# Operating Systems for Computer Engineering
# Assignment 2

December 10, 2018

**Abstract**

In practical session 3 you were introduced to the idea of signals. Using signals is a method of Inter-Process Communication (IPC). In this part of the assignment you are requested to implement a basic signal framework which will allow you to send basic information (an integer) between processes.

## 1 Signal Framework

This implementation will support only a single signal handler but unlike the signals you learned about in the practical session, this signal handler 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 that you will create includes 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);
```

The following subsection describes in detail the above system calls together with their implementation requirements. You are required to follow the instructions given and implement the requested data-structures and system calls.

## 2 Storing and Changing the signal handler

In order to store the signal handler, you will add a new field to struct proc (see ***proc.h***). This field will hold a pointer to the current handler (or -1 if no handler is set). As described in practical session 3, both fork and exec system calls modify the signal handlers.

We will copy the signal related behaviour of fork and exec for our signals implementation.

- The ***fork*** system call will copy the parent process' signal handler to the newly created child process
- The ***exec*** system call will reset the signal handler to be the default (-1)

The new ***sigset*** system call will replace the process signal handler with the provided one and return the previously stored signal handler.

## 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 handeled 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.

```
// defines an element of the concurrent struct
struct cstackframe {
  int sender_pid;
  int recepient_pid;
  int value;
  int used;
  struct cstackframe *next;
};

// defines a concurrent stack
struct cstack {
  struct cstackframe frames[10];
  struct cstackframe *head;
};

// adds a new frame to the cstack which is initialized with values
// sender_pid, recepient_pid and value, then returns 1 on success and 0
// if the stack is full
int push(struct cstack *cstack, int sender_pid, int recepient_pid, int value);

// removes and returns an element from the head of given cstack
// if the stack is empty, then return 0
struct cstackframe *pop(struct cstack *cstack);
```

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).

# 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., you should 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) or executing a signal handler when it returns to user space. In order to force the execution of the signal handler in user space you will have to modify the user space stack and the instruction pointer of the process. This requires knowledge of conventions of function call. You can refresh your your memory regarding function call conventions here.

There are three major steps that must be performed in order to make a function call:

- Push the arguments for the called function on the stack
- Push the return address on the stack
- Finally jump to the body of the called function

Pushing arguments and the return address on the user space stack is a straightforward operation over the process ***trapframe*** once you understand the function call conventions.

In order to execute the body of the signal handler upon return to user space, one must update the instruction pointer's value to be the address of the desired function.

When the signal handler finishes its run, the user space program should continue from the point it was stopped before execution of the signal handler. Thus, one can naturally think that the return address that should be placed on the stack as the return address of the signal handler should be the previous instruction pointer (before changing it to point to the signal handler). However, this will not work. Since the signal handler can change the CPU registers values this can cause unpredictable behaviour of the user space program once jumping back to the original code.

In order to solve this problem, you must save the CPU registers values before the execution of the signal handler and restore them after the execution of the

signal handler finishes. You should create a new field inside ***struct proc*** that will hold the original registers values. When the signal handler finishes, your code must return to kernel space in order to restore them. This is the responsibility of the ***sigret*** system call, which will only restore the CPU registers values for the user space execution (you may backup the whole old ***trapframe*** in a new field inside ***struct proc***).

The main problem here is that the signal handler can accidently not call the sigret system call on exit and this may cause an unpredictable behavior in the user code. To solve this problem, you need to "inject" an implicit call to the sigret system call after the call to the signal handler. This can be done by putting this code onto the user space stack and setting the return address from the signal handler to point to the beginning of the injected code. You can do so by using the ***memmove*** function which is located at ***string.c*** .

# 5 Hints

## 5.1 Storing and Changing the signal handler

### 5.1.1 Fork and Exec system calls

The fork copy the new allocated process and you have to copy the parent's signal handler to child signal handeler ,the changes have to be in fork(void) function on ***proc.c*** mean while the exec should initialize the signal handler to -1 , the changes have to be in ***exec.c*** file

### 5.1.2 Defining System Call and Signal Handler Prototype

You should add the three system calls as you done in the previous homework, the prototype typedef void (*sig_handler)(int pid, int value) you can add it to the file ***types.h***

### 5.1.3 Set Signal Handler

You should using ***sigset*** to override the current signal handler,in main function you have to use it sigset((sig_handler)(address of your own function)) it's the same concept of override "signal handler" we learned in the class .

## 5.2 Sending a signal to a process

### 5.2.1 cstackframe and cstack

***cstackframe*** struct represent information of sent signal from sender to receiver while the ***cstack*** is represent stack using array of ***cstackframe*** and pointer, you should implement the 2 main function "push" and "pop" for the stack.
to refresh your memory how can pointer and array implement stack you can see link here.