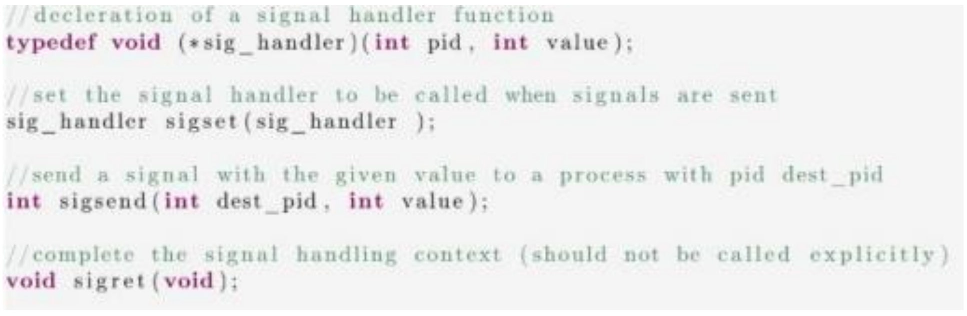
Operating Systems Synchronization and Signals

Main Goal: To implement a basic signal framework which will allow to send basic data between processes.

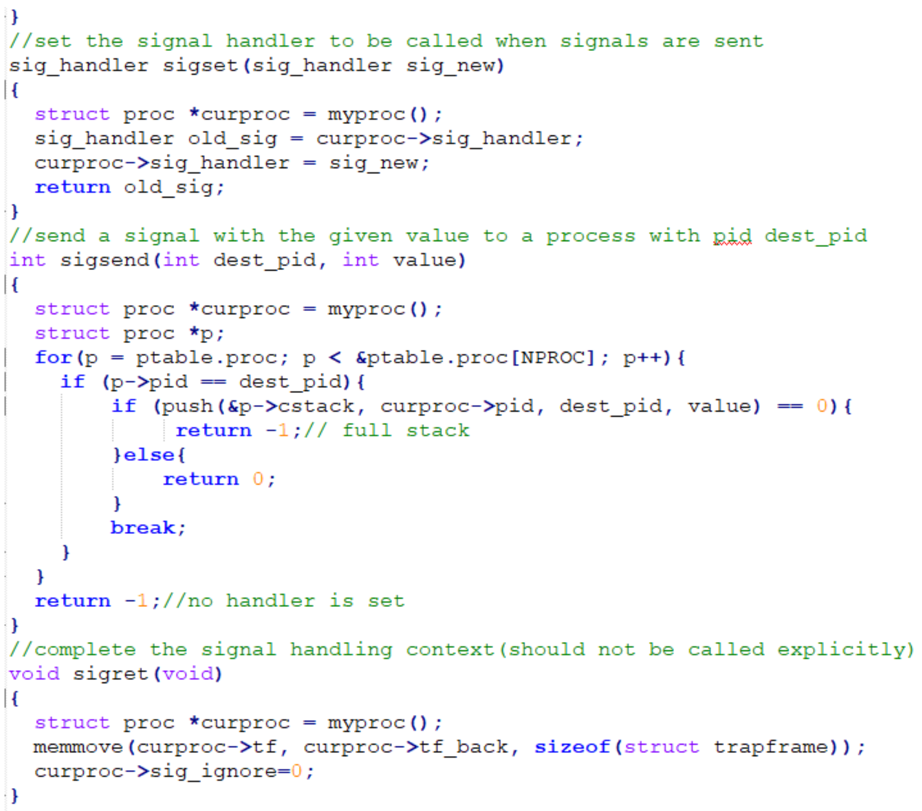
Part 1: Signal Framework

This implementation will support only a single signal handler which can receive two numbers: the pid of the sending processes and the other is a number that is sent by the sending process. The signal framework will include 3 system calls:



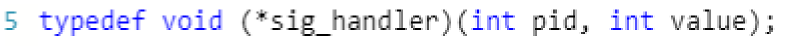
```
//decleration of a signal handler function
typedef void (*sig_handler)(int pid, int value);

//set the signal handler to be called when signals are sent
sig_handler sigset(sig_handler );

//send a signal with the given value to a process with pid dest_pid
int sigsend(int dest_pid, int value);

//complete the signal handling context (should not be called explicitly)
void sigret(void);
```

