Operating System
Assignment 3

Eran Aflalo 209343722

Omer Luxembourg 205500390

1 Signal Framework

We implemented a signal handler that can receive 2 numbers – pid of the sending process and a number that is sent by the sending process.

We defined a typedef which is a decleration of a signal handler function:

```
1  typedef unsigned int   uint;
2  typedef unsigned short ushort;
3  typedef unsigned char  uchar;
4  typedef uint pde_t;
5  //------ 06.01.21 ------
6  typedef void (*sig_handler)(int pid, int value);
7  //-------------------------
```

In addition we defined the system call function **sigset** and **sigsend** and **sigret** (for the user space also…):

```
27 //------ 06.01.21 ------
28 //set the signal handler to be called when signals are sent
29 sig_handler sigset(sig_handler);
30 // send a signal with the given value to a process with pid dest_pid
31 int sigsend(int dest_pid, int value);
32 //------ 09.01.21 ------
33 void sigret(void);
34 //-------------------------
```

Sigret will be mentioned in the 4th part.

The three of them been added as in task 1, to the system call chain. (**usys.S**, **syscall.h**, **syscall.c,** etc)

System call route:

```
103
104 //PAGEBREAK: 16
105 // proc.c
106 int             cpuid(void);
107 void            exit(void);
108 int             fork(void);
109 int             growproc(int);
110 int             kill(int);
111 struct cpu*     mycpu(void);
112 struct proc*    myproc();
113 void            pinit(void);
114 void            procdump(void);
115 void            scheduler(void) __attribute__((noreturn));
116 void            sched(void);
117 void            setproc(struct proc*);
118 void            sleep(void*, struct spinlock*);
119 void            userinit(void);
120 int             wait(void);
121 void            wakeup(void*);
122 void            yield(void);
123 //------ 06.01.21 ------
124 int             sigsend(int,int);
125 //------ 09.01.21 ------
126 void            sigret(void);
127 //-------------------------
```

```
18 #define SYS_mknod  17
19 #define SYS_unlink 18
20 #define SYS_link   19
21 #define SYS_mkdir  20
22 #define SYS_close  21
23
24 //------ 06.01.21 ------
25 #define SYS_sigset 22
26 #define SYS_sigsend 23
27 //------ 09.01.21 ------
28 #define SYS_sigret 24
29 //-------------------------
```

```
28 SYSCALL(dup)
29 SYSCALL(getpid)
30 SYSCALL(sbrk)
31 SYSCALL(sleep)
32 SYSCALL(uptime)
33
34 #------ 06.01.21 ------
35 SYSCALL(sigset)
36 SYSCALL(sigsend)
37 #------ 09.01.21 ------
38 SYSCALL(sigret)
```

```
103 extern int sys_wait(void);
104 extern int sys_write(void);
105 extern int sys_uptime(void);
106 //------ 06.01.21 ------
107 extern int sys_sigset(void);
108 extern int sys_sigsend(void);
109 //------ 09.01.21 ------
110 extern int sys_sigret(void);
111 //--------------------
```

```
131 [SYS_mknod]   sys_mknod,
132 [SYS_unlink]  sys_unlink,
133 [SYS_link]    sys_link,
134 [SYS_mkdir]   sys_mkdir,
135 [SYS_close]   sys_close,
136
137 //------ 06.01.21 ------
138 [SYS_sigset] sys_sigset,
139 [SYS_sigsend] sys_sigsend,
140 //------ 09.01.21 ------
141 [SYS_sigret] sys_sigret,
142 //--------------------
143 };
```

The rest of the implementation is in the next sections.

2 Storing and Changing the Signal Handler

To store the signle handler, we will add a new field to struct proc (in **proch.h**). This field will hold a pointer to the current handler (or -1 if no handler is set). Both **fork** and **exec** system calls modify the signal handler.

