Concurrent Programming

Session 5: Shared Variable Programming

Computer Engineering Department
Iran University of Science and Technology
Tehran, Iran

Instructor: Hadi Salimi
Distributed Systems Lab.
Computer Engineering Department,
Iran University of Science and Technology,
hsalimi@iust.ac.ir

Definitions

- State of a Program
  - The state consists of the values of the program variables at a point of time.
  - Both implicit and explicit variables
- History (trace)
  - A particular execution of a program can be viewed as a history: S0 → S1 → S2 → … → SN
- Question?
  - Starting from an initial state, is it possible to make two different histories?
    - For both parallel and sequential program?

Histories

- About histories
  - Some are not valid at all
  - Some are not desirable
- The role of synchronization is to constrain the possible histories to desirable ones.

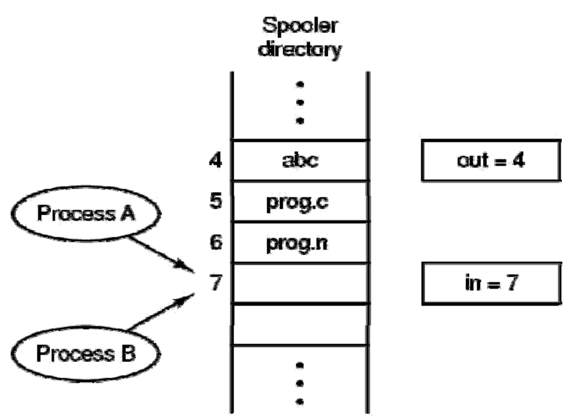
Program Properties

- A property is an attribute that is true of every possible history of that program.
- There are two kind of attributes:
  - Safety: the program never enters a bad state
  - Liveness: the program eventually enters a good state.
- Mutual Exclusion is an example of safety.
- Termination is an example of liveness.
- How one can go about safety or liveness satisfaction?

Critical Section

- From some viewpoint, a Critical Section is the implementation of an atomic action in software.
- Unlike hardware, the atomic structures should be constructed at software level (in which layer?).

Race Conditions



- Two processes want to access shared memory at same time. What happens if they try to access it simultaneously?

Race Conditions (cont.)

- Situations like this are called race conditions.
- What will happen if two processes execute the following code?

```
X=0;
…
Read(x);
X++;
Write(x);
```

Critical Sections

- We should prohibit more than one process from reading and writing the shared data at the same time.
- In other words, what we need is mutual exclusion.
- The part of the program where the shared memory is accessed is called the critical section or critical region.

Critical Sections

- Four properties should satisfied:
  - Mutual Exclusion
  - Absence of Deadlock
  - Absence of Unnecessary Delay
  - Eventual Termination
- Which one is a safety property?
- Which one is a liveness property?

Locks (an example)

```
#define FALSE  0
#define TRUE   1
#define N      2                          /* number of processes */

int turn;                                 /* whose turn is it? */
int interested[N];                        /* all values initially 0 (FALSE) */

void enter_region(int process);           /* process is 0 or 1 */
{
      int other;                          /* number of the other process */

      other = 1 – process;                /* the opposite of process */
      interested[process] = TRUE;         /* show that you are interested */
      turn = other ;                      /* set flag */
      while (turn == other && interested[other] == TRUE)     /* null statement */ ;
}

void leave_region(int process)            /* process: who is leaving */
{
      interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Locks?

- Is there any scalability problem with the mentioned solution?
- What's the solution?

Spin Locks

- In the case of an spin lock, all processes spin around a single lock variable
- The lock variable can be protected by means of hardware or software techniques.

```
enter_region:
    TSL REGISTER,LOCK                | copy lock to register and set lock to 1
    CMP REGISTER,#0                  | was lock zero?
    JNE enter_region                 | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered

leave_region:
    MOVE LOCK,#0                     | store a 0 in lock
    RET | return to caller
```

Sleep and Wakeup

- The described solutions requiring busy waiting.
- Is also can have unexpected effects like **priority inversion**:
  - There is two processes H and L.
  - H has higher priority than L.
  - L is in its critical section and H becomes ready.
  - What happens?

Semaphore

- In many problems there is a need to count an event, like producing an item or consuming it.
- Accessing to this counter should be protected against concurrent processes.
- Such a protected counter is called a semaphore which has more features.

Semaphore (cont.)

Two operators are defined on a semaphore: Down and Up (generalizations of sleep and wakeup)

```
Down(int& x){
If (x > 0)
    x--;
else
    Sleep() };
```

```
Up(int& x) {
If (there is any waiting process)
    Pick a process from queue and make it
    ready;
else
    x++ };
```

Semaphore (cont.)

- How to protect a critical section using semaphores?

```
int s = 1;
Down(s);
Critical Section
Up(s);
```

Semaphore (cont.)

Consider a resource which can be shared by 3 processes. How accessing this device can be protected using semaphores?

```
int x = 3;
Down(x);
…
Accessing the shared resource.
…
Up(x);
```

Monitors

- To make it easier to write correct programs, a higher level primitive called monitor is introduced.
- It is a collection of procedures, variables and data structures that are all grouped in a package.
- An important property:
  - Only one process can be active in a monitor at any time.

Monitors (cont.)

monitor *example*

int x;

procedure *producer(x)*

…

end;

procedure *consumer(x)*

…

end;

Monitors (cont.)

- Monitors are a programming language construct, so the compiler should handle calls to procedures.
- When a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active or not.