them)</td>
<td>Cooperation by communication</td>
<td>Results of one process may depend on information obtained from others</td>
<td>Deadlock (consumable resource)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timing of process may be affected</td>
<td>Starvation</td>
</tr>
</tbody>
</table>

Table 5.2 Process Interaction

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

---

Mutual exclusion: Software Approaches

- Software approaches can be implemented for concurrent processes that execute on a single-processor or a multiprocessor machine with shared main memory

- These approaches usually assume elementary mutual exclusion at the memory access level

- Dijkstra reported an algorithm for mutual exclusion for two processes, designed by the Dutch mathematician Dekker

- Following Dijkstra, we develop the solution in stages

---

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.

---

Figure 5.1 Mutual Exclusion Attempts (page 1)

(a) First attempt

```c
/* PROCESS 0 */
while (turn = 0); /* do nothing */
/* critical section*/
turn = 1;
```

```
/* PROCESS 1 */
while (turn = 0); /* do nothing */
/* critical section*/
```

(b) Second attempt

```c
/* PROCESS 0 */
while (!flag()); /* do nothing */
/* critical section*/
flag() = false;
```

```
/* PROCESS 1 */
while (!flag()); /* do nothing */
/* critical section*/
flag() = false;
```

(c) Third attempt

```c
/* PROCESS 0 */
flag() = true;
while (!flag()); /* do nothing */
/* critical section*/
flag() = true;
flag() = false;
```

```
/* PROCESS 1 */
flag() = true;
while (!flag()); /* do nothing */
/* critical section*/
flag() = true;
flag() = false;
```

(d) Fourth attempt

```c
/* PROCESS 0 */
flag() = true;
while (flag()); /* delay */
/* critical section*/
flag() = true;
```

```
/* PROCESS 1 */
flag() = true;
while (flag()); /* delay */
/* critical section*/
flag() = true;
```

© 2017 Pearson Education, Inc., Hoboken, NJ. All rights reserved.
Mutual exclusion: Hardware approaches

- Interrupt disabling

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>Works with uniprocessors only</td>
</tr>
<tr>
<td></td>
<td>Degrades OS efficiency</td>
</tr>
</tbody>
</table>

- Atomic compare&swap operations

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Works with any number of processors</td>
<td>Busy waiting</td>
</tr>
<tr>
<td>Simple</td>
<td>Deadlocks</td>
</tr>
<tr>
<td>Can support multiple critical sections</td>
<td>Starvation</td>
</tr>
</tbody>
</table>
Semaphores

A variable that has an integer value upon which only three operations are defined:

- There is no way to inspect or manipulate semaphores other than these three operations

1) A semaphore may be initialized to a nonnegative integer value
2) The semWait operation decrements the semaphore value
3) The semSignal operation increments the semaphore value

Consequences

There is no way to know before a process decrements a semaphore whether it will block or not

There is no way to know which process will continue immediately on a uniprocessor system when two processes are running concurrently

You don’t know whether another process is waiting so the number of unblocked processes may be zero or one

struct semaphore {
    int count;
    queueType queue;
};
void semWait(semaphore s) {
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignal(semaphore s) {
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}

Figure 5.6 A Definition of Semaphore Primitives

struct binary semaphore {
    enum (zero, one) value;
    queueType queue;
};
void semWait(binary semaphore s) {
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignal(binary semaphore s) {
    if (s.value == one)
        s.value = one;
    else {
        /* place process P on ready list */;
    }
}

Figure 5.7 A Definition of Binary Semaphore Primitives
**Strong/Weak Semaphores**

- A queue is used to hold processes waiting on the semaphore

**Strong Semaphores**

- The process that has been blocked the longest is released from the queue first (FIFO)

**Weak Semaphores**

- The order in which processes are removed from the queue is not specified

---

```
/* program mutual_exclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
    while (true)
    {
        semWait(s);
        /* critical section */;
        semSignal(s);
        /* remainder */;
    }
} void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}
```

**Figure 5.9 Mutual Exclusion Using Semaphores**

---

**Figure 5.10 Processes Accessing Shared Data Protected by a Semaphore**

---

**Figure 5.8 Example of Semaphore Mechanism**
Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data

Monitor Characteristics

- Local data variables are accessible only by the monitor’s procedures and not by any external procedure
- Process enters monitor by invoking one of its procedures
- Only one process may be executing in the monitor at a time

Synchronization

- A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor
- Condition variables are a special data type in monitors which are operated on by two functions:
  - cwait(c): suspend execution of the calling process on condition c
  - csignal(c): resume execution of some process blocked after a cwait on the same condition

Figure 3.18 Structure of a Monitor