fork –

Takes parent handler

```
187  // Caller must set state of returned proc to RUNNABLE.
188  int
189  fork(void)
190  {
191    int i, pid;
192    struct proc *np;
193    struct proc *curproc = myproc();
194
195    // Allocate process.
196    if((np = allocproc()) == 0){
197      return -1;
198    }
199
200    // Copy process state from proc.
201    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
202      kfree(np->kstack);
203      np->kstack = 0;
204      np->state = UNUSED;
205      return -1;
206    }
207    np->sz = curproc->sz;
208    np->parent = curproc;
209    *np->tf = *curproc->tf;
210
211    // Clear %eax so that fork returns 0 in the child.
212    np->tf->eax = 0;
213
214    for(i = 0; i < NOFILE; i++)
215      if(curproc->ofile[i])
216        np->ofile[i] = filedup(curproc->ofile[i]);
217    np->cwd = idup(curproc->cwd);
218
219    safestrcpy(np->name, curproc->name, sizeof(curproc->name));
220
221    pid = np->pid;
222
223    acquire(&ptable.lock);
224    //------ 06.01.21 ------
225    np->sigHandler =  curproc->sigHandler;
226    //-----------------------
227    np->state = RUNNABLE;
228
229    release(&ptable.lock);
230
231    return pid;
```

exec –

exec body function

default handler (-1)

```
types.h | proc.h | proc.c | exec.c | user.h | usys.S | syscall.h
74      goto bad;
75    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
76    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
77      goto bad;
78    ustack[3+argc] = sp;
79  }
80  ustack[3+argc] = 0;
81
82  ustack[0] = 0xffffffff;  // fake return PC
83  ustack[1] = argc;
84  ustack[2] = sp - (argc+1)*4;  // argv pointer
85
86  sp -= (3+argc+1) * 4;
87  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
88    goto bad;
89
90  // Save program name for debugging.
91  for(last=s=path; *s; s++)
92    if(*s == '/')
93      last = s+1;
94  safestrcpy(curproc->name, last, sizeof(curproc->name));
95
96  // Commit to the user image.
97  oldpgdir = curproc->pgdir;
98  curproc->pgdir = pgdir;
99  curproc->sz = sz;
100 curproc->tf->eip = elf.entry;  // main
101 curproc->tf->esp = sp;
102 //------ 06.01.21 ------
103 curproc->sigHandler = (sig_handler)(-1);
104 //-------------------------
105 switchuvm(curproc);
106 freevm(oldpgdir);
107 return 0;
108
109 bad:
110  if(pgdir)
111    freevm(pgdir);
112  if(ip){
113    iunlockput(ip);
114    end_op();
115  }
116  return -1;
117 }
118
```

Sigset system call (a.k.a. **sys_sigset**) is defined in **sysproc.c** to replace the destenation process' signal handler with a new one, and return the old signal handler.

```
sysproc.c | types.h | proc.h | proc.c | exec.c | user.h | usys.S | syscall.h | syscall.c
93  //------ 06.01.21 ------
94  //system call - replace the old signal handler to new signal handler and return the old one.
95  int
96  sys_sigset(void)
97  {
98    sig_handler new_sigHandler;
99    if(argptr(0, (void*)&new_sigHandler,sizeof(sig_handler)) < 0)
100     return -1;
101   sig_handler old_sigHandler = myproc()->sigHandler;
102   myproc()->sigHandler = new_sigHandler;
103   cprintf("sysproc.c : sys_sigset nom.  for process %d \n",myproc()->pid);
104
105   return (int)(old_sigHandler);
106 }
```

3 Sending a Signal to a Process

The new **sigsend** system call sends a signal to a destination process. When a signal is sent to a process it is not handled instantly since the destination process may be already running or even blocked. This means that each process must store all the signals which were sent to it but still not handled in a data structure that we will refer to as the pending signals stack, Since multiple processes can send signals to the same recipient then he must save the signals in structure.

The **sigsend** system call will add a record to the recipient pending signals stack. It will return 0 on success and -1 on failure (if pending signals stack is full). Systemcall function added to **sysproc.c**:

```
107
108  //system call - send signal "value" from current processs to dest_pid process.
109  int
110  sys_sigsend(void)
111  {
112      int dest_pid;
113      int value;
114      if(argint(0, &dest_pid) < 0)
115          return -1;
116      if(argint(1, &value) < 0)
117          return -1;
118      cprintf("(sysproc.c) sys_sigsend: from process %d - to pid %d with value %d.\n",myproc()->pid, dest_pid, value);
119      return sigsend(dest_pid,value);
120  }
```

