Operating Systems

Homework-4

5.8: The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, *P*0 and *P*1, share the following variables:

boolean flag[2]; /* initially false */

int turn;

```
do {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
} while (true);
```

Figure 5.21 The structure of process P_i in Dekker's algorithm.

The structure of process *Pi* (i == 0 or 1) is shown in Figure 5.21. The other process is *Pj* (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

Solution:

(1) **Mutual exclusion**: is ensured using the flag and turn variables. If both processes set their flag to true, only one will succeed. Namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn.

(2) **Progress** is provided, again through the flag and turn variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access their critical section, it can set their flag variable to true and enter their critical section. It only sets turn to the value of the other process upon exiting its critical section. If this process wishes to enter its critical section again before the other process, it repeats the process of entering its critical section and setting turn to the other process upon exiting. [only flag[i] or flag[j] == true, not both]

Submitted by
K. Chaitanya Krishna
ID: 013772642

(3) **Bounded waiting** is preserved by the turn variable. Assume two processes wish to enter their respective critical sections. They both set their value of flag to true, however only the thread whose turn it is can proceed, the other thread waits. If bounded waiting were not preserved, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered and exited its critical section. However, Dekker's algorithm has a process set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.

(Referred from pages 206-207 of Operating Systems Concepts, 9th Edition)

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

```
enum pstate {idle, want in, in critical section};

   pstate flag[n];

   int turn;
```

```
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;
   }

      /* critical section */

   j = (turn + 1) % n;

   while (flag[j] == idle)
      j = (j + 1) % n;

   turn = j;
   flag[i] = idle;

      /* remainder section */
} while (true);
```

Figure 5.22 The structure of process P_i in Eisenberg and McGuire's algorithm.

Submitted by
K. Chaitanya Krishna
ID: 013772642

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.

Solution:

This algorithm satisfies the three conditions. Before we show that the three conditions are satisfied, we give a brief explanation of what the algorithm does to ensure mutual exclusion. When a process i requires access to critical section, it first sets its flag variable to want in to indicate its desire. It then performs the following steps: (1) It ensures that all processes whose index lies between turn and i are idle. (2) If so, it updates its flag to in critical section and checks whether there is already some other process that has updated its flag to in critical section. (3) If not and
if it is this process's turn to enter the critical section or if the process indicated by the turn variable is idle, it enters the critical section. Given the above description, we can reason about how the algorithm satisfies the requirements in the following manner:

(1) **Mutual exclusion** is ensured: Notice that a process enters the critical section only if the following requirements is satisfied: no other process has its flag variable set to in critical section. Since the process sets its own flag variable set to in critical section before checking the status of other processes, we are guaranteed that no two processes will enter the critical section simultaneously.

(2) **Progress** requirement is satisfied: Consider the situation where multiple processes simultaneously set their flag variables to in critical section and then check whether there is any other process has the flag variable set to in critical section. When this happens, all processes realize that there are competing processes, enter the next iteration of the outer while (1) loop and reset their flag variables to want in. Now the only process that will set its turn variable to in critical section is the process whose index is closest to turn. It is however possible that new processes whose index values are even closer to turn might decide to enter the critical section at this point and therefore might be able to simultaneously set its flag to in critical section. These processes would then realize there are competing processes and might restart the process of entering the critical section. However, at each iteration, the index values of processes that set their flag variables to in critical section become closer to turn and eventually we reach the following condition: only one process (say k) sets its flag to in critical section and no other process whose index lies between turn and k has set its flag to in critical section. This process then gets to enter the critical section.

(3) **Bounded-waiting** requirement is met: The bounded waiting requirement is satisfied by the fact that when a process k desires to enter the critical section, its flag is no longer set to idle. Therefore, any process whose index does not lie between turn and k cannot enter the critical section. In the meantime, all processes whose index falls between turn and k and desire to enter the critical section would indeed enter the critical section (due to the fact that the system always makes progress) and the turn value monotonically becomes closer to k. Eventually, either turn becomes k or there are no processes whose index values lie between turn and k, and therefore process k gets to enter the critical section.

(Referred from pages 206-207 of Operating Systems Concepts, 9th Edition)

Submitted by
K. Chaitanya Krishna
ID: 013772642

5.14: Describe how compare_and_swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

Solution:

Compare and Swap: compare_and_swap() is an atomic instruction that is used in multithreading to attain synchronization. It compares the contents of a memory location with a given value and, only if they are same, modifies the contents of the location with the new given value. This is done as a single atomic operation. The atomicity guarantees that the new value is calculated based on up-to-date information; if the value had been updated by another thread in the meantime, the write would fail.

```
int compare_and_swap (int *ptr, int expected, int new){

int actual = *ptr;

if (actual == expected)

*ptr = new;

return actual;

}

void acquire (lock *mutex){

while(compare_and_swap(&mutex->available,0,1)==1)
//critical_section
}

void release(lock *mutex){

mutex->available=0;
}
```

(Referred from pages 209-212 of Operating Systems Concepts, 9th Edition)

5.16: The implementation of mutex locks provided in Section 5.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

Solution:

The spinlock should be held for < 2xT. Any longer than this duration it would be faster to put the thread to sleep (requiring one context switch) and then subsequently awaken it (requiring a second context switch.)

(Referred from pages 212-213 of Operating Systems Concepts, 9th Edition)

Submitted by
K. Chaitanya Krishna
ID: 013772642