COL334 Assignment1

Arshdeep Singh

2016CS50625

1 System Calls

Some changes that are generic to implementation of every system call in xv6.

- Changes need to be made in file usys.S so that the system call can be recognized and correct system call id is put in *eax* register.
- Header syscall.h is also changed to allocate an unique id to new system call.
- Header user.h is changed so that the system call is visible to user program.
- Actual implementation of system call is done in sysproc.c to keep with convention.

1.1 Toggling the trace mode

System call corresponding to `sys_toggle()` has been made which changes the state from TRACE_OFF to TRACE_ON and vice versa. A flag indicator `int show_trace` was kept in `syscall.c` to maintain the status. This also resets the counts that are kept for the `sys_print_count` when the state is set back to TRACE_ON.

1.2 System call trace

Initially the system starts with it's state in TRACE_OFF (`show_trace=0`), and I maintain an indicator in `syscall.c` to modify it, when `sys_toggle()` is called, as mentioned above. I kept an array in the `syscall.c` to maintain the count of the system calls made after the state is set to TRACE_ON. Also, a mapping from system call number to it's name has been maintained. So, now whenever a call is made to `sys_print_count()` it prints all counts corresponding to each system call made after the state is set to TRACE_ON

1.3 Add system call

Add system call has been implemented and a user program has also been made, namely `user_add.c` It simply takes integer arguments using `argint` and returns the result.

1.4 Process List

In this part a system call, `sys_ps()` has been implemented which calls it's helper function, `void print_running(void)` in `proc.c` which goes over the process table and prints all the processes, which are in the state SLEEPING, RUNNABLE or RUNNING.

2 Distributed Algorithm

The distributed algorithm works in the following manner. The coordinator process creates a number of child processes, and the child processes compute the partial sum and send back the partial sum back to the parent/coordinator process. Now, if we want to calculate variance as well, the child process then registers it's signal handler and goes to sleep, until interrupted by the coordinator process. The coordinator process takes the partial sums and calculates the average and the multicasts the sum back to the child processes. The way it is achieved, is that the coordinator process sends a signal back to the receiver, and also wakes it up from sleep. After waking up from sleep the receiver processes calculates the partial variances and sends it back again to the sender using unicast, after which the sender calculates the complete variance.

3 Inter Process Communication

In this assignment, I implemented both unicast and multicast model of IPC.

3.1 Unicast Model

For the unicast model for IPC, the following system calls were implemented

- `int sys_send(int sender_pid, int rec_pid, void *msg)`
- `int sys_recv(void *msg)`

I used an array of 64 queues which corresponds to the maximum number of processes in xv6. Each queue, has a fixed maximum size and stores message of size 8 bytes. Also, each queue is *separately* (improves performance) protected by a lock, which is acquired while using the queue. Now the `sys_send` function takes up the message and stores it in the queue assigned to the process with `rec_pid` process id. When, `sys_recv` is called by the process, I dequeued the message from the queue corresponding to the current process and copied back to the address given as a parameter to the system call. If the receive is called first by the process and there is no message in the queue the process is put to sleep and is later woken up when a message arrives.

3.2 Multicast Model

To implement the multicast model for IPC, I made the following new system call,
`int sys_signal(sighandler_t handler)`

where `sighandler_t` corresponds to `typedef void (*sighandler_t)(void)` What this system call does is that, it takes a signal handler defined in the user program and registers it, to be used later (explained below). I also made some modification to the process structure defined in the file `proc.h`, where I added two fields `int signal_pending` which is set to 1, when a coordinator process sends a signal to a receiver process. Also, I kept a field for `sighandler_t signal_handler`.

Now, to implement the given system call, `int sys_send_multi(int sender_pid, int rec_pids[], void *msg, int length)`, what I did was for every receiver pid, i made a call to a helper function `void send_multicast(int send_pid, int recv_pid, char* msg)` in `proc.c`. What `send_multicast` does is that it acquires the lock corresponding to the receiver queue id, and puts the message in the receiver's queue and sends a signal by setting it's receiver process' pending signal to 1, after which it releases the lock.

For completion, I modified the `allocproc` in `proc.c` and initialised `signal_pending` to 0 for every process to indicate that a signal has not been sent. Now, the only part that is left to call the signal handler when there is a pending signal for a particular receiver process. Now to take care of that, I modified the code in the `void scheduler(void)` in `proc.c` where it checks for all processes,if there has been a pending signal of a particular process (call it p), where I used `char* uva2ka(pde_t *pgdir, char *uva)` from `vm.c` to map user virtual address to kernel address and then opened up a new frame and set the eip of the trapframe for the process p to the signal handler, registered for the process. By this, I acheive the purpose of calling the signal handler for a receiver, which then fetches it's message from it's queue and after processing sends back the message to the sender using the unicast model of IPC.