CSCI 509 - Operating Systems, Fall 2019
Assignment 3: Kernel Resource Management

Due Date: Tuesday, October 22, 2019
Points: 125

1 Overview and Goal

In this assignment, you will implement three monitors that will be used in our Kernel. These are the ThreadManager, the ProcessManager, and the FrameManager. The code you write will be similar to other code from the previous assignments in that these three monitors will orchestrate the allocation and freeing of resources.

2 Download New Files

This assignment starts the development of your kernel. Check out your master branch. Create a new directory called "os" on your master branch and you will work in this directory for the rest of the project. **DO NOT** create an "a3" directory. You should do all your development for Assignment 3 through 7 on branch "master" in the directory "os". When you finish this assignment, you will create the branch "a3" for a turn-in branch. But to start with, you should get the files at the following URL and add them to branch "master" in the "os" directory. *Do not create the "a3" branch until you are done with this assignment.*

The URL is `https://facultyweb.cs.wwu.edu/~phil/classes/f19/509/a3`

Even though some of the files have the same names, be sure to get new copies for this assignment. Put all these files in your "os" directory. In general some files may be modified. For this assignment, you should get the following files:

```
makefile
DISK
Runtime.s
Switch.s
System.h
System.k
List.h
List.k
BitMap.h
BitMap.k
Main.h
Main.k
Kernel.h
Kernel.k
```

The packages called Thread and Synch have been merged together into one package, now called Kernel. This package contains quite a bit of other material as well, which will be used for

later assignments. In this and the remaining assignments, you will be modifying the Kernel.k and Kernel.h files. Don't modify the code that is not used in this assignment; just leave it in the package.

The Kernel.k file contains the following stuff, in this order:

```
Thread scheduler functions
Semaphore class
Mutex class
Condition class
Thread class
ThreadManager class
ProcessControlBlock class
ProcessManager class
FrameManager class
AddrSpace class
TimerInterruptHandler
other interrupt handlers
SyscallTrapHandler
Handle functions
```

In this assignment, you can ignore everything after TimerInterruptHandler. The classes called ThreadManager, ProcessManager, and FrameManager are provided in outline, but the bodies of the methods are unimplemented. You will add implementations. Some other methods are marked "unimplemented;" those will be implemented in later assignments.

The BitMap package contains code you will use; read over it but do not modify it.

The makefile has been modified to compile the new code. As before, it produces an executable called os.

You may modify the file Main.k while testing, but when you do your final run, please use the Main.k file as it was distributed. (In other words, don't commit any changes to Main.k to your git repo.) In the final version of our kernel, the Main package will perform all initialization and will create the first thread. The current version performs initialization and then calls some testing functions.

3 Task 1: Threads and the ThreadManager

In this task, you will modify the ThreadManager class and provide implementations for the following methods:

- Init
- GetANewThread
- FreeThread

In our kernel, we will avoid allocating dynamic memory. In other words, we will not use the heap. (Kernel.k will not use the KPL "alloc" feature.) All important resources will be created at startup time and then we will carefully monitor their allocation and deallocation.

An example of an important resource is Thread objects. Since we will not be able to allocate new objects on the heap while the kernel is running, all the Thread objects must be created ahead of time. Obviously, we can't predict how many threads we will need, so we will allocate a fixed number of Thread objects (e.g., 10) and re-use them.

When a user process starts up, the kernel will need to obtain a new Thread object for it. When a process dies, the Thread object must be returned to a pool of free Thread objects, so it can be recycled for another process.

Kernel.h contains the line:

```
const MAX_NUMBER_OF_PROCESSES = 10
```

Since each process in our OS will have at most one thread, we will also use this number to determine how many Thread objects to place into the free pool initially.

To manage the Thread objects, we will use the ThreadManager class. There will be only one instance of this class, called threadManager, and it is created and initialized at startup time in Main.k.

Whenever you need a new Thread object, you can invoke threadManger.GetANewThread. This method should suspend and wait if there are currently none available. Whenever a thread terminates, the scheduler will invoke FreeThread. In fact, the Run function has been modified in this assignment to invoke FreeThread when a thread terminates-thereby adding it to the free list-instead of setting its status to UNUSED.

