HW2Report

Muhammed Yasir Fidan

May 2021

1 New System Calls

I implement 6 new system calls in spim for this homework. Those are threadCreate,InitThread,ThreadExit,ThreadJoin,MutexLock and MutexUnlock. I implement those in syscall.cpp file. Just replace spim files system.h and system.cpp with my system files for running project.

```
62
63  //I IMPLEMENT THIS SYSCALLS
64  //IN syscall.cpp
65
66  #define THREAD_CREATE 18
67  #define THREAD_JOIN 19
68  #define THREAD_EXIT 20
69  #define INIT_THREAD 21
70  #define MUTEX_LOCK 22
71  #define MUTEX_UNLOCK 23
```

Figure 1: 6 new Syscalls

1.1 Thread Create

I create a thread structure in system.cpp file. This structure used for holding my threads. Threads has some shared parts in memory like text and data segments, so I didn't hold them because all threads will have same text and data segments. Each threads have different registers, Id,state, Program counter(of course) and stack. Also there is a processName attribute but because my program has 1 process and multiple thread, each thread process name will be same and it will be "init" as told in pdf. I hold it in thread structure too for printing thread attributes make easy.

```
108
109  int threadIdCount = 0;
110  int currentThread = 0;
111  int waitCount = 0;
112  struct Thread{
113    /*
114      All threads have separete Id,name,Program count
115      Stack,State and registers.
116    */
117    int ThreadID;
118    char ThreadName[50];
119    char ThreadState[10];
120    char processName[10]; // Process name alwaays "in
121    mem_addr Thread_PC; // Program counter for Thread
122    mem_addr ThreadStack; // New Stack for Thread
123    //St
124    mem_word *thread_stack_seg;
125    short *thread_stack_seg_h;
126    BYTE_TYPE *thread_stack_seg_b;
127    mem_addr thread_stack_bot;
128    int thread_stack_size;
129    // Registers for Separete threads
130
131    reg_word ThreadReg[R_LENGTH];
132    reg_word ThreadHI, ThreadLO;
133    reg_word ThreadCCR[4][32], ThreadCPR[4][32];
134  };
```

Figure 2: Thread Structure

Then I create 2 vectors that has type from this structure. This vectors hold ThreadTable and RoundRobin. Whenever a thread switching occur I use this RoundRobin structure for chosing new thread. This algorithm like a fifo structure. In ROundRobin vector first element is currently running thread, whenever a interrupt or thread exit happining, I delete first element from roundrobin and add this at the end of the vector. Then choose new front Thread to switch.

```
140
141  //My Thread table as a vector
142  vector<Thread> ThreadTable;
143  //Round-robin scheduling
144  vector<Thread> roundRobin;
145  bool threadExit = false;
146
```

Figure 3: Thread Table and Robin

```
//create thread
struct Thread newThr;
//Fill thread
newThr.ThreadID = threadIdCount++;
char temp[50];
snprintf(temp,50,"Thread_%d",newThr.ThreadID);
strcpy(newThr.ThreadName,temp);
newThr.Thread_PC = PC+4;
strcpy(newThr.ThreadState,"Ready");
strcpy(newThr.processName,"init");

newThr.thread_stack_size = ROUND_UP(initial_stack_size, BYTES_PER_WORD);
newThr.thread_stack_seg =  (mem_word *)malloc(newThr.thread_stack_size);
memcpy(newThr.thread_stack_seg,stack_seg,newThr.thread_stack_size);
newThr.thread_stack_seg_h = stack_seg_h;
newThr.thread_stack_seg_b = stack_seg_b;
newThr.thread_stack_bot = stack_bot;

//newThr.ThreadStack = R[29];
cout<<"Thread ID:\t"<<newThr.ThreadID<<endl;
cout<<"Thread Name:\t"<<newThr.ThreadName<<endl;
cout<<"Thread PC:\t"<<newThr.Thread_PC<<endl;
cout<<"Thread state:\t"<<newThr.ThreadState<<endl;
cout<<"Thread stackPointer:\t"<<newThr.thread_stack_seg<<endl;
cout<<"Process Name:\t"<<newThr.processName<<endl;
//PC += 4;
for(int i=0; i<R_LENGTH; i++){
  newThr.ThreadReg[i] = R[i];
}
for(int i=0; i<4; i++){
  for(int j=0; j<32; j++){
    newThr.ThreadCCR[i][j] = CCR[i][j];
    newThr.ThreadCPR[i][j] = CPR[i][j];
  }
}
newThr.ThreadHI = HI;
newThr.ThreadLO = LO;
newThr.ThreadReg[2] = 0; // Make this thread v0 = 0

ThreadTable.push_back(newThr);
roundRobin.push_back(newThr);
break;
```

