Porting OSE Systems to Linux

June 10, 2010

Fredrik Eriksson
fen05001@student.mdh.se
Abstract

OSE is a real-time operating system from ENEA. One of the key features in OSE is the signaling interprocess communication (IPC) system. ENEA has also released a similar IPC system called LINX, that is platform independent.

For the purpose of running OSE programs under Linux, I have evaluated two software libraries that are designed to simulate the OSE programming environment. The libraries in question are called SoftApi and OSEPL, where SoftApi was developed by RealTime Logic and modified by me to utilize LINX for interprocess communication, and OSEPL was designed by me to overcome some of the limitations in SoftApi. Neither of those libraries proved to be suited for production systems, but it is deemed possible to implement a similar library that is.
Contents

1 Introduction ........................................ 1
  1.1 Overview ........................................... 1
  1.2 Emulation .......................................... 1
  1.3 Embedded Systems ................................. 2
  1.4 Real-Time Systems ................................. 2
  1.5 Operating Systems ................................. 3
  1.6 Purpose ............................................ 3

2 Related Work .......................................... 4
  2.1 Linux in Real-Time Systems ....................... 4
  2.2 OSE and Linux ...................................... 4
  2.3 Performance Measurement ......................... 5
  2.4 Software Projects .................................. 5

3 Problem Definition and Analysis ...................... 6
  3.1 Limitation and Future Work ....................... 6

4 The Operating Systems ................................ 7
  4.1 Operating System Embedded (OSE) .................. 7
    4.1.1 The Signaling API ............................. 7
  4.2 Linux .................................................. 8

5 APIs and Porting Layers ............................... 9
  5.1 LINX .................................................. 9
  5.2 Special Considerations ............................ 10
    5.2.1 hunt() .......................................... 10
    5.2.2 Internode communication ...................... 11
    5.2.3 Reserved signal numbers ...................... 11
    5.2.4 Redirection .................................... 11
    5.2.5 Context ......................................... 11
  5.3 SoftApi .............................................. 12
  5.4 Combining SoftApi and LINX ....................... 12
  5.5 OSEPL ............................................... 13
    5.5.1 Internal communication ....................... 13
    5.5.2 Errors .......................................... 14
    5.5.3 Process Types ................................ 15
    5.5.4 Other Limitations .............................. 15

6 Installing Linux ...................................... 16

7 Performance .......................................... 17
  7.1 What is performance? .............................. 17
  7.2 Performance Metrics ............................... 17
  7.3 Test Platform ...................................... 18
  7.4 Timing Accuracy ................................... 19
  7.5 Tests ................................................. 19
    7.5.1 Tests for Standard System Calls ............. 19
    7.5.2 Tests for OSE System Calls .................. 20
    7.5.3 Tests for OSE Signaling calls ................ 20

Fredrik Eriksson
E-Mail: fen05001@student.mdh.se
Phone: 0733-729 828
8 Results

8.1 SoftApi vs. OSEPL ........................................ 22
  8.1.1 OSE System Calls ..................................... 22
  8.1.2 OSE Signaling Calls .................................. 23
8.2 Conclusions ............................................... 23
8.3 Future Work ............................................... 24

Appendices

A Tests

A.1 tests.h ...................................................... I
A.2 timing.c .................................................... II
A.3 tests.c ...................................................... III

B Test Values ................................................ IX
1 Introduction

This section describe a few basic concepts and the purpose of my work.

1.1 Overview

Sometimes when using a computer, you find that there are better ways to solve something, and better tools to use. The many programs and usages of computers is one of the things that makes some of us find them so interesting. Ironically it is also one of the things that makes them confusing for others. For desktop computers there are in fact three different major software platforms. There is the widely used Windows-platform; there is the growing competitor Mac-platform, and there is the free alternative Linux.

Usually people just use the operating system that comes with the computer. Some people know that they prefer another operating system and installs that the first thing they do on the new computer. It happens, however, that after using one operating system for a while you want to change to another for some reason; for example if there is a program you need or want that can only be run on a specific platform.

Another reason for changing operating system is cost. Both Windows and Mac OS costs money to upgrade to a new major system version; Linux however is free of cost. Therefore, some people may choose to use Linux instead of upgrading their proprietary operating system.

The problem with changing operating system is that, just like one of the reasons for changing can be to get a specific program running, other programs that you frequently use may not be compatible with the new platform. Sometimes you can find an equivalent program to replace it, and sometimes you might realize that that program is not necessary after all. Sometimes there is an actual port of the program that can run on the new platform, and if the particular platform switch is from Windows to a Unix-platform, the simulation layer Wine may be an alternative.

1.2 Emulation

The terms simulation, emulation and virtualization are commonly used as synonyms, and although they are all quite similar one should still be aware of the differences. In terms of software platforms, simulation can be considered to be the most general term and virtualization the most specific. Simulation is when a system mimics the abilities of another system, if it is done on a very low level (for example interpreting and translating assembly code from one architecture to another) then it can be considered emulation. Virtualization is advanced emulation where not just processor instructions are handled, but the entire hardware environment with processor, IO-devices, peripherals etc. are emulated.

An example of a simulation system that is not emulation is Wine. Wine is a project which aims to let you run Windows applications in Unix-based systems. The name is commonly mistaken for “Windows Emulator”, but is in fact a recursive acronym: “Wine Is Not an Emulator” [23]. Wine is a free implementation of Win32 API, and although
not the entire API has been implemented, it has been used for years to run Windows programs under Linux and BSD systems.

Just like the name implies Wine does not emulate a system. When executing a Windows-program in Wine the machine code is interpreted by the hardware, just like it is when the program runs in Windows, without any software layer that translates it. It does however load the Wine libraries instead of the ones that exist in Windows.

Wine is commonly used in Unix-like desktop environments, as many commercial games and office programs are written specifically for Windows. More specifically these programs are written for Windows running on a x86-architecture (or perhaps the 64-bit implementation x86_64). As Wine is not an emulator, it can only run those programs on the same architecture. In the desktop world this is not much of a problem since almost all desktop systems use processors from the x86-family.

Desktop systems aside, there are also many other kinds of computer systems.

### 1.3 Embedded Systems

Michael Barr and Anthony Massa defines embedded systems as “a combination of computer hardware and software—and perhaps additional parts, either mechanical or electronic—designed to perform a dedicated function” in their book *Programming Embedded Systems: with C and GNU Development Tools* [3].

Most of the electrical devices people use daily is in fact embedded systems. A washing-machine, microwave and multimedia equipment does in fact contain one or more such systems. Sometimes it can be hard to differentiate between an embedded system and a general purpose computer system. Just five years ago mobile phones were considered to be embedded devices, but with the growth of the smartphone market the phones sold today have almost as much functionality as any general purpose personal computer.

### 1.4 Real-Time Systems

Sometimes, especially with embedded systems, it is not enough that the system does the right things; it must also do the right thing at the right time.

A commonly used example of a real-time system is the control system for an airbag. The operation of an airbag is simple. If the car collides the airbag is inflated and will soften the impact for the driver. However; if the airbag is inflated too late the driver will already have hit his head in the instrument panel.

Since the airbag begins to deflate shortly after deployment it is equally important that the airbag is not deployed too early, as it quickly becomes ineffective.

Most real-time systems are not safety-critical like the airbag, anti-lock braking systems or emergency shut down systems of a nuclear power plant [19], but nevertheless it is still important that, for example, the sound is in sync with the video on the TV, or that your phone starts ringing the moment someone calls and not ten minutes later. Generally what is important for real-time devices is that they must have (more or less, depending on the system) predictable response times.

