G1 rapport

Sebastian Ostenfeldt Jensen,
Thomas Wolff Rosenqvist and
Nikolaj Østergaard

17. February 2014

Task 1

Task 2

syscall write

The way we implemented the `syscall_write`, was by adding a function with the name `sycall_write` inside `proc/syscall.c`. And in the same file we added a case more in the function `syscall_handle`, that called our `syscall_write` with register A1, A2 and A3, and saved the result in V0. Below the code for `syscal_write` is shown in chunks

```
1  device_t *dev;
2  gcd_t *gcd;
3  int len = 0;
4  /*
5  * we always write to terminal, and we don't need fhandle
6  * assign it to itself to avoid compiler warnings ie errors
7  */
8  fhandle = fhandle;
```

First of we `device_t dev` to hold the console, the we create a `gcd_t gcd` to do the writing at last we set fhandle to it self, because we don't need it, and if we didn't we would get unused variable error.

```
1  /*Find the system console (first tty)) */
2  /* Should be FILEHANDLE_STDOUT, which is 1, but when
3  * using 1 and not 0 i get kernel assert failed on dev
4  */
5  dev = device_get(YAMS_TYPECODE_TTY,0);
6  if(dev == NULL){
7    return -1;
8  }
9  /* Set generic char device*/
10 gcd = ( gcd_t *) dev->generic_device;
11 if(gcd == NULL){
12   return -1;
13 }
```

In the code above on line 5 we set the device to stdin, as stated in the code comment, when setting it to stdout i.e. 1, the code would return -1, when setting to 0, it would write to console.

```
1  len = gcd->write(gcd, buffer, length);
2
3  return len;
```

When the gcd has been set, we use it to write to the screen

syscall read

The read syscall is implemented in the same way as write, the only difference is that instead of calling `syscall_write` we call `syscall_read`, the arguments are the same and we also save the return value in register V0. The code is shown below. The first 20 lines are the same as write, where we initialize gcd and makes sure it is not NULL, after that we reach the code below.

```
1  while(len <= length && !( *(char*) (buffer+len-1)== 13)){
2      /* we read one byte at a type, so we increment len, and
3       * store the next byte on the offset of len
4       */
5      len += gcd->read(gcd, buffer + len, length);
6      /* If we don't hit enter, write to
7       * the screen what the user typed in
8       */
9      if(!( *(char*) (buffer+len-1)== 13)){
10       // should only write one char
11       syscall_write(0,buffer+len-1,1);
12     }
13   }
```

We decided to implement read, so that it could read more the one byte per syscall, the while loop runs until we reach the maximum length we are allowed to read, or when the user presses enter. In ascii 13 is carriage return, which is enter. This implementation can easily be change, so that it also stops on EOF, newline or whatever is preffered. The reason we choose enter for now, is because we only read from console, where it is normal to pres enter when done typing. One line 11 we echo the byte the user just entered. We only want to echo if the user didn't press enter. After exiting the loop we just return len, code not shown.

readwrite test

Here we explain the readwrite program used to test the two syscalls.