The Design of *HAL9000* Operating System

Trick-or-Thread

March 31, 2019

Chapter 1

Design of Module *Threads*

1.1 Assignment Requirement

1.1.1 Initial Functionality

1.1.2 Requirements

The requirements of the "Threads" assignment are the following:

1. *Timer.* You have to change the current implementation of the timer, named `EX_TIMER` and located in the file "`ex_timer.c`", such that to replace the busy-waiting technique with the sleep – wakeup (block – unblock) one. A sleeping thread is one being blocked by waiting (in a dedicated waiting queue) for some system resource to become available. You could use executive events (`EX_EVENT`) to achieve this.

2. *Priority Scheduler — Fixed-Priority Scheduler.* You have to change the current Round-Robin (RR) scheduler, such that to take into account different priorities of different threads. The scheduling principle of the priority scheduler is "any time it has to choose a thread from a thread list, the one with the highest priority must be chosen". It must also be preemptive, which means that at any moment the processor will be allocated to the ready thread with the highest priority. The priority scheduler does not change in any way the threads' priorities

(established by threads themselves) and for this reason could also be called fixed-priority-based scheduler.

3. *Priority Donation.* As described in the *HAL9000* documentation after the fixed priority scheduler is implemented a priority inversion problem occurs. Namely, threads with medium priority may get CPU time before high priority threads which are waiting for resources owned by low priority threads. You must implement a priority donation mechanism so the low priority thread receives the priority of the highest priority thread waiting on the resources it owns.

1.2 Design Description

1.2.1 Needed Data Structures and Functions

For the Priority Donation, a new variable will be added to the thread struct, to hold the initial priority of the thread. Some list structures might be added in order to keep track of the thread and locks relationship, but that is further to be discussed. Functions that require modifications are the ones related to the locks, such as SpinLockAcquire, SpinLockRelease, MutexAcquire, MutexRelease. And a new function might be required to be implemented, in order to set the priority of a certain thread, because the existing one only modifies for the current thread.

1.2.2 Detailed Functionality

Some questions you have to answer (inspired from the original Pintos design templates):

1. Timer

    - Briefly describe what happens in a call to *ExTimerWait()*, including the effects of the timer interrupt handler.
    - What steps are taken to minimize the amount of time spent in the timer interrupt handler?

- How are race conditions avoided when multiple threads call *ExTimerWait()* simultaneously?
- How are race conditions avoided when a timer interrupt occurs during a call to *ExTimerWait()*?

2. Priority Scheduler

   - How do you ensure that the highest priority thread waiting for a mutex or executive event wakes up first?
   - How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

3. Priority Donation

   - When a thread with a high priority level is scheduled, or in a ready-state, the current running thread should give away the CPU to the higher priority one. The problem rises when the higher priority thread requires access to the resources owned by the lower priority thread.
   - Basic scenario:
     - (0) The CPU runs the scheduled thread "Low"
     - (1) Due to higher priority, the "High" thread starts running
     - (2) "High" requires access to resources held by "Low"
     - (3) "Low" releases the lock and "High" can access the resources
     - (4) CPU continues executing the higher priority thread
   - Algorithm description
     - The first thing to do whenever a thread is scheduled, is to check whether the resources it uses are locked by a lower priority thread.
     - The call to a lock acquire function will help us identify the "conflicting" threads and the Priority Donation can occur. (2)