Fredrik Eriksson  
E-Mail: fen05001@student.mdh.se  
Phone: 0733-729 828
1.5 Operating Systems

Simple embedded systems can be implemented without operating systems. In systems that require multitasking, networking or have similar relatively advanced features, it is more often than not easier to use an operating system. As can be seen on the Wikipedia page of operating systems; there are quite a few such systems made for embedded devices and many of those are made for usage in special purpose devices like mobile phones, network routers or similar [22].

Generally an operating system consist of a kernel, which is the core of the system, as well as userspace libraries and tools [20]. As mentioned earlier some operating systems are developed for usage in systems with specific purposes, and so they also have different functionality. An operating system developed for usage in network routers must for example have good network capabilities, while a system for MP3-players may not require networking at all.

The operating systems that are relevant to the work in this paper are Operating System Embedded (which I will refer to as OSE) and Linux. I will describe these operating systems closer in section 4.1 and 4.2 respectively. The main difference between them is that while OSE is designed and developed to be a reliable and predictable soft real-time operating system, Linux has gradually evolved real-time support over the years [5]. The Linux kernel can still be improved for even more predictable response times, but it has reached a state where it is useful for real-time systems [2].

1.6 Purpose

Ericsson is one of the leading developer and provider of telecom equipment. Today much software is built upon the OSE operating system from ENEA. While OSE is a very capable operating system, Linux has a few advantages over it.

Due to the open nature of Linux it is a very popular platform for software development and customisations, which has given birth to countless of tools and utilities to ease both development and management. Just like the Linux kernel itself most of the tools are available free of charge. The large user base also means that there are many developers on the employment market who knows the system.

When porting a large system from OSE to Linux you do not want to rewrite all the programs that you have already developed for the platform; therefore it is desirable that the existing programs can be run natively directly under Linux without any manual modification of the source code.

The primary goal of my work has been to implement an API-layer so that programs written for OSE can run under Linux. This layer will work similarly to Wine described in section 1.2 in that it should allow the execution of programs written for one platform (here OSE) to be done on another platform (primary Linux).

My secondary goal has been to evaluate the performance of OSE programs that run under Linux. Getting the programs to work is of course necessary for migrating a system developed for OSE to Linux, but if the performance penalty is too high it is not practically useful. Therefore I have compared both common system calls (such as malloc() and free()) and system calls that is OSE-specific (such as send() and receive()).
2 Related Work

Since Linux in real time systems and performance measurement in software are quite old phenomena there are tons of papers that in one way or another can be seen as related to my work. Here I have collected only the few that are most relevant to this paper.

2.1 Linux in Real-Time Systems

How to take advantage of Linux in a Real-Time environment has been researched since at least 1996 when Barbanov and Yodaiken wrote a paper about their work in creating a Linux for real time systems [1]. Their work eventually resulted in RTLinux, which was the first implementation of a Linux for hard real time systems [25]. Today there are at least two more open source projects for this purpose: Xenomai and RTAI [10, 14].

These projects are also used and maintained by various corporations that use and sell Linux for real time systems. MontaVista is one such company and their guide for how to port legacy real time applications served as a good starting point for the work in this paper (see section 3) [21].

It should be noted though that none of those projects are “true” Linux in the sense that it only runs a Linux kernel. Instead they use a real-time core that cooperate with the Linux kernel to achieve real-time characteristics. Real-time support for the standard Linux kernel became better with the release of version 2.6, and the ability for user-space programs to preempt non-critical sections of the kernel [5].

2.2 OSE and Linux

Viveka Sjöblom has compared various performance aspects of Linux and OSE [18]. Sjöblom reaches the conclusion that comparisons between operating systems (both in her and my case these operating systems are Linux and OSE) are possible, and best comparison is done by comparing execution time of system calls. System calls are good since they usually have similar capabilities in different systems, and any program that runs on the system will use the system calls extensively.

There are a few differences between the work in this thesis and what Sjöblom did. First off, Sjöblom used a Linux distribution provided by the hardware manufacturer which probably is optimized and well tested for that particular hardware. In this thesis a generic distribution that can be installed and used on many different hardwares is used.

The main difference between mine and Sjöblom’s work is that I have not just tested an arbitrary Linux system, but Linux with a porting layer to mimic OSE functionality. The tests of system calls that are natively available on both platforms should not differ from Sjöblom’s work.

Boman and Rutgersson also ported Linux and a bootloader to the MPC8360 PowerPC platform [17]. Their goal was to test if Linux could replace OSE in soft real-time systems. The report contains an in depth comparison in various aspects of OSE and Linux. They reached the conclusion that although there are some trade-offs, Linux is a viable alternative to OSE when it comes to performance and stability.
2.3 Performance Measurement

Generally papers on software performance measurement deals with more or less complex systems [4, 11, 13]. While my work is focused on more modest performance measurements, such papers has still served as inspiration and introduction to the subject. Most apparent inspiration comes from Collberg and Hugne, who used hardware performance counters to measure performance in real time systems [4]. In my work I have used the hardware clock counter to measure execution time to make the procedure as non-intrusive as possible.

2.4 Software Projects

There are two software projects that is of great importance to my work. The first is SoftApi developed by RealTime Logic. SoftApi is an implementation of the OSE API. It was originally designed to allow OSE programs to run in a Windows NT environment, but was later ported to run under Linux as well. The second software project is LINX, which is an open source project from ENEA which is very similar to the signaling system used for interprocess communication in OSE. These two pieces of software are discussed in more detail in section 5.3 and 5.1 respectively.

With these projects as background my hypothesis is that OSE-programs running under Linux should be both possible and feasible.
3 Problem Definition and Analysis

As described earlier there are various reasons why you would like to port an OSE system to Linux. The problem is to find a way to do this both for small systems with just a few custom programs, and large systems with hundreds of such programs. Of course the system should still be useful afterwards, so there must not be any major performance loss. It is not unusual to migrate embedded real-time systems to Linux. William Weinberg from MontaVista Software has written a white paper about the general process of such migrations [21].

Weinberg mentions three ways for porting real-time systems to Linux:

1. Full native Linux application port
2. Run-time partitioning with virtualization
3. RTOS API emulation over Linux

A full native Linux port might very well be considered for systems that consist of only a few small programs that mainly use standard POSIX system calls. This does not describe Ericsson’s system as it consists of hundreds of different programs and extensively uses OSE specific system calls.

The second option basically describes that the real-time operating system can run in parallel to a native Linux environment. Since one of the goals from Ericsson’s point of view is to save cost by not using a proprietary operating system, this is not an option either.

What Weinberg describes as “RTOS API emulation” is what was described in the introduction as a Wine-like layer. A simpler term, and the term that will be used in the rest of the paper, is “porting layer” as described by Raghavan et.al[16]. This seemed to be the best option to use for this particular system. By creating a API-layer for OSE system calls all the hundreds of programs already developed for the platform should be able to run without any modification.

3.1 Limitation and Future Work

Focus in this paper is functionality and performance of a single node. Complete analysis of a complete multi-node system is left out for future research. This also means that compatibility of OSE and Linux nodes is not tested. Furthermore the performance analysis in this paper is focused on execution time (or response time) of system calls (see section 7 for more information about different performance aspects).
4 The Operating Systems

This section describes relevant functionality and differences between OSE and Linux.

4.1 Operating System Embedded (OSE)

OSE is a real-time operating system developed by ENEA. OSE is flexible and used in a wide variety of systems, ranging from mobile phones to industrial control systems [8]. The operating system itself is quite small and only provides basic functionality. One of the more central functions of OSE is the advanced signaling API.