Here is the definition of ThreadManager as initially distributed:

```
class ThreadManager
  superclass Object
  fields
    threadTable: array [MAX_NUMBER_OF_PROCESSES] of Thread
    freeList: List [Thread]
  methods
    Init ()
    Print ()
    GetANewThread () returns ptr to Thread
    FreeThread (th: ptr to Thread)
endClass
```

When you write the Init method, you'll need to initialize the array of Threads and you'll need to initialize each Thread in the array and set its status to UNUSED. (Each Thread will have one of the following as its status: READY, RUNNING, BLOCKED, JUST_CREATED, and UNUSED.) Threads should have the status UNUSED iff they are on the freeList. You'll also need to initialize the freeList and place all Threads in the threadTable array on the freeList during initialization.

You will need to turn the ThreadManager into a "monitor." To do this, you might consider adding a Mutex lock (perhaps called threadManagerLock) and a condition variable (perhaps called aThreadBecameFree) to the ThreadManager class. The Init method will also need to initialize threadManagerLock and aThreadBecameFree.

The GetANewThread and FreeThread methods are both "entry methods," so they must obtain the monitor lock in the first statement and release it in the last statement.

GetANewThread will remove and return a Thread from the freeList. If the freeList is empty, this method will need to wait on the condition of a thread becoming available. The FreeThread method will add a Thread back to the freeList and signal anyone waiting on the condition.

The GetANewThread method should also change the Thread status to JUST_CREATED and the FreeThread method should change it back to UNUSED.

We have provided code for the Print method to print out the entire table of Threads.

Note that the Print method disables interrupts. The Print method is used only while debugging and will not be called in a running OS so this is okay. Within the Print method, we want to get a clean picture of the system state-a "snapshot"-(without worrying about what other threads may be doing) so disabling interrupts seems acceptable. However, the other methods-Init, GetAThread and FreeThread-must NOT disable interrupts, beyond what is done within the implementations of Mutex, Condition, etc.

In Main.k we have provided a test routine called RunThreadManagerTests, which creates 20 threads to simultaneously invoke GetAThread and FreeThread. Let's call these the "testing threads" as opposed to the "resource threads," which are the objects that the ThreadManager will allocate and monitor. There are 20 testing threads and only 10 resource thread objects.

Every thread that terminates will be added back to the freeList (by Run, which calls FreeThread). Since the testing threads were never obtained by a call to GetANewThread, it would be wrong to add them back to the freeList. Therefore, each testing thread does not actually terminate. Instead it freezes up by waiting on a semaphore that is never signaled. By the way, the testing threads are allocated on the heap, in violation of the principle that the kernel must never allocate anything on the heap, but this is okay, since this is only debugging code, which will not become a part of the kernel.

In the kernel, we may have threads that are not part of the threadTable pool (such as the IdleThread), but these threads must never terminate, so there is no possibility that they will be put onto the freeList. Thus, the only things on the freeList should be Threads from threadTable.

You will also notice that the Thread class has been changed slightly to add the following fields:

```
class Thread
  ...
  fields
    ...
    isUserThread: bool
    userRegs: array [15] of int    -- Space for r1..r15
    myProcess: ptr to ProcessControlBlock
  methods
```

```
    ...
endClass
```

These fields will be used in a later assignment. The Thread methods are unchanged.

4 Task 2: Processes and the ProcessManager

In our kernel, each user-level process will contain only one thread. For each process, there will be a single ProcessContolBlock object containing the per-process information, such as information about open files and the process's address space. Each ProcessControlBlock object will point to a Thread object and each Thread object will point back to the ProcessControlBlock.

There may be other threads, called "kernel threads," which are not associated with any user-level process. There will only be a small, fixed number of kernel threads and these will be created at kernel start-up time.

For now, we will only have a modest number of ProcessControlBlocks, which will make our testing job a little easier, but in a real OS this constant would be larger.

```
const MAX_NUMBER_OF_PROCESSES = 10
```

All processes will be preallocated in an array called processTable, which will be managed by the ProcessManager object, much like the Thread objects are managed by the ThreadManager object.

Each process will be represented with an object of this class:

```
class ProcessControlBlock
  superclass Listable
  fields
    pid: int
    parentsPid: int
    status: int                    -- ACTIVE, ZOMBIE, or FREE
    myThread: ptr to Thread
    exitStatus: int
    addrSpace: AddrSpace
    fileDescriptor: array [MAX_FILES_PER_PROCESS] of ptr to OpenFile
  methods
    Init ()
    Print ()
    PrintShort ()
endClass
```

Each process will have a process ID (the field named pid). Each process ID will be a unique number, from 1 on up.