As you can see the implementation of these system calls is in the proc.c file:



```
}
//set the signal handler to be called when signals are sent
sig_handler sigset(sig_handler sig_new)
{
  struct proc *curproc = myproc();
  sig_handler old_sig = curproc->sig_handler;
  curproc->sig_handler = sig_new;
  return old_sig;
}
//send a signal with the given value to a process with pid dest_pid
int sigsend(int dest_pid, int value)
{
  struct proc *curproc = myproc();
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->pid == dest_pid){
        if (push(&p->cstack, curproc->pid, dest_pid, value) == 0){
            return -1;// full stack
        }else{
            return 0;
        }
        break;
    }
  }
  return -1;//no handler is set
}
//complete the signal handling context(should not be called explicitly)
void sigret(void)
{
  struct proc *curproc = myproc();
  memmove(curproc->tf, curproc->tf_back, sizeof(struct trapframe));
  curproc->sig_ignore=0;
}
```

The type of sig_handler is defined in the file types.h:



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

The sigset function defines a new signal handler for the current process – changes the new field sig_handler of the proc struct to the new signal handler it received.

The sigsend function will send a signal according to the dest_pid var received with the value. When it finds the relevant process by its pid it pushes the handler to the process's stack.

The sigret cannot be called explicitly by the user (the implementation for its call in the following parts). It backups the trapframe for the current process.

Part 2: Storing and Changing the signal handler

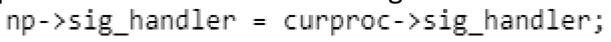
In order to store the signal handler, we've added a new field to struct proc at proc.h. This field will hold a pointer to the current handler (or -1 if no handler is set). As we learned, both fork and exec system calls modify the signal handlers. We've copied the signal related behavior of fork and exec for our signal's implementation.

• The fork system call will copy the parent process' signal handler to the newly created child process

• The exec system call reset the signal handler to be the default (-1)

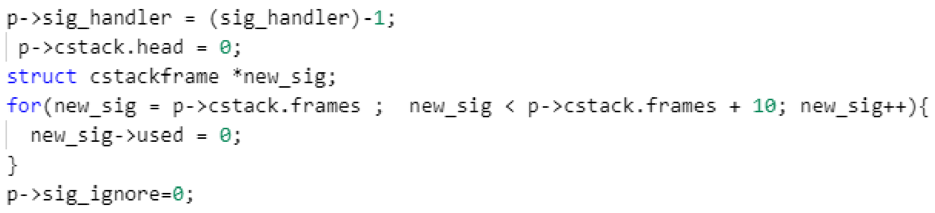
The new sigset system call replaced the process signal handler with the provided one and returns the previously stored signal handler.

In order to implement this, we made changes to exec and fork. In addition, we added new fields to the relevant structs in proc.h file.

1. In the fork function(proc.c) we added the next row in order to copy the parent handler when executing fork:


```
np->sig_handler = curproc->sig_handler;
```

2. In allocproc function(proc.c) we added rest for the new fields of proc (signal handler sets to the default handler):



```
p->sig_handler = (sig_handler)-1;
 p->cstack.head = 0;
struct cstackframe *new_sig;
for(new_sig = p->cstack.frames ;  new_sig < p->cstack.frames + 10; new_sig++){
  new_sig->used = 0;
}
p->sig_ignore=0;
```

Part 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.

As mentioned in part 1 the sigsend function seeks the dest_pid process by its pid in the ptable. When it finds the relevant process, it adds the handler from to the process stack.

As you can see the implementation is in proc.s file:

```
int push(struct cstack *cstack, int sender_pid, int recepient_pid, int value) {
  acquire(&ptable.lock);
  struct cstackframe *new_frame;
  int result = 1;
  for (new_frame = cstack->frames; new_frame < cstack->frames + 10; new_frame++){
    if (new_frame->used == 0){
        new_frame->used=1;
        break;
    }
  }
  if (new_frame == cstack->frames + 10) { // stack is full
    result = 0;
  }
  else {
    new_frame->sender_pid = sender_pid;
    new_frame->recepient_pid = recepient_pid;
    new_frame->value = value;
    do{
        new_frame->next = cstack->head;
    } while (cstack->head != new_frame->next);
    cstack->head = new_frame;
  }
  release(&ptable.lock);
  return result;
}

struct cstackframe *pop(struct cstack *cstack) {
  acquire(&ptable.lock);
  struct cstackframe *top;
  do{
      top = cstack->head;
      if(top==0){
          break;
      }
  }while (cstack->head != top);
  if(top!=0){
    cstack->head = top->next;
    top->used = 1;
  }
  release(&ptable.lock);
  return top;
}
```

Multiple process are able to send a handler to the same process at the same time, therefore we are using the ptable.lock for keeping synchronization.

