Report – Assignment 3 – signals

302957824 – שחר בר
340895499 – ראובן פרתי

1. Signal framework

First, we added a new type to xv6 – sig_handler:

```c
typedef void (*sig_handler)(int pid, int value);
```

Then, in order to introduce the signal handling to xv6, we added to the proc structure in proc.h the following fields from line 109:

```c
 95 struct proc {
 96   uint sz;                     // Size of process memory (bytes)
 97   pde_t* pgdir;                // Page table
 98   char *kstack;                // Bottom of kernel stack for this process
 99   enum procstate state;        // Process state
100   int pid;                     // Process ID
101   struct proc *parent;         // Parent process
102   struct trapframe *tf;        // Trap frame for current syscall
103   struct context *context;     // swtch() here to run process
104   void *chan;                  // If non-zero, sleeping on chan
105   int killed;                  // If non-zero, have been killed
106   struct file *ofile[NOFILE];  // Open files
107   struct inode *cwd;           // Current directory
108   char name[16];               // Process name (debugging)
109   sig_handler sighandler;      // Signal handler function address
110   struct cstack cstack;        // Pending signals concurrent stack
111   int ignoreSignals;           // Currently handling signals, or not
112   struct trapframe2 oldTf;     // Backup trapframe for signal handling logic
113 };
```

To handle the signals we created three system calls: **sigset, sigsend** and **sigret**. To do so we added the corresponding fields into **usys.S, syscall.h, syscall.c, defs.h** and **user.h**:

```
SYSCALL(sigset)
SYSCALL(sigsend)
SYSCALL(sigret)
```

```c
#define SYS_sigset   22
#define SYS_sigsend  23
#define SYS_sigret   24
```

```c
extern int sys_sigset(void);
extern int sys_sigsend(void);
extern int sys_sigret(void);
```

```c
sig_handler     sigset(sig_handler);
int             sigsend(int, int);
void            sigret(void);
```

```c
[SYS_sigset]   sys_sigset,
[SYS_sigsend] sys_sigsend,
[SYS_sigret]   sys_sigret,
```

2. Sigset system call

The sigset system call is in charge of replacing the process signal handler with the provided one. It returns the previously stored signal handler.

The signal handler is initialized during the exec system call to be the default -1. If a thread is created by the fork system call, it will inherit its parent process' signal handler.

To implement this functionality, we modified the exec (exec.c) and the fork (proc.c) functions:

```
curproc->sighandler = (sig_handler)(-1);
```

exec.c – line 102

```
np->sighandler = curproc->sighandler;
```

proc.c – line 217

The system call itself (sys_sigset) is defined in sysproc.c and calls a the function sigset(sig_handler) located in proc.c:

```
int sys_sigset(void)
{
    int sighandler;
    argint(0, &sighandler);
    return (int) sigset((sig_handler)sighandler);
}
```

```
sig_handler sigset(sig_handler new_handler)
{
    struct proc *curr_proc = myproc();
    sig_handler old_handler = curr_proc->sighandler;
    curr_proc->sighandler = new_handler;

    return old_handler;
}
```

3. Sigsend system call

The sigsend system call send a signal to a destination process. Because the process cannot handle instantly the received signal, sigsend adds a record to the recipient pending signals stack. It will return 0 on success and -1 on failure.

In order to implement this functionality, we added a new data structure **cstack**, a concurrent stack of **cstackframes** with two functionalities – push and pop, to interact with it (proc.h):

```
struct cstackframe {
  int sender_pid;
  int recepient_pid;
  int value;
  int used;
  struct cstackframe *next;
};
```

```
struct cstack {
  struct cstackframe frames[10];
  struct cstackframe *head;
};
```

```
int push(struct cstack*, int, int, int);
struct cstackframe *pop(struct cstack*);
```

Because each process gets its own stack, we needed to initialize it.

allocproc() line 115-118 in proc.c

```
p->cstack.head = &p->cstack.frames[0];
int i;
for(i = 0; i < 10; ++i)
  p->cstack.frames[i].used = 0;
```

```
struct cstackframe *pop(struct cstack *cstack) {
    struct cstackframe *top = cstack->head;
    if (top == NULL) // stack is empty
        return 0;
    cstack->head->used = 0;
    cstack->head = top->next;
    return top;
}
```

```c
int push(struct cstack *cstack, int sender_pid, int recepient_pid, int value)
{
    struct cstackframe *newSig;
    for (newSig = cstack->frames; newSig < cstack-> frames + 10; newSig++)
    {
        if (newSig->used == 0)
        {
            newSig->used = 1;
            newSig->sender_pid = sender_pid;
            newSig->recepient_pid = recepient_pid;
            newSig->value = value;
            newSig->next = cstack->head;
            cstack->head = newSig;
            return 1;
        }
    }
    // stack is full
    return 0;
}
```

```c
int sigsend(int dest_pid, int value)
{
    struct proc *p;
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if (p->pid == dest_pid)
        {
            int pushed = push(&p->cstack, myproc()->pid, dest_pid, value);
            release(&ptable.lock);
            return pushed - 1; // pushed == 0 -> Error (return -1), pushed == 1 -> Valid (return 0)
        }
    release(&ptable.lock);
    return -1; // dest_pid is not a valid pid
}
```

