SGP REPORT NACHOS

In this report we have discussed our work on thread class (thread.cc) , synchronization primitives (synch.cc) and Exception handling (exception.cc).

Shashi Mohan Reddy Ravula

Ishan Tiwari

Exception Handling (exception.cc)

System Call and Exception handling

The System calls is nachos are called by the User programs by executing syscall instruction in sys.s file which generates a trap into the Nachos kernel. which in turn invokes the RaiseException() routine in machine.cc by passing appropriate arguments. RaiseException() , in turn, calls the routine ExceptionHandler() in exception.cc .

We then handle the exception by passing ExceptionHandler with argument indicating kind of exception raised and also switch between several syscalls handlers based on the argument type from to 0 to 35 .

We have modified certain parts of the exception.cc so as to handle the system calls such as

- SC_P (Semaphore Wait)
- SC_V (Semaphore Signa)l
- SC_SEM_CREATE
- SC_SEM_DESTROY
- SC_LOCK_CREATE
- SC_LOCK_DESTROY
- SC_LOCK_ACQUIRE
- SC_LOCK_RELEASE
- SC_COND_CREATE
- SC_COND_DESTROY
- SC_COND_WAIT
- SC_COND_SIGNAL
- SC_COND_BROADCAST

let us now consider the implementation SC_SEM_CREATE which is triggered when user program calls system call SemCreate() . We need to create variable of type semaphore to store

```
case SC_SEM_CREATE: {
        DEBUG('e', (char*)"SEMAPHORE CREATE\n");
        Semaphore* Semid;
        int sizep;
        int addr;
        addr = g_machine->ReadIntRegister(4);
        sizep = GetLengthParam(addr);
        char debug[5];
        GetStringParam(addr,debug,sizep);          //Getting the first string argument
        int32_t sid;
       //creating the semaphore
       Semid = new Semaphore(debug,g_machine->ReadIntRegister(5));

        if(Semid)                    //checking for any error while creating a Semaphore
        {
              sid = g_object_ids->AddObject(Semid);
              g_syscall_error->SetMsg((char*)"",NoError);

              //returning the semaphore ID for other use
              g_machine->WriteIntRegister(2,sid);

        }

        else

        {
              sid = g_object_ids->AddObject(Semid);

              //showing the error message for InvalidSemaphoreId msgerror.cc

              g_syscall_error->SetMsg(msg, OutOfMemory);
              sprintf(msg,"%d",sid);
              g_machine->WriteIntRegister(2,-1); // returning the error code via reg 2
        }
              break;
        }
```

The above code for the case SC_SEM_CREATE will first declare a Semaphore variable Semid. It will read the first argument coming from sem_Create and add it to the debug char array using getStringParam(). Then it will create the semaphore object using the first argument in addr and reading the second argument directly Intregister. It will check the condition whether the semaphore object is created or not and will return the sid which is obtained during adding the object. There is map in objid.h where id, object pair is stored, where we are adding the object. Else if no semaphore object is created it will throw an error (returns -1 through register 2).

Similarly all the other case are also implemented and specific calls to the respictive methods at synch.cc and thread.cc are made just like *new Semaphore ()*

Nachos Threads (thread.cc)

Nachos Thread class is defined in “nachos-3.4\code\threads\thread.h” and the interfaces are implemented in “nachos-3.4\code\threads\thread.cc”

We had to implement following methods in the thread.cc

Thread::Start(Process *owner, int32_t func, int arg)

In order to Implement this method we had to Initialize several parameters by calling functions like .This method is just like fork() it creates a new thread.

StackAllocate() - to create a new stackspace of size **UserStackSize**
AllocBoundedArray() - useful for catching overflow beyond fixed-size thread execution stacks.
InitSimulatorContext(KStack,SIMULATORSTACKSIZE) - Sets the initial values for simulator context
InitThreadContext(func,(int32_t)stackPointer,arg) - sets the initial values for thread context

Thread::Finish ()

Finish() is called when a thread is done executing its function.The thread invoking Finish() ceases to exist and the thread at the head of the ready list is assigned the CPU.This mechanism is called by the **static threadStart()** method**.** The global variable **g_thread_to_be_destroyed** points to the current thread, but does not actually terminate it. Finish then calls **Sleep()**, which terminates the thread .When the scheduler starts running another thread using switchto(), the newly scheduled thread examines the **g_thread_to_be_destroyed** variable (Null or not) and finishes the job.

