Operating System

Design Doc for Project 1

Instructed by *Xu Wei*

Chen Xiaoqi, Ko Lok Sun, Wu Yijie, Zhang Hanrui

Due on Mar. 31, 2015

Thread System Design

TASK I: KThread.join()

Description

Current thread call thread1.join(), and wait thread1 finish Notice that Current thread != thread1.

Requirment

1. No busy waiting
2. Return immediately when finished.
3. join() is called at most once for each thread object

Solution

We can add `waitQueue` to hold threads that have to wait until this thread is finished. And move the threads in waitQueue to readyQueue when this thread is done. Since the threads in waitQueue are in status blocked, scheduler will not dispatch them until this thread is finished. Note that there are two prerequisites:

1. This thread can not be currentThread, and
2. The waitQueue has been initialized before `join()` is called (same as readyQueue).

Pseudocode

In KThread.join():

```
if(this.status != finished){
  add currentThread to waitQueue
  currentThread.sleep() //cause it to be blocked
}
```

In KThread.finish()

```
 while(waitQueue has nextThread){
    nextThread.ready()
    remove nextThread from waitQueue
 }
 currentThread.status = finished //existing code
 sleep()
```

Question

1. Do we need to stop interrupt?
2. Do we need to do some clean up? (tcb will be destroy in restoreState())
3. Do we need to handle priority donation here?

TASK II: Condition Variables

Description

Implement directly without semaphore. Lock and interrupt disable can be used

Requirement

1. No busy waiting
2. Sleeper must no miss the wake up. (No preempt between wake)

Solution

We complete the following three functions of `Condition2` class: `sleep()`, `wake()`, `wakeAll()`.

Pseudocode

```
constructor:
  using the queue in scheduler to solve priority inversion problem
sleep(){

  release conditional lock
  disable interrupt
  add this thread to queue
  this thread sleep
  enable interrupt
  acquire conditional lock

}
wake(){
  disable interrupt
  if(queue not empty){
    queue.removeFirst().ready
  }
  enable interrupt
}
wakeAll(){
  while(queue not empty){
    queue.removeFirst().ready()
  }
}
```

Question

1. How to make sure the sleeper only waked by wake call?
2. Will the 'enable machine interrupt' in sleep() crash the OS?
3. Do we need to 'disable machine interrupt' in wakeAll()?

TASK III: Alarm Class

Description

Reimplement the waitUntil call without using busy waiting.

Requirement

1. No busy waiting.
2. Put the threads in ready queue after they wait AT LEAST (NOT EXACTLY) the right amount of time.
3. waitUntil will be called by more than one thread (So we need array / linked list)

Solution

The class maintains a list of upcoming threads-to-wake, in time sequence, and add new items in corresponding order.

Upon each timer interrupt arrives, we scan the list to remove and wake up the expired items.

Use heap to maintain the priority queue, to quickly insert and remove.

Psuedocode

```
construct:
    Create a heap of pairs (thread, time)

waitUtil:
    put (thread, waitTime) in the heap;
    thread.yield()

timerInerrupt:
    currentThread.yield()
    while(heap top's waitTime > currentTime)
        thread=heap.pop()
        thread.ready()
```

Question

1. Do we need to sort the linked list?
2. Is it ok without using machine interrupt.disable()

TASK IV: Synchronous Send/Receive

Description

one-word messages.

Requirement

1. No busy waiting
2. No message buffer
3. Send only return after a reader read the message

Solution

This task is similar to the producer-consumer model. The sender will wait upon receiver if message buffer occupied, and wake up a receiver after storing message; the receiver will try waking up a sender if there's no message, then wait upon sender if no message available.

Psuedocode

```
Lock lock;
Condition Variable send(lock),receive(lock)
Word buffer;

Send(message):
    lock.acquire();
        while(buffer!=null)
            send.await();
        buffer=message;
        receive.signal();
    lock.release();

Receive():
    lock.acquire();
        if(buffer==null)
            send.signal();
        while(buffer==null)
            receive.await();
        tmp=buffer;
        buffer=null;
        send.signal();
    lock.release();
    return tmp;
```

TASK V: PriorityScheduler Class

Description

Implement priority scheduling in Nachos.

Requirement

1. No busy waiting
2. Solve priority inversion problem

Solution

We use a heap to construct a priority queue of threads sorted by priority, to realise insertion and pop-maximum in log(n) time.

Each time a lock allocates a queue, it is also monitored by the scheduler; we find the lock acquiring relationship between threads by these queue, and calculate the new priority after priority donation.

Use the heap to find the appropriate next thread with highest priority.