Each function is located in proc.c. The following code is the sigsend system call in sysproc.c.

```c
int sys_sigsend(void)
{
    int dest_pid, value;
    if (argint(0, &dest_pid) < 0)
        return -1;
    if (argint(1, &value))
        return -1;
    return sigsend(dest_pid, value);
}
```

4. Sigret system call

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.

To implement this functionality we first modified the trapret function (int trapasm.S) to call the checkSignals(struct trapframe) function that checks whether there are signals pending to execution. First we push the argument for the function and then we call the function checkSignals. Because the function might change the CPU registers values, we need to make sure to save the CPU registers values before the execution of the signal handler and restore them after the execution of the signal handler. To do so we added a field in the proc structure (c.f page 1 - old_tf) to save the CPU registers' state before the call.

```
  # Return falls through to trapret...
.globl trapret
trapret:
  pushl %esp
  call checkSignals  // Check for pending signals
  addl $4, %esp
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

Finally once we return from the checkSignals, we added an implicit call to the sigret system call (invokesigret.S) in order to resume the CPU registers' state before the call (that have been saved in old_tf).

```
int sys_sigret(void) {
  sigret();
  return 1;
}
```

```
void sigret(void) {
    struct proc *p = myproc();
    memmove(p->tf, &p->old_tf, sizeof(struct trapframe));
    p->ignoreSignals = 0; // enable handling next pending signal
}
```

```
.globl invoke_sigret_start
.globl invoke_sigret_end
invoke_sigret_start:
  movl $SYS_sigret, %eax
  int $T_SYSCALL
invoke_sigret_end:
```

```
//invokesigret.S
void            invoke_sigret_start(void);
void            invoke_sigret_end(void);
```

```
void checkSignals(struct trapframe *tf) {
    struct proc *p = myproc();
    if (p == 0)
        return; // no proc is defined for this CPU
    if (p->ignoreSignals)
        return; // currently handling a signal
    if ((tf->cs & 3) != DPL_USER)
        return; // CPU isn't at privilege level 3, hence in user mode

    struct cstackframe *top = pop(&p->cstack);
    if (top == (struct cstackframe *) 0)
        return; // no pending signals
    if (p->sighandler == (sig_handler) -1)
        return; // default signal handler, ignore the signal
    p->ignoreSignals = 1;

    memmove(&p->old_tf, p->tf, sizeof(struct trapframe));//backing up trap frame

    uint length = (uint) &invoke_sigret_end - (uint) &invoke_sigret_start;
    p->tf->esp -= length;
    memmove((void *) p->tf->esp, invoke_sigret_start, length);
    *((int *) (p->tf->esp - 4)) = top->value;
    *((int *) (p->tf->esp - 8)) = top->sender_pid;
    *((int *) (p->tf->esp - 12)) = p->tf->esp; // sigret system call code address
    p->tf->esp -= 12;
    p->tf->eip = (uint) p->sighandler; // trapret will resume into signal handler
    top->used = 0; // free the cstackframe
}
```

Note that if there is a a pending signal, that is different from the default signal handler, we handle it by poping the head frame from the stack and pushing the required arguments (value and sender_pid) and the corresponding sighandler location. It will be executed after the checkSignals function. Note the next PC after the signal handler will be the implicit_sigret.

5. Tests:

We performed a total of 4 tests to verify our implementation. First we created two signal handlers – simple_handler1 and simple_handler2 as followed.
Note fibonacci(int n) is the n-th element in the fibonacci series.