Thread::SaveProcessorState()

This function is used to save the state of the user programs on context switch .We have read all the register values of **INT** , **FLOAT** and **ConditionCode(CC**) **Registers** then stored into thread context.

Thread::RestoreProcessorState()

This Restore the CPU state of a user program on a context switch. We have read all the register values from the thread context and written them to **FLOAT, INT** and **CC Registers.**

We have extensively tested all the above methods by writing testcases like threadtest.c, prodcon.c, multprodcon.c, condtest.c,locktest.c

Synchronization Primitives (synch.cc)

Thread Communication is very essential since the input of one thread may depend on the output of another, or they may need to access the same resource and need to do so in an orderly manner. The key to avoiding race conditions is to provide mutual exclusion - when one thread uses a shared variable other threads are blocked.

Thread Synchronization (Inter Thread Communication) in Nachos is achieved by few synchronization primitives like

- ***Semaphores***
- ***Locks***
- ***Condition Variables***

We have implemented the following methods in synch.cc

Semaphore::P()

```
while (value == 0) {
          queue->Append((void *)g_current_thread);  // semaphore is not available and the
thread is blocked (sleep)
          g_current_thread->Sleep();

          }
        value--;
```

The above code will check if the value of the counter which is initialised when semaphore is created equals to to zero or not . If it is zero then it append the calling thread to semaphore waiting queue , then it blocks the calling thread puts it to sleep by appending it to waiting queue. It also decrements the value after successfully blocking one thread.

Semaphore::V()

```
thread = (Thread *)queue->Remove();
       if(thread!=NULL)
          g_scheduler->ReadyToRun(thread);
        value++;
```

The above code will remove the first thread which is in the semaphore queue and also adds the calling thread to the ready queue.

Lock::Acquire()

```
if(g_current_thread == owner) {
   g_machine->interrupt->SetStatus(oldLevel); // enable interrupts
   return;
 }
 if(free) {
   free = false;                              // lock is no longer free
   owner = g_current_thread;                 // now owned by currentThread
 } else {
   sleepqueue->Append((void *)g_current_thread); // add to lock wait queue
   g_current_thread->Sleep();                   // put to sleep
 }
```

The above code will check if the calling thread is already the owner if yes then it will return nothing. If not it will check if the lock id is free if its free it assigns the calling thread .if lock id is not free then it appends calling thread to the lock wait queue and puts it to sleep (blocks).

Lock::Release()

```
 if(g_current_thread!= owner) {
   if(owner!= NULL){
   printf ("Error: Thread (%s) is not the lock owner (%s)!\n",
              g_current_thread->GetName(), owner->GetName());     // print error message
   }

   else{
     printf("Error: There is no lock owner; thread (%s) cannot release the lock!\n",
                                                          g_current_thread->GetName());
   }
   return;
 }
 if(!(sleepqueue->IsEmpty())) {
   thread = (Thread *)sleepqueue->Remove(); // remove a thread from lock's wait queue
   if(thread != NULL) {
     g_scheduler->ReadyToRun(thread);    // put in ready queue in ready state
     owner = thread;               // make lock owner
   }
```

The above code checks for the ownership of the lock if it is not the owner then it will return error because only thread which locks and is the owner can release it . So if any other thread tries to unlock or release it ,it throws an error. If the calling thread is the owner then it checks if lock wait queue is not empty and then it removes the waiting thread and puts it back to ready queue .

Condition::Wait()

```
waitqueue->Append((void*)g_current_thread);   //add the thread in the list
```

```
g_current_thread->Sleep();     //wait for a signal
```

This code will simply put the calling thread in condition queue and the scheduler waiting queue.

