COL331: Operating Systems Assignment 1 Report

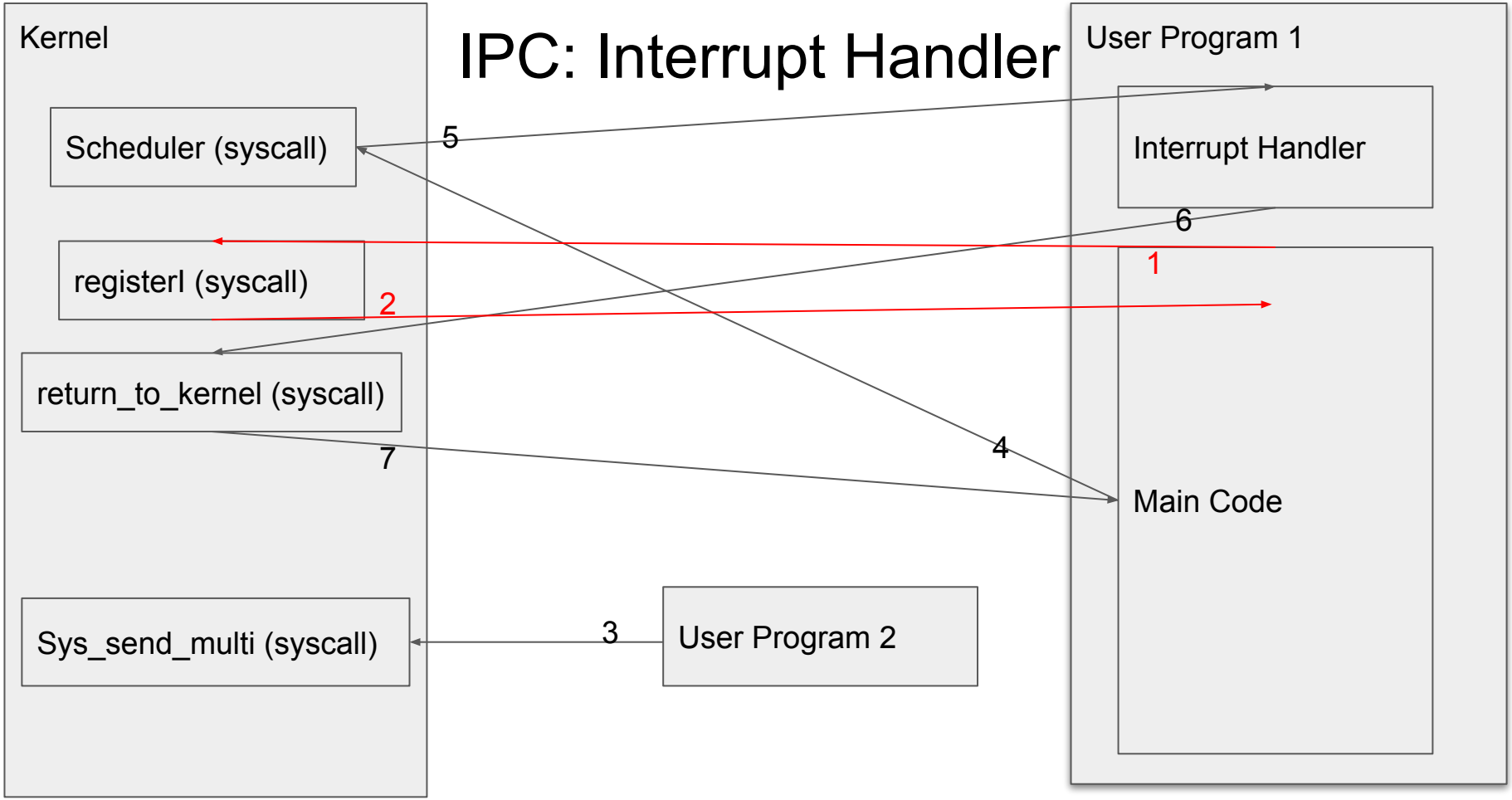
By Shashwat Shivam

Distributed Algorithm (Phases & Steps)

1. Initialization (Phase 1) :-
   1.1. The parent process initializes children and gives everyone a tid (starting from 0 for parent process) and also stores id of parent (to communicate back messages).
   1.2. The pid of each child thread created is store by the parent.
   1.3. Each child thread registers it's interrupt handler in the kernel using the registerI system call.
   1.4. Using the value of their tid, all the threads (including the parent process) calculate the number(range) of items of the array to process by each thread using the formulas :-
      - num_of_items_to_process = ceil(size/num_threads)
      - Start = tid * num_of_items_to_process
      - End = min((tid+1)*num_of_items_to_process , size)
2. Local Sum Phase with communicating result (Phase 2) :-
   2.1. Each thread sums it's range of array into a local sum.
   2.2. Each thread (except parent) communicates it's local sum value to parent thread. (exit here if type = 0)
   2.3. Parent thread retrieves values of local sums using a recv system call equal to the number of children and adds the received sum into it's own sum.

Distributed Algorithm (Continued)

3. Calculate Mean and message mean (Phase 3) :-
    3.1. The parent calculates the mean using the sum of the array.
    3.2. Using the sys_send_multi system call, it messages the mean to every child process.
    3.3. The child processes which were waiting for mean to get updated receive this message using the signal handler which updates the value of the mean in a global variable.
4. Calculate local squared sum and message to parent (Phase 4) :-
    4.1. All the threads calculate local squared distance sum from the mean.
    4.2. The child threads message their local sums to the parent using unicast method.
    4.3. The parent receives the local sums and adds them to it’s own local sum and finally calculates mean by dividing by size.


IPC: Interrupt Handler
Kernel
Scheduler (syscall)
registerI (syscall)
return_to_kernel (syscall)
Sys_send_multi (syscall)
User Program 1
Interrupt Handler
Main Code
User Program 2
1
2
3
4
5
6
7

IPC: Interrupt Handler (Continued)

The following points to be noted before reading the interrupt handler implementation :-

1. A registerI system call is done by the user program which takes as argument a pointer to a function with a parameter (void*msg).
2. The system call stores the address of the function pointer in the kernel indexed by the process id.
3. The interrupt handler defined in the user process takes the passed message and stores it into the global variable (with proper typecasting) to be used by the program.

IPC: Interrupt Handler (Continued)

Steps to send message using sys_send_multi and interrupt handling :-

1. Any process sends a msg using sys_send_multi with the valid parameters.
2. In the kernel, the sys_send_multi stores a boolean true for the processes in the array (sendInterruptSignal) at the index to whom the message has to be sent to. The message for each process is also copied to kernel space.
3. Now when the scheduler picks up a process to run, if there is a true in the array sendInterruptSignal at index of the process, we send the message to the process using the following steps.
4. We backup the current trapFrame to restore it later.
5. Now we modify the eip of the trapframe so that the instruction pointer points to the registered interrupt handler.

IPC: Interrupt Handler (Continued)

6. We copy the message to the top of the stack space (MSGSIZE) using the esp (stack pointer) of the trapframe of the process.
7. We add another 4 bytes to store an integer with the value equal to address of starting of next 8 bytes, so that this can be interpreted as a pointer to the value which needs to be read. (This value will be the parameter passed to the interrupt handler which will act as void*).
8. After the interrupt handler has done the operations using the passed value, it calls the return_to_kernel syscall which goes to kernel space.
9. The return_to_kernel syscall restores the trapframe to the backed up trapframe and the process can now continue from the point it paused execution.