COL331 A1 Report

Sachin Kumar Prajapati
2016CS10355

2 - System Calls

- ***sys_print_count*** : An array is maintained which saves the count of the number of times the system call has been called. On invoking the system call. The system calls containing non-zero counts get printed on screen with their respective count
- ***sys_toggle***: A variable toggle is kept which changes its value from 0 to 1 or from 1 to 0 on each invocation. Whether the count of system call is incremented or not depends on the value of toggle. Also, on each change of toggle from 0 to 1, the array containing the count is cleared to 0.
- ***sys_add:*** Simply take the arguments of 2 int and returns their sum.
- ***sys_ps:*** Lists all the process whose state is not **UNUSED** with name and pid.

3-IPC: Unicast

- A struct for **message queue** (as shown in following figure)is added in the struct for proc in **proc.h**(for storing the messages). The variable head and tail are initialised to 0 at the time of allocproc as all process start with empty queue. **NUM_MSG=32, MSGSIZE=8.**

```
43 struct message_list{
44   int sender_id[NUM_MSG];
45   char messages[NUM_MSG][MSGSIZE];
46   int head;
47   int tail;
48 };
49
```

- On invoking ***sys_send*** , the process enqueues the message(**8 bytes**) into the message queue of the receiving process using **memmove()** function. In addition to this it also wakes up the receiving process by using **wakeup()** in case it was already expecting a message and was blocked.
- On invoking the ***sys_recv,*** the process first checks if its message queue is empty, if this is the case then it goes to sleep using **sleep()** till the time some process sends it a message and it gets wakeup again, else it dequeues the first message from the queue and copies it into the user space using function **memmove()**.

4-IPC: Multicast

Implementation of Signal Handler:-

- Following system calls are added in order to implement interrupt handler(the usage is given as comment):-

```
28 int sig_set(signal_handler);  //For setting up the signal handler for a user program
29 int sig_send(int to_pid, int signum); //For sending a signal(signal no. = signum) to process with pid=to_pid
30 int sig_ret(void);  //Used internally by kernel to return from the signal handler
31 int sig_pause(void); //For waiting a signal to arrive
```

- In addition to this, the following struct is added to the proc in proc.h to store the signals received by a process. MAX_SIG is the maximum no. of signals a process can hold at once. For now it's set to 5.

```
52 struct signals{
53   int sig_num[MAX_SIG];
54   int s_pid[MAX_SIG];
55   int head;
56   int tail;
57 };
```

- The signal handler used in this case is defined as a function which takes a void pointer as argument.

```
7  typedef void (*signal_handler)(void *sent_message);
```

4-IPC: Multicast

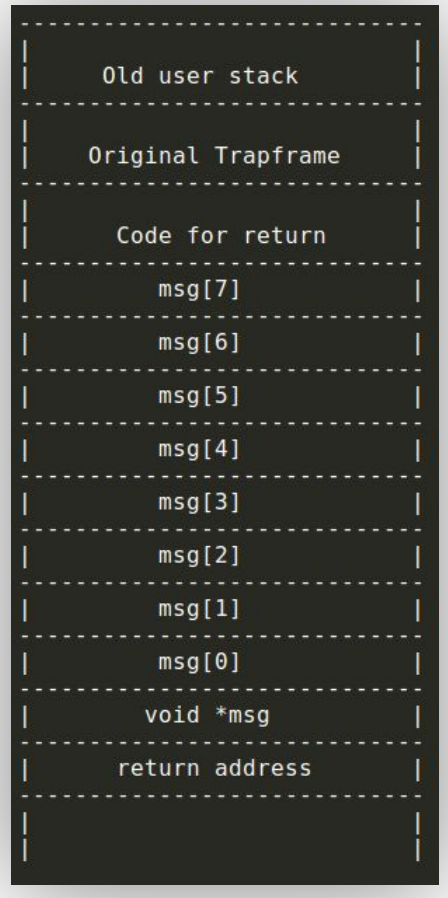
Steps for handling signals:-

1. When a signal is sent to a process it gets queued up in the signal queue of the process.
2. Now whenever this process is about to scheduled on a CPU, a context switch is made. After this the kernel uses the trapframe stored in the process to restore the user context of the process by using the **trapret** function.
3. Now for handling signals, the **trapret** function is modified. The **trapret** function now before doing its regular work first checks whether a signal has been sent to this process and handles the signal if it is the case.
4. To handle the signals, the function called by the trapret (**handle_signals()** in this case) modifies the original trapframe of the process in order to handle the signals.
5. The following modifications are done to the stack stored in the trapframe:-
    - At first the original trapframe is pushed to the user stack(this would be used when we return to the original process after handling signal).
    - Secondly, a assembly code is pushed to the user stack, this code would help us in returning back to the kernel safely after handling the signal. Note the return from signal handler is not given in the hand of user program for safety issues, it is controlled by the kernel, to ensure that the signal handler returns safely to the kernel upon execution.
    - After this, the message to be passed is pushed on to the stack. Following which is the pointer to this message(this will act as the argument for the signal handler)
    - At last, we have to provide a return address for the signal handler. This we give as the pointer to the code we pushed onto the stack. So after returning the control would jump to this code and it would ensure safe return to the kernel.

4-IPC: Multicast

Steps for handling signals:-

6. After making these modifications the user stack will look like this ---------->
7. After having done this, our stack is ready. The last thing which needs to be done is to jump to the signal handler upon reaching back to user. We do this by setting the trapframe's **eip** = pointer to the signal handler of the process.
8. The trapret continues and restores this modified trapframe. The eip is already at the signal handler and the arguments are provided to the stack. So the signal handler runs as a normal user program in the user space. The message is transferred and at the time of return, the control reaches the special code on the stack which is used to return.
9. Now this code, makes a system call ***sig_ret*** , as a system call is made the process goes into the kernel mode and a new trapframe is stored. The job of this system call is to replace this new trapframe with the old trapframe which still resides on the user stack. The user stack could be accessed by the stack pointer in the new trapframe.
10. As, the original trapframe is restored the next context switch to this process will return to user space, with exactly the same context as before handling signal.



Distributed Algorithm: Working

- At first the following signal handler is registered for the process.

```
19  void message_handler(void* msg){
20      //Message received is 8 bytes
21      mean_received = *(float*)msg;
22      sreceived[this_child_id] = 1;
23      return;
24  }
```

- The parent process forks 5 childrens. The pid of these 5 children is stored in array. Each process also gets an id(from 0-4).
- The arr is divided for the childrens to work upon. The child processes then first calculate their partial sum of the array and send it to the parent process using ***sys_send***. The parent process calls ***sys_recv*** 5 times to receive the partial sums. The parent then adds these partial sum to calculate total sum and calculates the mean of arr.
- The parent process then multicasts this calculated mean to its children using ***sys_send_multi.***
- The child processes then again work on their part of array to calculate partial sum for variance and sent it to the parent process using ***sys_send***.
- The parent then collects the partial variance sum using ***sys_recv()*** and calculates the actual variance.
- The parent then calls **wait()** 5 times to wait for child processes to exit and then exits.

Extra Work

- Could handle signals for multiple signal handlers.
- Interrupts are not disabled while executing the signal handler.
- Can make system calls inside signal handlers.
- Return from signal handler is done by the kernel for safety.
- Waiting for signal is blocking hence not waiting CPU cycles.

References

- http://courses.cms.caltech.edu/cs124/lectures/CS124Lec15.pdf
- https://www.cs.bgu.ac.il/~os162/wiki.files/Assignment_2.pdf