4.1.1 The Signaling API

The main method for interprocess communication in (and between) OSE systems is by OSE signals. The OSE signaling system is very simple to use, but is nevertheless quite powerful. The signaling API consists of only a few system calls where some of the most important ones are:

- `hunt()`
- `alloc()`
- `free_buf()`
- `send()`
- `receive()`

When a process is started in OSE it has to have a name assigned to it. This name is a simple (human readable) text string that can then be used by other processes that wants to communicate with the process. The name is sent to the `hunt()` call which is used to obtain the ID of the process. The ID can then be used as an address to send signals to.

Signals can vary in size and are allocated dynamically with the `alloc()` system call. An allocated signal can be sent to other processes with one of the `send()` system calls or be freed with the `free_buf()` call. When a signal has been sent the recipient can retrieve the signal with `receive()` or one of the similar system calls.

Besides being used for communication between processes within a single system, signals can also be used for communication between different systems. For this purpose OSE uses a link-handler that keeps track on where to find the neighbouring systems. The process of communicating between systems is just the same as communicating within the same system, with the exception that the process name has to be preceded by a path which describes which node the process is running on.

Below is an example of a function that takes a process name as input argument, hunts for the process and sends a signal to it.
void send_sig(char *name) {
    PROCESS serv_proc;
    SIGSELECT sigsel_hunt[] = { 1, OSE_HUNT };
    /* using alloc() to allocate memory for a hunt signal */
    union SIGNAL *sig = alloc(sizeof(OSE_HUNT), OSE_HUNT);
    /* using hunt() to find the process. */
    hunt(name, 0, (PROCESS *) NULL, sig);
    /* hunt() consumed the signal, so we can freely use that 
     * variable for a new signal. sigsel_hunt is specified so 
     * that we will only accept OSE_HUNT signals. */
    sig = receive(sigsel_hunt);
    /* The process can be found using sender() */
    serv_proc = sender(sig);
    /* we don’t need the signal anymore */
    free(buf(&sig));
    /* and now we can send any signal to the server process. 
     * I will use PING_SIG in this example, although it’s not 
     * an OSE standard signal */
    sig = alloc(sizeof(struct ping_sig), PING_SIG);
    send(&sig, serv_proc);
    /* just like hunt(), send() consumes the signal, so no 
     * memory needs to be freed here */
}
5 APIs and Porting Layers

Standard C functionality can be found in both OSE and Linux, so file control, string manipulation and similar capabilities will work out of the box on Linux systems. For other functionality there is need for a porting layer that mimic the OSE API. The API layers for running OSE programs in Linux is described in Figure 1. Two porting layers have been tested: SoftApi and OSEPL.

5.1 LINX

LINX is an interprocess communication API and service for Linux and OSE. It is developed by ENEA and boasts with being highly scalable, reliable and open source. In Linux LINX consists of a few kernel modules as well as a programming library. LINX can be used for communication within a single system and communication with other LINX nodes. Communication between nodes can be done either using TCP/IP or raw Ethernet. The advantage of using LINX in the porting layers is that it is very easy to translate OSE signaling calls to LINX calls.

Below is the `send_sig()` function from 4.1.1 implemented using LINX calls. The `LINX` pointer is obtained when initializing LINX using `linx_open()` and is needed for almost all LINX-calls. As can be seen, this looks very similar to the OSE implementation.

```c
void send_sig(LINX *linx, char *name)
{
    LINX_SPID serv_proc;
    LINX_SIGSELECT sigsel_hunt[] = { 1, LINX_OS_HUNT_SIG };  
    union LINX_SIGNAL *sig;

    /* using linx_hunt() to find the process. LINX uses a 
    * LINX_OS_HUNT_SIG by default so there is no need to 
    * allocate it here. */
    linx_hunt(linx, name, NULL);

    /* linx_hunt() consumed the signal, so we can freely use that 
    * variable for a new signal. sigsel_hunt is specified so 
    * that we will only accept LINX_OS_HUNT_SIG signals. */
    sig = linx_receive(linx, sigsel_hunt);

    /* The process can be found using linx_sender() */
    serv_proc = sender(linx, &sig);

    /* we don’t need the signal anymore */
}
```

Fredrik Eriksson
E-Mail: fen05001@student.mdh.se
Phone: 0733-729 828
linx_free_buf(linx, &sig);

/* and now we can send any signal to the server process. */
* I will use PING_SIG in this example, although it's not
* an OSE standard signal */
sig = linx_alloc(linx, sizeof(struct ping_sig), PING_SIG);
linx_send(linx, &sig, serv, proc);

/* just like linx_hunt(), linx_send() consumes the signal, so no
* memory needs to be freed here */
}

In November 2009 when I started my work the latest version of LINX was 2.2.2. This version was incompatible with the lab-environment, and so I created three patches to fix this. Two of the patches fixed the incompatibility with newer versions of the Linux kernel (namely in the /proc and /sys file systems in 2.6.31), and the third patch was used to make the build-process compatible with the Gentoo portage system. In December ENEA released LINX version 2.3.0, which fixed the kernel issues, and therefore only the portage-patch was needed.

5.2 Special Considerations

LINX is used as a communication back-end for both SoftApi and OSEPL. This section describes a few issues with LINX that may be of importance for further evaluation of the porting layers. Note that these issues noted on version 2.3.0 of LINX, which is used unmodified in both SoftApi and OSEPL.

5.2.1 hunt()

According to the OSE-kernel data sheet [7] there are special processes called “blocks”. All prioritized processes (which is the process type used for ordinary user-space processes) belongs to a single block. The blocks are handled just like any other process in the system, with the exception that some operations that are done on the block affects all processes inside that block.

The blocks are also important for the hunt() call. hunt() is used to search for processes with a certain name and according to the OSE API [6], the search is done in a certain order:

1. Processes in the callers block.
2. Other blocks.
3. Processes in other blocks.
4. Processes on other nodes.

In LINX the hunt() call does not search in any particular order. In fact the LINX API [9] states that it is not defined which instance will be found by the hunt in case more then one processes has the same name. This can potentially be a problem for existing OSE programs that is run through one of the porting layers, if they for example name a block the same as one of the processes in the block.
5.2.2 Internode communication

LINX can be used just like OSE signals to send messages to processes running on other systems (although SoftApi may not be reliable in that aspect, see 5.4). I have not examined how, or even if, LINX messages passed on a network differs from external OSE signals, therefore there is no guarantee that nodes running LINX and OSE can communicate with each other.

LINX only implements the communication API, with send(), receive() and similar calls, but there are also other calls in the OSE API that use the same addressing system (in OSE it is simply a PROCESS ID, in LINX it is called SPID). Just as send(sig, pid) sends the signal sig to the process referred to with pid regardless if the process is running on the local system or on another node, kill(proc(pid)) is supposed to kill the process even if it runs on another system. This functionality can not be directly implemented with the aid of LINX. Neither SoftApi nor OSEPL has this functionality implemented.

5.2.3 Reserved signal numbers

The first 4 bytes in a LINX signal is a signal number that can be used to identify the message type. In LINX signal number 0 is invalid and numbers between 250 and 255 are reserved for internal use. None of the available documentation for OSE mentions anything about reserved signals, therefore it might be possible that some programs written for OSE use these signal numbers. If so, then those programs has to be re-written to use other signal identifiers.

It is possible to automatically translate invalid signal numbers in the porting layers. However, since the porting layer does not know what signal numbers are used in the rest of the running applications, there is no way to guarantee that the translation (which must be done both on the sending and receiving side) does not interfere with any other signal number.