The push function seeks the last available frame (if the stack is full it returns 0). When it finds a frame, it sets all the relevant fields and updates the stack.

The pop function removes the relevant frame from the top of the stack.

Part 4: Signal Handling

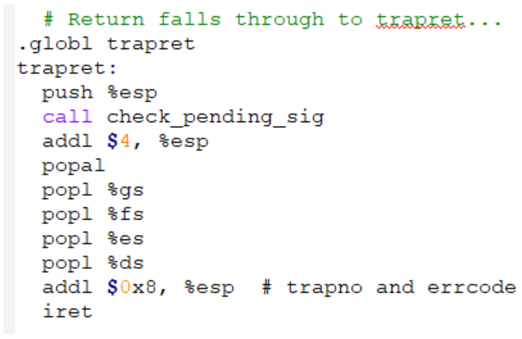
The signal handler can change the CPU registers values, this can cause unpredictable behavior of the user space program once jumping back to the original code.

In order to solve this problem, we saved the CPU registers values before the execution of the signal handler and restored them after the execution of the 3-signal handler finishes.

We've created a new field inside struct proc that holds the original registers values. When the signal handler finishes, we return to kernel space in order to restore them, using the sigret system call, which will only restore the CPU registers values for the user space.

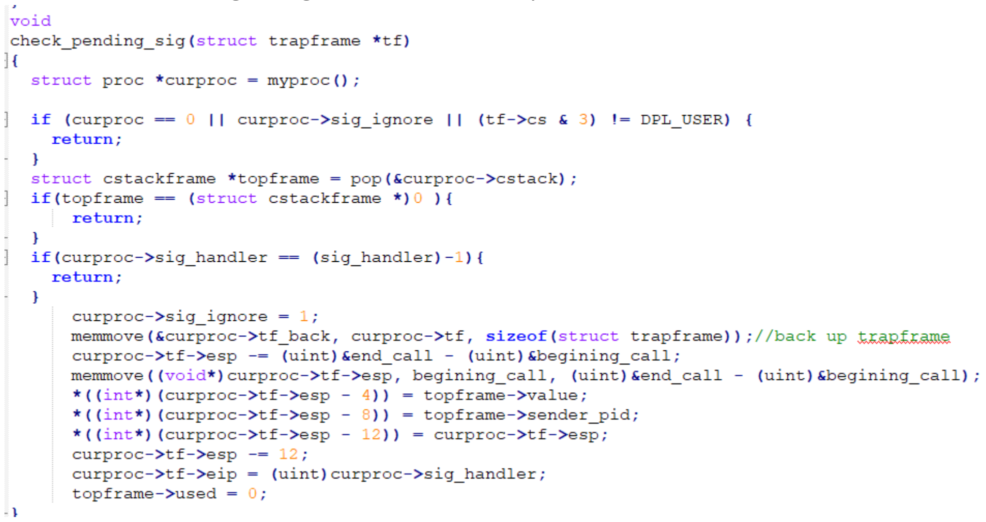
In the userspace we are making a check if there are any pending signals. In case there are, we take the top frame from the stack and call the relevant handler.

As seen here in the code, in trapasm.S file we added the call to the check_pending_sig function (proc.c).



