CONTENTS

A.4 Initial system configuration ........................................ 161

B  The GIT and Cogito tools ........................................... 167
   B.1 Installation ....................................................... 168
   B.2 Branches ........................................................ 169
   B.3 Checking out a tree ............................................ 171
   B.4 Keeping a tree up to date ..................................... 172

C  Building a custom kernel for the Cell .............................. 173
   C.1 Fetching the sources ........................................... 173
   C.2 Setting up the cross-compiler ................................ 174
   C.3 Generic build process .......................................... 175
   C.4 PlayStation 3 details .......................................... 175
   C.5 IBM SystemSim details ........................................ 176
Preface

This is the final report for my Projecte Final de Carrera (PFC) for the Enginyeria en Informàtica degree. This is a research project that focuses on heterogeneous multiprocessor systems, looking for better ways to represent their components and to optimize their management.

This preface is here to guide you through the lengthy report. It describes, among other things, how it is organized and the typographical conventions used in the printed text. Of course, it also includes some acknowledgements.

Abstract

Heterogeneous multiprocessor systems are becoming popular nowadays but, unfortunately, current operating systems do not handle them properly. On the one hand they fail to abstract all their processors at a generic level so they duplicate complex algorithms to manage each kind of processor. Furthermore, all these processor are not properly exposed to the user because some of them are seen as Input/Output (I/O) devices.

On the other hand, these operating systems provide primitive programming mechanisms to deal with the variety of processors in a heterogeneous multiprocessor systems. This makes writing efficient, parallel applications a very complex matter because they way to handle all the processors is unnatural when compared to the traditional execution of programs.

This project investigates the problems just described and proposes novel approaches to resolve them or, at least, improve the current situation by providing more natural alternatives.
• Constants are printed using a fixed-width typeface. For example: MAX_LENGTH.

• C type names are printed using a fixed-width typeface. For example: enum mode.

• Commands that can be entered on the command line, with or without parameters, are printed using a fixed-width typeface. For example: ls -l /tmp.

• Environment variables are printed using a fixed-width typeface. For example: PATH.

• Function names are printed using a fixed-width typeface followed by a couple of empty parenthesis regardless of the function's signature. For example: do_something().

• File names and paths are printed using a fixed-width typeface. Directory names have a trailing slash. For example: /usr/local/bin/ or /etc/fstab.

• Standard commands and function names are shown as manual page references. This is to disambiguate the cases where a single name can refer to multiple different items. They are printed using a fixed-width typeface. For example: ls(1) or printf(3). For the list of possible sections and their meaning see table 1.

<table>
<thead>
<tr>
<th>Section</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>General commands</td>
</tr>
<tr>
<td>2</td>
<td>System calls</td>
</tr>
<tr>
<td>3</td>
<td>Library functions</td>
</tr>
<tr>
<td>4</td>
<td>Kernel interfaces</td>
</tr>
<tr>
<td>5</td>
<td>Configuration files</td>
</tr>
<tr>
<td>6</td>
<td>Games</td>
</tr>
<tr>
<td>7</td>
<td>Miscellaneous information</td>
</tr>
<tr>
<td>8</td>
<td>System manager's commands</td>
</tr>
<tr>
<td>9</td>
<td>Kernel internals</td>
</tr>
</tbody>
</table>

Table 1: Standard manual sections.

• Variable definitions are printed using a fixed-width typeface. For example: int count.

• Variable names are printed using a fixed-width typeface. For example: count.

• Mathematical notation is printed using an italic typeface. For example: \( i \in [0..100] \).
ACKNOWLEDGMENTS

Thanks go to the individuals at the cbe-oss-dev and lkm mailing lists for their support in all the implementation doubts that arose; to the people at the LCAC, in special to Roman Valls, for their assistance when dealing with the department’s machines and resources; to Víctor Jiménez for giving me a brief overview on his initial asynchronous execution model; to Jordi Caubet for assisting me in using the BSC resources; and to Dimitris Nikolopoulos and Daniel Jiménez for providing us with real-world Cell applications.
Part I

Project settlement
Chapter 1

Introduction

The computers we use today are built around the Von Neumann architecture, just as they have been for a very long time. This abstract model divides a general-purpose computing machine in three conceptual blocks: a Central Processing Unit (CPU) (which can be composed of one or more processors), a certain amount of main memory and zero or more I/O devices. All of these elements communicate through an interconnection bus as depicted in figure 1.1.