5.2.4 Redirection

With the OSE set_redirection() system call it is possible to redirect signals sent to a process to another process instead. There is also a addresser() function that can analyze a signal to determine which process it was originally sent to. Neither of these functions are implemented in LINX. Since these are functions closely related to the signaling system, it is probably better to try to build in this functionality in LINX rather than letting the porting layer keep track of redirections. Neither SoftApi or OSEPL has this functionality implemented.

5.2.5 Context

As mentioned all LINX calls requires the use of a local LINX pointer obtained by calling linx_open(). This pointer cannot be used out of context. If a process opens a LINX pointer and forks, the pointer must be closed by either the parent or the child. Another noteworthy observation is that the LINX pointer cannot be opened by a process and

Fredrik Eriksson
E-Mail: fen05001@student.mdh.se
Phone: 0733-729 828
then used by another thread within the process. This means that the LINX pointer must always be opened by the thread that is going to use it.

5.3 SoftApi

SoftApi is a Linux/Windows NT-implementation of the OSE API written by RealTime Logic. The main purpose of the API seems to be testing of OSE programs in a Windows NT environment.

At first glance SoftApi seems to be a full implementation of the OSE API, but after checking the source code you can see that it is probably not very useful for production environments. The main drawback is the implementation of the signaling system, since a program running in the OSE API can only communicate within one single Linux process. Within this Linux process you can have several different OSE processes, but since those processes refer to each other using internal PIDs no external process can communicate with them.

The API also seem to suffer from quality issues. The code is very sparsely documented and hard to read. It even utilizes goto statements at some places in the code, which is commonly seen as bad practice. The standard makefile for the API does not use any warning flags when compiling. When adding warning flags to the compilation command, the build output showed hundreds of lines with warnings and the compilation was aborted with errors.

5.4 Combining SoftApi and LINX

To make SoftApi able to communicate with other SoftApi processes on a system, it was necessary to replace the communication system implemented in SoftApi. The implementation from RealTime Logic required all OSE processes to run in a single process to have access to the same memory area.

At first glance it may seem like a simple task to integrate LINX into SoftApi. The API is almost the same so making for example send() call linx_send() internally is quite easy to implement. But of course some special care is needed to make sure that the system still behaves as it is supposed to.

SoftApi is limited in that it only allows a single Linux process. All OSE processes started in SoftApi are started as threads within that single process. While it may be faster (especially when creating new OSE processes) and easier to emulate a precise environment for the application, it also has disadvantages. The main disadvantage is apparent when you start many different OSE programs. Since they all share the same Linux process, if a single OSE program crashes, all other applications will also be terminated.

SoftApi has some shared memory in which information about the processes is stored. In RealTime Logic’s implementation of the signaling API this shared memory is also used to store signals. That design makes it impossible to send a signal to a process that has no knowledge about the shared memory, and therefore communication between different instances of SoftApi is not possible. This is not limited to the signals, but also applies to all calls that uses process IDs, as the IDs are used to identify where in the shared memory the process has its data.
When integrating LINX in SoftApi there were two choices: Either create a new ID system where the LINX_SPID can be used as IDs, or add the LINX_SPID as a separate ID and let the signaling functions translate the SoftApi ID to a LINX_SPID. The second option was chosen since that would only need minor changes in the SoftApi code, and it would not result in any major slowdown for any of the system calls, with the exception of `sender()` which is the only function that needs to translate a LINX_SPID to a process ID.

The only functionality that is known to have been broken in the integration of LINX is redirection (see 5.2.4).

These are the known issues of the combination SoftApi and LINX:

- Non-communication calls that take a process ID (like `kill_proc()`) can only be used on processes within a SoftApi instance (not between SoftApi processes regardless if they run on the same or different systems).
- For reasons unknown SoftApi only works if it is statically linked into the program, no dynamic linking.
- Programs compiled for an OSE environment will not work, probably due to type and/or macro declarations. The source code needs to be compiled using SoftApi.
- Every instance of a SoftApi program starts their own sysDeamon and tickDeamon.
- Redirection does not work.

5.5 OSEPL

As SoftApi suffered from quality issues, could not collaborate with other instances and, most of all, run all processes within a single Linux process, I wanted to create an API without these limitations. The result was OSEPL; a quick implementation of the most fundamental OSE functionality. OSEPL is a proof of concept and even some basic functionality (such as semaphores) is not implemented yet. Nevertheless it is more reliable when it comes to multiprocessing than SoftApi is.

Just like SoftApi OSE processes are started as POSIX threads within the API. However, in OSEPL blocks are started as separate Linux processes. The idea is that processes should be started by the block they where created in. Due to time constraints the managing of block process is not completely implemented at the moment, but should still be trivial to implement.

5.5.1 Internal communication

Some OSE calls can be used to request information from other processes. As OSEPL does not have any memory that is shared by all OSE processes this information has to be obtained by some other interprocess communication method.

Since OSEPL uses LINX as backend for the OSE signaling system, all limitations mentioned in section 5.2 applies. Instead of reserving more LINX signals for internal communication, OSEPL processes uses POSIX message queues. The choice of POSIX message
<table>
<thead>
<tr>
<th>Call</th>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start()</td>
<td>SIGUSR1</td>
<td>Start a suspended process</td>
</tr>
<tr>
<td>stop()</td>
<td>SIGUSR2</td>
<td>Suspend a running process</td>
</tr>
<tr>
<td>kill_proc()</td>
<td>SIGTERM</td>
<td>Terminate a process</td>
</tr>
</tbody>
</table>

Table 1: Linux signal used by OSEPL

queues was made for mainly two reasons. First off they are simple to use and supports notification when a new message arrive. The second reason is that POSIX message queues are designed to be deterministic and efficient, which makes them useful in a real-time environment [12].

OSEPL differs from SoftApi in that the OSE process IDs are the same as the LINX_SPID for the process, that way there is no need to translate between different IDs. Whenever a process starts it first opens a LINX endpoint and obtain the local LINX_SPID. Then the process opens a POSIX message queue named “/OSEPL_ID” where ID is the process ID. Whenever a message is sent to this message queue a new thread is started (with the OSE process thread as parent) which takes care of the message.

The following calls will use POSIX message queues when called with an ID that is not the ID of the calling process:

- `get_bid()`
- `get pcb()`
- `set pri for()`
- `start()`
- `stop()`
- `kill proc()`

`get_bid()` and `get pcb()` will return the requested value through a separate message queue named “/OSEPL_REQ_ID”, where ID is the ID of the called process. By using one-way communication through the message queues there is no risk that a request from the above system calls is accidentally read by the wrong process.

For `start()`, `stop()` and `kill proc()` that should affect the execution of the target process, Linux signals are used in addition to message queues. When a process calls for example `stop()` to stop another process, first a message is sent through the message queue that tells the process that stop has been called. The message is received by a separate thread in the process which in turn sends a SIGUSR2-signal to the thread running the OSE process. The signal is caught by a signal handler which halts the process execution by calling `pause()`. The Linux signals used in OSEPL are listed in table 1.

5.5.2 Errors

As mentioned earlier OSEPL is a proof of concept implementation. One of the areas where there is still much work that needs to be done is the error handling. None of the

Fredrik Eriksson
E-Mail: fen05001@student.mdh.se
Phone: 0733-729 828
functions described in ose.h that has to do with error handling is implemented. Like-
wise internally in OSEPL errors are not always taken care of. In some areas (especially
when creating new processes), when something goes wrong the API prints an error and
then exits, while other areas may just report that something went wrong but try to con-
tinue anyway. In other places error codes are not checked at all.

5.5.3 Process Types

OSE describes several different process types, where the most common are prioritized
processes, interrupt processes and block processes. Block processes are implemented
to a certain extent (see section 5.5), but otherwise only prioritized processes can run in
OSEPL at the moment.