Figure 4: Thread creation

Thread create syscall basically create a new Thread type object, init its value and add it to the thread table and round robin. At the beginning it copies parent register but each thread has its own set of threads and I create new stack position for new thread. Also I init v0 value for new thread as 0. So I can use this register to seperate threads in program execution. Like after creation thread check v0 value if its 0 jump somewhere, so new created thread jump and goes this way but main thread continue same place.

1.2 Spim Interrupt handler

```
  ThreadTable[currentThread].Thread_PC=PC;
  //ThreadTable[currentThread].ThreadStack=R[29];
  strcpy( ThreadTable[currentThread].ThreadState,"Ready");
  ThreadTable[currentThread].ThreadHI = HI;
  ThreadTable[currentThread].ThreadLO = LO;
  ThreadTable[currentThread].thread_stack_seg =  stack_seg;
  ThreadTable[currentThread].thread_stack_seg_h = stack_seg_h;
  ThreadTable[currentThread].thread_stack_seg_b = stack_seg_b;
  ThreadTable[currentThread].thread_stack_bot = stack_bot;
  for(int i=0; i<R_LENGTH; i++){
    ThreadTable[currentThread].ThreadReg[i] = R[i];
  }
  for(int i=0; i<4; i++){
    for(int j=0; j<32; j++){
      ThreadTable[currentThread].ThreadCCR[i][j] = CCR[i][j];
      ThreadTable[currentThread].ThreadCPR[i][j] = CPR[i][j];
    }
  }
  roundRobin[0].Thread_PC = ThreadTable[currentThread].Thread_PC;
  strcpy(roundRobin[0].ThreadState,ThreadTable[currentThread].ThreadState );
  roundRobin[0].thread_stack_seg = ThreadTable[currentThread].thread_stack_seg;
  //move this thread to end of the list
  roundRobin.erase(roundRobin.begin());
  if(!threadExit){
    roundRobin.push_back(ThreadTable[currentThread]);
  }


  cout<<"Switching Thread "<<currentThread<<" to ";
//Switch to next thread now next thread is in 0 position
currentThread = roundRobin[0].ThreadID;


PC = ThreadTable[currentThread].Thread_PC;
//R[29] = ThreadTable[currentThread].ThreadStack;
strcpy( ThreadTable[currentThread].ThreadState,"Running");
HI = ThreadTable[currentThread].ThreadHI;
LO = ThreadTable[currentThread].ThreadLO;
stack_seg = ThreadTable[currentThread].thread_stack_seg;
stack_seg_h = ThreadTable[currentThread].thread_stack_seg_h;
stack_seg_b = ThreadTable[currentThread].thread_stack_seg_b;
stack_bot = ThreadTable[currentThread].thread_stack_bot;
for(int i=0; i<R_LENGTH; i++){
  R[i] = ThreadTable[currentThread].ThreadReg[i];
}
for(int i=0; i<4; i++){
  for(int j=0; j<32; j++){
```

Figure 5: Spim Interrupt Handler

In spim interrupt handler I make context switch between threads. I use round Robin vector here. First I save current thread attributes(like registers, stack pointer,program counter etc) then erase this thread from round robin and push it back at the end of structure. After that take new front thread and copy its attributes.

1.3 Init create

This syscall same as normal thread create, I use this just create main thread and make process name "init".

1.4 Thread exit

Thread exit syscal delete permenantly thread from round robin structure than switch another thread for continue.

1.5 Thread join

```
435     }
436
437     case THREAD_JOIN:
438     {
439       if(waitCount > 0){
440
441         R[10] -= waitCount;
442         waitCount = 0;
443       }
444       //cout<<"ThreadJoin"<<
445
446     break;
447     }
```

Figure 6: Thread Join

I have a variable called waitCount. In main thread I hold thread count in t2 register. whenever a thread exit they will increase waitCount and when join called this waitCount will be decrease from t2 of the main thread. So main thread wait for all threads exit. When all thread exit main thread t2 will be 0 and it can be continue.

