Operating Systems Practical Assignment – Alternate Alarms

Sander van Loo and Erik van Zijst
August 19th, 2004

Vrije Universiteit,
Faculty of Sciences, Department of Mathematics and Computer Science,
Amsterdam, The Netherlands

{sander,erik}@marketxs.com

Introduction

This document describes our work for the third practical assignment of the 2004 operating systems course. In this assignment we were to add additional alarm functionality to the minix operating system. Alarms allow programs to be notified by the operating system after some specified time. However, since minix allows a process to have only a single alarm set, the new alarm that is to be added must work independent of the standard alarm.
Our work consists of two components: the modifications we made to the kernel and a set of user-level programs to test the new alarm under various circumstances. We will discuss both in separate chapters.

Kernel Modifications

To offer alternate alarms, the assignment states that we should add a system call `alarm2()` that is used to schedule the new alarm. The system call must be added to the memory manager. Also, a new signal definition `SIGALRM2` is to be added so it can be caught by applications.

The changes are reflected in the following files:

- `include/alarm2.h`: the header file that contains the prototype of our new system call
- `include/signal.h`: contains the definition of our new `SIGALRM2` signal and _NSIG
- `include/minix/callnr.h`: definition of the call number for alarm2()
- `include/minix/com.h`: here we defined field names for the messages sent between the memory manager and the kernel
- `include/minix/const.h`: added NR_ALARMS indicating the number of available alarm queues per process
- `src/fs/table.c`: adds null-mapping to the file system for our system call that is already implemented in the memory manager
- `src/mm/mproc.h`: added a flag that indicates per process whether the alternate alarm is scheduled
- `src/mm/forkexit.c`: unsets the alternate alarm when a process dies
- `src/mm/proto.h`: prototypes for `do_alarm2()` and `set_alarm2()`
- `src/mm/table.c`: maps the new system call number to its implementation
- `src/mm/signal.c`: added a generic framework for handling *n* alarms per process, exposing only the standard alarm and the new alternate alarm timers
- `src/kernel/proc.h`: modified to maintain `NR_ALARMS` alarm queues per process, rather than only one
- `src/kernel/system.c`: now initializes `NR_ALARM` alarm timers, instead of only one per process
- `src/kernel/clock.c`: added the generic `set_alarmn()` call, capable of handling `NR_ALARMS` alarm queues

We started by modifying the kernel to maintain an array of alarm queues, rather than only a single one. This way it is trivial to add one or more alternate alarms. This caused us to change the initialization process of the alarm timers of processes which is controlled by the kernel module SYSTEM. As our kernel in principle supports any number of alarms, we added a generic `do_setalarmn()` routine that is passed the number of the alarm that is to be modified in the message. This routine is called by the memory manager, using the

`SET_ALARMN` message. The old `do_setalarm()` call is now merely a wrapper around `set_alarmn()` that simply defaults to alarm number 0 (the standard alarm). This approach minimizes the amount of redundant code, especially if the kernel was ever to support even more alternate alarms. The memory manager no longer uses the old `SET_ALARM` message to communicate with the clock module, but uses our new `SET_ALARMN` message. The alarm and alarm2 system calls in the memory manager now both call the new internal `set_alarmn()` routine, each passing along their alarm number. The alarm numbers are defined as 0 for the standard alarm and 1 for our alternate alarm. The new, private `set_alarmn()` routine in the memory manager constructs an instance of the new `SET_ALARMN` message that contains the number of the alarm to set and sends it to the clock module. Our implementation does not support alternate synchronous alarms. Synchronous alarms still use the standard alarm queue (alarm number 0).

Our initial implementation worked fine and supported the use of both alarms independently. However, when a process scheduled both alarms to go off at exactly the same time, the process froze instead of executing its signal handlers. This happens because when both the alarms go off at the same time, the kernel places the signal handlers on the stack of the target process. However, because the kernel never calls the `lock_ready()` routine to tell the process to continue, the process remains blocked and doesn't execute its signal handlers. The reason why the process remains blocked is because the kernel corrupts up the `PENDING` and `SIGPENDING` bits of the process when two signals are delivered simultaneously. Right after `inform()` resets the `PENDING` bit, but before the kernel can process it, the next alarm sets the bit again, causing the kernel to leave to process in the blocked state. One way of avoiding this problem is to make sure the kernel never delivers two alarms to a process at the same time, but to delay the next alarm at least one clock tick so it will be processed individually. Of course this affects the accuracy of the alarms, but since alarms can only be set with the granularity of seconds, delaying an alarm for one clock tick is considered rather irrelevant. This leads to an implementation of the `do_clocktick()` method that iterates through the alarm queues until it finds one that should go off. If the `do_clocktick()` method delivers an alarm for a process it delays the delivery of any concurrent alarms for this process to the next clock tick. The `do_clocktick()` method also implements the mapping between alarm numbers and signals. It currently maps alarm 0 to `SIGALRM` and alarm 1 to `SIGALRM2`. If `NR_ALARMS` would be incremented to support more than 2 alarms so would the number of signals.

