Assignment 5 COM S 352

Due: February 23, 2018

5.10(10 points) 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.

This is because if a user-level program is given the ability to disable interrupts, then the timer interrupt can be disabled and the context switching from taking place can be prevented. Thus, allowing the program it self to use the processor without letting other processes to execute.

5.11(10 points) Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

This is because that interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled. Thereby the process disabling interrupts cannot guarantee mutually exclusive access to program state, and there are no limitations on what processes could be executing on other processors

5.16(10 points) 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.

This would be very similar to the changes made in the description of the semaphore. Associated with each mutex lock would be a queue of waiting processes. When a process determines the lock is unavailable, they are placed into the queue. When a process releases the lock, it removes and awakens the first process from the list of waiting processes.

5.23(10 points) Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the test and set() instruction. The solution should exhibit minimal busy waiting.

```
int control = 0;
int semaphore value = 0;
wait()
```

```
{
while (TestAndSet(&control) == 1);
if (semaphore value == 0) {
atomically add process to a queue of processes
waiting for the semaphore and set control to 0;
} else {
semaphore value--;
control = 0;
}
}
signal()
{
while (TestAndSet(&control) == 1);
if (semaphore value == 0 &&
there is a process on the wait queue)
wake up the first process in the queue
of waiting processes
else
semaphore value++;
control = 0;
}
```

5.35 (20 points) Design an algorithm for a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a function tick() in your monitor at regular intervals.
The following restrictions apply:

Pseudocode:

```
monitor alarm{
condition awake;
int current =0;
void delay(int ticks){
int time;
time = current + ticks;
while(current<time ) awake.wait(time);
Awake.signal;
}
```

```
void tick(){
Current++;
awake.signal;
}}
```

6) (40 points) write a program so that one thread will print "hello ", one thread will print "world"and another thread will print the exclamation mark “!”, and the main function will print the trailing "\n", using pthread_create(), pthread_exit(), pthread_yield(), and pthread_join()..

Hints & Tips:

1) You must use a synchronization method to ensure that the "world" thread runs after the "hello" thread.

2) You must use a synchronization method to assure that the main thread does not execute until after the "world" and “exclamation” threads.

```
/* Include Files */
 #include <stdio.h>
/* External References
*/
extern int world( void );
extern int hello( void );
extern int exclamation(void);

int main( int argc, char *argv[] ) {
world();
hello();
exclamation();
printf( "\n" );
return( 0 );
}

/* world - print the "world" part. */

int world( void ) {
printf( "world" );
return 0 ;
}
```

```
/* hello - print the "hello" part. */

int hello( void ) {
printf( "hello " );
return 0 ;
}

/* exclamation – print “!”.*/
Int exclamation(){
Printf(“!”);
Return 0;
}
```