Bronson Edralin

EE468 OS

Homework 5

10/16/14

**Problem 5.9** [1 pt] The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of n-1 turns was presented by Eisenberg and McGuire. The processes share the following variables:

enum pstate {idle, want\_in, in\_cs};

pstate flag[n];

int turn;

All the elements of flag are initially idle. The initial value of turn is immaterial (between 0 and n-1). The structure of process Pi is shown in Figure 5.22. Prove that the algorithm satisfies all three requirements for the critical-section problem.

*The structure of process Pi in Eisenberg and McGuire’s algorithm:*

do {

while (true) {

flag[i] = want in; j = turn;

while (j != i) {

if (flag[j] != idle) {

j = turn;

else

j = (j + 1) % n;

}

flag[i] = in cs;

j = 0;

while ( (j < n) && (j == i || flag[j] != in cs))

j++;

if ( (j >= n) && (turn == i || flag[turn] == idle))

break;

}

j = (turn + 1) % n;

while (flag[j] == idle)

j = (j + 1) % n;

turn = j;

flag[i] = idle;

/\* remainder section \*/

} while (true);

This algorithm satisfies all three requirements for the critical-section problem, which was Mutual exclusion, Progress, and Bounded waiting.

*Proving Mutual-exclusion:*

Process Pi can enter critical section only if (() and (turn=i or flag[turn]=idle)). implies flag[0], flag[1], …flag[n-1] except flag[i] is not equal to in-cs. This implies that P1,P2,…Pn-1, except Pi is currently not in the critical section. After Pi has successfully entered its critical section, it has set flag[i] to in-cs so other processes will not be able to increase their j until j=n. This will prevent them from entering the critical sections.

*Proving progress requirement:*

Process Pi can be prevented from entering critical section only if it is stuck in a loop until it reaches the condition (() and (turn=i or flag[turn]=idle)). When there is no process executing in critical section, j=n may occur. Note that since a process Pk exiting the critical section set turn to some value corresponding the processes which are waiting to enter the critical section and set flag[k] to idle. This will allow waiting processes to enter the critical section to proceed.

*Proving bounded waiting:*

A process Pk exiting the critical section will set turn to next process that is not idle in cyclic ordering (k+1, k+2,….., n-1, 0 …, k) and set flag[k] to idle. It designates first process in order with the entry section as the next one to enter the critical section. Any process waiting to enter critical section will have to do it within n-1 turns (bounded).

**Problem 5.10** [1 pt] Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place. This allows the program to use the processor without letting other processes to execute.

**Problem 5.14** [1 pt] Describe how the compare\_and\_swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirements.

The available hardware synchronization instructions cannot satisfy progress and bounded waiting. There are two shared variables Boolean lock initialized to False and Boolean waiting [n] initialized to False for all processes i=0 to n-1; where if lock=False, no other process is in critical section and if waiting[i] = False, process Pi is not ready to enter its critical section.

do {

waiting [i] = TRUE;

key = TRUE; // key is local to Pi

while (waiting[i] && key)

swap(&lock, &key);

waiting [i] = FALSE;

// Execute in Critical Section

j = (i+1)%n; //select next process

while (j!=i && !waiting[i])

j = (j+1)%n;

if (j==i)

lock = FALSE;

else

waiting [j] = FALSE;

// Execute in remainder section

}while (TRUE);

**Problem 5.21** [1 pt] This problem is about limited number of open connections in a server.

Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

This may be accomplished by initializing the semaphore ‘availableConn’ to a value ‘N’. When a new connection is opened, the semaphore value should be decremented by executing P (availableConn) to decrement the value of currently available connections. When one connection is closed, then V(availableConn) can be executed indicating availability of one more socket for a connection.

**Problem 5.28** [1 pt] This is about fairness and throughput in the readers-writers problem.

Discuss the tradeoff between fairness and throughput of operations in the readers–writers problem. Propose a method for solving the readers–writers problem without causing starvation.

Throughput in readers-writers problem is increased by favoring the multiple readers rather than allowing a single writer to exclusively access the shared values. Favoring readers may result in starvation for writers. Starvation in this problem could be avoided by incorporating timestamps for the waiting processes. When a writer is finished with its task, it would wake up the process that has been waiting for the longest duration. If a reader arrives and sees that another reader is accessing the database, it would only enter the critical section only if there are no waiting writers. These restrictions would guarantee fairness.

**Problem 5.30** [1 pt] This is about the signal() statement in Section 5.8.

Suppose the signal() statement can appear only as the last statement in a monitor function. Suggest how the implementation described in Section 5.8 can be simplified in this situation.

wait(mutex);

...

body of F

...

if (next \_count > 0)

signal(next);

else

signal(mutex);

Mutual exclusion within a monitor is ensured.

Describing how condition variables are implemented as well.

For each condition x, we introduce a semaphore x sem and an integer variable x count, both initialized to 0. The operation x.wait() can now be implemented as

x count++;

if (next \_count > 0)

signal(next);

else

signal(mutex);

wait(x\_sem);

x\_count--;

The operation x.signal() can be implemented as

if (x-count > 0) {

next\_count++;

signal(x\_sem);

wait(next);

next\_count--;

}

When thread T1 calls signal(), the control switches to another thread T2 that is waiting for the condition immediately. In Hoare’s implementation, when T2 leaves monitor or calls another wait(), the control should be switched back to T1 and not to other threads who are waiting to get in monitor. Since T1 is still inside monitor, even if inactive, it would be better to let T1 finish its work before admitting others into monitor. The next semaphore and the next\_count variable are used to keep track of this situation. With signal() being the last statement of all monitor procedures, it is ensured that T1 is no longer in monitor. This means that we will not need next semaphore and next\_count variable anymore.