Processes will be related to other processes in a hierarchical parent-child tree. Each process will know who its parent process is. The field called parentsPid is a integer identifying the

parent. One parent may have zero, one, or many child processes. To find the children of process X, we will have to search all processes for processes whose parentsPid matches X's pid.

The ProcessControlBlock objects will be more like C structs than full-blown C++/Java objects: the fields will be accessed from outside the class but the class will not contain many methods of its own. Other than initializing the object and a couple of print methods, there will be no other methods for ProcessControlBlock. We are providing the implementations for the Init, Print and PrintShort methods.

Since we will have only a fixed, small number of ProcesControlBlocks, these are resources which must be allocated. This is the purpose of the monitor class called ProcessManager.

At start-up time, all ProcessControlBlocks are initially FREE. As user-level processes are created, these objects will be allocated and when the user-level process dies, the corresponding ProcessControlBlock will become FREE once again.

In Unix and in our kernel, death is a two stage process. First, an ACTIVE process will execute some system call (e.g., Exit()) when it wants to terminate. Although the thread will be terminated, the ProcessControlBlock cannot be immediately freed, so the process will then become a ZOMBIE. At some later time, when we are done with the ProcessControlBlock it can be FREEd. Once it is FREE, it is added to the freeList and can be reused when a new process is begun.

The exitStatus is only valid after a process has terminated (e.g., a call to Exit()). So a ZOMBIE process has a terminated thread and a valid exitStatus. The ZOMBIE state is necessary just to keep the exit status around. The reason we cannot free the ProcessControlBlock is because we need somewhere to store this integer.

For this assignment, we will ignore the exitStatus. It need not be initialized, since the default initialization (to zero) is fine. Also, we will ignore the ZOMBIE state. Every process will be either ACTIVE or FREE.

Each user-level process will have a virtual address space and this is described by the field addrSpace. The code we have supplied for ProcessControlBlock.Init will initialize the addrSpace. Although the addrSpace will not be used in this assignment, it will be discussed later in this document.

The myThread field will point to the process's Thread, but we will not set it in this assignment.

The fileDescriptors field describes the files that this process has open. It will not be used in this assignment.

Here is the definition of the ProcessManager object.

```
class ProcessManager
  superclass Object
  fields
    processTable: array [MAX_NUM_OF_PROCESSES] of ProcessControlBlock
    processManagerLock: Mutex
    aProcessBecameFree: Condition
    freeList: List [ProcessControlBlock]
    aProcessDied: Condition
  methods
```

```
    Init ()
    Print ()
    PrintShort ()
    GetANewProcess () returns ptr to ProcessControlBlock
    FreeProcess (p: ptr to ProcessControlBlock)
    TurnIntoZombie (p: ptr to ProcessControlBlock)
    WaitForZombie (proc: ptr to ProcessControlBlock) returns int
endClass
```

There will be only one ProcessManager and this instance (initialized at start-up time) will be called processManager.

```
    processManager = new ProcessManager
    processManager.Init ()
```

The Print() and PrintShort() methods for ProcessControlBlocks are provided for you. You are to implement the methods Init, GetANewProcess, and FreeProcess. The methods TurnIntoZombie and WaitForZombie will be implemented in a later assignment and can be ignored for now.

The freeList is a list of all ProcessControlBlocks that are FREE. The status of a ProcessControlBlock should be FREE if and only if it is on the freeList.

We assume that several threads may more-or-less simultaneously request a new ProcessControlBlock by calling GetANewProcess. The ProcessManager should be a "monitor," in order to protect the freeList from concurrent access. The Mutex called processManagerLock is for that purpose. When a ProcessControlBlock is added to the freeList, the condition aProcessBecameFree can be Signaled to wake up any thread waiting for a ProcessControlBlock.

Initializing the ProcessControlManager should initialize

- the processTable array
- all the ProcessControlBlocks in that array
- the processManagerLock
- the aProcessBecameFree and the aProcessDied condition variables
- the freeList

All ProcessControlBlocks should be initialized and placed on the freeList.

The condition called aProcessDied is signaled when a process goes from ACTIVE to ZOMBIE. It will not be used in this assignment, but should be initialized nonetheless.

