Shivanshu Gupta
Akash Tanwar
11 May 2016

Running JOS on ARM

Operating Systems Final Project Report

Task 1 -> Getting the already existing code to run on Raspberry Pi board

- **Creating the bootable SD Card:** An 8GB SD Card is made bootable with the Raspbian OS. Then the kernel.img file of the Raspbian OS is replaced with the bin file of JOS kernel image which is cross compiled by running the command *make arch=raspberrypi***.** Then the raspberry pi's bootloader, loads the JOS kernel image during booting.
- **Booting Process:**
  1. The 14th and 15th General Purpose Input Output pins were connected with jumper cables to a USB Serial Adaptor.
  2. The USB cable was then connected to a laptop with ubuntu system was installed.
  3. Minicom is then run on the linux terminal with the following command : *sudo minicom -b 115200 -D /dev/ttyUSB0* where ttyUSB0 is the identifier of the port connected to the serial adaptor and 115200 is the bit rate for the communication.
  4. The raspberry pi is then attached to the power supply and booted up with the SD card.
  5. The output of the pi is communicated over the serial port and is visible in the minicom terminal.
  6. Minicom terminal accepts input too from the user and communicates respectively with the kernel.

Task 2 -> Extending support for exo-kernel features

To support exo-kernel features we needed some basic system calls which we implemented.

sys_page_alloc(int pid, uint32_t va, int perm): This allocates a page of memory and maps it at 'va' with permission 'perm' in the address space of 'pid' and the page's contents are set to 0. If a page is already mapped at 'va' then that page is unmapped.

sys_page_map(int srcpid, uint32_t srcva, int destpid, uint32_t destva, int perm): Maps the page of memory at 'srcva' in scrpid's address space at 'dstva' in dstpid's address space with permission 'perm'.

Perm has the same restrictions as in sys_page_alloc, except that it also must not grant write access to a read only page.

syscall_page_unmap(int pid, uint32_t va): Unmaps the page of memory at 'va' in the address space of 'pid'. If no page is mapped, the function silently suceeds.

syscall_exofork(): Allocates a new environment.

To add a system call in arunOS we need to create the respective wrappings and make appropriate entires in sys call tables at a few locations. Then we can create the right files to implement it.

Inter Process communication:

*syscall_ipc_try_send(int pid, uint32_t *data):* Tries to send the 'value' to the target 'pid'. If srcva < USER_TOP, then also send page currently mapped at 'srcva' so that receiver gets duplicate mapping of the same page. If the sender wants to send a page but the receiver isn't asking for one then no page mapping is transferred, but no error occurs.

syscall_ipc_recv(uint32_t dstva): Block until a value is ready. Record that we want to receive using the proc_ipc_recving and proc_ipc_dstva fields of struct Env, marking ourself READY and then yielding.

Note that in arunOS the proc structure doesn't have the required fields available for these two system calls. Hence we added the following fields to the proc structure :

```
bool proc_ipc_recving;              // proc is blocked receiving
uint32_t proc_ipc_dstva;            // VA at which to map received page
uint32_t proc_ipc_value;            // Data value sent to us
int proc_ipc_from;          // procid of the sender
int proc_ipc_perm;          // Perm of page mapping received
```

Also note that in arunOS code there is no 'environment' or 'process' mapping available in the user space so to retrieve the values in sys_ipc_recv we need to make another system call which is:

*syscall_ipc_data(int *srcpid, uint32_t *data_stores[]):* This call takes in 3 pointers which are srcpid, value_store and perm_store which are in data_stores[0] and data_stores[1] respectively. These pointers are filled by the kernel with the respective values of the process that called it.

After the sys calls are implemented we made a library routine available to the user for proper ipc send and receive of page and values.

Attempt at User level page fault handling:

After a lot of research and trials we figured our how an interrupt table and exception tables are formed on ARM and how to register a handler in them. Then we learnt about page faults in ARM which are called data aborts. Then we created a stub for data aborts as well but it turned out that handling them appropriately is not easy. There are different types and cases of data aborts which have to be handled and since we were not so aware of ARM's conventions and with no good enough source s to understand it, at hand, we decided not to implement the data aborts.

Specific Yield option to user:

We realised that yielding to a specific process is not safe as a processes may keep forking to each other in a cycle thus take hold of the processor only for themselves, that is, no other process except the ones in a cycle may get a chance. There are many such sensitive safety issues with such a system call and thus we chose not to provide it to the user.

Task 3 -> Application to improve a parameter of interest

We made the application that increments a counter using two processes in the manner ping pong is played. This application is a clear demonstration of the immense possibilities of pc_send and ipc_recv. For example sorting using multiple processes. One such algorithm is bionic sort in which every process has number and finally after numerous compare and exchange all the numbers get sorted such that they have numbers in increasing order of there process id's. The process with higher pid has a higher number than the process with a lower pid and vice-versa.