CSC 4420
Assignment 2
May Wandyez
Gq5426
6/26/2024

CSC 4420 Assignment 2

Question 1 (2 points): The register set is listed as a per-thread rather than a per-process item in the following table. Why? After all, the machine has only one set of registers.

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Processes run on threads. Registers are manager per thread because each thread represents a independent execution within a process, and each thread requires its own set of registers to store the execution state, control information, and temporary data. Isolating the registers from the process enables multiple threads to run concurrently and independently – the alternative would be a process clumsily overwriting shared registers for calculations, loss of the ability to execute independently as threads would need to coordinate register usage, and increase the complexity of scheduling the tasks in each thread.

Question 2 (2 points): What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

Busy waiting involves a process/thread continuously checking to see if conditions are met suitable for it to run (such as awaiting availability of computational resources, or completion of a prerequisite task). The object that is busy waiting is not being blocked from running, so it instead takes up CPU space for the purpose of continuously checking to see if conditions are met to run.

There are other types of waiting, such a timeouts (checking periodically at defined intervals), blocking waiting (putting a process to sleep until the condition for waiting is met), event waiting (being brought out of waiting for when an event occurs), interrupt waiting (just interrupt the CPU when conditions are met, skip the line!).

Busy waiting can be avoided altogether, and replaced with blocking and synchronization variables, or be triggered by events and interrupts instead of a continuous loop for checking conditions. However, busy waiting has the benefit of high response time as it avoids context

switching that blocking mechanisms may require. In systems where response time matters, or the complexity of blocking mechanisms is a strain on the limit resources, busy waiting can be used instead.

Question 3 (3 points): The program shown below uses the Pthreads API. What would be the output from the program at LINE C and LINE P? Motivate your answer.

```
//assignment 2 question 3
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>

int value = 0;
void *runner(void *param);//this is the thread

int main(int argc, char *argv[])
{
   pid_t pid;
   pthread_t tid;
   pthread_attr_t attr;

   pid = fork();

   if (pid ==0){
      //this is the child process because the pid is 0
```

```
        pthread_attr_init(&attr);
        pthread_create(&tid,&attr,runner,NULL);
        pthread_join(tid,NULL);
        printf("CHILD: value = %d", value);//THIS IS LINE C
    }else if(pid >0) {
        //this is the parent process because the pid is GREATER than 0
        wait(NULL);
        printf("PARENT: value=%d",value); //THIS IS LINE P
    }
}


void *runner(void *param){
    value = 5;
    pthread_exit(0);
}
```

For convenience, I rewrote the code and corrected some of the mistakes. The value at line C (for child) is 5, while the value at line P (for parent) is 0. This is because when the child of the parent is created through the fork, the child creates a new thread to run the 'runner' function. The runner function changes value to 5, the child then waits for the thread tid to complete before resuming the child.

Question 4 (3 points): Consider the following solution to the mutual-exclusion problem involving two processes P0 and P1. Assume that the variable turn is initialized to 0. Process P0's code is presented below.

```
/* Other code */

while (turn != 0) { }; /* Entry Section. */

Critical Section;
```

```
turn = 0;  /* Exit Section*/

/* Other code */
```

For process P1, replace all 0 by 1 in above code. Determine if the solution meets all the three required conditions for a correct critical section problem solution.

Process P1:

While turn(!= 1){

//Entry section

};

Critical section ;

Turn = 1;

/*other code*/

The critical section is the part where shared resources are accessed or modified by multiple

thread or processes. The critical section problem requires a solution where there is mutual exclusion (only one process can be inside critical section at any time), progress (when work is done, critical section is freed), and bounded waiting (each thread has FINITE waiting time).

However, looking at the code, the entry section spins infinitely while awaiting its turn (its turn is 1), mutual exclusion is not guaranteed to be met unless this is the only section of code with a turn of 1, and similarly progress does not occur at ALL because it assigns its own turn value AFTER the critical section is completed. Further, there is NO bounded waiting, there is no guarantee that the turn will ever be 1 after some certain period of time – thus the piece of code waits forever.

This pseudo code might solve the problem instead:

```
while(turn!=1){
        //spin while waiting for turn
        //if turn not received within 100 seconds, exclude others.
        If timewaited > 100 seconds{
                Turn = 1
        }
}
While(turn =1){
/*other code goes here*/
Turn = 0 //relinquish turn to turn 0
}
```

Question 5 (3 points) Does the busy waiting solution using the turn variable (see figure below) work when the two processes are running on a shared-memory multiprocessor, that is, two CPUs sharing a common memory? Is there anything else necessary compared to the single-CPU case?

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a) Process P0

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b) Process P1

Yes, it SHOULD work when running on a share memory processor, however because each CPU has its own cache, there might be an issue where modifications to the turn variable in one CPU's cache do not synchronize. This can be solved through mutexes, or through a controller that both CPUs access to determine the value of turn.

Question 6 (3 points): Show that, if the wait() and signal() semaphore operations are not executed atomically, then mutual exclusion may be violated.

Atomicity means that when an operation begins, it follows through to completion without being interrupted. If wait() and signal() are possible to interrupt, this may lead to race conditions that violate mutual exclusion. For instance, wait() could be executes on the exact same semaphore concurrently, leading to the semaphore's value being adjusted by both, potentially leading to operations concurrently trying to access the same critical section in violation of mutual exclusion.