5.5.4 Other Limitations

The current implementation of OSEPL is in no way optimized. To simplify debugging
some of the critical sections are far longer than they need to be, and some of the pro-
cedures could be made asynchronous to, for example, take advantage of multi-core
systems. Some of the system calls (especially create_process() and kill_proc() might actually do far more than is needed at that point of execution. In whole OSEPL
is not suited for production environment at this point, but serves more as proof that it
is possible to create a porting layer that takes more advantage of Linux-specific features
than SoftApi does.

Just like for SoftApi, it is not possible to run programs that are compiled for OSE directly
due to differences in macro and/or type declarations in the header files. Instead the
programs has to be compiled against the OSEPL porting layer.
6 Installing Linux

The performance measuring will be on two different software platforms, OSE and Linux, but both will be running on the same hardware. The hardware platform is a GPB64 card with a PowerPC processor. The hardware has almost full support in the Linux kernel so no modifications was needed for it. For these tests the root filesystem was mounted over NFS.

There are many Linux distributions available for the PowerPC architecture, and I choose to install Gentoo. There are mainly three reasons for this:

1. The base installation is fairly small with only basic tools available.
2. It comes with a ready-to-use development environment.
3. I have used Gentoo for a long time and therefore I am very familiar with the system.

Installation of Gentoo is simple, all that is needed is to download a “stage3” snapshot for powerpc (which basically is an archive containing a full root filesystem) and extract it to the NFS root on the server. The only thing that actually needs configuration is one line in /etc/fstab to get the root filesystem mounted automatically, setting root password and configuring the ssh server to start at boot (which in Gentoo is done by running $ eselect rc add sshd default).

Since the development card does not have a working real time clock the time has to be set manually at every boot. This could easily be fixed by letting the card sync the clock over NTP.

Configuration and compilation of the Linux kernel, as well as initialization procedure to run the Linux kernel on the hardware, was done according to instructions provided by Ericsson.

Fredrik Eriksson
E-Mail: fen05001@student.mdh.se
Phone: 0733-729 828
7 Performance

This section describes the methodology and tests used for the performance measurements.

7.1 What is performance?

It is common that software for personal computers (especially games which I will use as examples) comes with a description of minimal system requirements. Often it is possible to run the software on hardware that does not quite fulfill these requirements, but most likely you will run into some sort of problem. Most common is that the application runs too slow, but if you don not have enough memory in the computer or the software is optimized for a different processor, it will probably crash. Either way, running software on a platform that does not perform as well as the software expects is generally a bad idea.

The software developer specifies the system requirements so the end-user knows if it is possible to run the software on his/her computer. But how does the developer know what hardware to specify as requirement?

For game development the developers have to make some trade-offs. Only the most devoted gamers will buy a game that is unplayable on anything but the latest hardware. Nevertheless modern computer games are expected to have advanced 3D-features and realistic physics which generally requires powerful computers. The developer will have to make a decision based on what hardware will be commonly used when the software is released and what kind of features the game should have, to decide the system requirements.

Of course games are not the only area in software where performance is a concern. A web page that takes half a minute to render because of large file sizes and/or advanced scripts is not useful. In this case it might not be the end-users hardware that is the bottleneck, but it may also be the network capacity or the load on the web server.

A third example is VoIP-applications (Voice over IP). Even if the application only uses a small amount of the hardware resources, and even if it the network bandwidth usage is sparse, the user may still experience poor performance if the response time is too long. When speaking with someone over telephone or VoIP-systems we expect that our word reaches our partner at the other end of the line without any delay. Of course instant voice-transportation is not possible, but when the delays can be counted in seconds it gets difficult to have a conversation.

7.2 Performance Metrics

Even if it is fairly easy as an end-user to say that a program performs poorly (after all, the only times we think about performance is when we notice that it is poor), it is very difficult to find a single metric for performance. That said, depending on the application there might be a single property that is of special interest for the application performance. This is especially true if we move away from the area of personal computers and take a look at embedded systems, network equipments and web-based services.
Although there are no universal performance metric, there are a few areas of special interest. Molyneaux defines the following four metrics for performance [15]:

**Availability** defines the amount of time the application is usable for the end-user. This is seldom a problem on personal computers, where the end-user actively has to start the program, and influences from other users/systems are limited. It is a more relevant metric for embedded devices and web-based applications.

**Response time** is the time it takes for the application to respond on input. Both the web-page and VoIP examples above deal with response times. Response times are very important in network equipment such as routers and switches, as they should not be bottle necks for the many network applications that requires short response times.

**Throughput** is a rate metric for the application. It can be how many network packages a router can handle per second, or the frame rate of a game.

**Utilization** describes how the application uses resources. For shared resources a low utilization is generally good (for example will a low memory usage leave more memory for other programs in the system). For non-shared resources a low utilization can mean that you could save money by using cheaper hardware.

For this paper I have only focused on response times, which in this case is execution time of system calls; leaving the other performance aspects for future work.

### 7.3 Test Platform

These tests have been run on a GPB64-card from Freescale, featuring a mpc8641 PowerPC processor. For the tests, all libraries were compiled without debugging symbols and with the gcc optimize flag set to 2.

The Linux userland was a Gentoo stage 3 snapshot from 2009-11-15. The test program was compiled natively on the platform with the gcc and glibc versions below. Other than uninstalling and reinstalling the different porting layers, and recompilation of the test program, nothing was changed between the test run of SoftApi and OSEPL.

- kernel 2.6.31.4
- LINX 2.3.0
- glibc 2.9
- gcc 4.3.4

The OSE installation was at version 4.6.1 and was provided by Ericsson. As the OSE installation did not contain a compiler, programs written for OSE had to be compiled on another platform. This cross compiled environment was provided by Ericsson, and used gcc version 3.4.4.
7.4 Timing Accuracy

One of the difficult things when measuring execution time is to do accurate measurements. Always when doing performance testing using software, the measurement software itself will affect the result [4]. There are many other factors that can affect the performance measurement including hardware interrupts, context switches, caching, memory swapping and more. These factors are relevant even for small installations without many processes and interference from the outside world; like the current test setup. By running many iterations of each test the mean value should give a good indication of the execution time even without having to address every single one of the possible interference factors.

Sjöblom mentions that it was difficult to find a method for measuring time equally on OSE and Linux, but she does not mention what methods she actually used [18]. Instead of focusing on seconds, and letting the operating system do the time measurement, I choose to count the time in clock cycles. I have used inline assembly to read the CPU cycle count directly from the hardware. That way I am sure to measure the time equally in OSE and Linux. As a bonus using assembly makes this operation very fast, thus does not have any major effect on the test result. A test to benchmark the timing overhead showed that about 3-4 clock cycles was needed for the assembly instructions.

7.5 Tests

As mentioned earlier I have, just as Sjöblom did, focused on testing execution time for system calls. The system calls I have chosen to explore can be divided into three different groups. The first group are system calls that exists in both OSE and Linux, and the result of these tests are expected to match the results Sjöblom reached. The second group are system calls that are OSE specific, but not related to the signaling system, and the third group are the signal API calls. The tested system calls in their respective groups are described in table 2.

The test code is available in appendix A. The most crucial code for the tests is the BENCH() macro defined in tests.h. As can be seen from the code the macro lets you define, not only what function(s) should be tested, but also what functions to run before and after the test. For the results in this paper, the tests where run 1000 times, and the timing from every single iteration was saved to a file for later analysis.

7.5.1 Tests for Standard System Calls

These system calls where chosen because of Sjöbloms work. By benchmarking the same system calls, the result can be used to confirm her findings. The tests can also be used together with Sjöbloms tests as references to analyze the impact a porting layer has on the other functions.