1.6 Mutex Lock and Unlock

I also have a structure for mutexes mutexes vector hold locked mutex infos. So

Figure 7: Mutex Structure

when a thread try to lock a mutex it first check if its already locked or not. If its locked it cant take lock and wait context switch because can't continue. If a lock is free it can take lock and add lock info to this vector, so when another thread try to take lock it first check vector and see its taken by some other thread and it can't take lock.
Mutex unlock basically just remove mutex info from mutexes vector, so another thread can take it and continue.

2 Merge Program

My first kernel is doing a merge sort by using multithreading. In this program every thread sort a different part of array and main thread will wait them. After all thread sorting part is done main thread sort remaining part and print sorted array at terminal.

```
.data
    array: .word 7 20 12 60 11 5 75 200 21 33 44 68 99 105 54 190 17 345 0 1
    MAX: .word 20
    ThreadCount: .word 4

    mutex: .asciiz "mutex"
    after: .asciiz "After Wait Main Thread Continue\n"
    sorted: .asciiz "New array after merge sort:\n"
    space: .asciiz " "
    part: .word 0


.text


main:
    lw $t2,ThreadCount

    li $v0,21
    syscall     #init process

    addi $t4,$0,0

threadCreateLoop:
    beq $t4,$t2,wait

    li $v0,18   #create threaed
    syscall
    beqz $v0,thread2
    addi $t4,$t4,1
    j threadCreateLoop




wait:
    li $v0,19 # Thread join
```

Figure 8: Merge Program beginning

Here this is my merge program beginning. Max is size of array and thread-Count is number of thread. First in main thread I assign t2 = 4 because thread join work by looking t2 value, so it must init to thread count. You need to change Max, Thread count and array if you want to try this for different thread numbers. For example, This is 4 thread example but if you make Threadcount value 2 this will work fine for 2 thread too.
In main first I assign t2 to thread count than create init process bu calling syscall 21(this is my init thread syscall). Then in a loop I create my threads (4 thread in this example). And after thar main thread will wait other threads.

