Interprocess Communication

Jonathan Windle

University of East Anglia

J.Windle@uea.ac.uk

June 9, 2017

Overview I

Cooperating Processes

- Processes may either be:
  - Independant
  - Cooperating
- Cooperating processes need an interprocess communication mechanism to exchange data
  - Shared memory
  - Message passing (i.e. signals)

Shared Memory IPC

- IPC using shared memory requires communicating processes to establish a region of shared memory
- Typically this resides in the address space of the process creating the sshared memory segment
- Processes are normally only able to access their own memory segment so this restriction must be relaxed.

Critical Sections

- To avoid race conditions the key is to find some way to prohibit more than one process from reading and writing the shared data at the same time i.e. we need mutual exclusion.
- Mutual Exclusion can be defined as some way of making sure that if one process is using the shared variable or file, the other process will be excluded from doing the same thing.
- If we arranged matters so that no two processes were ever in their critical sections at the same time, ,we could avoid a race condition.
- A solution must satisfy:
  1. Mutual exclusion (No two must be in their critical section)
  2. Progress: No process running outside its critical section may block another process.
  3. Bounded Waiting: No process should have to wait forever to enter its critical section.

Guaranteeing Mutual Exclusion

- The simplest solution is to have each process disable ALL interupts just after entering the critical section,and re-enable them just before leaving it.
  - Since a process can only be switched as the result of an interrupt the CPU cannot be switched to another process.
  - But this approach is unnatractive, since it is generally unwise to give processes the power to turn off interrupts.
- Initially, a single shares lock variable might appear to be a good idea but since it's a shared variable it's also susceptible to races.

Coroutines

```
int turn = 0;  // shared
```

```
while (TRUE) {
  while (turn != 0)
    ; // loop
  critical_section( );
  turn = 1;
  noncritical_region( );
}
```

```
while (TRUE) {
  while (turn != 1)
    ; // loop
  critical_section( );
  turn = 0;
  noncritical_region( );
}
```

- Mutual exclusion is guaranteed, but:
  - The processes must alternately access their critical sessions
  - If a process fails then the other process is hopelessly deadlocked i.e. the solution violates the no process outside the critical section can block another process.

Software Semaphores

```
Shared bool EinCS = FALSE;
Shared bool WinCS = FALSE;
```

```
void main( ) { /* EAST BOUND */
 while (TRUE) {
  while WinCS
   ; // loop
  EinCS = TRUE;
  critical_section( );
  EinCS = FALSE;
  noncritical_region( );
}
```

```
void main( ) { /* WEST BOUND */
 while (TRUE) {
  while EinCS
   ; // loop
  WinCS = TRUE;
  critical_section( );
  WinCS = FALSE;
  noncritical_region( );
}
```

- Overcomes problem of lockstep synchronization
- Program does not satisfy requirement for mutual exclusion:
  - East want CS and steps over while test
  - Assume East is unloaded from CPU and replaced by West
  - West wants CS and can enter CS because EinCS is still false.

Semaphores - Fixed

```
Shared bool EwantsCS = FALSE;
Shared bool WwantsCS = FALSE;
```

```
void main( ) { /* EAST BOUND */
  while (TRUE) {
    EwantsCS = TRUE;
    while WwantsCS
      ; // loop
    critical_section( );
    EwantsCS = FALSE;
    noncritical_region( );
}
```

```
void main( ) { /* WEST BOUND */
  while (TRUE) {
    WwantsCS = TRUE;
    while EwantsCS
      ; // loop
    critical_section( );
    WwantsCS = FALSE;
    noncritical_region( );
}
```

- Indicate desire to enter CS
- If EnterCS check passes, assume OK to proceed.
- Leads to deadlock... Let's hope semaphores don't come up... :D

Hardware Support

- The difficulty in software solutions for mutual exclusion was created by random interleaving of read and writes to memory
- If the process was unloaded from the CPU between the read and write instructions then we cannot guarantee mutual exclusion
- The problem can be eliminated if the hardware provides a memory access mechanism that allows the process to read a shared memory location, test it, and optionally change the value stored (depending on the result of the test)
  - These operations must be performed indivisibly within one instruction (bus cycle)
  - The instruction is called a `Test-and-Set-Lock` (TSL) or `Read-Modify-Write` (RMW).

Test-and-Set-Lock

- Many computers, especially those designed with multiple processors have a TSL instruction:
  - Syntax: TSL RX, LOCK; // RX = CPU register, LOCK = Memory location
- TSL reads the contents of the memory word LOCK into register RX and then stores a non-zero value at the memory address LOCK.
  - A key feature of the execution odTSL is that it locks the memory bus to prohibit other CPIs from aaccessing memory until it is done.
  - This also prevents the OS from unloading the process executing a TSL until it is done.
  - That is as one atomic, uninterruptable unit.

Using TSL

- We can use TSL to prevent two processes from simultaneously entering their critcal sections by implementing a simple LOCK.
- A process sets the LOCK before executing its critical section and releases the LOCK when it has left the critical section.

```
ENTER_CS  TSL    REGISTER, LOCK    ; copy LOCK to register and set LOCK to 1
          CMP    REGISTER #0       ; was LOCK 0?
          JNE    ENTER_CS          ; If it was non zero, LOCK was set, so loop
          RET                      ; return to caller, CS entered
```

```
LEAVE_CS  MOV    LOCK, #0          ; store 0 in LOCK
          RET                      ; return to caller
```

TSL-Summary

- The TSL instruction makes the solution to the critical section problem straightforward.
  - Process must call `ENTER_CS` and `LEAVE_CS` at appropriate times for the solution to work.
- There are some problems:
  - One process might block another indefinitely
  - The proces waits ina busy-wait loop (wastes CPU cycles)
  - Extending to N processes?

Summary

- Solutions to shared memory IPC use semaphores
- Two types of Semaphore
- General Semaphore:
  - Can take any non-negative integer value, often called a counting semaphore
  - Useful if a number of processes need access to a resource
- Binary Semaphore:
  - Value can either be 1 or 0
  - Often called a mutex (as used to enforce mutual exclusion)
- Solution:
  - Acccess shared variables withing critical section and ensure access to critical sections are atomic.
  - Software solution: Dekkers
  - Hardware solution: TSL/RMW instructions.

The End