Design Document

Prepared for: OS Project

Prepared by: Akash Tanwar, Shivanshu Gupta

26 April 2016

Topic: JOS on ARM

JOS ON ARM

Objectives

Running JOS on ARM using the code on https://github.com/pykello/arunos

Goals

a) Getting the kernel running over Raspberry Pi board.
b) Extending it to support exo-kernel features. Currently it doesn't support so.
c) Demonstrating an application to improve a particular parameter of interest (e.g. page faults, execution time, etc.)

Solutions

1. Booting the Kernel on the Pi -

1. **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. Then the pi bootloader, loads the JOS kernel image during booting.
2. **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.

2. Exo-kernel features -

1. User Level Page Fault Handling:

The ArunOS kernel code does not contain any user level page fault handling. So we will follow the steps given in our lab 4 to do this. Also the given code has no trap handling in it and thus we will have to implement traps from scratch.

Once traps and page fault handling are set up, we will implement the system call *sys_env_set_pgfault_upcall()* so that the user can register his own page fault handler in the kernel.

We will also need to implement *set_pgfault_handler()* function.

2. Copy On Write Optimization in fork:

The present JOS kernel only has *dumbfork()* like *fork()* implementation in it. On forking a process the kernel copies the entire address space of the parent into the child using *memcpy()*.

Since we will have implemented the support for a user level page fault handler now, we can implement "*exofork*" which just copies the registers of the parent process to the child and also sets all the pages of the parent to *PTE_COW* to save time.
For this we will implement *sys_exofork()* which creates a new process using *proc_create() and* sets the process status to SLEEPING and then copies the registers.

Whenever any of the processes tries to write, the user level page fault handler will kick in and copy the particular page to a new memory location for a process private copy of it.

For doing this we will need the following supporting system calls:-

- *sys_page_alloc(envid_t envid, void *va, int perm)*: creates a page at address va with permissions perm. It will also use additional support like map_page(), unmap_page(), etc
- *sys_page_unmap(envid_t envid, void *va)*: Remove the page mapped at virtual address va in the process envid.
- *sys_page_map(envid_t srcenvid, void *srcva ,envid_t dstenvid, void *dstva, int perm)*: It maps the page of one process (parent) to the virtual address dstva of child.

3. Bringing a specific yield option To the User

We will implement a function *user_yield(int pid)* which takes as argument a process id corresponding to the process to which the current process wants to yield. If the process id is invalid then, it yields to the first "runnable" process. We will also need to ensure that the process who was yielded to doe not yield back to the same process because otherwise 2 user processes can starve out all the other processes.

4. Inter process communication

We will implement the system calls *sys_ipc_send()* and *sys_ipc_recv()*. To receive a message, a process calls *sys_ipc_recv*. This system call de-schedules the current process and does not run it again until a message has been received.To send a value, a process calls *sys_ipc_try_send* with both the receiver's process id and the value to be sent. If the named process is actually receiving (it has called *sys_ipc_recv* and not gotten a value yet), then the send delivers the message and returns 0. Otherwise the send returns an error value to indicate that the target process is not currently expecting to receive a value. For this purpose we need to have the following supporting function:-

- *page_lookup(pde_t *pgdir, void *va)* - This function returns the page mapped at virtual address va in the page directory *pgdir*