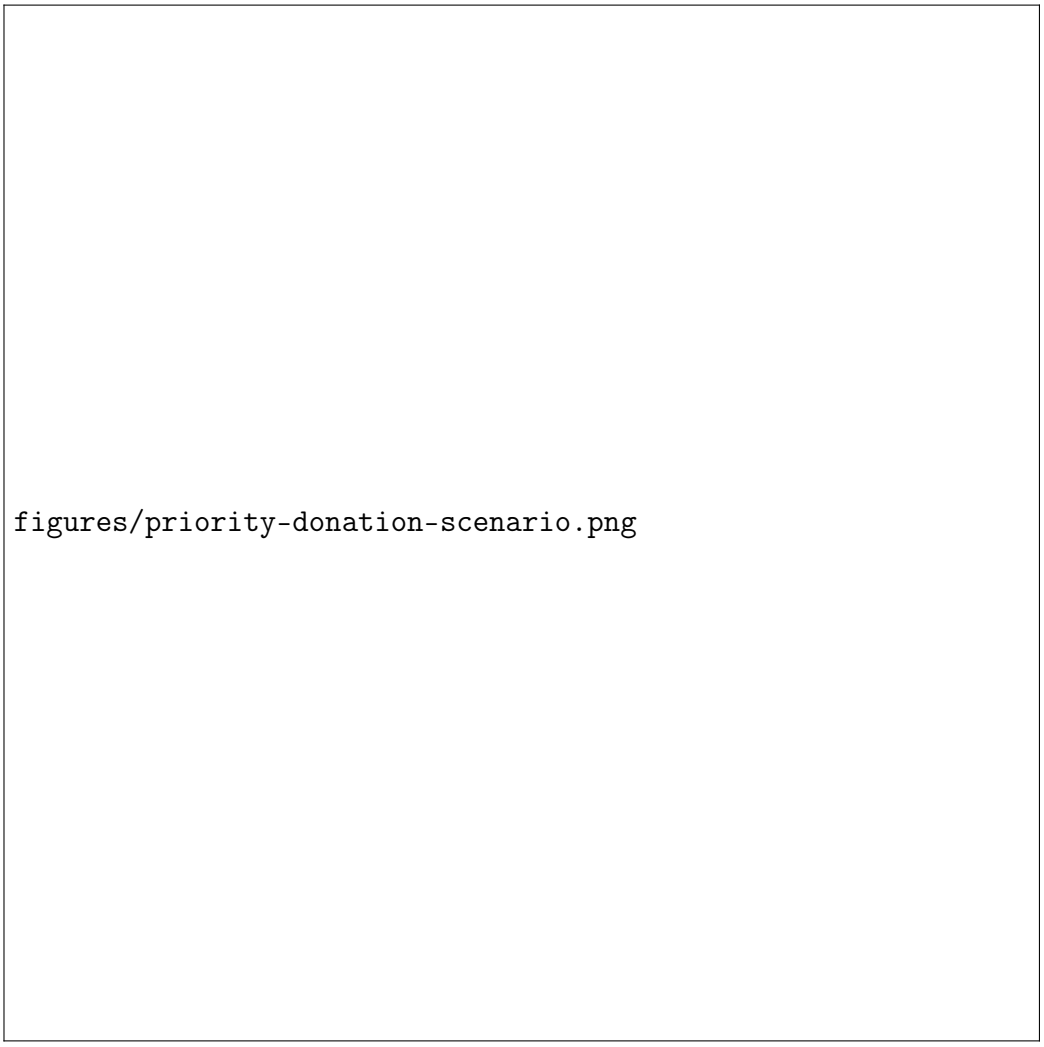

Figure 1.1: Priority Donation Scenario.

- After identifying the involved threads, the lock‘s holder (reciever thread) priority is stored in the variable holding the initial priority of the thread structure, and the intrinsec priority is modified to the one of the higher priority (donor thread).
- When the resource held by the lower priority thread is no longer required by the higher thread, the lock is released and the priority changes are reversed. (3)
- After releasing the lock and reversing the priority donation,

the higher priority thread continues it‘s execution.

- Algorithm implementation

```
LockAcquire(lock)
if(holder = NIL)
holder = currentThread
while(holder != currentThread)
{
holder.initPriority = holder.Priority
holder.Priority = currentThread.Priority
}
LockRelease(lock);
currentThread.Priority = currentThread.initPriority
```

1.2.3 Explanation of Your Design Decisions

1. Timer
2. Priority Scheduler
3. Priority Donation

   The aim of this algorithm is to keep the CPU "busy" handling the priority donation cases only when they occur, and return to the higher priority thread as fast as possible.

1.3 Tests

1. Timer
2. Priority Scheduler
3. Priority Donation

Chapter 2

Design of Module *Userprog*

2.1 Assignment Requirements

1. argument passing + validation of system call arguments (pointers)
2. system calls for process management + file system access
3. system calls for thread management

2.2 Design Description

1. validation

   When issuing a system call, we must make sure that the pointers recieved from the user program are valid. Some error cases might rise when:

   - a null pointer is provided
   - trying to access an un-mapped user address
   - the user adress doesnt have the required rights to perform the action
   - acessing a virtual adress If the action would harm the kernel, the call must be suspended and an error message returned.

