User-Traps

CS3500 Operating Systems Jul-Nov 2025

Sanjeev Subrahmaniyan S B

EE23B102

1 Objective

The objective of this lab is to modify the xv6 operating system to periodically(determined by the arguments to a syscall) alert running processes and call a defined handler function, in the form of user level interrupts. In specific:

1. Add a system call to set the running process to call a handler function every time it consumes a specified number of CPU ticks. Note that a tick is defined as an interrupt of the hardware timer and is fairly arbitrary.
2. Add a system call to return from the handler function to permit the application to resume where it left off. This is accomplished by storing information about the running process when the handler is called and recovering the same when done, in a way similar to how context switches happen.

2 Code

A number of changes were made in the operating system code as part of the assignment. The files and the corresponding changes made in them are enumerated here.

2.1 Makefile

```
diff --git a/Makefile b/Makefile
index c6d1a2b..be5f3c7 100644
--- a/Makefile
+++ b/Makefile
@@ -144,6 +144,7 @@ UPROGS=\
    $U/_trace\
    $U/_PingPong\
    $U/_Mycat\
+   $U/_alarmtest\
```

In the makefile, an entry is made for the alarmtest file to be executed as a user program under the UPROGS section.

2.2 user/user.h

```
diff --git a/user/user.h b/user/user.h
index 694227f..0610266 100644
--- a/user/user.h
+++ b/user/user.h
@@ -23,6 +23,8 @@ char* sbrk(int);
 int sleep(int);
 int uptime(void);
 int trace(int);
```

```
+int sigalarm(int ticks, void (*handler)());
+int sigreturn(void);
```

The prototypes are added for the system calls here.

2.3 user/usys.pl

```
iff --git a/user/usys.pl b/user/usys.pl
index 9c97b05..b4c0619 100755
--- a/user/usys.pl
+++ b/user/usys.pl
@@ -37,3 +37,5 @@ entry("sbrk");
 entry("sleep");
 entry("uptime");
 entry("trace");
+entry("sigalarm");
+entry("sigreturn");
```

An entry is made in the perl script to generate the stub of code for the system call during compilation.

2.4 kernel/proc.c

```
diff --git a/kernel/proc.c b/kernel/proc.c
index bdfa81f..752cd0a 100644
--- a/kernel/proc.c
+++ b/kernel/proc.c
@@ -124,6 +124,11 @@ allocproc(void)
 found:
   p->pid = allocpid();
   p->state = USED;
+
+  p->alarm_ticks = 0;
+  p->alarm_ticks_left = 0;
+  p->alarm_handler = 0;
+  p->alarm_active = 0;

   // Allocate a trapframe page.
   if((p->trapframe = (struct trapframe *)kalloc()) == 0){
```

Entries are made in the process control block to hold the data required to implement the alarm functionality. These are initialized to default values of zero in the allocproc() function inside the file.

2.5 kernel/proc.h

```
diff --git a/kernel/proc.h b/kernel/proc.h
index c8ed154..14a9f45 100644
--- a/kernel/proc.h
+++ b/kernel/proc.h
@@ -92,7 +92,13 @@ struct proc {
   int xstate;                  // Exit status to be returned to parent's wait
   int pid;                     // Process ID
   int trace_mask;               // mask for which system calls must be traced
-
+
+   int alarm_ticks;              // to hold the number of ticks per alarm for this
    process
+   int alarm_ticks_left;     // hold the number of ticks left before next alarm
+   uint64 alarm_handler;         // hold the pointer to the handler function
+   struct trapframe alarm_trapframe; // hold the trapframe when a switch is made
+   int alarm_active;             // flagging if the alarm is active to prevent
    reentrancy
+
    // wait_lock must be held when using this:
    struct proc *parent;         // Parent process
```

The data types required to implement the alarm functionality are added as entries to the PCB struct. Note that an entry is also made to hold a copy of the trapframe when the handler has to called. This is saved inside the PCB struct. The requirements for each of those data types are specified in the comments alongside them.

2.6 kernel/syscall.c

```
diff --git a/kernel/syscall.c b/kernel/syscall.c
index d4efa71..814fc6f 100644
--- a/kernel/syscall.c
+++ b/kernel/syscall.c
@@ -102,6 +102,8 @@ extern uint64 sys_link(void);
 extern uint64 sys_mkdir(void);
 extern uint64 sys_close(void);
 extern uint64 sys_trace(void);
+extern uint64 sys_sigalarm(void);
+extern uint64 sys_sigreturn(void);

 // An array mapping syscall numbers from syscall.h
 // to the function that handles the system call.
@@ -128,6 +130,8 @@ static uint64 (*syscalls[])(void) = {
 [SYS_mkdir]   sys_mkdir,
 [SYS_close]   sys_close,
 [SYS_trace]   sys_trace,
+[SYS_sigalarm]   sys_sigalarm,
+[SYS_sigreturn]   sys_sigreturn,
 };

 char *syscallnames[] = {
@@ -153,6 +157,8 @@ char *syscallnames[] = {
   "mkdir",
   "close",
   "trace",
+  "sigalarm",
+  "sigreturn",
 };
```

This file incorporates the changes required to expose the system calls implentation. For this, the function prototypes are defined, entries are made on the function pointer array, and in the array of function names for the tracing utility implmented in one of the previous assignments.