![Diagram of the Von Neumann architecture]

Figure 1.1: The Von Neumann architecture.

With the advance in technology – smaller and more efficient chips –, single-processor machines morphed into multiprocessor computers which bundled more than one CPU of the same kind. These are known as homogeneous multiprocessor system or, depending on the technology used to achieve this cooperation, Symmetric MultiProcessing (SMP) machines (further described in section 2.3).
1.1 Current status

Multiprocessor systems are just starting to become popular nowadays. Unfortunately, it is yet unknown what is the best way to leverage their power when executing common user-level tasks such as, for example, word processing or gaming. All these applications are generally single-thread bounded, so they cannot take advantage of multiprocessor machines, and it is not easy to turn them into a multi-threaded design due to the current programming models and operating system abstractions.

Things get worse when these applications are put in heterogeneous multiprocessor systems because it is harder to think how to take advantage of the processing power offered by all the PEs. It is hard because there is no support in programming languages to make this transparent and because programmers are not used to such machines.

It is expected that research in heterogeneous multiprocessor system will be very important in the coming years. There are many areas within this topic that are worth to inspect, and each of them is a world on its own. The following sections try to outline some of the current problems in nowadays' operating system to manage these machines.

1.1.1 Description of PEs

Current operating systems do not have an abstraction that is able to describe any PE in the system: they are only able to describe boot-time PEs at a Machine-Independent (MI) level. In general, it is completely impossible to manage PEs that are different to the boot ones simply because this case is not considered in traditional kernels. But there are some exceptions; in these cases, the PEs that are not equal to the boot ones are exposed to the system in a completely separate way, typically in the form of I/O devices. That is, the processor appears to the system as a device where you send a command (start the program on that processor), then wait until a finalization event is raised and then collect the results of the command (the program results).

One could say that this is actually adequate and that it works in practice. In fact it does because this model is currently in use in some heterogeneous multiprocessor system. But now consider an hypothetical machine with two general-purpose processors: a PowerPC and an Intel i386-compatible one. In this situation, none of the two processors would be more important
1.1.3 Memory management

We can classify multiprocessor system architectures into three different groups as regards the way they see and access the main memory:

**SMP** Two or more identical processors are connected to a single, shared main memory.

**Non-Uniform Memory Access (NUMA)** Processors are divided into independent nodes, each one with a certain amount of local memory. The local memories of all nodes are grouped together to form the machine's main memory. As is easy to imagine, accessing the local memory is a very fast operation, but reaching the memory in another node—typically through a optical fiber network—has a severe performance penalty.

**Asymmetric MultiProcessing (ASMP)** Not all memory is available to all processors, and they needn't be identical. They may share only a portion of the main memory or may have completely decoupled local storage.

Heterogeneous multiprocessor systems form the less-restricted multiprocessor model, so they can be seen as ASMP systems. As PEs are internally different, each one may only be able to see part of the total memory or, even more, provide some memory on its own that is only visible from inside it.

Current systems do not present these distinct levels of memory in a natural way. The programmer may be required to issue manual Direct Memory Access (DMA) transfers among memory zones, which is not the typical way to access system memory. It could be interesting to investigate and abstract these concepts to see if it is possible to present such processor-specific memory access patterns to the system in a more transparent fashion.

1.1.4 Binary formats

General-purpose processors can perform any task but, obviously, they are not as efficient as if a specific-purpose processor ran that same task. This is the main reason for having extra chips in a system aside from lowering the load of the main system processors. At the moment, the user must be aware of differences in PEs and manually submit the tasks (binary programs at their roots) to the most appropriate one.
1.2. GOALS

reachable from any other subsystem in the kernel, and not only from Machine-Dependent (MD) ones.

