180 C Homework 3

Name:D Venkata Sai Sri Hari

SJSU ID-014533571

Reference- Silberschatz textbook, internet, manual textbook

5.8

The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:
boolean flag[2]; /* initially false */
int turn;
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

(1) Mutual exclusion is ensured using the flag and turn variables. Even though both variables set their flags to true only based on turn it enters. So, turn can be given to only one, so mutual exclusion is obtained. Other process can enter only when this comes out of critical section.

(2) Progress is provided, through the flag and turn variables. 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

(3) Bounded waiting is preserved using 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, only 1 process will repeatedly enter it, but it is not the case in bounded waiting.

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 was presented by Eisenberg andMcGuire. 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.

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

Progress requirement is satisfied: When multiple processes simultaneously set their flag variables to in_cs and then check with other process that has the flag variable set to in_cs. 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_cs is the process whose index is closest to turn. 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_cs become closer to turn and eventually we reach the following condition: only one process (say k) sets its flag to in_cs. This process then gets to enter the critical section.

Bounded-waiting requirement is met: The bounded waiting requirement is satisfied by the way that when a process k wants to enter the critical section, its section is never again 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

5.14

Describe how the compare and swap() instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

We can take help of these instructions and write an algorithm who can satisfy these requirements. There are 2 shared variables: Boolean lock initialed to FALSE and Boolean waiting [n] initialed to FALSE for all processes i =0 to n-1 ; where if lock = FALSE, no other process is in critical section(CS) and if waiting [i] = FALSE, process Pi is not ready to enter its CS.

```
do
{
waiting [i] = TRUE;
key = TRUE; // key is local to Pi
while (waiting[i] && key) 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);
```

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.

So to solve this the following must be implemented. For each mutex lock there would be of queue process. Whenever the lock is unavailable for a process, they are placed in the queue. When a process releases the lock, it removes and awakens the first process from the list of waiting processes.