The GetANewProcess method is similar to the GetANewThread method, except that it must also assign a process ID. In other words, it must set the pid. The ProcessManager will need to manage a single integer for this purpose. (Perhaps you might call it nextPid). Every time a ProcessControlBlock is allocated (i.e., everytime GetANewProcess is called), this integer

must be incremented and used to set the process's pid. GetANewProcess should also set the process's status to ACTIVE.

The FreeProcess method must change the process's status to FREE and add it to the free list.

Both GetANewProcess and FreeProcess are monitor entry methods.

5 Task 3: The Frame Manager

The lower portion of the physical memory of the BLITZ computer, starting at location zero, will contain the kernel code. It is not clear exactly how big this will be, but we will allocate 1 MByte for the kernel code. After that will come a portion of memory (called the "frame region") which will be allocated for various purposes. For example, the disk controller may need a little memory for buffers and each of the user-level processes will need memory for "virtual pages."

The area of memory called the frame region will be viewed as a sequence of "frames". Each frame will be the same size and we will have a fixed number of frames. For concreteness, here are some constants from Kernel.h.

```
PAGE_SIZE = 8192                                  -- in hex: 0x00002000
PHYSICAL_ADDRESS_OF_FIRST_PAGE_FRAME = 1048576 -- in hex: 0x00100000
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512              -- in hex: 0x00000200
```

This results in a frame region of 4 MB, so our kernel would fit into a 5 MByte memory.

The frame size and the page size are the same, namely 8K. In later assignments, each frame will hold a page of memory. For now, we can think of each frame as a resource that must be managed. We will not really do anything with the frames. This is similar to the dice in the gaming parlor and the forks for the philosophers... we were concerned with allocating them to threads, but didn't really use them in any way.

Each frame is a resource, like the dice of the game parlor, or the philosophers' forks. From time to time, a thread will request some frames; the frameManager will either be able to satisfy the request, or the requesting thread will have to wait until the request can be satisfied.

For the purposes of testing our code, we will work with a smaller frame region of only a few frames. This will cause more contention for resources and stress our concurrency control a little more. (For later assignments, we can restore this constant to the larger value.)

```
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 27            -- For testing only
```

Here is the definition of the FrameManager class:

```
class FrameManager
  superclass Object
  fields
    framesInUse: BitMap
    numberFreeFrames: int
    frameManagerLock: Mutex
```

```
    newFramesAvailable: Condition
  methods
    Init ()
    Print ()
    GetAFrame () returns int          -- returns addr of frame
    PutAFrame ( frameAddr: int )      -- done using a frame
    GetNewFrames (aPageTable: ptr to AddrSpace, numFramesNeeded: int)
    ReturnAllFrames (aPageTable: ptr to AddrSpace)
endClass
```

There will be exactly one frameManager object, created at kernel start-up time.

```
frameManager = new FrameManager
frameManager.Init ()
```

With frames (unlike the ProcessControlBlocks) there is no object to represent each resource. So to keep track of which frames are free, we will use the BitMap package. Take a look at it. Basically, the BitMap class gives us a way to deal with long strings of bits. We can do things like (1) set a bit, (2) clear a bit, and (3) test a bit. We will use a long bit string to tell which frames are in use and which are free; this is the framesInUse field. For each frame, there is a bit. If the bit is 1 (i.e., is "set") then the frame is in use; if the bit is 0 (i.e., is "clear") then the frame is free.

The frameManager should be organized as a "Monitor class." The frameManagerLock is used to make sure only one method at a time is executing in the FrameManager code.

We have provided the code for the Init, Print, and GetAFrame methods; you'll need to implement PutAFrame, GetNewFrames and ReturnAllFrames.

The method GetANewFrame allocates one frame (waiting until at least one is available) and returns the address of the frame. The method PutAFrame returns a single frame. These will be used later when the kernel needs a frame for kernel work that is not directly associated with an address space.

The GetNewFrames method gets a number of frames for an AddrSpace and needs to make a note of which frames have been allocated. It does this by storing the address of each frame it allocates (the address of the first byte in each frame) into an AddrSpace object.

An AddrSpace object is used to represent a virtual address space and to tell where in physical memory the virtual pages are actually located. For example, for a virtual address space with 10 pages, the AddrSpace object will contain an ordered list of 10 physical memory addresses. These are the addresses of the 10 "frames" holding the 10 pages in the virtual address space. However, the AddrSpace object contains more information. For each page, it also contains information about whether the page has been modified, whether the page is read-only or writable, etc. The information in an AddrSpace object is stored in exactly the format required by the CPU's memory management hardware. In later assignments, this will allow us to use the AddrSpace object as the current page table for a running user-level process. At that time, when we switch to a user-level process, we'll have to tell the CPU which AddrSpace object to use for its page table. In addition to looking over the code in AddrSpace, you may want to review the BLITZ architecture manual's discussion of page tables.