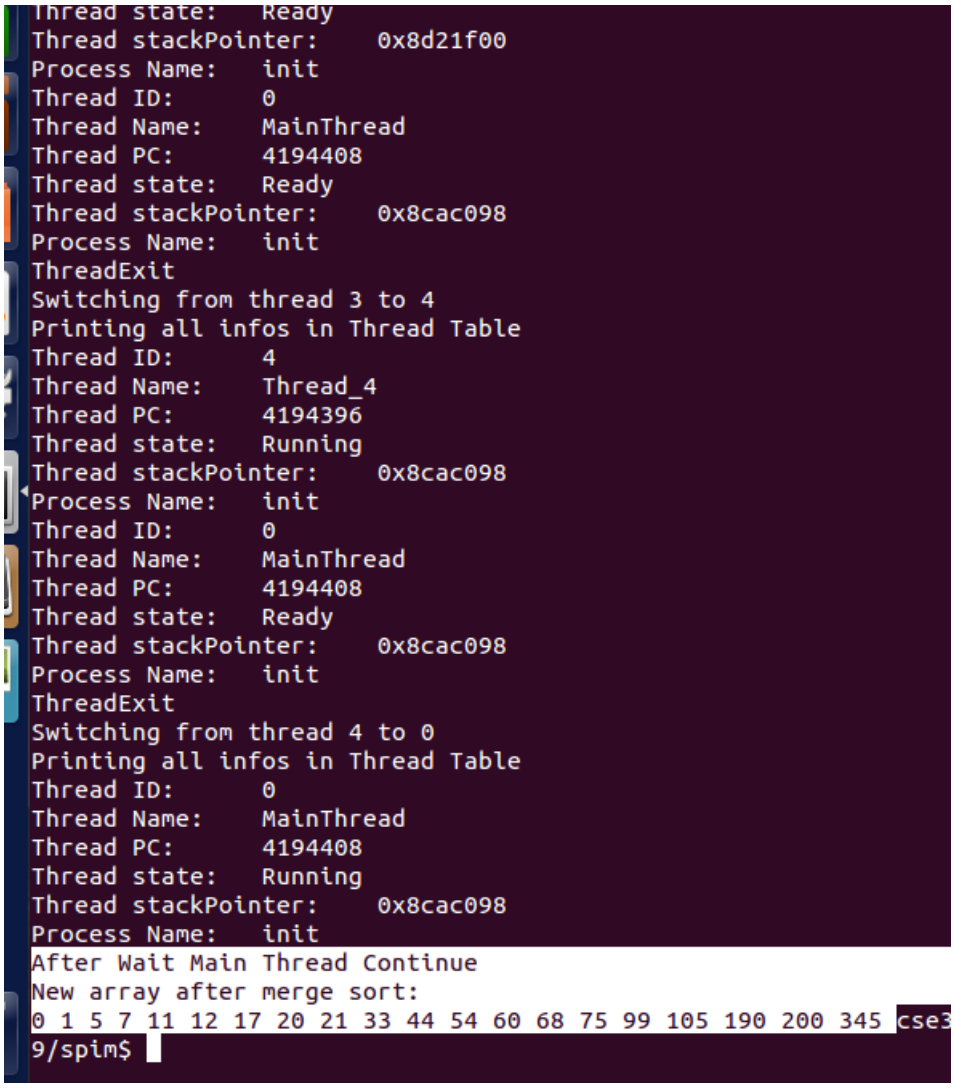

Figure 9: Merge Program output

Here this is my merge program output, I print round robin threads information when a thread exit or interrupt occur, And as youcan see main thread wait other threads then it will print sorted array. You can execute program as ./spim -f SPIMOS_GTU_1.s

3 Producer-Consumer problem

My second kernel is Producer consumer problem. 1 thread for producer and 1 thread for consumer and both of them work 10 times. When execute program, user needs enter 1 or 2, 1 for producer consumer with mutexes and 2 for no mutex producer consumer. Also because we have just mutex not semaphores, instead of using real buffer when they called they start and print "Producing..." or "Consuming..." for simulate produce and consume step. Also there is a loop that count from 0 to 1000 for simulate producing and consuming and give tim to kernel to switch between threads, so we can see more clearly difference between motex and non mutex solution. This may be hard to see difference because my interrupt handler print threads info whenever a intterupt happen, so output can be little bit confusing but just focus "Producing start","producing end", "consumer start", "consumer end", "producing..." and "consuming..." output messages and you will see difference between mutex and non mutex implementation.