```
void
simple_handler1(int pid, int value){
    printf(1, "handler1 exec, sender = %d, value = %d\n", pid, value);
}

void
simple_handler2(int pid, int value){
    printf(2, "handler2 exec, sender = %d, value = %d, (fibonacci(value) == %d)\n", pid, value, fibonacci(value));
}
```

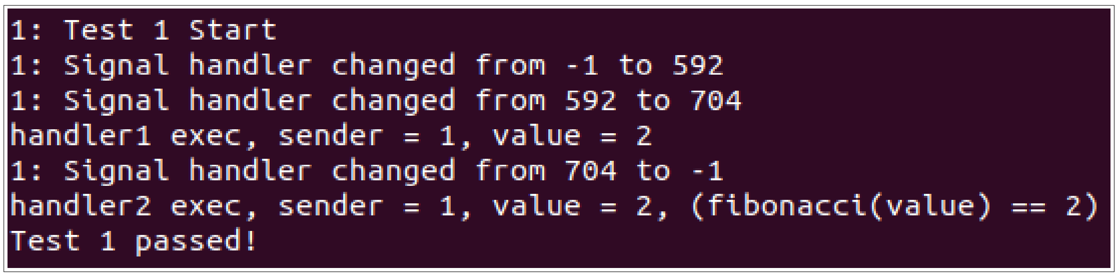
a. Test 1 – Sigset

Input:

```
int child_pid;
int father_pid = getpid();
sig_handler handler;

// Test 1: Check that the element returned by sigset is indeed the previous signal handler
printf(1, "1: Test 1 Start\n");
handler = sigset(&simple_handler1);
printf(1, "1: Signal handler changed from %d to %d\n", handler, &simple_handler1);
handler = sigset(&simple_handler2);
printf(1, "1: Signal handler changed from %d to %d\n", handler, &simple_handler2);
handler(1, 2);
handler = sigset((sig_handler) -1);
printf(1, "1: Signal handler changed from %d to %d\n", handler, -1);
handler(1, 2);
printf(1, "Test 1 passed!\n\n");
```

Output:



```
1: Test 1 Start
1: Signal handler changed from -1 to 592
1: Signal handler changed from 592 to 704
handler1 exec, sender = 1, value = 2
1: Signal handler changed from 704 to -1
handler2 exec, sender = 1, value = 2, (fibonacci(value) == 2)
Test 1 passed!
```

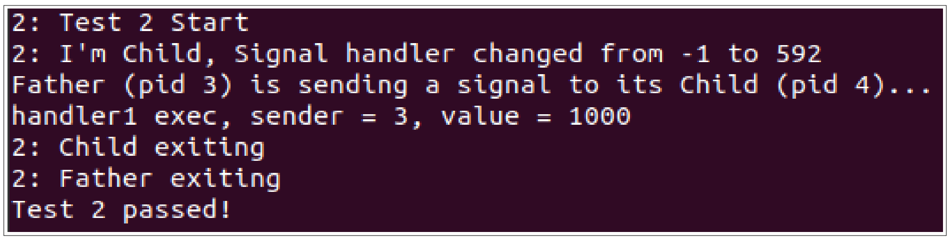
Explanation: In this test we changed 3 times the signal handler of the main process. As we can see during the first call, the handler returned from the system call is -1 which is the default one (as expected). The second is the address of simple_handler1 and the third is the address of simple_handler2. As predicted when calling the handler returned we get that the associated functions are called.

b. Test 2 – Sigsend

Input:

```c
// Test 2: Check that when exiting, the child will handle the signal sent by his father with the value 100
printf(1, "2: Test 2 Start\n");
child_pid = fork();
if(child_pid == 0) // child pid
{
    handler = sigset(&simple_handler1);
    printf(1, "2: I'm Child, Signal handler changed from %d to %d\n", handler, &simple_handler1);
    Busy(30);  // Busy
    printf(1, "2: Child exiting\n"); //the process ignored the signal
    exit();
}
else // parent
{
    Busy(30); // Busy
    printf(1, "Father (pid %d) is sending a signal to its Child (pid %d)...\n", father_pid, child_pid);
    sigsend(child_pid, 1000);
    wait();
    printf(1, "2: Father exiting\n");
}
printf(1, "Test 2 passed!\n\n");
```

Output:



```
2: Test 2 Start
2: I'm Child, Signal handler changed from -1 to 592
Father (pid 3) is sending a signal to its Child (pid 4)...
handler1 exec, sender = 3, value = 1000
2: Child exiting
2: Father exiting
Test 2 passed!
```

Explanation: This test checks the proper execution of the signal handler sent from a father to its child. As we can see, the child first change its handler to be simple_handler1 and its father sends a signal. When the son’s thread goes from the user to the kernel space (during printf) the signal handler is executed, with the corresponding arguments.

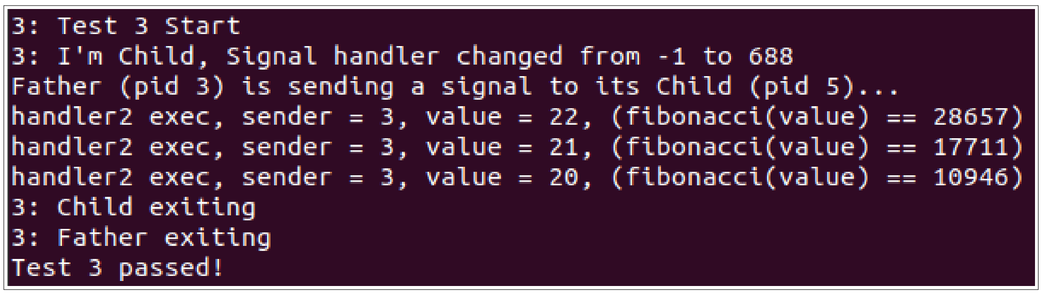
c. Test 3 – Sigret and cstack

Input:

```c
//test 3 - Check that pending signals' stack works (father sends 3 signals)
printf(1, "3: Test 3 Start\n");
child_pid = fork();
if(child_pid == 0)  // child
{
    handler = sigset(&simple_handler2);
    printf(1, "3: I'm Child, Signal handler changed from %d to %d\n", handler, (int) &simple_handler2);
    Busy(30);
    printf(1, "3: Child exiting\n"); //the process ignored the signal
    exit();
}
else{
    Busy(30);
    printf(1, "Father (pid %d) is sending a signal to its Child (pid %d)...\n", father_pid, child_pid);
    sigsend(child_pid, 20);
    sigsend(child_pid, 21);
    sigsend(child_pid, 22);
    wait();
    printf(1, "3: Father exiting\n");
}
printf(1, "Test 3 passed!\n\n"); //the process ignored the signal

exit();
```

Output:



```
3: Test 3 Start
3: I'm Child, Signal handler changed from -1 to 688
Father (pid 3) is sending a signal to its Child (pid 5)...
handler2 exec, sender = 3, value = 22, (fibonacci(value) == 28657)
handler2 exec, sender = 3, value = 21, (fibonacci(value) == 17711)
handler2 exec, sender = 3, value = 20, (fibonacci(value) == 10946)
3: Child exiting
3: Father exiting
Test 3 passed!
```

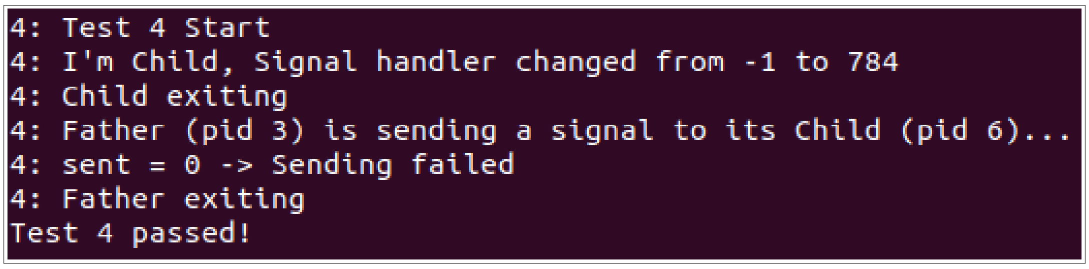
Explanation: This time, similarly to the previous test, the father sends signals to its son. However this time there are several (3) signals pending. Because the signals are introduced into a stack, the first to be executed will be the last one pushed (LIFO). Hence, as we can see, the execution is in the right order (22, 21 and 20).

d. Test 4 – Signal lost

Input:

```
// Test 4 - Signal lost, the son exit before the father send the signal
printf(1, "4: Test 4 Start\n");
child_pid = fork();
if(child_pid == 0) // child pid
{
    handler = sigset(&simple_handler1);
    printf(1, "4: I'm Child, Signal handler changed from %d to %d\n", handler, &simple_handler1);
    printf(1, "4: Child exiting\n"); //the process ignored the signal
    exit();
}
else // parent
{
    Busy(30); // Busy
    printf(1, "4: Father (pid %d) is sending a signal to its Child (pid %d)...\n", father_pid, child_pid);
    int sent = sigsend(child_pid, 1000);
    printf(1, "4: sent = %d -> %s\n", sent, sent ? "Signal sent!" : "Sending failed");
    wait();
    printf(1, "4: Father exiting\n");
}
printf(1, "Test 4 passed!\n\n");
```

Output:



```
4: Test 4 Start
4: I'm Child, Signal handler changed from -1 to 784
4: Child exiting
4: Father (pid 3) is sending a signal to its Child (pid 6)...
4: sent = 0 -> Sending failed
4: Father exiting
Test 4 passed!
```

Explanation: As we can see, the child finishes running before its father has a chance to send the signal. Hence, during the sigsend system call, when the kernel will look for the child_id (6) to add the pending signal to its cstack, it won’t find the corresponding pid and will terminate with an error (return 0).