The test for malloc() is similar, but not identical to the one Sjöblom did. Just like her I have done two tests of malloc(), one that allocates a small amount of memory, and a second that allocates more memory. Sjöblom did not define exact values for these tests but used pseudo-random numbers in each iteration. Since I want all iterations to do exactly the same thing, I defined the small amount of memory to be 1024 bytes and the
Table 2: Different groups of system calls

<table>
<thead>
<tr>
<th>Group</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>free()</td>
</tr>
<tr>
<td></td>
<td>malloc()</td>
</tr>
<tr>
<td></td>
<td>memcpy()</td>
</tr>
<tr>
<td></td>
<td>memmove()</td>
</tr>
<tr>
<td></td>
<td>memset()</td>
</tr>
<tr>
<td>OSE System</td>
<td>create_process()</td>
</tr>
<tr>
<td></td>
<td>get_bid()</td>
</tr>
<tr>
<td></td>
<td>kill_proc()</td>
</tr>
<tr>
<td>OSE Signals</td>
<td>receive()</td>
</tr>
<tr>
<td></td>
<td>send()</td>
</tr>
<tr>
<td></td>
<td>sender()</td>
</tr>
</tbody>
</table>

large amount to be 1,000,000 bytes. For all other memory functions I only tested for 1024 bytes.

memset() was done by filling a buffer with the number 123, memcpy() and memmove() was done by moving/copying memory within a buffer that was twice the size of the test data (1024 bytes). memcpy() and memmove() was tested twice, the first time the data was copied from the beginning of the buffer to the end, and the second time from the end of the buffer to the beginning. All in all, these tests should be identical to the ones Sjöblom performed.

7.5.2 Tests for OSE System Calls

create_process() is an important call in OSE as it is the only recommended way to create a new processes. It is also one of the calls that are bound to have some overhead in the porting layers compared to the Linux pthread_create(), clone() or fork() calls. Noteworthy is that in OSE a process is created in a stopped state, which means that no time consuming context switch is needed for the create_process() call. It is possible to mimic this in Linux with the clone() call, but neither of the porting layers do this, so context switches are to be expected on Linux.

kill_proc() is almost as important as create_process() as it deals with killing processes. In the porting layers the call is important since all memory and listeners that has been set up for a process must be removed/freed when this call is used.

The last call in this group is the get_bid() call. This is mostly because of the implementation in OSEPL, where the call is forced to use POSIX message queues when figuring out the block ID of a process, unless the call is used on the calling process itself. By testing this call you can see how using POSIX message queues affects the execution time of an otherwise trivial system call.

7.5.3 Tests for OSE Signaling calls

Testing send() and receive() is a given when it comes to testing the OSE signaling system. These calls (along with send_w_s(), receive_w_tmo() and the rest of the fam-
illy) are widely used in OSE program for interprocess communication, and it is important that these calls perform well in the porting layers. Both of the porting layers use LINX as backend for these calls, the only thing that differs between the APIs is how the local process accesses its LINX pointer. These calls are tested both on their own and together in the sequence `send() \rightarrow receive()`. The process to which the signal is sent responds by sending the signal back to the sender as soon as it receives a signal.

The tests of `send()` and `receive()` were done with three different sizes of messages. The smallest messages tested was 128 bytes and the larger messages where 1024 and 4096 bytes.

The benchmarking of the `sender()` call is done because of SoftApi. SoftApi uses internal process IDs that cannot automatically be translated to a LINX_SPID address, which means that SoftApi has to store both a process ID and a LINX_SPID for every process. Every time SoftApi makes a call to a function from the LINX API which uses LINX_SPID (for example `send()` it must translate the local PID. `sender()` differs from all the other calls as the translation is reversed. Instead of translating from an internal PID to a LINX_SPID, the translation goes from a LINX_SPID to a PID. The implementation differs depending on which way the translation should be done, and therefore it might be interesting to see how the `sender()` function from SoftApi differs from OSEPL.
8 Results

The tests mentioned in section 7.5 were supposed to be run on three different software platforms: OSE, Linux with SoftApi and Linux with OSEPL. Unfortunately there were some unforeseen difficulties with the OSE software platform on the test hardware, and therefore a complete test of OSE could not be carried out.

It would seem that the card these tests were run on has much faster memory than the platform Sjöblom tested, as in her tests memcpy(), memmove() and memset took far longer time than malloc() and free() while in these tests the difference in execution time was much smaller. Because of this, and since there are no results from OSE on this platform it is difficult to draw any conclusions about Sjöbloms work.

This section will compare the results of the tests of the different porting layers. The actual test data is available in appendix B for authorized individuals only.

8.1 SoftApi vs. OSEPL

As neither SoftApi nor OSEPL use any wrapper for the standard system calls, the difference between the test results of those functions is negligible. An analysis of the rest of the calls, the OSE system calls and OSE signaling calls, follows in the sections below.

8.1.1 OSE System Calls

Somewhat expected, OSE system calls are far slower in OSEPL compared to SoftApi. Both create_process() and kill_proc() took about 100 times longer in OSEPL.

For create_process() the slowness can probably be explained by the heavy initialization procedures in OSEPL. OSEPL currently initializes the pcb-struct at process creation. This is probably unnecessary for OSE programs that does not extensively uses the get pcb() call, as that is the only call that actually use the struct. Not all initialization procedures can be cut out though, as opening message queues and setting up notification for messages is very important for the API to function.

On multi core systems it might be possible to speed up the creation of processes in OSEPL by making some of the initialization of the new process asynchronous. In the current implementation create_process() initializes as much as possible before creating the new thread. After that it waits until the new thread has run all initializations before it returns. Instead it should be possible to quickly create a new thread, letting the thread initialize LINX, and then return and let the rest of the initialization be done in parallel while the calling process continues its execution.

Even kill_proc() could be done much faster in OSEPL by making the call asynchronous. The current implementation waits for the dying process to send a message back to the caller right before it terminates, letting the caller know that the process has exited. It should be noted that SoftApi does not seem to completely terminate a process as soon as kill_proc() is called, therefore shared resources (such as memory) may not be freed just because kill_proc() has returned.
OSEPL also falls short in terms of predictability in these calls. For `create_process()` most values for OSEPL was measured within a span of about 6 000 000 clock cycles, while SoftApi kept the execution time within a span of 55 000 clock cycles (most of them within 10 000). For `kill_proc()` the difference was not quite as remarkable, but still very apparent.

In `get_bid()` OSEPL falls even further behind as the mean execution time in OSEPL is over 500 times longer than SoftApi. As mentioned earlier the test of `get_bid()` was chosen because of the use of POSIX message queues in the OSEPL implementation, and this test clearly shows that using message queues takes a very long time compared to fetching data from shared memory. To be fair, while it is true that it takes 500 times longer for OSEPL the actual execution time is still very short. It should be noted though that the mean execution time for a short send and receive test with LINX signals was much shorter than `get_bid()` from OSEPL, which implies that using LINX could be faster than POSIX message queues.

`sender()` that was benchmarked because of the somewhat complex implementation in SoftApi does not show any major performance difference between the porting layers. The average execution time differed with just one clock cycle in favour of OSEPL. It should be noted though that OSEPL had a more homogeneous execution time than SoftApi in this test.

8.1.2 OSE Signaling Calls

In the signaling tests OSEPL is marginally faster than SoftApi on all calls. The main difference between the porting layers when it comes to these calls are where the LINX pointer is stored. Most likely the reason SoftApi is slower is because of the function calls used to access the LINX pointer. In OSEPL this pointer can be accessed by the process directly without the need of helper functions.