For each process, we created the following –

```
1   //------ 06.01.21 ------
2
3   // defines an element of the concurrent struct
4   struct cstackframe{
5       int sender_pid;
6       int recipient_pid;
7       int value;
8       int used;
9       struct cstackframe *next;
10  };
11
12  //defines a concurrent stack
13  struct cstack{
14      struct cstackframe frames[10];
15      struct cstackframe *head;
16  };
17
18  // adds a  new frame tot the cstack which is initialized with values send_pid,
19  // recipient_pid and value,
20  // then returns 1 on success and 0 if the stack is full
21  int push (struct cstack *cstack, int sender_pid, int recipient_pid, int value);
22
23  // removes and returns an element from the head of given cstack. if the stack is empty, then return 0
24  struct cstackframe *pop(struct cstack *cstack);
25
26  //------------end--------------
```

```
60  };
61
62  enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
63
64  // Per-process state
65  struct proc {
66    uint sz;                     // Size of process memory (bytes)
67    pde_t* pgdir;                // Page table
68    char *kstack;                // Bottom of kernel stack for this process
69    enum procstate state;        // Process state
70    int pid;                     // Process ID
71    struct proc *parent;         // Parent process
72    struct trapframe *tf;        // Trap frame for current syscall
73    struct context *context;     // swtch() here to run process
74    void *chan;                  // If non-zero, sleeping on chan
75    int killed;                  // If non-zero, have been killed
76    struct file *ofile[NOFILE];  // Open files
77    struct inode *cwd;           // Current directory
78    char name[16];               // Process name (debugging)
79
80    //------ 06.01.21 ------
81    sig_handler sigHandler;
82    struct cstack pending_signals;
83    //--------------------
84
85  };
```

We initialized in **proc.c** – **allocproc()** function the cstack pending_signals frames and head:

```
114
115     //------ 07.01.21 ------
116     //reset the used of the pending_signals and the head to &frames[0]
117     int i;
118     for(i=0;i<10;i++)
119         (p->pending_signals).frames[i].used=0;
120     (p->pending_signals).head=&((p->pending_signals).frames[0]);
121     //----------------------
122
123   return p;
124 }
```

The **sigsend**, **push** and **pop** functions –

```
546
547   //------ 06.01.21 ------
548   int
549   sigsend(int dest_pid,int value)
550   {
551     struct proc *p;
552     acquire(&ptable.lock);
553     // finding the destination process
554     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
555       if(p->pid == dest_pid){
556         // try to push the signal to cstack of the destination process
557         int success = push(&(p->pending_signals),myproc()->pid,dest_pid,value);
558         release(&ptable.lock);
559         return (success-1); //returns 0 on success and -1 if failed (full cstack)
560       }
561     }
562     release(&ptable.lock);
563     return -1; //returns -1 because failed to find the process
564   }
565
566
567   // adds a  new frame tot the cstack which is initialized with values send_pid,
568   // recipient_pid and value,
569   // then returns 1 on success and 0 if the stack is full
570   int push (struct cstack *cstack, int sender_pid, int recipient_pid, int value)
571   {
572     int i;
573     //make the new frame which will be the new head of the stack (head become second)
574     struct cstackframe new_csframe = {sender_pid,recipient_pid,value,1,cstack->head};
575     for(i=0; i<10;i++){
576       if((cstack->frames[i]).used==0){
577         cstack->frames[i] =new_csframe;
578         cstack->head = &(cstack->frames[i]);
579         return 1;
580       }
581     }
582     return  0; //if full
583     //if(cstack->head==null)
584   }
```

Sending the destination process a signal.

```
595
596   // removes and returns an element from the head of given cstack. if the stack is empty, then return 0
597   struct cstackframe *pop(struct cstack *cstack)
598   {
599     struct cstackframe* oldHead = cstack->head;
600     if(oldHead->used==0) //the stack is empty
601       return 0;
602     else{
603       //remove the head of the stack
604       (cstack->head)->used=0;
605       cstack->head = oldHead->next;
606       return oldHead;
607     }
608
609   }
610
```

4 Signal Handling

When a process is about to return from kernel space to user space (using the function **trapret** which can be found at **trapasm.S**) it must check its pending signals stack. If a pending signal exists and the process is not already handling a signal (i.e., we did not support handling multiple signals at once) then the process must handle the signal.

The signal handling can be either discarding the signal (if the signal handler is default we printed a message) or executing a signal handler when it returns to user space. To force the execution of the signal handler in user space we have to modified the user space stack and the instruction pointer of the process.

We changed the following to make the function call: pushed the arguments for the called function, pushed the return address on the stack, and jump to the body of the called function. In addition in the return address we pointed it to a code we injected to the stack – implicit sigret.

Also, we saved the CPU registers values and restored them when returning from the called function – signal handler.