2.7 kernel/syscall.h

```
diff --git a/kernel/syscall.h b/kernel/syscall.h
index cc112b9..12da2da 100644
--- a/kernel/syscall.h
```

```
+++ b/kernel/syscall.h
@@ -21,3 +21,5 @@
 #define SYS_mkdir  20
 #define SYS_close  21
 #define SYS_trace  22
+#define SYS_sigalarm 23
+#define SYS_sigreturn 24
```

Likewise, indices are assigned for each of the system calls. This enables the system call handler to identify which system call was made through its arguments and call the appropriate implementation function.

2.8 kernel/sysproc.c

```
diff --git a/kernel/sysproc.c b/kernel/sysproc.c
index 377bb66..02944eb 100644
--- a/kernel/sysproc.c
+++ b/kernel/sysproc.c
@@ -113,3 +113,40 @@ sys_trace(void)
   release(&p->lock);
   return 0;
 }
+
+uint64
+sys_sigalarm(void)
+{
+  int alarm_ticks;
+  argint(0, &alarm_ticks);
+
+  uint64 alarm_handler;
+  argaddr(1, &alarm_handler);
+
+  struct proc *p = myproc();
+  acquire(&p->lock);
+  p->alarm_ticks = alarm_ticks;
+  p->alarm_handler = alarm_handler;
+
+  if(alarm_ticks > 0){
+    p->alarm_ticks_left = alarm_ticks;
+  } else {
+    p->alarm_ticks_left = 0;
+    p->alarm_active = 0;
+  }
+  release(&p->lock);
+  return 0;
+}
+
+uint64
+sys_sigreturn(void)
+{
+  struct proc *p = myproc();
+  memmove(p->trapframe, &p->alarm_trapframe, sizeof(struct trapframe));
+
+  acquire(&p->lock);
+  p->alarm_active = 0;
+  release(&p->lock);
+
+  return p->trapframe->a0;
```

```
+}
```

The system calls are implemented here.

For the sys_sigalarm() function, the arguments passed(number of ticks and the handler) are copied into kernel space using the dedicated functions. After that, the data entries in the PCB of the process are updated with the arguments passed to the system call. Note that it is also checked if the number of ticks is positive and accordingly updated. If not, the behavior of disabling the alarm is implemented by setting the alarm_active entry to zero.

The sys_sigreturn() function implements the copy of the trapframe information when the handler is about to return. This is implemented by copying the values in the alarm_trapframe entry of the PCB, which was populated by a copy of the trapframe when the call was made, into the process' original trapframe. Additionally, it is ensured that the value of the register a0 is preserved by returning the value from the trapframe, which is stored in a0 when the funtion returns.

