Project 1

Draft Due: 6/29/15

CSE 150- Operating Systems

Landon Taylor

Lenny Khoden

Isaac Flores

Andrew Yfantis

1. Implement KThread.join(). A KThread can be used to execute Nachos kernel code. Kthread.join() is a function that checks the current thread to see if it is completed.

It would make sense to use a condition variable to ensure the system keeps track of available resources.

There are two test cases: call join on a thread that is terminated, and call join on a thread that is the current thread. The final result is that the parent thread will wait until the targeted thread to complete.

/\* Waits for this thread to finish. If this thread is already finished, return immediately. This method must only be called once; the second call is not guaranteed to return. This thread must not be the current thread. \*/

**public** **void** join() {

if(currentThread is completed)

{

Return;

}

Lib.debug(dbgThread, "Joining to thread: " + toString());

Lib.assertTrue(this != currentThread);

Join\_count++

disable interrupts

save currentTCB

put current thread into wait queue

Re-enable interrupts

}

1. Implement condition variables directly, without semaphores. This is to be done in Condition2. This function disables interrupts for synchronization.

Our test cases would be running functions on a various amount of threads and checking for reasonable output (order of execution, data issues).

Public void sleep()

{

Disable interrupts;

While(current thread exists)

{

Wait to be current thread;

}

//do things until done

//when done go to waiting

Signal ( waiting to be ready. )

Enable interrupts;

}

Public void wake(){

Disable interrupts;

If( something is sleeping)

{

Wake one up; ////put in ready queue=wake

}

Enable interrupts;

}

Public void wakeall(){

Disable interrupts;

If( anything is sleeping)

{

Wake all up; //put in ready queue=wake

}

Enable interrupts;

}

1. Complete the implementation of the Alarm class by the waitUntil(long x). This method puts the current thread to sleep for x ticks, and wakes it in the timer interrupt handler.

This class will also be tested with a various amount of threads, with different wait times. We then check and see if the times are reasonable with the current time. We can check the interrupt method by specific calls and checking for reasonable output.

waitUntil( long x)

{

wakeTime = currentTime + x;

while(wakeTime > currentTime)

{

Disable interrupts

Save current TCB

Current thread put into waitQueue

Restore interrupts

}

}

timerInterrupt()

{

kThread.currentThread().yield();

disable interrupts;

//put current into readyQueue

restore interrupts;

}

1. Implement synchronous send and receive of one word messages, using **condition variables**. Implement the Communicator class with operations:void speak (int word) and int listen().

We have to test listen if nothing has spoken. We will check how listening works with two speakers. We will test how two speakers and two listeners can interact.

Int reading=0;

Int writing=0;

Public void speak(int word){

Disable interrupts

While( reading | writing ){sleep}

//Writing++

Enable interrupts

//write pseudocode

Disable interrupts

//writing--

Signal() ; //give up turn

Enable interrupts

}

Public int listen(){

Disable interrupts

While( writing ){sleep;}

//reading++

Enable interrupts

//reading pseudocode;

Disable interrupts

//Reading --

Signal() ; //give up turn

Enable interrupts

}

1. Implement priority scheduling in Nachos by completing PriorityScheduler class. It is necessary to change nachos.conf that specifies the scheduler class to use.

Having a ranking with priorities will require tests with a larger amount of threads with a range of priority. There should be cases where higher priority are dependent on lower priority threads. Another case is to have multiple threads with the same priority level.

The levels given are 0-7, with a default of 1.

public KThread nextThread() {

Lib.assertTrue(Machine.interrupt().disabled());

// implement me

return null;

}

protected ThreadState pickNextThread() {

// implement me

return null;

}

public void print() {

Lib.assertTrue(Machine.interrupt().disabled());

// implement me (if you want)

}

public int getEffectivePriority() {

// implement me

return priority;

}

public void setPriority(int priority) {

if (this.priority == priority)

return;

this.priority = priority;

// implement me

}

public void waitForAccess(PriorityQueue waitQueue) {

// implement me

}

public void acquire(PriorityQueue waitQueue) {

// implement me

}

1. The boat problem. Many people are stuck on the island of Oahu (at least two children), and need to go to the island Molokai on a single boat. This boat can only hold up to two children, or one adult. Example 3 people waiting:
2. Two children row to Molokai
3. One child rows back to Oahu
4. One adult rows to Molokai
5. The child that was left rows back to Oahu
6. Both children go to Molokai (done)

We can see there needs to be a loop where if an adult crosses, there must be a child waiting at Molokai to bring the boat back, if the adult was not the last person on Oahu.

Our test cases must result in the same amount of adults and children that start and end, and they move across within the defined parameters. Having an initial 2 children makes the variable of other threads a fairly straightforward.

static void AdultItinerary()

{

If(child at Molokai)

{

//get in boat as adult

//row across

//get out of boat

}

//countAdultMolokai++

//countAdultOahu--

}

static void ChildItinerary()

{

//one child keeps bringing other children across

//brings back for adult(s) near end of completion.

While( people at oahu)

{

If( at oahu)

{

If( boatempty)

//join as passenger || pilot //(is one more preferable?)

If( boatpassenger)

{

//Join as pilot

//row to molokai

}

If( pilot)

//Join as passenger

}

Else //at Molokai

{

If( amPassenger)

{

//get off of boat

countChildOahu--;

countChildMolokai++;

}

If(amPilot)

{

//make sure there are no passengers

//pilot to oahu

}

}

}