Operating System Homework II

106061146 陳兆廷

1. Proof of *TestAndSet()* for meeting 3 requirements
   i. Mutual exclusion

   As shown in Fig 1., Parameter *lock* is responsible for controlling the usage of critical section for processes. *lock* is a global variable set by multiple processes. Only when *lock* being *False* allow other process to access critical section. If one process has entered the critical section, as shown in Fig 2., *TestAndSet* set *lock* to *True*. This way of controlling critical section can ensure mutual exclusion by locking other processes out when one is in critical section.

   ii. Progress

   By setting *waiting[j]* to True before entering critical section and to *False* when exiting, letting other process enter critical section. When there exist 2 processes, P0 and P1, P0 wishes to enter the critical section while the other does not. Then *waiting[0]* is *True* and *waiting[1]* is *False*. Even if the key of P1 is *True*, P1 will not enter critical section, eventually P0 can enter. Thus, it achieves the goal of progress, which indicates that processes cannot be postponed indefinitely when entering critical section.

   iii. Bounded waiting

   The number of the process is *n* in the algorithm. By *j = (i + 1) % n, j = (j + 1) % n*, we get the number left divided by *n*. The number decides which process to be checked next. This mechanism ensures the quality of bounded waiting since the processes can enter critical section in at most *n* turns.

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = TestAndSet(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE; (No one is waiting)
        else
                waiting[j] = FALSE; (process j enters next)
                // remainder section
} while (TRUE);
```

Fig 1.

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

Fig 2.

2. Program to solve 2nd readers-writers program
   i. By adding a new semaphore: *rd* to indicate the status of reading, the writer

can lock the readers out and prevent them from accessing the critical section.

ii. Also, semaphore *mutex* is changed to *rmutex* and *wmutex* for reader and writer mutex, to ensure mutual exclusion when *counts* are changing.

iii. The first writer locks readers from accessing critical section by *wait(rd)*.

iv. The last writer unlocks readers by *signal(rd)*.

Pseudo code:

Reader	Writer

`do`
`{`
`    wait(rd);`
`    //a reader is trying to enter`
`    wait(rmutex);`
`    readcount++;`
`    if (readcount == 1)`
`        wait(wrt);`
`    signal(rmutex);`
`    signal(rd);`
`    //done trying to enter`
 	
`        // reading is performed`
 	
`    wait(rmutex);`
`    readcount –;`
`    if (readcount == 0)`
`        signal(wrt);`
`    signal(rmutex);`
`}`
	`do`
	`{`
	`    wait(wmutex);`
	`    writecount++;`
	`    if (writecount == 1)`
	`    //1st writer`
	`        wait(rd);`
	`        //lock readers out`
	`    signal(wmutex);`
	`    wait(wrt);`
	
	`        // writing is performed`
	
	`    signal(wrt);`
	`    wait(wmutex);`
	`    writecount –;`
	`    if (writecount == 0)`
	`    //last writer`
	`        signal(rd);`
	`        //unlock readers`
	`    signal(wmutex);`
	`}`