8.2 Conclusions

The main purpose of this paper was to explore the possibilities of running programs written for OSE under Linux. In this paper I have shown that this is possible to accomplish using a porting layer like SoftApi or OSEPL. Neither of these porting layers can run programs that are not compiled against their respective header files, and without knowledge about how macros and data types are defined in OSE, this limitation is probably difficult to avoid.

Neither one of the porting layers is a clear winner in the performance tests. While SoftApi performs better in everything except communication, the single-process design makes it unreliable in production systems. OSEPL has better design when it comes to multitasking, but still falls short on performance on process creation and termination. Furthermore neither of the porting layers can handle system calls to processes on other systems (see section 5.2.2). One way to solve that particular problem is to examine how OSE behaves on those calls and mimic this functionality, either in the porting layer or directly in LINX. That way it can be made compatible with OSE for clusters with mixed Linux and OSE nodes. Another way (which may not be compatible with OSE) is to dedicate a unused LINX signal number which tells the API that that particular message should be handled by the API.

Fredrik Eriksson
E-Mail: fen05001@student.mdh.se
Phone: 0733-729 828
The recommendation to anyone who is interested in running OSE programs in Linux is to develop a new porting layer. LINX is a good start for the interprocess communication protocol but it will need to be modified if support for redirections is desired. The process design from OSEPL where an OSE block is represented by a Linux process and an OSE process is a POSIX thread is also recommended as it keeps the process hierarchy similar between the platforms. It also isolates the OSE blocks from each other, so that if one process crashes it will not affect the rest of the processes in the system.

Of course it is possible to go all the way and give every OSE process their own Linux process, but processes are grouped together in a block for a reason. More often than not does the processes within a block need each other for correct execution. If one of those processes crashes the others can not continue to execute anyway. Furthermore it is faster to create threads than processes, therefore using processes as blocks and threads as OSE processes is a good compromise between stability and speed.

Since SoftApi stores the information of all processes within the API in shared memory it is very fast for system calls that requests that information. It is probably not possible to create a porting layer that is as fast as SoftApi while still being able to execute these system calls for processes running on external systems. It is more realistic that the performance will be somewhere around the results for the test of `get_bid()` or the send/receive test on OSEPL.

### 8.3 Future Work

The work in this paper can easily be extended for a more complete comparison between the systems. Aside from the performance measurement from OSE that did not make it into this report, the real-time capabilities of Linux can be further examined, for example by comparing the different scheduling methods in Linux. For the porting layers they still need testing in the areas of correctness, stability and predictability.

This report has focused on single-node performance, but one of the advantages of OSE is the clustering capabilities (especially for the signaling system), so functionality and performance testing for multi-node systems should be of interest. In that aspect it would also be interesting to know if it is possible to mix Linux and OSE nodes in a cluster, and what impact that would have on the cluster as whole.
References


Appendices

A Tests

The following three source code files contain the source code for the benchmark tests I have done. It is not the complete program, but only the parts that are relevant to how the measurement was done.

A.1 tests.h

```c
#include <TESTS.H>

/* Some configure options for the tests */
#define SIZE_SMALL 1024
#define SIZE_LARGE 1000000
#define MSG_TINY 128
#define MSG_SMALL 1024
#define MSG_LARGE 4096
#define ITERATIONS 1000

/* where to store files, without trailing slash */
#define FILE_PATH ".
#define NAME_SIZE 64

/* End of configurable options */

#define PING_SIG 0x32
struct ping_sig
{
    SIGSELECT sig_no;
};

union SIGNAL
{
    SIGSELECT sig_no;
    struct ping_sig ping;
};

struct bench_t
{
    char file [NAME_SIZE];
    uint64_t bench_max;
    uint64_t bench_min;
    uint64_t bench_mean;
    uint64_t bench_total;
    uint64_t bench_values [ITERATIONS];
};

#define BENCH(PRE_TEST, TEST, POST_TEST, BENCH_STRUCT) \\"}
uint32_t s_up, s_low, e_up, e_low;
uint64_t bench_total, bench_max, bench_min; \\"
```
```c
uint64_t *bench_values; \
int i = 0; \
bench_total = 0; \
bench_max = 0; \
bench_min = UINT64_MAX; \
bench_values = BENCHSTRUCT->bench_values; \
while (i < ITERATIONS) \
{ \
  PRE_TEST \
  get_time(&s_up, &s_low); \
  TEST \
  get_time(&e_up, &e_low); \
  POST_TEST \
  bench_values[i] = get_diff(s_up, s_low, e_up, e_low); \
  if (bench_values[i] < bench_min) \
    bench_min = bench_values[i]; \
  if (bench_values[i] > bench_max) \
    bench_max = bench_values[i]; \
  bench_total += bench_values[i]; \
  i++ \
} \
BENCHSTRUCT->bench_mean = bench_total/ITERATIONS; \
BENCHSTRUCT->bench_total = bench_total; \
BENCHSTRUCT->bench_min = bench_min; \
BENCHSTRUCT->bench_max = bench_max; \
}
void write_result(struct bench_t *result);
struct bench_t *bench_overhead(struct bench_t *result);
struct bench_t *bench_memcpy(struct bench_t *result, size_t cpy_size);
struct bench_t *bench_memmove(struct bench_t *result, size_t cpy_size);
struct bench_t *bench_memset(struct bench_t *result, size_t size);
struct bench_t *bench_free(struct bench_t *result, size_t size);
struct bench_t *bench_malloc(struct bench_t *result, size_t size);
struct bench_t *benchmark(size_t size);
struct bench_t *benchmark2(struct bench_t *result, size_t cpy_size);
struct bench_t *benchmark_send(struct bench_t *result, PROCESS pid, size_t size);
struct bench_t *benchmark_receive(struct bench_t *result, PROCESS pid, size_t size);
struct bench_t *benchmark_send_receive(struct bench_t *result, PROCESS pid, size_t size);
struct bench_t *benchmark_get爱(struct bench_t *result, PROCESS pid);
struct bench_t *benchmark_create_process(struct bench_t *result, OSENTRYPOINT entrypoint);
struct bench_t *benchmark_kill_process(struct bench_t *result, OSENTRYPOINT entrypoint);

/* timing */
uint64_t get_diff(uint32_t s_up, uint32_t s_low, uint32_t e_up, uint32_t e_low);
void combine(uint32_t up, uint32_t low);
void get_time(uint32_t *up, uint32_t *low);
#endif

A2 timing.c

#include <stdint.h>
/**
 * get how many clock cykles has passed since boot. This is architecture
 * specific and works on powerpc (if POWERPC is defined at compile time)
 * and x86/x86_64.
 * \param up pointer to a 32 bit int to store the upper value
 * \param low pointer to a 32 bit int to store the lower value.
 **/

void get_time(uint32_t *up, uint32_t *low)
{
#ifdef POWERPC
  uint32_t tmp;
```
do
    {
        .asm_
            volatile("mftbu_%[up]"
                : [up] "=r" (+up)
            );
            volatile("mftbu_%[low]"
                : [low] "=r" (+low)
            );
        .asm_
            volatile("mftbu_%[tmp]"
                : [tmp] "=r" (tmp)
            );
    } while (tmp != *up);
#endif

/* combine two 32 bit unsigned integer to a single 64 bit.
 * \param up upper value
 * \param low lower value
 * \return a 64 bit int with the combined value.
 */
uint64_t combine(uint32_t up, uint32_t low)
{
    uint64_t ret = up;
    ret <<= 32;
    ret +=low;
    return ret;
}