2. system calls for thread management Implementation of the following system calls: SyscallThreadExit, SyscallThreadCreate, SyscallThreadGetTid, SyscallThreadWaitForTermination, SyscallThreadCloseHandle

2.2.1 Needed Data Structures and Functions

- validation

  Functions such as:

  ```
  validateAddress(PVOID Address) - checks if the provided address is below
  the kernel virtual adress-space
  validateThreadHandle(UM_HANDLE handle) - checks if the specified handle
   corresponds to a thread, rather than to a file or process
  validateFileHandle(UM_HANDLE handle)
  validateProcessHandle(UM_HANDLE handle)
  ```

- system calls for thread management

  I have decided to implement a new data structure that holds the thread and it's assigned handle, and a handle system data structure for keeping information about existing handles in the system.

  ```
  struct _THREAD_HANDLE
  {
  UM_HANDLE ThreadHandle;
  PTHREAD  pThread;
  };

  struct _HANDLE_SYSTEM_DATA
  {
  LIST_ENTRY AllThreadHandlesList;
  };
  ```

  Similarily to the threads tracking implementation, the thread handles are stored inside a static structure

```
static HANDLE_SYSTEM_DATA m_handleSystemData;
PHANDLE_SYSTEM_DATA GetHandleSystemData(); - for access to the
 structure within system calls implementation
```

2.2.2 Detailed Functionality

1. system calls for thread management

   - SyscallThreadExit - removes the executing thread and it‘s handle from the system‘s AllThreadHandlesList and exits with exit status

     ```
     while( iterator < list.size){
     if(list[iterator]->thread == CurrentThread){
     list.remove(thread);
     ThreadExit(ExitStatus);
     break;
     }
     iterator++;
     }
     ```

   - SyscallThreadCreate - creates a new thread, generates a handle for it, adds the paired structure to the AllThreadHandlesList and returns the generated handle as output parameter.

     ```
     THREAD_HANDLE pThreadHandle;
     pThreadHandle->pThread = ThreadCreate();
     pThreadHandle->ThreadHandle = GenerateHandle();
     list.add(pThreadHandle);
     ```

   - SyscallThreadGetTid - iterates through the AllThreadHandlesList and searches for the thread coresponding to the input handle and returns its id

     ```
     while( iterator < list.size){
     if(list[iterator]->ThreadHandle == inputHandle){
     return list[iterator]->pThread->Id;
     ```

```
}
iterator++;
}
```

- SyscallThreadWaitForTermination - iterates through the AllThreadHandlesList and searches for the thread coresponding to the input handle and waits for it‘s termination

```
while( iterator < list.size){
if(list[iterator]->ThreadHandle == inputHandle){
threadWaitForTermination(list[iterator]->pThread,status);
break;
}
iterator++;
}
```

- SyscallCloseHandle - iterates through the AllThreadHandlesList and searches for the input handle and removes it from the list, closing access to it

```
while( iterator < list.size){
if(list[iterator]->ThreadHandle == inputHandle){
list.remove(list[iterator]);
}
iterator++;
}
```

2.2.3 Explanation of Your Design Decisions

Another solution I first came up with, was to add fields to the THREAD structure, holding list entries with handles and related threads, but that means wasted memory by kernel-initialized threads, that do not require handles. This solution solves this problem and provides a way to validate the input handle parameters, verifying that they correspond to the needed resource (thread/process/file).

2.3 Tests

The test applications for the thread management system calls involve testing the following syscalls, divided into several steps

1. ThreadWait
    - ThreadWaitTerminated
    - ThreadWaitNormal
      Creates two threads, sends a termination status as parameter and checks the termination status returned by the syscalls.
    - ThreadWaitClosedHandle, ThreadWaitBadHandle
      If the syscall executes successfully, even if a bad handle or a closed handle to the thread is recieved as parameter, the test fails.
2. ThreadCreate
    - ThreadCreateWithArguments
      Validates the arguments recieved as parameters, the test should fail if any of the parameters are not as expected.
    - ThreadCreateOnce, ThreadCreateMultiple, ThreadCreateTwice
      Verifies the return status and the number of threads created by the system call.
    - ThreadCreateBadPointer
      Should fail in case a bad reference is sent.
3. ThreadClose
4. ThreadExit
5. ThreadGetTid