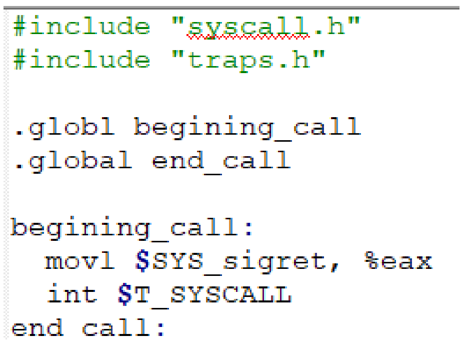
```
  # Return falls through to trapret...
.globl trapret
trapret:
  push %esp
  call check_pending_sig
  addl $4, %esp
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

The check_pending_sig function checks validity of the proc and if there are any frames in the stack it handles the top one. First it backups the trapframe, next it sets the relevant address of the a code calling to sigret function to be performed in the end.



```
void
check_pending_sig(struct trapframe *tf)
{
  struct proc *curproc = myproc();

  if (curproc == 0 || curproc->sig_ignore || (tf->cs & 3) != DPL_USER) {
    return;
  }
  struct cstackframe *topframe = pop(&curproc->cstack);
  if(topframe == (struct cstackframe *)0 ){
      return;
  }
  if(curproc->sig_handler == (sig_handler)-1){
    return;
  }
      curproc->sig_ignore = 1;
      memmove(&curproc->tf_back, curproc->tf, sizeof(struct trapframe));//back up trapframe
      curproc->tf->esp -= (uint)&end_call - (uint)&begining_call;
      memmove((void*)curproc->tf->esp, begining_call, (uint)&end_call - (uint)&begining_call);
      *((int*)(curproc->tf->esp - 4)) = topframe->value;
      *((int*)(curproc->tf->esp - 8)) = topframe->sender_pid;
      *((int*)(curproc->tf->esp - 12)) = curproc->tf->esp;
      curproc->tf->esp -= 12;
      curproc->tf->eip = (uint)curproc->sig_handler;
      topframe->used = 0;
}
```

The beginning_call and end_call are fround in a new assembly file we added – call_sys_sigret.S.



```
#include "syscall.h"
#include "traps.h"

.globl begining_call
.global end_call

begining_call:
  movl $SYS_sigret, %eax
  int $T_SYSCALL
end call:
```

We implemented this in order to make sure the sigret is getting called.

5: Tests

After running the “make qemu” run “test”. We added the file test.c that will be executed:

1. First we test the sigset (it can be called from userspace):

```
printf(1,"sigset test:\n");
printf(1,"signal handler: %d\n", (int)sigset((sig_handler)5));
printf(1,"signal handler: %d\n", (int)sigset((sig_handler)3));
```

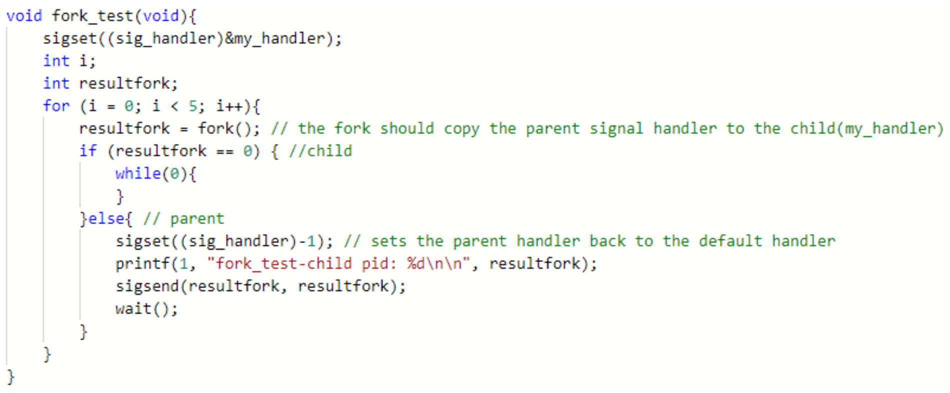
We print in order to check it overrides the current signal handler.

2. Next, we test the stack. Therefore, we created a new function in proc.c named testing that executes push and pop. We call that function and expect that the first 10 pushes will return 1 – push is ok, and the 11 push will return 0 – stack is full. In addition, we expect to see the current values in the first 10 pops and the 11 pop should return an undefined value.