/* return the time difference between two intervals.
 * \param s_up upper value for start
 * \param s_low lower value for start
 * \param e_up upper value for end
 * \param e_low lower value for end
 * \return a 64 bit unsigned int with the difference.
 */
uint64_t get_diff(uint32_t s_up, uint32_t s_low, uint32_t e_up, uint32_t e_low)
{
    uint64_t ret;
    ret = combine(e_up, e_low) - combine(s_up, s_low);
    return ret;
}

A.3 tests.c

#include <ose.h>
#include <tests.h>
#include <string.h>
#include <stdio.h>
void write_result(struct bench_t *result)
{ FILE *fd;
    char tmp[32];
    int i;
    fd = fopen(result->file, "w");
    for(i = 0; i < ITERATIONS; i++)
    {
        snprintf(tmp, 32, "%llu\n", result->bench_values[i]);
        fwrite(tmp, strlen(tmp), 1, fd);
    }
    fclose(fd);
}

struct bench_t *bench_overhead(struct bench_t *result)
{
    snprintf(result->file, NAME_SIZE, "%s/%s", FILE_PATH, func);
    BENCH(
        void *test;
        test = malloc(size);
        if(test == NULL)
            return NULL;
        free(test);
        result
    );
    return result;
}

struct bench_t *bench_malloc(struct bench_t *result, size_t size)
{
    snprintf(result->file, NAME_SIZE, "%s/%s\d", FILE_PATH, func, size);
    BENCH(
        void *test;
        test = malloc(size);
        if(test == NULL)
            return NULL;
        free(test);
        result
    );
    return result;
}

struct bench_t *bench_free(struct bench_t *result, size_t size)
{
    snprintf(result->file, NAME_SIZE, "%s/%s\d", FILE_PATH, func, size);
    BENCH(
        void *test;
        test = malloc(size);
        if(test == NULL)
            return NULL;
        free(test);
        result
    );
    return result;
}

struct bench_t *bench_memset(struct bench_t *result, size_t size)
{
    snprintf(result->file, NAME_SIZE, "%s/%s\d", FILE_PATH, func, size);
    BENCH(
        void *test;
        test = malloc(size);
        if(test == NULL)
            return NULL;
        free(test);
        result
    );
    return result;
}
memset(test, 123, size);
free(test);
result = result;
}

struct bench_t *bench_memcpy(struct bench_t *result, size_t cpy_size)
{
    snprintf(result->file, NAME_SIZE, "%s/%s.%d", FILE_PATH, func, cpy_size);
    BENCH(
        int index;
        void *test;
        void *ptr;
        test = malloc(cpy_size+2);
        if (test == NULL) return NULL;
        memset(test, 123, cpy_size);
        ptr = test+cpy_size;
        memcpy(ptr, test, cpy_size);
        for(index = 0; index < cpy_size+2; index++)
            if ((char *)test)[index] != 123)
                return NULL;
        free(test);
        result = result;
    )
    return result;
}

struct bench_t *bench_memcpy2(struct bench_t *result, size_t cpy_size)
{
    snprintf(result->file, NAME_SIZE, "%s/%s.%d", FILE_PATH, func, cpy_size);
    BENCH(
        int index;
        void *test;
        void *ptr;
        test = malloc(cpy_size+2);
        if (test == NULL) return NULL;
        ptr = test+cpy_size;
        memset(ptr, 123, cpy_size);
        memcpy(ptr, test, cpy_size);
        for(index = 0; index < cpy_size+2; index++)
            if ((char *)test)[index] != 123)
                return NULL;
        free(test);
        result = result;
    )
    return result;
}

struct bench_t *bench_memmove(struct bench_t *result, size_t cpy_size)
{
    snprintf(result->file, NAME_SIZE, "%s/%s.%d", FILE_PATH, func, cpy_size);
    BENCH(
        int index;
        void *test;
        void *ptr;
        test = malloc(cpy_size * 1.5);
        if (test == NULL)


```c
return NULL;
ptr = test+(cpy_size/2);
memset(test, 123, cpy_size);
memmove(ptr, test, cpy_size);
for(index = 0; index < cpy_size*1.5; index++)
    if(((char*)test)[index] != 123)
        return NULL;
free(test);
return result;
}

struct bench_t *bench_memmove2(struct bench_t *result, size_t cpy_size)
{
    snprintf(result->file, NAME_SIZE, "%s/%s", FILE_PATH, _func_, cpy_size);
    BENCH(
        int index;
        void *test;
        void *ptr;
        test = malloc(cpy_size*1.5);
        if(test == NULL)
            return NULL;
        ptr = test+(cpy_size/2);
        memset(ptr, 123, cpy_size);
        memmove(test, ptr, cpy_size);
        for(index = 0; index < cpy_size*1.5; index++)
            if(((char*)test)[index] != 123)
                return NULL;
        free(test);
    }
    return result;
}

struct bench_t *bench_create_process(struct bench_t *result, OSENTRYPOINT entrypoint)
{
    snprintf(result->file, NAME_SIZE, "%s/%s", FILE_PATH, _func_);
    BENCH(
        PROCESS pid;
        pid = create_process(OS_PRI_PROC, "test_proc", entrypoint, 1000, 15, 0,0,0,0,0);
        kill_proc(pid);
    }
    return result;
}

struct bench_t *bench_kill_proc(struct bench_t *result, OSENTRYPOINT entrypoint)
{
    snprintf(result->file, NAME_SIZE, "%s/%s", FILE_PATH, _func_);
    BENCH(
        PROCESS pid;
        pid = create_process(OS_PRI_PROC, "test_proc", entrypoint, 1000, 15, 0,0,0,0,0);
        kill_proc(pid);
    }
    return result;
}
```

Fredrik Eriksson
E-Mail: fen05001@student.mdh.se
Phone: 0733-729 828
```c
struct bench_t *bench_sender(struct bench_t *result, PROCESS pid)
{
    SIGSELECT sigsel,any[] = { 0 },
    snprintf(result->file, NAME, "%s" , FILE_PATH, func);
    BENCH(
        union SIGNAL *sig;
        sig = alloc(sizeof(struct ping), PINGSIG);
        send(&sig, pid);
        sig = receive(sigsel,any);
        pid = sender(&sig);
        free_buf(&sig);
        result;
    );
    return result;
}
```

```c
struct bench_t *bench_send_receive(struct bench_t *result, PROCESS pid, size_t size)
{
    SIGSELECT sigsel,any[] = { 0 },
    snprintf(result->file, NAME, "%s" , FILE_PATH, func);
    BENCH(
        union SIGNAL *sig;
        sig = alloc(size, PINGSIG);
        send(&sig, pid);
        sig = receive(sigsel,any);
        free_buf(&sig);
        result;
    );
    return result;
}
```

```c
struct bench_t *bench_receive(struct bench_t *result, PROCESS pid, size_t size)
{
    SIGSELECT sigsel,any[] = { 0 },
    snprintf(result->file, NAME, "%s" , FILE_PATH, func);
    BENCH(
        union SIGNAL *sig;
        sig = alloc(size, PINGSIG);
        send(&sig, pid);
        sig = receive(sigsel,any);
        free_buf(&sig);
        result;
    );
    return result;
}
```

Fredrik Eriksson
E-Mail: fen05001@student.mdh.se
Phone: 0733-729 828
BENCH(
    union SIGNAL *sig;
    sig = alloc(size, PING_SIG);
    send(&sig, pid);
    sig = receive(sigsel,any);
    free_buf(&sig);
    result
);
return result;
}

struct bench_t *bench_get_bid(struct bench_t *result, PROCESS pid)
{
    snprintf(result->file, NAME_SIZE, "%s/%s", FILE_PATH, __func__);
    BENCH(
        , get_bid(pid);
        ,
        result
    );
return result;
}
B Test Values

Available for authorized individuals only.