Experiments

To verify whether our new alarm works as required, we ran a number of tests to see how it performed when running exclusively, or concurrently with the existing minix alarm. Note that we let our test programs display a timestamp on every line of output to let the user verify the duration of the alarms.

Concurrent Alarms

With our `alarm2a.c` program we can test what happens when we schedule both the normal alarms, as well as the new alarm and let them expire individually, or simultaneously. The program accepts two arguments. The first is the amount of time in seconds for scheduling the standard alarm, while the second argument specifies the interval for the new alarm.

When the program is run with the arguments "1" and "2", it will cause `SIGALRM` to fire one second after `SIGALRM2`. The output of the test is given below:

```
# ./alarm2a 1 2
Thu Aug 19 14:51:02 2004: Alarm 1 is scheduled to arrive in 1 seconds
Thu Aug 19 14:51:02 2004: Alarm 2 is scheduled to arrive in 2 seconds
Thu Aug 19 14:51:03 2004: Alarm 1 received
Thu Aug 19 14:51:04 2004: Alarm 2 received
#
```

When the arguments are reversed, both signals fired in reverse order as expected:

```
# ./alarm2a 2 1
Thu Aug 19 14:51:06 2004: Alarm 1 is scheduled to arrive in 2 seconds
Thu Aug 19 14:51:06 2004: Alarm 2 is scheduled to arrive in 1 seconds
Thu Aug 19 14:51:07 2004: Alarm 2 received
Thu Aug 19 14:51:08 2004: Alarm 1 received
#
```

An interesting test is to see what happens when both alarms fire simultaneously. As we described earlier, our first implementation failed this test as the kernel corrupted the `PENDING` and `SIGPENDING` bits of the program while passing both signals to the process in one pass. The result was that the process froze. As the output below shows, our current version does not freeze and handles concurrent alarms correctly:

```
# ./alarm2a 1 1
Thu Aug 19 14:51:10 2004: Alarm 1 is scheduled to arrive in 1 seconds
Thu Aug 19 14:51:10 2004: Alarm 2 is scheduled to arrive in 1
seconds
Thu Aug 19 14:51:11 2004: Alarm 1 received
Thu Aug 19 14:51:11 2004: Alarm 2 received
#
```

Remaining Time

To test whether the new alarm behaves correctly when it is scheduled a second time before it expired, we wrote `alarm2b.c`. This program schedules both alarms, but each time an alarm goes off, its signal handler routine reschedules the other alarm and displays the number of seconds the other alarm had still pending. The program terminates after 4 alarms. Its output is given below:

```
# ./alarm2b 1 2
Thu Aug 19 14:51:36 2004: Alarm 1 is scheduled to arrive in 1
seconds
Thu Aug 19 14:51:36 2004: Alarm 2 is scheduled to arrive in 2
seconds
Thu Aug 19 14:51:37 2004: Alarm 1 received
Thu Aug 19 14:51:37 2004: Alarm 2 is rescheduled to arrive in 2
seconds (1 seconds remaining)
Thu Aug 19 14:51:39 2004: Alarm 2 received
Thu Aug 19 14:51:39 2004: Alarm 1 is rescheduled to arrive in 1
seconds (0 seconds remaining)
Thu Aug 19 14:51:40 2004: Alarm 1 received
Thu Aug 19 14:51:40 2004: Alarm 2 is rescheduled to arrive in 2
seconds (0 seconds remaining)
Thu Aug 19 14:51:42 2004: Alarm 2 received
Thu Aug 19 14:51:42 2004: Alarm 1 is rescheduled to arrive in 1
seconds (0 seconds remaining)
Thu Aug 19 14:51:42 2004: Done
#
```

The test shows that both alarm calls correctly return the number of seconds until they expire when they are scheduled twice.

Conclusion

In this text we discussed our additions to the minix 2.0.0 kernel in order to support a second, independent alarm to user-level applications. To this end we added a new signal definition and a new system call to the kernel that offers the same, but totally independent functionality as the standard alarm. This allows applications to schedule two concurrent alarms, rather than merely one. We explained our implementation and difficulties we encountered and presented a number of experiments to test our kernel modifications under various circumstances. In all cases our tests confirmed that the new alarm worked as expected.