The code in method

GetNewFrames (aPageTable: ptr to AddrSpace, numFramesNeeded: int)

needs to do the following:

1. Acquire the frame manager lock.
2. Wait on newFramesAvailable until there are enough free frames to satisfy the request.
3. Do a loop for each of the frames

   for i = 0 to numFramesNeeded-1

   (a) determine which frame is free (find and set a bit in the framesInUse BitMap)

   (b) figure out the address of the free frame

   (c) execute aPageTable.SetFrameAddr (i, frameAddr) to store the address of the frame which has been allocated
4. Adjust the number of free frames
5. Set aPageTable.numberOfPages to the number of frames allocated.
6. Unlock the frame manager

The code in method

```
ReturnAllFrames (aPageTable: ptr to AddrSpace)
```

needs to do more or less the opposite. It can look at aPageTable.numberOfPages to see how many are being returned. It can then go through the page table and see which frames it possessed. For each, it can clear the bit.

```
for i = 0 to numFramesReturned-1
  frameAddr = aPageTable.ExtractFrameAddr (i)
  bitNumber = ...frameAddr...
  framesInUse.ClearBit(bitNumber )
endFor
```

It will also need to adjust the number of free frames and "notify" any waiting threads that more frames have become available. PutAFrame is similar, except this is used where the Kernel needs a frame that is not part of a user page table. These frames are returned one at a time and will also need to "notify" any waiting threads that one more frame has become available.

You'll need to do a Broadcast, because a Signal will only wake up one thread. The thread that gets awakened may not have enough free frames to complete, but other waiting threads may be able to proceed. A broadcast should be adequate, but perhaps after carefully studying the Game Parlor problem, you will find a more elegant approach which wakes up only a single thread.

Also note that there is a possibility of starvation here. It is possible that one large process will be waiting for a lot of frames (e.g., 100 frames). Perhaps there are many small processes which free a few frames here and there, but there are always other small processes that grab those frames. Since there are never more than a few free frames at a time, the big process will get starved.

This particular scenario for starvation, (where processes are competing for frames) is a very real danger in an OS and a "real" OS would need to ensure that starvation could not happen. However, in our situation, it is acceptable to provide a solution that risks starvation.

The test program for the frame manager tests all functions you need to write. There is a variable called testPutAFrame that may be set to false so you test only GetNewFrames and ReturnAllFrames first. Once you get those working, you can set the testPutAFrame variable to true so the loop also tests PutAFrame. Your final tests must test all three functions.

6 Modifications

Do not commit modifications to any file except for "Kernel.h" and "Kernel.k".

Do not create global variables (except for testing purposes). Do not modify the methods we have provided. Do not modify the code for the AddrSpace class.

7 Relation to future assignments

The previous assignments were designed to get you up to speed and to let you know how much time the programming portion of this class will take. The previous assignments were independent, in the sense your solution code did not need to be 100% correct in order to continue with the BLITZ kernel assignment.

Things change with this assignment. Buggy code in this assignment is not good and will make it very hard to complete the future assignments.

In particular, you must complete this assignment before you can begin the next assignment. You must get your code working and pass the test programs before going on, unlike the previous assignments in which successful completion was not a barrier to moving on.

If you are unable to complete this assignment before the late due date, you will still need to keep working on them until your code works 100% correctly. If you find yourself having difficulty in completing this assignment on time, you should put some extra effort into catching up to the schedule. Continual late or super late (meaning worthless) assignments will impact all remaining assignments as you must complete each assignment in order to be ready for the next one. Please make sure you request help from your professor if you find yourself in this situation. It very important for your assignment grades to get these working.

8 What to Hand In

Once you have your code *working on master*, then create a new "a3" branch as your turn-in for this assignment. Remember to push your "a3" branch as you have done before. Please make

sure that you do not commit changes to Main.k or other files you change in your testing. Only the files Kernel.h and Kernel.k should be changed.

Please make a script file of a run of your working kernel. Name the file "a3-script" and commit that file in your "a3" branch. Put the file in the root directory of your gitlab project, not in the "os" directory.

