CSE 120: Principles of Operating Systems

Lecture 4

Synchronization

October 7, 2003

Prof. Joe Pasquale
Department of Computer Science and Engineering
University of California, San Diego

© 2003 by Joseph Pasquale
Before We Begin ...

Read Chapter 7 (Process Synchronization)

Programming Assignment #1
  • Due Sunday, October 19, midnight
Synchronization

Synchronize: to (arrange events to) happen at same time

Process synchronization

- when one process has to wait for another
- points in each process occur “at the same time”

Uses of synchronization

- avoid race conditions
- wait for resources to become available
Example of a Race Condition

The Credit/Debit Problem

Process $P_0$

Credit (a)
int a;
{
    int b;
    b = ReadBalance ();
    b = b + a;
    WriteBalance (b);
    PrintReceipt (b);
}

Process $P_1$

Debit (a)
int a;
{
    int b;
    b = ReadBalance ();
    b = b - a;
    WriteBalance (b);
    PrintReceipt (b);
}
To Avoid Race Conditions

Identify critical sections

• sections of code executed by multiple processes
• must run *atomically* with respect to each other

Enforce *mutual exclusion*

• only one process active in a critical section
Four Rules for Mutual Exclusion

1. No two processes inside critical sections at same time
2. No process outside a critical section may block others
3. No process may wait forever to enter critical section
4. No assumptions about speeds or number of CPU’s
How to Achieve Mutual Exclusion?

Surround critical section with entry/exit code

< entry code >  < entry code >
< critical section >  < critical section >
< exit code >  < exit code >

Entry code should act as a barrier

• if another process is in their critical section, block
• otherwise, allow process to proceed

Exit code should act to release other entry barriers
Possible Solution: Software Lock?

Lock indicates whether any process is in critical section

```
shared int lock = OPEN;

P_0
while ( lock == CLOSED );
lock = CLOSED;
< critical section >
lock = OPEN;
```

```
P_1
while ( lock == CLOSED );
lock = CLOSED;
< critical section >
lock = OPEN;
```

Race condition in while loop (breaks Rule 1)
Possible Solution: Take Turns?

Alternate which process can enter critical section

```plaintext
shared int turn = P0;

P0
while ( turn != P0 );
< critical section >
turn = P1;

P1
while ( turn != P1 );
< critical section >
turn = P0;
```

Prevents entry even if OK to enter (breaks Rule 2)
Possible Solution: State Intention?

Each process states it wants to enter critical section

shared boolean flag[2] = {FALSE, FALSE};

\[P_0\]
flag\[P_0\] = TRUE;
while ( flag\[P_1\] );
\langle critical section \rangle
flag\[P_0\] = FALSE;

\[P_1\]
flag\[P_1\] = TRUE;
while ( flag\[P_0\] );
\langle critical section \rangle
flag\[P_1\] = FALSE;

Race condition: prevents entry forever (breaks Rule 3)
Peterson’s Solution

If there is competition, take turns; otherwise, enter

```
int turn;
shared boolean flag[2] = {FALSE, FALSE};

P0
flag[P0] = TRUE;
turn = P1;
while ( flag[P1] && turn == P1 );
< critical section >
flag[P0] = FALSE;

P1
flag[P1] = TRUE;
turn = P0;
while ( flag[P0] && turn == P0 );
< critical section >
flag[P1] = FALSE;
```

Works! Extends to n processes. Problem: busy-waiting
What about Turning Off Interrupts?

By turning off interrupts, context-switching can’t occur:

\[ P_0 \]
\begin{align*}
\text{InterruptsOff ();} \\
\langle \text{critical section} \rangle \\
\text{InterruptsOn ();}
\end{align*}

\[ P_1 \]
\begin{align*}
\text{InterruptsOff ();} \\
\langle \text{critical section} \rangle \\
\text{InterruptsOn ();}
\end{align*}

Too restrictive: locks out all processes, even those not in critical section (breaks Rule 2)

Doesn’t work on a multiprocessor (breaks Rule 4)
Hardware Solution: Test&Set Instr.

TSET mem: simultaneously test mem and set it to 1
  • \{ reg ← mem, mem ← 1, set cond code } atomically
  • C func: tset (lock) sets lock to 1, returns orig value

shared int lock = 0;

\[P_0\]
\[
\text{while ( tset (lock) == 1 );}
\]
< critical section >
\[
\text{lock = 0;}
\]

\[P_1\]
\[
\text{while ( tset (lock) == 1 );}
\]
< critical section >
\[
\text{lock = 0;}
\]

Simple, works for n processes (but still busy-waits!)
Semaphores

Semaphore: integer variable used for synchronization

• has integer value
• list of waiting processes

Works like a gate

• If value > 0, gate is open, and value indicates number of processes that can enter
• Otherwise, gate is closed, possibly with waiting processes
Semaphore Operations

Semaphore \( s = n \)  /* declare and initialize */

Wait (s)

if \( s \) is zero, block process (and associate with \( s \))

decrement \( s \) (note: occurs after process unblocks)

Signal (s)

increment \( s \)

if any blocked processes, unblock one of them

No other operations permitted (e.g., can’t test value)
Mutual Exclusion with Semaphores

Use a “mutex” semaphore initialized to 1

sem mutex = 1

\[ P_0 \]
Wait (mutex);
< critical section >
Signal (mutex);

\[ P_1 \]
Wait (mutex);
< critical section >
Signal (mutex);

Allows only one process to enter critical section

Simple, works for \( n \) processes, no busy-waiting (really?)
Synchronization with Semaphores

Use a “synch” semaphore initialized to 0

```
sem synch = 0;

P_0                                      P_1
< to be done before P_1 >                 Wait (synch);  
Signal (synch);                           < to be done after P_0 >
```

Allows a process to wait for another before proceeding

Semaphores provide pure synchronization

- no way for a process to tell it blocked
- i.e., no information transfer
Atomicity of Semaphore Operations

Wait (sem) and Signal (sem) are atomic operations
  • their bodies are critical sections

Mutual exclusion achieved using a lower-level mechanism
  • test-and-set locks
  • turning off interrupts (on a uniprocessor)

Therefore, problems such as busy-waiting still exist
  • but at a “lower-level”
  • for brief (and known) periods of time