We start this work by characterizing the wide variety of processors available in the market and studying how Linux manages them at the moment. We later propose a design for our PE system, implement it and evaluate its behavior in a real-world heterogeneous multiprocessor system.

This subproject is detailed in part II.

1.2.2 Scheduling improvements

Once we have got the PE system working, we can unify the separate process schedulers in a current system and create a single MI scheduler that can manage tasks for all the PE in the system.

To do this successfully, we first study how different types of processors are currently scheduled in a real system. Later on, we define the application concept, which allows us to group execution threads for independent PEs into a single representation. At last, we define new behaviors for the process scheduler to take into account the application concept. This way the system can take more intelligent decisions as regards scheduling, as it can see the real requirements of an application as a whole, and not at the thread-level only.

Aside from doing conceptual proposals for improvements, we also implement one of them—the simple one—to test it on real hardware and evaluate how it stacks up with the original scheduling algorithms.

This subproject is detailed in part III.

1.2.3 Asynchronous execution

This subproject is quite separate from the others but is part of the global project because it proposes a novel way to deal with thread execution on non-boot PEs.

As we have already mentioned, the current model to execute a program in a PE involves launching an operation to start its processing and then waiting for its termination. This is very similar to a coroutines model. We improve this programming model by introducing the ability
Chapter 2

Preliminary concepts

This chapter presents several basic concepts that are required to understand the rest of this report. Feel free to skip it if you are familiar with the terms outlined here, or refer to this chapter whenever you need it. Of course, you can also check the definitions in the glossary.

2.1 Programs, processes and threads

A program is a sequence of instructions that perform some transformations of main memory’s contents or change the status of I/O devices. A binary program is the translation of a program written in a high-level language into a series of machine-level instructions. These instructions can be interpreted directly by the microprocessor.

A process is the run-time representation of a program. It includes the description of the resources used by the program such as memory areas, usage of I/O devices and its relationships with other processes. This is highly dependant on the operating system.

A thread is an execution flow within a process. Threads are represented by the current state of the microprocessor; basically this means the register’s values. Processes can have multiple threads within them.

Figure 2.1 shows the relationships between programs, processes and threads. The image illustrates the program as something that belongs to the disk; once executed, it is transformed into a process that contains an image of the program’s code and data as well as multiple execution threads that work on this code.
2.3. SMP AND SMT

scheduling from the operating system's point of view. This is a kind of homogeneous multiprocessor system machines and typically all the components in a SMP setup are the machine's native PEs.

Simultaneous MultiThreading (SMT) [Con07b], on the other hand, is a technique that allows a processor to simultaneously manage the execution context of multiple threads. This is done to reduce the penalty of context switches, which is very useful for synchronization purposes, and to keep the processor's pipeline full, as its fetch stage can retrieve instructions from different threads at each cycle. SMT exposes each of these execution threads as virtual PEs, so, for example, an operating system running on a single Intel Pentium 4 with SMT enabled will see two virtual processors.

The techniques to schedule processes among SMP and SMT systems vary slightly since the former works with different physical processors and the later with virtual ones. However, as both are homogeneous multiprocessor systems, they are fairly simple and similar to manage.
Chapter 3

Case study: The Cell Broadband Engine

In order to be able to simplify our analysis and test our proposed changes, we must focus the project on a specific heterogeneous multiprocessor system. This allows us to consider real hardware and software implementations, which in turn makes the overall project more realistic.

This chapter presents the Cell, the platform we have chosen for our development. The reasons for doing so are cited later on. Of course, all the ideas proposed in this report are applicable to other heterogeneous multiprocessor system; the implementation, however, will probably not be.

3.1 The PowerPC architecture

The PowerPC [Con07c] is a Reduced Instruction Set Computing (RISC) microprocessor architecture jointly created by IBM, Apple and Motorola. It is based on IBM's older POWER architecture, and remains compatible with it in many areas. This microprocessor was originally designed for personal computers, but has recently gained popularity in the embedded and high-performance marked due to its flexibility and efficiency. In fact, it is widely used in research projects nowadays.

Compared to the Intel i386 architecture, the PowerPC was cleanly designed from the ground up. It has few but very powerful instructions. It also has a big register bank which allows
The memory interface This provides shared access to the system’s main memory from either the PPEs or the SPEs.