This is the system call **sigret()**:

```
122  //------ 09.01.21 ------
123  //system call - sigret
124  void
125  sys_sigret(void)
126  {
127      sigret();
128      return;
129  }
130  //--------------------
```

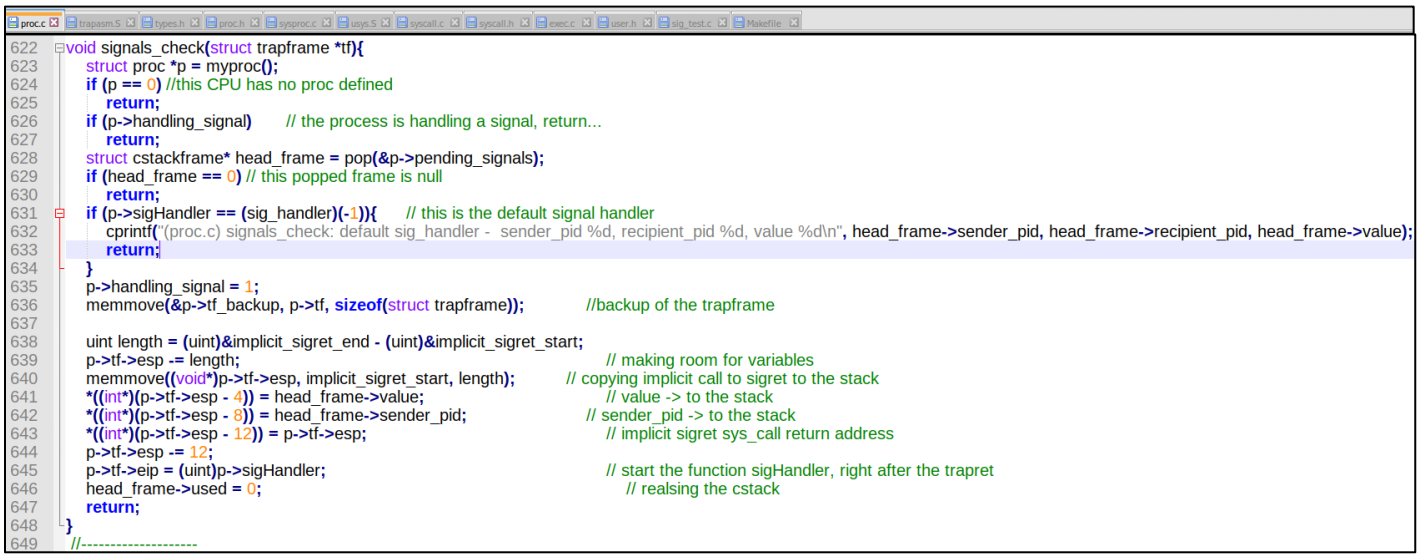
In **proc.c** we created the **sigret** system call, which only restoring our backup of the *trapframe*.

```
612  //------ 10.01.21 ------
613  // a system call function to handle signal return - using the old trapframe and handling signal to 0
614  void
615  sigret(void)
616  {
617      struct proc *p = myproc();
618      memmove(p->tf, &p->tf_backup, sizeof(struct trapframe));        // restoring the old backup of the trapframe
619      p-> handling_signal = 0;        // turning off the handling signals flag - now other signals can be handled
620  }
```

In the **trapasm.S** we updated the stack to call the **signals_check** (which is the signals checker function). In addition we added an **implicit_sigret** to the file, which will be loaded to the user's stack, to execute after returning from **signals_check** function.

```
26   # Return falls through to trapret...
27   .globl trapret
28   trapret:
29     # ------ 06.01.21 ------
30     pushl %esp
31     call signals_check      # check for pending signals, if there are, handle them if you available
32     addl $4, %esp
33     # ------------------------
34     popal
35     popl %gs
36     popl %fs
37     popl %es
38     popl %ds
39     addl $0x8, %esp  # trapno and errcode
40     iret
41
42
43   # ------ 10.01.21 ------
44   .global implicit_sigret_start
45   .global implicit_sigret_end
46   implicit_sigret_start:              # inserting a call to the sigret sys_call
47      movl $SYS_sigret, %eax
48      int $T_SYSCALL                   # User code makes a system call with INT T_SYSCALL.
49   implicit_sigret_end:
50   # ------------------------
```