Psuedocode

```
PriorityQueue:

  constructor:
    initialize the heap (Using newThreadQueue(True) to create)
  nextThread:
    pop the thread on heap top, then
    return the popped thread
  pickNextThread:
    return the thread on heap top
ThreadState:
  getEffectivePriority:
    scan the thread donation relation tree,
        and find the parent element with maximum priority;
    return that priority.
  wait for access:
    insert this thread to the priority queue
    if queue.priorityTransfer:
      donate the priority to the thread
  acquire:
    (everytime a thread got a resource,
        say lock , samephores or conditional variable)
    (this function will be called)
    (Tell the scheduler which thread will receive donation)
```

TASK VI: Boat Problem

Assumptions

First here comes our assumptions on the intelligence of people on the islands. It is reasonable that they should be able to figure out the island they are currently on, whether they are on the boat or not, the state of the boat. They should be able to follow some universal strategy for each child and adult. Also we reasonably assume that everyone on the islands have the ability to do simple arithmetic operations. That is, each one of them is able to:

1. Count the number of adults and children on his current island;
2. Decrease the corresponding number when seeing people leave his island;
3. Increase the corresponding number when seeing people arriving at his island.

All the arithmetic operations above would be implemented through operations on shared variables done by each thread individually.

Strategies

The strategies are stated as below:

1. For children, if they detect that there are more than 1 children on the "source" island, they try to board the boat and be the driver. If there is exactly 1 child, try to board as rider. Each time a boat reaches the "destination" island, they try to board the boat and set off then there is 1 child aboard. If there is only 1 child on the "source" island, he simply board as the driver and set off.
2. For adults, if they are on the "destination" island, or there are more than 1 children on the "source" island, they do nothing. Otherwise they try to board the boat and set off immediately.

When the procedure finalizes, the parent thread could be informed by observing that the number of children and adults on the "source" island becomes 0. Note that this is indeed an one-way communication, for that the parent thread never tries to modify that value. In fact, the parent thread do literally nothing except for starting and terminating the whole procedure.

Note that adults should not try to depart before at least two children has departed; otherwise, the boat cannot be driven back; also, when there's only one children, it should yield to adults for boarding.

Pseudocode

Peudocode comes as below (for convenience, assume that all the operations done to integer variables are atomic):

```
#Shared variables:

state of the boat
Integer adultsOnSource, childrenOnSource, childrenAboard
lock boarding, childrenUnload
corresponding condition variable bd, cu

Parent:
    start adults and children
    wait(0)
```

```
    while adultsOnSource + childrenOnSource > 0:
        wait()
    return

Adult:
    increase adultsOnSource
    while self on source:
        boarding.acquire()
            while childrenAboard > 0 or childrenOnSource > 1 or boat at dest:
                bd.wait()
            decrease adultsOnSource
            adultToDest()
            bd.signal()
        boarding.release()
    return

Child:
    increase childrenOnSource
    while true:
        while self on dest:
            boarding.acquire()
                while childrenAboard > 0 or adultsOnSource == 0 or boat at source:
                    bd.wait()
                childDriveToSource()
                increase childrenOnSource
                bd.signal()
            boarding.release()
        while self on source:
            boarding.acquire()
                increase childrenAboard
                isDriver = 1
                while childrenAboard == 1 and adultOnSource > 0:
                    if childrenOnSource == 1:
                        decrease childrenAboard
                        bd.wait()
                        increase childrenAboard
                    else:
                        isDriver = 0
                        bd.wait()
                bd.signal()
            boarding.release()
            if isDriver == 1:
                childrenUnload.acquire()
                    childDriveToDest()
                    decrease childrenOnSource
                    decrease childrenAboard
                    cu.signal()
                childrenUnload.release()
            else:
                childrenUnload.acquire()
                    while childrenAboard > 1:
                        cu.wait()
```

```
                childRideToDest()
                decrease childrenOnSource
                childDriveToSource()
                increase childrenOnSource
                decrease childrenAboard
            childrenUnload.release()
    return
```

Test Case

Task I

1. A traditional thread pool, which creates many thread and joins each of them once.
2. An erroneous scheduler who joins a thread twice.
3. A chain of thread each joining the subsequent one, until a certain depth, creating a chain of waiting. The last thread exits after some sleep/delay, and all other threads should also quit properly after the delay.

Task II

1. A database reader/writer application similar to the in-class example;
2. A bank account transaction model, let teller waiting for deposit, such that no two threads operate on the balance simultaneously and at any time the balance is nonnegative.
3. A signal/broadcaster model where the slept process is waken up immediately after a broadcast. Test such that no thread overslept.

Task III

1. Test the offset between expected sleeping time and actual slept time.
2. Implement SleepSort in user space.

Task IV

1. 5 Receivers wait and 2 senders came; after the racing condition, only 2 receivers got the message
2. 5 senders wait and 2 receivers came; after the racing condition, only 2 senders succeed
3. 50 senders and 50 receivers came; no one is left stuck after the racing condition.

Task V

1. Make priority inversion case and the system should not deadlock.
2. Make a reader/writer case and cause a livelock, to see the system is properly stucked (i.e. the writer is livelocked)
3. Implement PrioritySort in user space.

Task VI

1. Test the scope of information is proper.
2. Test the case when there's only many chindren, but no adult.
3. Test the case when there's only one children and one adult, and see if they depart simultaneously.
4. Test when there's no children and system should deadlock.

References

1. Example design doc