Condition::Signal()

```
Thread* thread = (Thread*) waitqueue->Remove();
        if(thread = NULL)
        {
                g_scheduler->ReadyToRun(thread);
        }
        else
        {
          printf("Error: No threads to signal");
        }
```

The above code will simply remove a thread from the condition queue and sends it to scheduler ready queue by calling readytorun()

Condition::Broadcast()

```
Thread* thread;
        while((thread=(Thread*)waitqueue->Remove()) != NULL)
        {
                g_scheduler->ReadyToRun(thread); //Wake up all the threads
        }
```

The above code removes all the threads from the condition queue and wakes them up by adding them to the scheduler ready queue

The major differences between semaphores locks and condition variables are that the counter values of ***semaphores*** can be incremented or decremented by any thread which calls it , so basically any thread can alter the counter value of semaphores

But for ***locks*** only the thread which acquire the lock can release it and the rest are blocked if they try to acquire .

Condition variables are used when a thread needs to wait for the output or data processed by another thread . In this case the thread which needs to wait for a data from another thread blocks itself by calling Wait() . When the other thread is done with its processing it signals the blocked thread by calling Signal() or Broadcas () (if necessary) . For example take Slow producer and Fast Consumer(needs to wait) .

We have tested these synchronization primitives by wiriting certain test case like prodcon.c, multprodcon.c, Locktest.c, condtest.c

Note : while implementing all the synchronization primitives we made sure operations are atomic by disabling the interrupts.

Test Programs:

threadtest.c

A Simple program to test the threads. We are creating a new thread from the main. Now the main thread and the new thread will randomly print the value of the argument.

Output:

```
Entering SimpleTest*** thread 4201088 looped 0 times
*** thread 2 looped 0 times
*** thread 4201088 looped 1 times
*** thread 2 looped 1 times
*** thread 4201088 looped 2 times
*** thread 2 looped 2 times
*** thread 4201088 looped 3 times
*** thread 2 looped 3 times
*** thread 4201088 looped 4 times
*** thread 2 looped 4 times
```

prodcon1.c

A Program for simple producer/consumer using semaphores and threads (to test basic p and v). We are creating two threads for a producer and a consumer. We are creating three semaphores emptybuf, fullbuf and a mutexuf for producer and consumer and protecting their shared variable while writing to it.

Output:

```
Starting thread
main started
Starting thread
Starting thread
main done
Producer created
Consumer created
for 5 iterations, the total is 10
```

multprodcon.c

A program creating multiple producer and consumer using 3 semaphores. We are using 3 buffers, 2 producer and 2 consumer. we are also using the Join() function in the main to run the producer and consumer.

Output:

produce 30
produce 31
produce 30
Consume 30
Consume 31
Consume 30
produce 32
produce 33
produce 31
Consume 32
Consume 33
Consume 31
produce 32
produce 33
Consume 32
Consume 33

locktest.c

A Program for testing the locks. We are creating 3 threads with three corresponding functions. First thread is acquiring the lock and holding the shared variable, the second one is changing the value of a shared variable and releasing the lock but it will give the error message since he is not the owner of the lock and hence he will not be able to change the shared variable. Now, the third will also not able to change the variable since it is locked.

Output:

Starting thread

Starting thread
value of shared var x in main is 1
value of x inside test1 is 10
value of x inside test2 is 10Error: Thread (test2) is not the lock owner (test1)!

value of x inside test3 is 10No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
Cleaning up...

condtest.c

A program to test the conditions. We are testing for condition wait and condition broadcast. We have created 3 threads corresponding to 3 functions. first and second function will wait and the third function will broadcast so that the other two functions will be able to execute its code.

Output:

Starting thread
Inside the first test now waiting for test3 to give latest value of y

Inside the second test now waiting for test3 to give latest value of y

Test3 : Broadcasting
value of x inside Test3 is 11
Test 1 is not waiting now and value of y is 31
Test 2 is not waiting now and value of y is 41 No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

signaltest.c

A program to test the condition Signal. We have created two threads and two functions corresponding to it. First function will wait and the second function will signal the first function. But here, when we are trying to call the signal it is not able to identify the waiting thread and is printing “No threads to signal” which is printed when there is no thread in the waiting queue.

Output:

Starting thread

Starting thread
Inside the first test now waiting for test2 to give latest value of y

value of x inside Test2 is 11Error: No threads to signalNo threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!