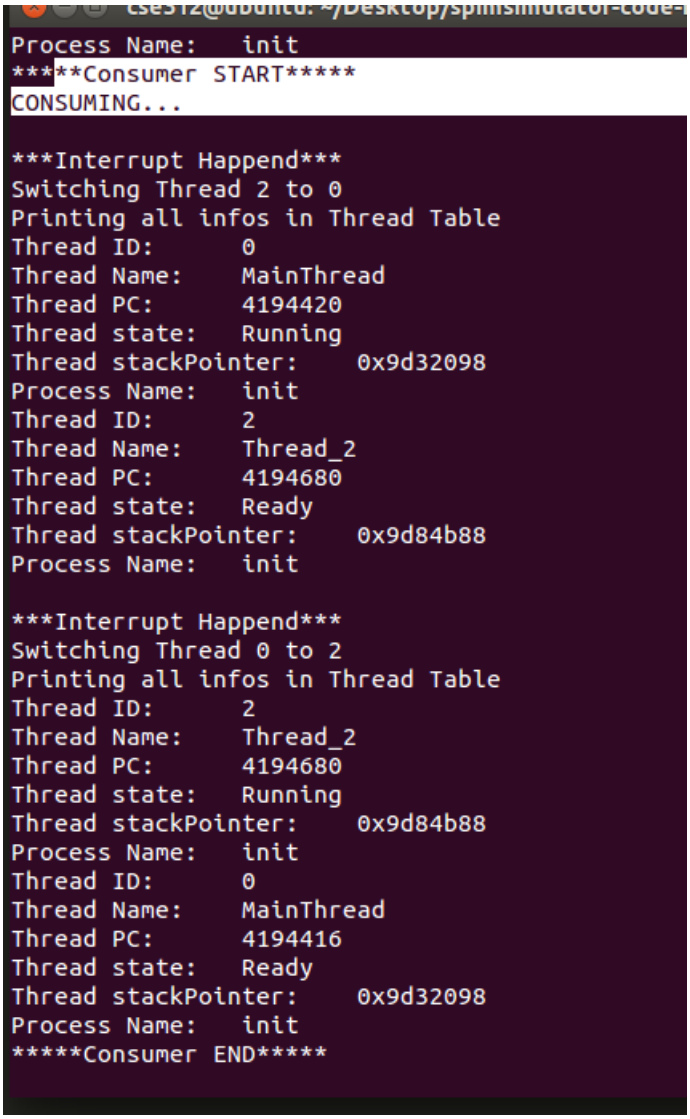

Figure 10: Producer consumer example output with mutexes

Here you can see with mutexes when a Producer or consumer start, it doesn't matter how much interrupt occur first consumer or producer must end before another producer or consumer period starts because mutex block other one to continue. Here as you can see There is "consumer start" it means consumer take the lock, so producer cant continue even an interrupt occur. After Consuming you can see in last line "Consumer end". After that consumer relese mutex so producer or consumer can continue again.
Whit mutexes all input will be this format atomically:

"*****Consumer START*****"
"*****CONSUMING...*****"
"*****Consumer END*****"
or for producer
"*****Producer START*****"
"*****PRODUCING...*****"
"*****Producer END*****"

This 2 print format can't be diveded because of mutexes. So you will not see any
"*****Consumer START*****"
"*****PRODUCING...*****"
"*****Producer END*****"
print example. Always if consumer start first it will end before producer start, or producer start, it will end before consumer or again producer start.

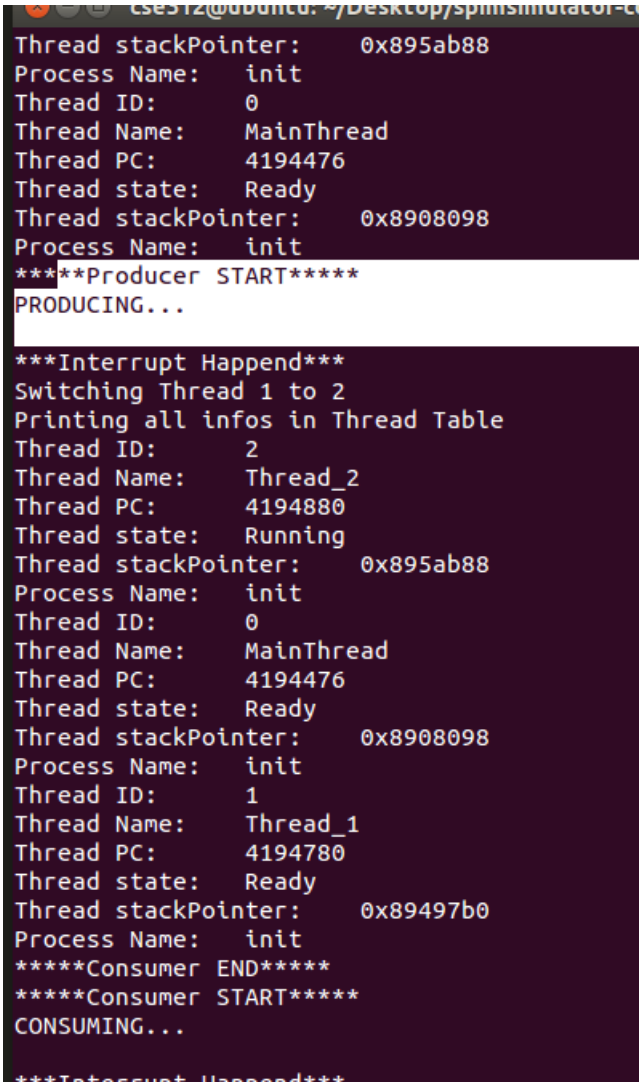

Figure 11: Producer consumer example output without mutexes

Here without using mutex you can see sometimes program Write
"***Producer START***"
"PRODUCING"
"***CONSUMER END***"
"***CONSUMER START***"
As you can see before consumer end a context switch happining then producer can start while consumer didn't end and before producer end context switch happining and consumer continue again. This is because we don't use mutexes so producer and consumer can't block each other.