Create a cover sheet and turn in that cover sheet.

9 Sample Output

Here is the output produced by our solution code. You should see something similar, if your program works correctly. It will not be exactly the same and some sections may have a lot more lines of data.

```
====================  KPL PROGRAM STARTING  ====================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...


*****  THREAD-MANAGER TEST  *****

123.4..1562.7839.1041112.13141.15....16....3295.17.8184..76.192010.111315
...14..12..161..9.173188..7.19.52.20....14410.12.91113...616...15571..183
20....122..17..10..9161114586157.......4.123.1.18217.13..20...9.19.516.14
11.2..8..110.15.64.12.1618.9.7.132019..11.3.15..14.812..61.9...2.2018..10
165.15.78..9...4.112013.3..191815....714...17.11.151216.194...18.9.102076
..11.12..15...14.84191013..93..2...1511518..8.714..9.616.19..1..2015...81
3.42.1210.9......520.4.2.181116.14915..4.2.1.1117..5.8.1018.15.2.4.14.9.1
1..1.56.8.2.15..4.1410..93..18.2013.2.15..7.17...1210914.8.20.2.....17515
123.14.9.20..16.819.12.5.3.15.1.16.14.20.8.3..513.12.15.18..1.8..209.193.
5.6.18.7.8....20.14153.511.12.4.7.8.3.14.15.9.11.12...1518.16.3..14..1117
.12..4.5..1710..14.26.1.7..123.16.8.19.17.6.20.2.10..12.165.3.19.1.17..4.
..6.8.1220216.19..7.13.13.6.11.19.16..20.4.2.1.7..13.6.11.17.19..16.14.7.
.1....18....1013619.16.1711.4.7.20..13.18.16.19.10.6.17..411.7..13.18.19.
..17.10.6..7.13..1918.17.10.6.....131817.19.106.13..17...10.13.17..13..

***** THREAD-MANAGER TEST COMPLETED SUCCESSFULLY *****



*****  PROCESS-MANAGER TEST  *****
```

```
123.4..1562783.9.104...1...11.738.12132149.156.5.1016.17..184...19..201.11
138.714.12.3....5.26.17164.3.20.19..1812..629.14..1015...17.137..16.84.182
0.1912...116.15.14.....572.3.89.116....10417513..15.12.1814...23..19.716.1
204.1511...14..5..9.6131237..162....18.10.1718..5..1119.61620....9184..13.
17..14108....612711.159..2..11317....20.11..16..7146.119515.10.4.168...714
..11.92.12.1..181720..14..6319...1219..7.168...2513...411.15..129.181.14..
19.32.4..6.15.128.16..19.1410.5.20.9.7..46.16.11.3.13.17.14.9.8.7.1.16.2..
5..192010.9.7..164.1.11...31317.5.9.7.6.12...19.13.11.20161.5.7.12.19.3.6.
.1110..113.5.12..719.11.6.2.3.20.16.12.5.19.11.6..207.2..9.1416.5.19.15.6.
11.20.2.14.9.12.4.5.17..1118.20.2.14.9...12.5154.11.20.6..152..18.16.14.10
.5.20..63.17..9.102.1.14.11.18.13.10...739.2.16.20.14.11.10.8.3..76.16..20
.13...8.3...18....12194.10.138.3...918..20.1310..124.19.3.1.15.17.13..4.14
.10.12...20.8..1913.4.17.10..18.8...13.19.17.4.18.15.8.10.17.13.19.18..15.
8.18.17.10...15.8.18.17.15...18.17.15.18.17.15.18..15..15.

***** PROCESS-MANAGER TEST COMPLETED SUCCESSFULLY *****



*****  FRAME-MANAGER TEST  *****

1234.57.68.9110...32.5..74...8.69..101.32.45.7.....6.9.8.310215.4......109
67138...2...5107.946...1.25..38.9..10.64..79.21..5..710.8.1.3.9.2..684.19.
.5..6310..48..9..65.1.7.53..2.8..93.10..12.47..1..4.1086..110..48.6..35..9
.710..9.85.3..4.78.6..310.4.7.3..58.10.6..94.1..37.8..6.92.5..74..12.4.2.3
.1..5.26.10.3.1.7.5.8.10.3.1.9.6.8.10.3.1.4.6.10.8.3..31...134...513..1.83
...3101..43..81..8..93.8.4..1.8.34..17.8.3.2.1.4.8.6.4.2.8.47...83.1.5.34.
.1.84.3.1..84..28..10..4107...4110..5.410..18.3.1.8.3..41..105..8..4510...
7105..610..75..71...1085..7..410.75...175..9.75.10..58.7..4.52.7.6..7.6.5.
4.10.85..10.6.7.47..5..106.7.510.7..56..26..69..2..652..28.6.1.2..36..52..
36..7.29...912..97..16...10.129..76...139..7.91..24..79..101..82.6..91.2.6
.9..71..93..61.10.9.1..62.8...298..810..48.1..6.810..1.210..8.510.8..110..
9.8.10..2810..98..710..79.8..7.210.6...27.109.8...1017.2.10.7.8.10.7..47.1
0.8.1.10..48.7..4.17..84.5.4.1.7..8.45.4..58.4..7.5.104..56...56.4..765..1
0.5.67.4..9.5.6.7.5.6.45.8.6.4.7.6.1.5.9.6..26..105.4.6.10.2..23..3.1.23..
6.5.23...623.10.2.6..54.2..5.3.26.3.7.2.3.1.3.1.9.2...613..41.10...1410..1
