COL331: Operating Systems **Date: February 24, 2019**

Assignment 1 Report

Name: Manish Tanwar **Entry Number: 2016CS10363**

1 Inter-Process Communication:

Unicasting:

- A queue is added for each process to hold the pending messages for that process.

```
struct msg_queue
{
  struct spinlock lock;
  char data[BUFFER_SIZE][MSGSIZE];
  int start;
  int end;
  int channel; // channel is used for sleep and wakeup
}MsgQueue[NPROC];
```

- System Calls:

```
int send(int sender_pid, int rec_pid, void *msg);
int recv(void *msg);
```

- Sending a Message: Syscall `send` enqueues the message to the receiver's message queue, it also wakeups the receiver process as it might be waiting for a message. It returns error(-1) if the queue is full.
- Receiving a Message: Syscall `recv` dequeues a message from it's message queue and stores it in `msg`. If the queue is empty, than the receiving process is blocked by calling `sleep` on a certain channel, so it's a blocking call.

Signal Handling:

- **Signal Handler Function Signature:**

```
typedef void (*sighandler_t)(void*);
```

  - `sighandler_t` is a function pointer which takes a void pointer as an argument and returns void.

- **System Calls for Signal Handling:**

```
// Set Signal handler for sig_num type where sig_num is b/w 0 to 3
int sig_set(int sig_num, sighandler_t handler);

/* Sends a signal to dest_pid process' sig_num Signal Handler
with argument sig_arg(Pointing to an 8 bytes message) */
int sig_send(int dest_pid, int sig_num, void *sig_arg);

// Block the process until a signal is received
int sig_pause(void);

/* Syscall for returning from signal handling(called by
wrapping code on stack) */
int sig_ret(void);
```

- **Added Fields in Proc Struct:**
  - Signal Handlers : It stores function pointers to signal handlers for that process, i.e if another process sends this process a signal with `sig_num` then the Signal Handler is implemented which is pointed by the stored function pointer. Pointer is set to 0, if no signal handler is set.
  - Signal Queue : A queue to store the pending signals for that process. It stores `sig_num` and `sig_arg` for each pending signal. The queue is implemented using an array with constant size 256 (circular array implementation).

```
struct sig_queue
{
  struct spinlock lock;
  char sig_arg[SIG_QUE_SIZE][MSGSIZE];
  int sig_num_list[SIG_QUE_SIZE];
  int start; // works as a channel(for sleep and wakeup) also
  int end;
};
```

- **Setting a Signal Handler:**
  - Function Signature:

```
int sig_set(int sig_num, sighandler_t handler);
```

  - The system call `sig_set` is called by the receiver to set the signal handler's function pointer.
  - The system call `fork` copies the parent's signal handlers to child's process.
  - The system call `exec` resets the signal handlers to default value 0.
  - The function `allocproc` resets the `sig_queue` to an empty queue.
- **Sending a Signal:**
  - Function Signature:

```
int sig_send(int dest_pid, int sig_num, void *sig_arg);
```

  - The sender calls syscall `sig_send` for sending a signal to process with pid `dest_pid`.
  - The receiver process will be implemented it's signal handler indexed with `sig_num` with argument `sig_arg`, only if there is a signal handler set to `sig_num` by the receiver already.
- **Handling a Signal:**
  - Whenever a process is about to return from kernel mode to user mode with function call `trepret` (implemented in trapasm.S) it checks the signal queue by calling the function `execute_signal_handler`.
  - Function `execute_signal_handler` does the following:
    - If there is no signal available for this process, then it returns back to `trepret`.
    - **Setting up user stack:** (for calling the signal handler)
      1. For saving the user mode's context, it saves the trapframe of this process from the kernel stack to the user mode stack.
      2. It changes the register `%eip` (stored in the trapframe) to the function pointer of the signal handler.
      3. We can push the parameter `sig_arg` on the stack, but this pointer points to an address in kernel space(which can't be accessed by the signal handler which is implemented in the user mode), so we first push 8 bytes of the message on the stack and then we push the pointer to the first byte pushed on to the stack as a pointer.
      4. Returning back to kernel mode is necessary to retrieve back the user mode context.
      5. For returning back from the user mode to kernel mode, we have to make a syscall `sig_ret` but the signal handler might not call this syscall due to other interrupts. To solve this problem we wrap up the system call code(written in `x86` assembly) on the stack and call it implicitly by pushing the return address for signal handler as the wrapped code's first instruction.