The debug and testing unit This provides performance counters and facilities to debug the execution of programs in the processor.

![Diagram of Cell chip internal representation](image)

Figure 3.1: Internal representation of a Cell chip.

For an introduction to programming the Cell, refer to [Bar07c], [Bar07d], [Bar07e], [Bar07f], [Bar07g] and [Bar07h].

3.2.1 The PowerPC Processing Element

The PPE is the generic-purpose processor found in the Cell. It is very similar to the PowerPC 970, described in section 3.1, in the sense that it supports the same instruction set and has a similar internal design, but has much reduced performance. Its basic tasks are the execution of the operating system’s kernel, the orchestration of the SPEs and the visualization of their results.

The *PowerPC Processing Unit (PPU)* is the processing core found inside the PPE, as the latter bundles some more components aside from the PowerPC processor. This PPU is presented to the operating system as two different virtual processors thanks to the SMT technology.
3.4 Linux on the Cell

While the Cell is able to run any operating system that supports the PowerPC 970, not all operating systems are able to manage the extra functionality included in the Cell such as its SPEs. Therefore, we have to focus on a system that currently has all this functionality bundled in so that we need not care about adding this support ourselves. This way we can directly start working on its resource management and abstraction techniques.

Nowadays, Linux 2.6 is the main development platform for the Cell architecture\(^1\). It has been ported to almost all machine configurations that have a Cell chip in them, such as the IBM Cell Blades, the PlayStation 3 or the IBM SystemSim for Cell.

We have chosen Linux 2.6 as the foundation for our work. See appendixes A and C for more details on how to get Linux to work on a Cell-based machine.

\(^1\)At the time of this writing, NetBSD was also ported to the PlayStation 3. However, it did not have any interface to manage the SPEs, so it was not useful to us.
Part II

Generalization of PEs
Chapter 4

Characterization and representation of PEs

PEs are very complex devices: they have a lot of internal details that define how they behave and how they interpret programs. Furthermore, there are a lot of PE architectures out there, each one with its own characteristics and features.

This chapter presents an overview of different properties that define a PE and how they could be abstracted to characterize all different designs. It later analyzes how the Linux kernel currently represents PEs and how they are exported to user-space programs. Its main goal is to gather the list of functional and non-functional requirements for the development of an abstraction layer to manage all available PEs in a machine. The results of this work are presented in chapters 5, 6 and 7.

The text in this chapter tries to be as objective as possible, presenting a concise analysis of the current representation of PEs.

4.1 Characterization of PEs

The following subsections list the major items that characterize a PE. Each one is described in detail to get a better understanding of how it can be abstracted.
4.1. CHARACTERIZATION OF PES

- GPRs: This set of registers is available to the software developer to store temporary data in them. Temporary data includes those that have been fetched from memory, those to be stored in memory, those to be fed to operations and those that are results of operations.

- FPRs: These registers are very similar to the GPRs but hold floating-point values instead of integral ones.

- Special-Purpose Registers (SPRs): Also known as control registers, these allow the programmer to customize how the processor behaves and to query its status. Among these we can find a flags register, which indicates several conditions after an arithmetic operation, a privilege level register to indicate the current execution mode, a register to mask external interrupts, etc.

However, specific-purpose PEs may have different sets of registers or lack them all. An example of this last case are some FPGAs, because they are only able to execute a predefined operation and need not execute a binary program provided by the user.

4.1.4 ISAs

As we have already mentioned, a processor is a computational unit that executes instructions [Con07h]. Each instruction is defined by an operation code (opcode) – which is a number that identifies the action to execute – and a set of parameters to that opcode. For example, we might have an arithmetical addition instruction identified by the 45 opcode accepting three parameters: one specifying the register on which to leave the result and the other two indicating the two source registers which have to be added.