Question 7 (3 points): Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: deposit(amount) and withdraw(amount). These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function and the wife calls deposit(). Describe how a race condition is possible and what might be done to prevent the race condition from occurring. NOTE: assume

a "current_balance" variable is shared to store the current account balance. You should write the source code of your solution.

A race condition is possible when one action potentially cancels out the actions of the other erroneously due to the shared resource being currently in use. This may be an issue as the functions may try to save the shared resource before the other updates it, for instance:

1.)The wife deposits 100 dollars into an account reading 0 dollars. The updated balance is saved as $100.00. This process completes quickly and is saved while the husband is withdrawing from the shared account.

2.)The husband goes at the same time, and withdraws 100 dollars from the account that has an initial balance of 0 dollars, leaving to an overdraft of 100 dollars, the new final total SHOULD be saved as -100.00. THIS VALUE OVERWRITES THE SAVED VALUE instead of the wife's deposit of 100.00, and as a result the husband and wife now both have an overdraft despite none actually occurring.

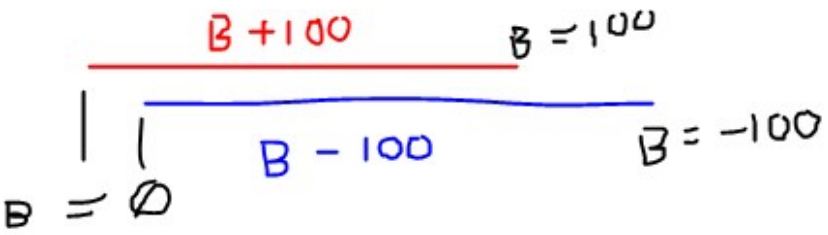

Figure 7 – Illustration of Race Conditions Occurring due to share memory being accessible by both tasks before either completes.

Here is some pseudo code using a simple flag to prevent simultaneous use of shared resources

```
Def UseHitSomeButtonsOnScreen{
while(accountInUse = True){
//busy waiting occurs here
}

//account is no longer in use by others, time to make sure no one else uses it while we use it
accountInUse = True
Withdraw() //this could also be Deposit()

//critical region actions performed, others are free to use resources.
accountInUse = False
}
```

Question 8 (3 sub-questions 2 points each, 6 points total):

Suppose we have an atomic hardware instruction **TestAndSet** whose action is described by the pseudocode:

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

The following pseudocode claims to implement mutual exclusion using TestAndSet:

```
SHARED:
   boolean lock = ???;

   line   Process 0:                              line   Process 1:
   0.0                                            1.0
   0.1       while (TRUE) {                       1.1       while (TRUE) {
   0.2          while (TestAndSet(&lock)) ;       1.2          while (TestAndSet(&lock)) ;
   0.3          /* CRITICAL SECTION */            1.3          /* CRITICAL SECTION */
   0.4          lock = FALSE;                     1.4          lock = FALSE;
   0.5          /* REMAINDER SECTION */           1.5          /* REMAINDER SECTION */
   0.6       }                                    1.6       }
```

a. **(2 points)** Assume the lock is initialize to False. Suppose the scheduler happens to cause statements to be executed in the order shown below. Complete the blank columns in the table.

LINE	LOCK VALUE	Explain Non-Obvious Occurrences
0.0	FALSE	EXECUTION BEGINS WITH PROCESS P0
0.1	FALSE	P0 While loop begins
0.2	TRUE	TestAndSet executed on lock, the value of the lock is set to true, however the value returned to the while loop is false

		due to the temporary variable rv. The while loop for lock is exited.
0.3	TRUE	P0 critical section executed
0.4	FALSE	P0 changes the value of the lock to false
0.5	FALSE	P0 remainder section executed
0.6		P0 IS INTERRUPTED, p1 IS PLACED IN RUNNING STATE
0.7	FALSE	
0.8	FALSE	
0.9	FALSE	
1.0	FALSE	
1.1	FALSE	P1 while true loop starts
1.2	TRUE	TestAndSet executed on lock, the value of the lock is set to true, however the value returned to the while loop is false due to the temporary variable rv. The while loop for lock is exited.
1.3	TRUE	P1 critical section executed
1.4	FALSE	P1 sets lock to false
1.5	FALSE	P1 remainder executed
1.6	FALSE	END OF TRACING

b. **(2 points)** Assume the lock is initialize to False. Suppose the scheduler happens to cause statements to be executed in the order shown below. Complete the blank columns in the table.

LINE	**LOCK VALUE**	**Explain Non-Obvious Occurrences**
0.0		EXECUTION BEGINS WITH PROCESS P0
0.1	FALSE	P0 While loop begins
0.2	TRUE	TestAndSet executed on lock, the value of the lock is set to true, however the value returned to the while loop is false due to the temporary variable rv. The while loop for lock is exited.
0.3	TRUE	P0 IS INTERRUPTED; p1 IS PLACED IN RUNNING STATE
0.4	TRUE	
0.5	TRUE	
0.6	TRUE	
0.7	TRUE	

0.8	TRUE	
0.9	TRUE	
1.0	TRUE	
1.1	TRUE	P1 while true loop starts
1.2	TRUE	TestAndSetExecuted on lock, the value of the current lock is returned as TRUE, the value of the lock is set to TRUE. The while loop thus continues indefinitely
1.3	TRUE	P1 remains in while loop.
…		END OF TRACING

c. **(2 points)** Assume the lock is initialized to True. What happens to the two processes?

Neither process would be able to run, as both processes would remain stuck in the TestAndSet while loop which would keep returning true, while also turning the value of the lock to True, meaning neither could escape it.