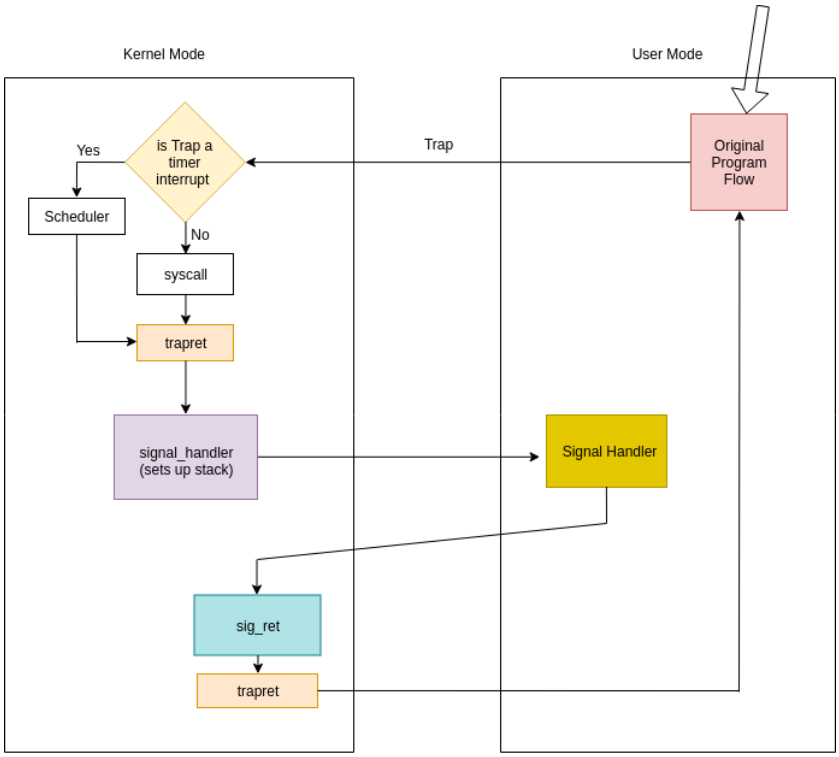

Figure 1: Flow Control of Receiver Processor on Signal Handling

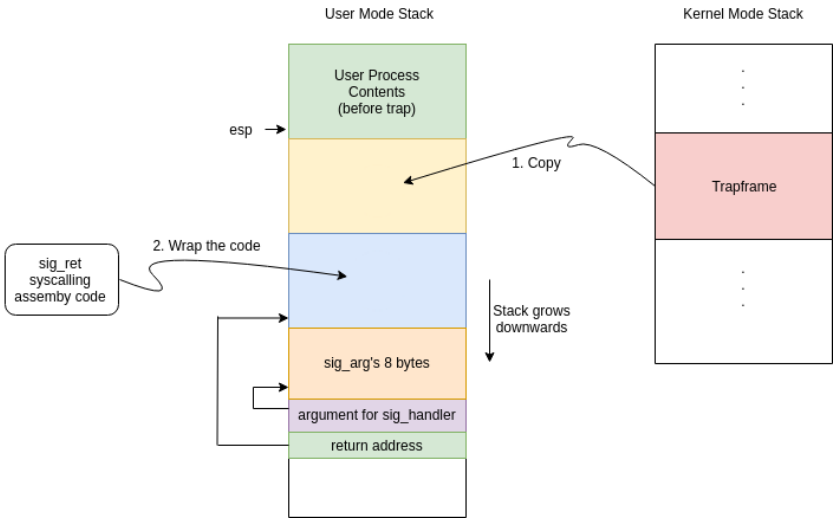

Figure 2: Setting up the User Stack

6. Return back to `trapret` function to call user mode's signal handler.

– Returning from Signal Handling:

* When the signal handler executes the wrapped code on the stack, it calls `sig_ret` syscall.
* `sig_ret` syscall just save the trapfram back from user stack to the kernel stack which retrieves back the original user mode context.

- **Waiting for a Signal:**

– A process may want to wait for a signal to arrive.

– A process can make `sig_pause` to get blocked until a signal arrives for it.

– `sig_pause` checks if there is any signal available for this process if there is not then it gets blocked by calling function `sleep` on a certain channel.

– Syscall `sig_send` wakeups the receiver process which might be waiting for a signal.

Multicasting:

- System Call Signature:

```
int send_multi(int sender_pid,int rec_pids[],void *msg,int rec_length);
```

- First, all the receiver has to set their signal handler (with default `sig_num` = 0) for receiving a message.
- Syscall `send_multi` sends signals to all the receiver processes (with pid's as `rec_pids[]`) with `sig_num` = 0 with the argument `sig_arg` = `msg`.
- Message is delivered when the receiver processes switch from kernel mode to user mode after a trap.

2 Distributed Algorithm:

- The coordinator(parent) process forks n child process(workers). n is kept equal to 8 but can be scaled up. (tested up to 32)
- Every worker process is assigned equal size subarrays to proceed on (with an exception of the last child process).
- Every worker process computes the partial sum of subarray assigned and sends it to the coordinator using unicasting function implemented.
- The coordinator collects the partial sums and computes the total sum. Then it multicasts the mean to each worker process.
- Every worker process computes the sum of the squares of the differences about the mean and sends to the coordinator using unicasting.
- The coordinator collects the partial square sums and computes the variance.

3 Extra Efforts:

- Traps are not disabled during the execution of the signal handler.
- We can make system calls in the signal handler.
- Can handle nested executions of signal handlers.
- Support of multiple signal handlers.
- Implemented a syscall waiting for a signal(blocking call).
- The message queue is implemented efficiently.
- The maximum number of worker processes in the distributed algorithm can be up to 32 as well.