OpCodes and their parameters are encoded as series of bytes, and the succession of different instructions compose a program. The way this encoding is done depends on the processor architecture. For example, Intel i386 machines use variable-length encoding, which means that each instruction can have a different length from all others depending on its parameters; on the other hand, the PowerPC architecture uses fixed-length encoding, meaning that all instructions have the same length regardless of their functionality or parameters.
4.1. CHARACTERIZATION OF PES

4.1.5 Local memory

Aside from the register bank, processors can have additional local memory. This memory is
different from main memory and the two may need to be accessed in different ways. Because of
this, the cost of accessing local memory may be different to the cost of accessing other memory.

Nowadays, most processors do not have local memory. The SPUs, however, do. The current
versions have a 256Kb local memory – known as the LS– that is seen by the SPU as its
main memory and thus can be accessed through regular load/store operations. Accessing the
machine’s main memory has to be accomplished by means of DMA, which is not transparent to
the programmer.

The processor’s local memory may also be accessible from other processors through different
techniques. It may be possible to map this local memory in some main memory address range
or it may be possible to access it through DMA.

We can characterize local memory through these properties:

**Initial local address** This is the local memory’s starting address; all addresses emitted by the
processor that fall into this memory must be greater than this initial value. In general, it is
expected for this address’ value to be zero (or fixed to a constant value) so that programs
loaded into local memory do not need to be relocated at runtime.

**Global mapping address** This is the main memory’s address on which the processor’s local
memory is mapped at. Only available if this mapping is possible.

**Size** The local memory’s size, in bytes.

**Global name** The local memory’s name, if any, through which other processors can refer to it.

4.1.6 Memory-management unit

The *Memory Management Unit (MMU)* [Con07i] is the processor’s functional unit in charge
of managing access to main memory. Whenever the processor executes an instruction that
refers to memory (typically a load or a store operation), the MMU traps this access and issues
transformations on the requested address. This way the machine can implement techniques such
as pagination and segmentation at the hardware level.
4.1. CHARACTERIZATION OF PES

**Hierarchy** The hierarchy to reach main memory from the processor. Most current processors have two cache levels: the first-level (L1) cache is small but very fast, and is placed very close to the processor; the second-level (L2) cache is much bigger, slower, and may be placed outside the processor's chip.

**Data/code separation** Some caches are split in two parts: one for instructions and one for data. This is the typical case for L1 caches.

**Size** The cache's size, in bytes. If it is split in multiple parts, it is worth to also store the size for each of them.

4.1.8 Power management

Power consumption is an important research area in current processors. The faster they are, the more they consume, hence generating more heat and drawing more power. Some techniques have been introduced to reduce the power consumption of processors depending on their load, such as automatic (or manual) frequency throttling, or different operation modes.

Operation modes can be characterized on their own because the processor has different features on each of them:

**Power consumption** The approximate power drawn in the given mode when running at full load.

**Performance** The approximate performance provided by the processor in the given mode. This may need to be specified as some relative index with respect to other modes.

4.1.9 Context switches

Context switching [Con07k] allows a single processor to stop the execution of a program – even if it has not yet finished – and starting the processing of a new one. This is typically used to simulate the execution of multiple tasks simultaneously.

For example: the PPE uses context switching to schedule processes during very short time intervals, creating the illusion of many different threads running at the same time. SPEs, on the
4.1. CHARACTERIZATION OF PES

is accessing memory outside its domains and hence manipulating processes it is not allowed to touch.

4.1.11 Input/output

Processors on their own cannot do much more than execute instructions and manage the memory’s contents: they need to communicate with different I/O devices to interact with the user, the network or the storage system. However, not all processors need to be able to do I/O because other processors in the system can do this work on their behalf.

A specific example: the SPEs in the Cell cannot do any I/O. However, they can ask the PPE to execute a service, and, in special, this service can be an I/O operation.

The I/O interface can be used to communicate with any kind of device, and the way to do so is highly dependent on the specific device’s programming interface. The device may provide Memory-Mapped I/O (MMIO) registers and/or I/O ports for communication. Even more, some of these devices may be processors on their own.

4.1.12 Interrupts

When a processor needs to receive notifications from external hardware, it can either use a polling mechanism or asynchronous notification through interrupts [Con07m]. The interrupts a processor can receive is something that is typically hard-wired in the hardware so, once the processor is attached to a mainboard, this set remains fixed.