```
void
testing(void)
{
  struct proc *curproc = myproc();
    struct cstack* test_s = &(curproc->cstack);
  cprintf("start testing\n");
    cprintf("push1: %d\n", push(test_s, 1,2,1));
  cprintf("push2: %d\n", push(test_s, 1,2,2));
  cprintf("push3: %d\n", push(test_s, 1,2,3));
  cprintf("push4: %d\n", push(test_s, 1,2,4));
  cprintf("push5: %d\n", push(test_s, 1,2,5));
  cprintf("push6: %d\n", push(test_s, 1,2,6));
  cprintf("push7: %d\n", push(test_s, 1,2,7));
  cprintf("push8: %d\n", push(test_s, 1,2,8));
  cprintf("push9: %d\n", push(test_s, 1,2,9));
  cprintf("push10: %d\n", push(test_s, 1,2,10));
  cprintf("push11: %d\n", push(test_s, 1,2,11));

  cprintf("pop1: %d\n", pop(test_s)->value);
  cprintf("pop2: %d\n", pop(test_s)->value);
  cprintf("pop3: %d\n", pop(test_s)->value);
  cprintf("pop4: %d\n", pop(test_s)->value);
  cprintf("pop5: %d\n", pop(test_s)->value);
  cprintf("pop6: %d\n", pop(test_s)->value);
  cprintf("pop7: %d\n", pop(test_s)->value);
  cprintf("pop8: %d\n", pop(test_s)->value);
  cprintf("pop9: %d\n", pop(test_s)->value);
  cprintf("pop10: %d\n", pop(test_s)->value);
  cprintf("pop11: %d\n", pop(test_s)->value);
}
```

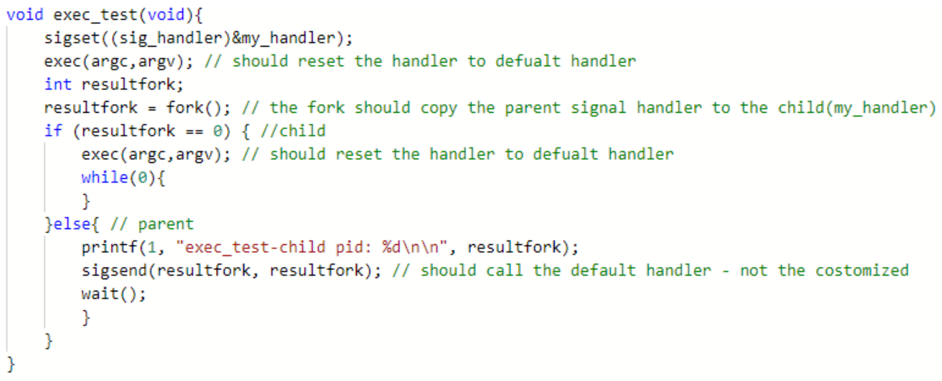
3. Fork_test: we want to check if the new child process copies the parent signal handler. Therefore, we run a loop that creates 5 child processes and the parent call their customized handler (they inherit from their parent)by calling sigsend. In the end of the signal handling the sigret will be called (not from userspace).

4. Fork_test: we test the default handler by calling sigset in the parent code – resting the handler to the default one.
5. Fork_test: as mentioned before we check that the child processes are executing the new costumed handler we set to the parent.



```
void fork_test(void){
    sigset((sig_handler)&my_handler);
    int i;
    int resultfork;
    for (i = 0; i < 5; i++){
        resultfork = fork(); // the fork should copy the parent signal handler to the child(my_handler)
        if (resultfork == 0) { //child
            while(0){
            }
        }else{ // parent
            sigset((sig_handler)-1); // sets the parent handler back to the default handler
            printf(1, "fork_test-child pid: %d\n\n", resultfork);
            sigsend(resultfork, resultfork);
            wait();
        }
    }
}
```

6. Exec_test: we check that when we use the exec function the signal handler is being set to the default one.



```
void exec_test(void){
    sigset((sig_handler)&my_handler);
    exec(argc,argv); // should reset the handler to defualt handler
    int resultfork;
    resultfork = fork(); // the fork should copy the parent signal handler to the child(my_handler)
    if (resultfork == 0) { //child
        exec(argc,argv); // should reset the handler to defualt handler
        while(0){
        }
    }else{ // parent
        printf(1, "exec_test-child pid: %d\n\n", resultfork);
        sigsend(resultfork, resultfork); // should call the default handler - not the costomized
        wait();
        }
    }
}
```