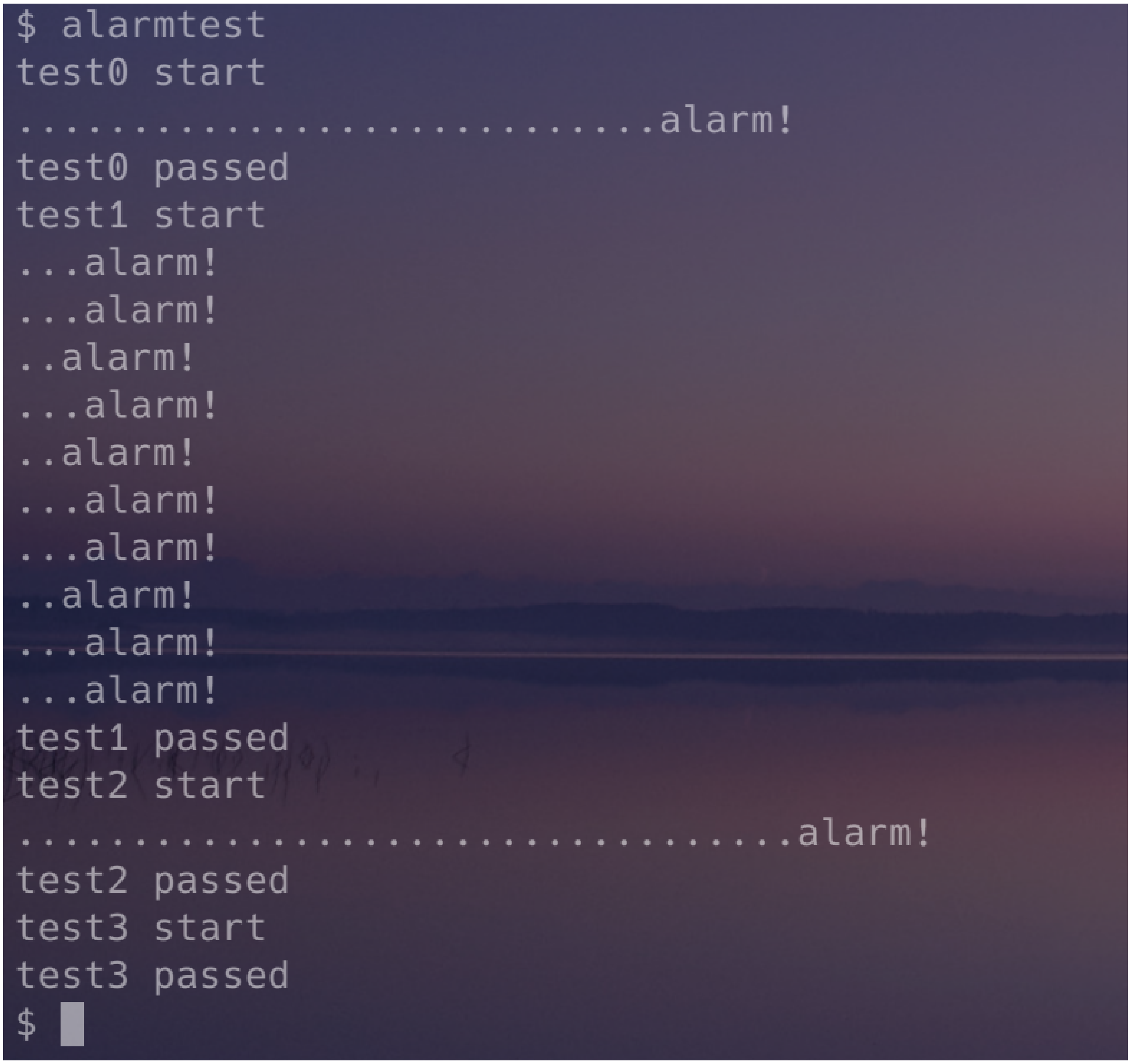
2.9 kernel/trap.c

```
diff --git a/kernel/trap.c b/kernel/trap.c
index 8a93454..db19b43 100644
--- a/kernel/trap.c
+++ b/kernel/trap.c
@@ -68,6 +68,20 @@ usertrap(void)
     syscall();
   } else if((which_dev = devintr()) != 0){
     // ok
+
+    // code to handle the alarm when ticks are received
+    struct proc *p = myproc();
+    if(p && p->state == RUNNING && p->alarm_ticks > 0){ // check if the process is
    running and has alarm armed
+      if(p->alarm_active == 0){ // prevent reentrant calls
+        if(--p->alarm_ticks_left <= 0){ // check if enough ticks have happened
+          memmove(&p->alarm_trapframe, p->trapframe, sizeof(struct trapframe)); //
    copy content
+
+          p->alarm_active = 1; // set alarm to active to prevent rentrancy
+          p->alarm_ticks_left = p->alarm_ticks; // reset ticks left for next alarm
+          p->trapframe->epc = p->alarm_handler; // point of return when syscall
   handler returns
+        }
+      }
+    }
   } else if((r_scause() == 15 || r_scause() == 13) &&
             vmfault(p->pagetable, r_stval(), (r_scause() == 13)? 1 : 0) != 0) {
     // page fault on lazily-allocated page
```

The trap handling is implemented in usertrap() function. First, it is checked if the trap was due to a timer interrupt. Then, it is checked if the process is currently in the running state, has a non-zero alarm_tick entry set by the system call and if it is not already active(which means the handler for the previous call has finished, preventing re-entrant system calls). Then, it is checked if the specified duration has elapsed after which the trapframe is copied. The alarm_active entry is set to indicate that the handler has been called, the number of ticks is reset and the epc(exception program counter - holds the instruction that will be run when returning from the trap) is set to the handler. After this, the return takes it to the handler which executes as required.

3 Execution and Output

The changes are tested with the alarmtest.c file provided as part of the assignment. The output when alarmtest is run in a docker container is:



```
$ alarmtest
test0 start
............................alarm!
test0 passed
test1 start
...alarm!
...alarm!
..alarm!
...alarm!
..alarm!
...alarm!
...alarm!
..alarm!
...alarm!
...alarm!
test1 passed
test2 start
..................................alarm!
test2 passed
test3 start
test3 passed
$ 
```