It is important to understand which interrupts can appear on a system as a whole and which processors are able to handle them, either because they are physically connected to receive such notifications or because they can run the necessary code – the Interrupt Service Routines (ISRs)– to manage them.

Furthermore, the execution of programs can raise exceptions, which are similar to interrupts but caused by software. Similarly, it is important to know which exceptions can arise and who can handle them for proper system operation. E.g. a SPE can generate exceptions but it is unable to resolve them on its own: it needs PPE services to fix the problems before it is able to continue execution.
4.2. REPRESENTATION OF CPUS UNDER LINUX

ready to be used, hence this other characterization. As happens with the present CPUs group, this one is also dynamic.

This group is represented by the cpu_online_map bitmap.

The size of the bitmaps described above is defined at compile time by setting the NR_CPUS constant to the maximum number of CPUs the kernel can support. New bitmaps can be defined by using the cpumask_t type.

Also note that, in the case of multi-core processors (including the virtual ones provided by SMT), every physical CPU will take multiple entries in each bitmap. This situation is tricky because it is not possible to know how much cores a non-present CPU may have, and hence how many entries should be reserved in the map. However, as we assume that all CPUs are identical, we size those entries as we size any other one that corresponds to a present CPU (the boot one, for example).

Figure 4.1 illustrates the contents of these maps on a motherboard with 6 sockets, three of which are empty and the other three being occupied by SMT-enabled CPUs. Of the three available CPUs, one of them is off because it failed initialization. Note that each map takes 16 bits as it is sized according to the NR_CPUS constant defined at build time, which was set to that value.

Figure 4.1: Example of the CPU maps on a motherboard with 6 sockets.
4.2. REPRESENTATION OF CPUS UNDER LINUX

struct rq {
    ...
};
DEFINE_PER_CPU(struct rq, runqueues);

Table 4.1 illustrates an example of a single structure definition versus its corresponding representation using the percpu framework.

    struct cpu {
        int ident;
        char name[8];
        struct rq runqueue;
    }

    struct cpu all_cpus[NR_CPUS];

    DEFINE_PER_CPU(int, idents);
    DEFINE_PER_CPU(char [8], names);
    DEFINE_PER_CPU(struct rq, runqueues);

Table 4.1: Comparison between a single structure and its corresponding percpu definitions.

Conceptually, the DEFINE_PER_CPU() macro reserves an array of struct rq elements of size NR_CPUS. This array is initialized lazily: the kernel’s boot process dynamically allocates an array of the given type, but it generally is of a much smaller size: instead of providing space for NR_CPUS elements, it provides space for the number of possible CPUs (as obtained by the cpu_possible_map bitmap already described). This allocation happens in the call to setup_per_cpu_areas() made by init/main.c; in other words, it is not safe to use percpu until this call has been made, which happens much after the basic machine initialization code has run (the one that initialize the CPUs themselves).

Later on, when the code desires to access a CPU’s specific field, it uses the per_cpu() accessor. This function also takes two parameters: the field to access and the CPU identifier. So, if we wanted to get a pointer to the run-queue of the third CPU, we could do:

struct rq *rq = per_cpu(runqueues, 3);

Now that we know the internals of the percpu framework’s basic internals, we can see why the kernel uses it instead of a single structure\(^3\):

\(^3\)http://lkml.org/lkml/2007/5/3/287
4.3. LACK OF INTEGRATION OF FOREIGN PES

CPU_ONLINE The CPU has been properly activated.

CPU_UP_PREPARE The CPU is going to be activated. Give all subsystems a chance to report errors before the CPU is really brought up.

CPU_UP_CANCELED Some listeners reported an error during the preparation phase, so the CPU was not activated.

CPU_DOWN_PREPARE Tell all listeners that the CPU is going to be removed; give them a chance to deny this operation by reporting an error or to complete any pending operation.

CPU_DOWN_FAILED Some listeners reported an error during the preparation phase, so the CPU cannot be removed and is left active.

CPU_DEAD The CPU was properly deactivated so it can safely be removed.