The function **signals_check(tf)** receiving the old *trapframe* and backing it up for future resotre (as we mentioned in **sigret**). We handeled the signal which was given by poping the head frame from the *cstackframe* of the process – the handling will be the deafult (a simple printing function) if the *sig_handler* is -1. If the ***sig_handler*** is initialized we are executing it by pushing the arguments (***value*** and ***send_pid***) and the ***sig_handler***. It will be executed after we are done with the function **signals_check**. The next PC after the execution of the handler will be the in the stack, where we entered the code for the **implicit_sigret**.



```
622 void signals_check(struct trapframe *tf){
623     struct proc *p = myproc();
624     if (p == 0) //this CPU has no proc defined
625         return;
626     if (p->handling_signal)       // the process is handling a signal, return...
627         return;
628     struct cstackframe* head_frame = pop(&p->pending_signals);
629     if (head_frame == 0) // this popped frame is null
630         return;
631     if (p->sigHandler == (sig_handler)(-1)){      // this is the default signal handler
632         cprintf("(proc.c) signals_check: default sig_handler -  sender_pid %d, recipient_pid %d, value %d\n", head_frame->sender_pid, head_frame->recipient_pid, head_frame->value);
633         return;
634     }
635     p->handling_signal = 1;
636     memmove(&p->tf_backup, p->tf, sizeof(struct trapframe));          //backup of the trapframe
637 
638     uint length = (uint)&implicit_sigret_end - (uint)&implicit_sigret_start;
639     p->tf->esp -= length;                                             // making room for variables
640     memmove((void*)p->tf->esp, implicit_sigret_start, length);        // copying implicit call to sigret to the stack
641     *((int*)(p->tf->esp - 4)) = head_frame->value;                    // value -> to the stack
642     *((int*)(p->tf->esp - 8)) = head_frame->sender_pid;               // sender_pid -> to the stack
643     *((int*)(p->tf->esp - 12)) = p->tf->esp;                          // implicit sigret sys_call return address
644     p->tf->esp -= 12;
645     p->tf->eip = (uint)p->sigHandler;                                 // start the function sigHandler, right after the trapret
646     head_frame->used = 0;                                             // realsing the cstack
647     return;
648 }
649  //-------------------
```

5 Signal Handling

We created the following function in the user space:

```
7   ////passing command line arguments
8
9   void theBestSignalHandler(int pid, int value){
10          printf (1, "new sig_handler: I am the best signal handler my pid %d from recipient_pid %d, value %d\n",getpid(),pid,value);
11  }
12
13  int main(int argc, char **argv)
14  {
15      printf(1, "========= Default sig_handler ==========\n");
16
17      int pid = fork();
18
19      if(pid == 0){
20          sleep(3);
21          printf(1, "Marco has woken up...\n");
22          exit();
23      }else if(pid < 0){
24          printf (1, "This is fork failed\n");
25      }
26      printf(1, "Anna (pid %d) is sending Marco (pid %d) a signal...\n", getpid(), pid);
27      sigsend(pid, 17);
28      wait();
29      printf(1, "Anna has done waiting for Marco...\n");
30
31      printf(1, "========== New sig_handler ===========\n");
32      pid = fork();
33      if(pid == 0){
34          sigset((sig_handler)(&theBestSignalHandler));
35          sleep(3);
36          printf(1, "Marco has woken up...\n");
37          exit();
38      } else if(pid < 0){
39          printf (1, "This is fork failed\n");
40      }
41      sleep(2);
42      printf(1, "Anna (pid %d) is sending Marco (pid %d) a signal...\n", getpid(), pid);
43      sigsend(pid, 17);
44      wait();
45      printf(1, "Anna has done waiting for Marco...\n");
46
47      printf(1, "=========== END TESTING ============\n");
48
49      exit();
50  }
```

We tested the system with 2 signal handlers – the first one is the default signal handler (print only), and the second can be anything, we created *the best signal handler* to handler it.

```
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ sig_test
========= Default sig_handler ==========
Anna (pid 3) is sending Marco (pid 4) a signal...
(sysproc.c) sys_sigsend: from process 3 - to pid 4 with value 17.
(proc.c) signals_check: default sig_handler -  sender_pid 3, recipient_pid 4, value 17
Marco has woken up...
Anna has done waiting for Marco...
========== New sig_handler ===========
(sysproc.c) sys_sigset: for process 5.
Anna (pid 3) is sending Marco (pid 5) a signal...
(sysproc.c) sys_sigsend: from process 3 - to pid 5 with value 17.
new sig_handler: I am the best signal handler my pid 5 from recipient_pid 3, value 17
Marco has woken up...
Anna has done waiting for Marco...
=========== END TESTING ============
$
```

The results, are as we expected.