0.6.1.104.1..9.10..219....31019.7..10.93..10.1.910..32.1.3..109.2.5.1.2.3.
10..97.1.10.5.9.4.9.1.2.10.3.9.2.9..110..93.9.1.7..71.7..27.5.7.4.7..42..7
.410.7..48...18.75..87.10..4.8.2.8.46.7...984.5.8..9.5..74.58..10.54..7.8.
5.47.8..65.4.7..5.84.5.1.5.4.9.5.3.5.8..95.6.6..10.65.8.7.6..26...926..72.
.6.2.54.2.3.6..102..56.10.2...7610...8610.1..2..110.4..1210.6..5.101.9..21
.6..110.2..410..81.4.10..71..410.8.2.6.1.7..76.10.5.1..7.10.2.71..106.1..1
02.7.10.1.2.7..74.7..57.9..95.9..39..5..931..57..3.6..39.510.7..2.95.8.7..
.1039.2.9.8.9..2.5.310.5.2.5.7.9.1..73.5..54..5.1.43.10.4.2...3.25.4.9..34
```

```
.25.9..34.5..79..2.3.4.21.4.7.3.5..24.9..3.52.3.4.5.2..82.6.2.5.4.7.2..94.
.3.2.9.6.1.6..69.6..56.7.6.4.2..46.9..46.9.6..3.69..49.6..92.6..39.5.9.3.9
..9..79.8.8.1.9..86.2..89.6..29.6..89.8.2.8..96.8..8.6.8..98.4.8.9.8.4.8.6
.8.4.8.6..3..3.2.3..93.8.3..63..83.9.3..3.2.3.8.9...3..3.3.3.3.3.3.3.3.3.

Here is a histogram showing how many times each frame was used:
0:  ************************************************************************
************************************************************************
************************************************************************
************************************************************************
************************************************************************
**************************************************
1:  ************************************************************************
************************************************************************
************************************************************************
************************************************************************
************************************************************************
**************************************************
2:  ************************************************************************
************************************************************************
************************************************************************
************************************************************************
************************************************************************
**************************************************


XXXXXXXXXXXXXXXXXXX  skipping  XXXXXXXXXXXXXXXXXXX


32:  ************************************************************************
************************************************************************
************************************************************
33:  ************************************************************************
************************************************************************
**********************************
34:  ************************************************************************
**************************************


***** FRAME-MANAGER TEST COMPLETED SUCCESSFULLY *****


====================  KPL PROGRAM TERMINATION  ====================
```

10 Coding Style

For all assignments, please follow our coding style. Make your code match our code as closely as possible.

The goal is for it to be impossible to tell, just by looking at the indentation and commenting, whether we wrote the code or you wrote the code. (Of course, your code will be circled!)

Please use our convention in labeling and commenting functions:

```
----------------------------  Foo  --------------------------------

  function Foo ()
    --
    -- This routine does....
    --
    var x: int
      x = 1
      x = 2
      ...
      x = 9
    endFunction
```

Methods are labeled like this, with both class and method name:

```
    ----------  Condition . Wait  ----------

    method Wait (mutex: ptr to Mutex)
      ...
```

Note that indentation is in increments of 2 spaces. Please be careful to indent statements such as if, while, and for properly.