CPU_LOCK_ACQUIRE Acquire all locks related to the CPU’s hotplugging.

CPU_LOCK_RELEASE Release all locks related to the CPU’s hotplugging.

4.3 Lack of integration of foreign PEs

This section describes multiple areas in the current Linux kernel in which we can clearly see that foreign PEs are not properly integrated with native ones.

4.3.1 List of available CPUs

There are several areas in the kernel that expose a list of available CPUs to the user, but these lists only include native PEs. Foreign PEs are not taken into account and are handled in completely different lists (if they exist).

The first example of this situation is the /proc/cpuinfo file. This text file shows a list of all available CPUs alongside a description of some of their properties. The current output of such file in a PlayStation 3 reports:

```
processor : 0
cpu : Cell Broadband Engine, altivec supported
```
4.3. **LACK OF INTEGRATION OF FOREIGN PES**

machine-specific parts (which may completely omit the registration of processing units inside the sysfs).

### 4.3.2 Task representation

Native processes and threads are currently represented by the `struct task_struct` type, described in more detail in section 8.1.1. Processes running on foreign PEs are represented using independent data structures and are used by completely different algorithms. For example, programs running on SPUs are abstracted by means of the `struct spu_context` type, described in 8.2.1. If we added another kind of PE to the machine, we could need another structure to represent its tasks because using the generic task structure could collide with their current usage in generic parts of the system.

It would be nice if the generic task structure could be used to represent *any* process in the system regardless of the type of the PE in which it was running. This way, machine-independent subsystems (such as the process scheduler) could deal with all active tasks, not only those running in native PEs.

Even more, and focusing on the Cell: this generalization could remove the need to keep a *SPU thread* to represent each SPU process given that the kernel could maintain a more detailed status (open file descriptors, memory mappings, etc.) for the SPU task on its own.

### 4.3.3 Process scheduling

The Linux process scheduler is only used to schedule tasks for native PEs. Foreign ones require a customized scheduler, if they need one. For more details on how PPEs and SPEs are currently scheduled, see chapter 8.

Furthermore, the system provides scheduling statistics to the user through the plain-text `/proc/schedstat` file:

```
version 14

timestamp 4294905440

cpu0  0  0  20  52  108  19586  5793  10838  9550  2021  5853  13793
domain0  3  2858  2803  10  25696  59  0  0  2803  18  16  0  1942  3  0  0  16  5808  5793  0
```
4.3. LACK OF INTEGRATION OF FOREIGN PES

If foreign PEs were treated as regular CPUs, their execution time could be accounted into a process' run time. The key idea is that those foreign threads could be incorporated into a process, hence counting the run time they take.

Similarly, and for the same reasons, a thread running in a foreign PE is not taken into account in the global machine load. When a process is waiting for a foreign thread to terminate, it appears as not busying any CPU in, e.g. the listing shown by `top(1)` or in the machine's load average shown by `uptime(1)`.

4.3.5 Binary execution

Almost all programs are nowadays provided as Executable Link Format (ELF) binaries [Com95]. The kernel provides a module, the binfof.elf, to load these binaries into memory and set up the caller’s process to execute them. This is what the `exec(2)` system call typically does.

ELF binaries have a field that indicates the ISA they use in their code. The kernel checks whether the binary’s ISA matches that of the kernel, and only allows those images that match to be executed. In the case of machines with support for multiple ISAs, such as the PowerPC 970, an additional module (binfof.elf32) must be provided to load binaries for those ISAs that do not match the kernel. I.e. the kernel is provided as a PowerPC 64-bit binary, but the processor is able to transparently run PowerPC 32-bit binaries too; this mismatch is what adds the necessity for an extra module.

Binaries for foreign PEs cannot be currently executed as regular programs even if they are distributed as ELF files. It is possible to resolve this issue by providing a binary-specific loader through the kernel’s binfof.misc functionality, but this is a hack: this module is provided with the idea to support different binary formats, not different ISAs.

Ideally the kernel could be intelligent enough to load any kind of ELF binary from an abstract point of view, check whether any PE can handle it and, if so, assign it to that PE.