Design Documentation

Daniel Svensson (z5086298) Brent O'Neil (z3375531)

March 5, 2017

1 Introduction

This design document roughly describes the milestones we completed in this project, note that not all milestones were finished, below is a list of the milestones completed.

The following milestones were completed:

- M0: Familiarisation
- M1: Timer driver
- M2: Memory Manager
- M3: Pager
- M4: System Call Interface
- M5: File-system

2 Timer Driver

The timer driver is initialized with a specific badge(Which number?) on the same interrupt endpoint used by the network. The timer badge is then used to determine which hardware interrupt has occurred in the `syscall_loop`. This endpoint is connected the interrupt sources `EPIT1` and `EPIT2`. Finally the `phy_addr` of the registers is mapped via `map_device()`. For handling registered callbacks a linked list is implemented where the head always points to the callback with the lowest timeout value if one exists.

2.1 Registers

The registers in use are `EPIT1` and `EPIT2`. Where `EPIT2` is to maintain the system time (Time since boot). `EPIT1` is used a öne shotẗimer where it is always counting towards the time of the callback nearest in the future.

When a new callback is registered it is inserted in the linked list at the appropriate location. This insertion maintains a sorted list of earliest callback first. This implementation is linear at insertion time as a function of registered callbacks. However it allows the interrupt vector to be linear as a function of the callbacks that are due, rather then registered.

2.2 Timer callbacks

Whenever `EPIT1` triggers an interrupt the head of the callback list is checked. If the system time is greater then the target time, the callback is fired. Otherwise the timer is simply reset, this can will occur when the first callback has a delay that is greater than the maximum possible timeout. If a callback is fired the list of callbacks is iterated until all callbacks that are due are issued. As this is a sorted list as soon as a callback is found that is scheduled in the future it is guaranteed every item in the list after it will be in the future.

`EPIT2` is used to maintain the current time-stamp by decreasing the `REG_Counter` until a interrupt occurs in which the number of interrupts is increased and the timer is restarted. In order to get the current time-stamp the current number of interrupts is multiplied by the number of milliseconds corresponding to a overflow defined as `EPIT_CLOCK_OVERFLOW`.
The time-stamp is then calculated in `epit_getCurrentTimestamp()`, where a race condition is identified and dealt with where it may be the case that a interrupt is triggered after reading the status registered but before calculating the time based on number of overflows and the current value of the count register. Any other race conditions with the hardware was not found.

3 Memory Management

3.1 Frametable

The Frametable is used to represent pages currently mapped into physical memory. It is located at virtual address `VMEM_START`. The table holds seL4 capability data about allocations done by `ut_alloc()`. To allocate/deallocate frames a linked list of free frames is used which only contains the frames above the frame table itself, thus the frames holding the frametable are never allocatable. When a new frame is allocated it is mapped to the virtual address in SOS corresponding to `VMEM_START + (index > seL4_PageSize)` with the physical memory given by `ut_alloc()`.

When a frame is deallocated it is unmapped and its capability is deleted, the memory is returned via `ut_free()`. The frame object contains a `pageTableEntry` which could be extended to include referenced and modified bits.

3.2 Pagetable

The page table is implemented as a two level Page table and is thus constant time look-up using the virtual address. The structure is given in the `pageDirectory` struct which keep track of the capability pointers to the `seL4_ARM_PageTable` page tables(2^{12} of them) each page table containing 2^8 `pageTableEntries`. Theese page tables are lazily allocated in `sos_map_page()`.
Memory regions are stored in `regions` which is a link list holding the base, size, permissions and if the region is treated as a stack. These permissions are checked in `vm_fault()`. A heap and IPC-buffer region has been defined is defined at process start up and the `sos_brk` syscall will dynamically increase the size of the heap region.

4 System call Interface

The system call (syscall) interface uses a single seL4 endpoint. Each process is given a badge at start time and this is used to determine which process is calling into SOS. Data about the system call,

such as numerical arguments and pointers, is passed through the IPC buffer. Data buffers, such as strings, are translated from user to SOS addresses using the function `get_shared_region()` creating what is known as a `shared_region`. This is simply a list of translations between `user_addr` and `sos_vaddr` for each page that a buffer resides on. All relevant permission and residency checking is done during this translation. Any permission error will result in a failure, whereas valid but unmapped memory will be mapped before proceeding. As described in 3.1 all process memory is also mapped as a consecutive region in SOS virtual memory starting at `VMEM_START`. This structure allows bulk data transfer, such as copying buffers, directly to and from user memory. In addition this allows us to avoid double buffering when passing data between two clients (in most cases between the user and nfs-library).

5 IO subsystem

The process have the notion of a file-description table, pointing to `fdnode` containing information about the resource. Further the `fdnode` indicates if the resource is a device or a file. Devices is a structure describing a device, it enforces limitations in the device (number of readers, writers) and also contains read and write functions. If the resource it a file it currently contains information specific to the `nfs-library`.

5.1 nfs-support

The nfs support is implemented using the given nfs-library. The nfs-callback functions refer to `fs_request` entries which contains specific information for a session to continue. This make it possible to use multiple read writes, the last reader or writer in a session will perform the reply back to the user. The `shared_region` explained in 4 removes the need for mapping in the system call path and also avoid double buffering since the nfs-callbacks can read/write directly to the `user_buffer`.

6 Execution Model

The execution model is simplified for milestone 5, much work was put into getting it working with milestone 6 to no avail. The execution model is currently assuming no more then one user process is running. It will handle a syscall in `syscall_loop`, if this syscall is blocking it will go back to `syscall_loop` and handle other events with the assumption to no other syscall could be happening. When the first syscall is completed it will reply to the user and is then again ready to handle another syscall.