Chapter 14

Development plan

This chapter describes the project's development plan. I have developed the project alone, so section 14.1 describes the steps I have taken alongside the development of the project and the amount of time it has taken.

However, to make the plan more real, section 14.2 details a complete development plan as if the project was developed by a fictitious software company. This involves multiple workers (instead of me), a set of well-defined development phases and the actual monetary costs for the project.

14.1 Personal plan

The PFC is a subject that takes 37.5 credits. It is expected to take around 600 hours which, if distributed as 8 work hours per day – not counting weekends – translates into 15 weeks (almost 4 months) of continuous work.

I did not write any development plan at the beginning of the project. In fact, it could have been very difficult for me to do it at that time and the results could have been highly inaccurate given that I was not aware of all the work that it involved. This is because:

- This is a research project. As such, the steps needed to reach the goals are not known beforehand: one has to keep studying how things work and dealing with unexpected events all the time.
1. Learned the basics of the PowerPC platform. This was a requirement because the Linux kernel we were going to modify was targeted to this architecture. This point involved reading several tutorials and took almost two weeks.

2. Learned the basics of the Cell architecture. This is the architecture we chose to do our work on, so I had to understand its main components and their relationships. This involved setting up the Cell SDK, learning how the IBM SystemSim worked, reading several tutorials, some manuals and implementing some sample programs. This took two weeks approximately.

Furthermore, I subscribed myself to the cbe-oss-dev mailing list\(^1\) to get involved into the open discussions about Cell development and to be informed of changes proposed by the developers. This required me to spend some time every day inspecting the mails I received, replying to some and asking some questions on my own.

3. Learned the basics of the Git and Cogito tools in order to be able to fetch the Linux development sources and maintain my own tree with custom modifications.

4. Set up a development system in the PlayStation 3. Initially we installed YellowDog Linux 5 plus the Cell SDK 2.0 and soon after we replaced the system with a Fedora Core 6 with the Cell SDK 2.1. This was done in two separate steps of the schedule.

5. Learned how to build a Linux kernel by using the development sources, both for the IBM SystemSim and the PlayStation 3.

6. Studied the kernel interface to manage the SPEs (the spufs).

7. Studied the differences between the legacy libspe 1.x and the new libspe 2.x. Also analyzed and described the internal execution flows of the latter and its interactions with the kernel.

8. Fixed some bugs that were encountered during the code analysis of the libspe.

9. Implemented the asynchronous SPE execution routines on top of the most recent versions of the Linux kernel.

\(^1\)https://ozlabs.org/mailman/listinfo/cbe-oss-dev
20. Wrote the project’s report. This task overlapped with many others.

<table>
<thead>
<tr>
<th>Task</th>
<th>Time spent</th>
<th>Hours per day</th>
<th>Total hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 weeks</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2 weeks</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>4 days</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>3 days</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>3 days</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>2 weeks</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>2 weeks</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>3 days</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>2 weeks</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>1 week</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>1 week</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>1 week</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>13</td>
<td>2 months</td>
<td>0.5</td>
<td>20</td>
</tr>
<tr>
<td>14</td>
<td>1 week</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>1 week</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>16</td>
<td>1 week</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>17</td>
<td>3 days</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>18</td>
<td>1 week</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>19</td>
<td>1 week</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>20</td>
<td>1 month</td>
<td>8</td>
<td>160</td>
</tr>
</tbody>
</table>

| Total | 5 months | N/A | 632 |

Table 14.1: Project summary.

At last, supposing I had a standard rate of 40,00 €/hr and an overtime rate of 50,00 €/hr, the total cost of the project could have been: \(40 \times 600 + 50 \times 32 = 25,600,00\) €.

14.2 Plan made by a fictitious software company

This section provides a simulation of the project’s cost as if it had been done by a software company. I have considered a standard iterative design model with two iterations, and have separated the project’s development plan into multiple phases.

I also take into account that there are several roles within the company, each one with its own rate and tasks. Given that each worker focuses on different parts of the project, he has to communicate with other workers that have different roles. This communication is done at the beginning and end of each phase, so it involves writing documents and reading them.
generate a document describing the problems they have found when realizing the abstract model in the Linux code.

As happens with the analysts, the company has two programmers. Their focus is different:

**Programmer 1 (P1)** Implements Project 1.

**Programmer 2 (P2)** Implements Project 2.

**Tester (T)** The company has a single tester. He is in charge of ensuring that the code remains stable during the development and that the new features work as expected. He also checks that the code remains API and Application Binary Interface (ABI) compatible with the code before the changes.

I also consider the **Client (C)** in the project plan, as he interacts with the developers at the end of each iteration for evaluation of the work done. Note that, however, I do not take him into account when calculating costs because his work hours are not paid by the software company.

Table 14.2 shows the salaries each worker has. In order to calculate the total monetary cost and for the sake of simplicity, I assume that there has been no deviation from the project’s baseline plan, so we have no need to apply any overtime rates to the workers.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Standard rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manager</td>
<td>60,00 €/hr</td>
</tr>
<tr>
<td>Analyst 1</td>
<td>50,00 €/hr</td>
</tr>
<tr>
<td>Analyst 2</td>
<td>50,00 €/hr</td>
</tr>
<tr>
<td>Programmer 1</td>
<td>30,00 €/hr</td>
</tr>
<tr>
<td>Programmer 2</td>
<td>30,00 €/hr</td>
</tr>
<tr>
<td>Tester</td>
<td>40,00 €/hr</td>
</tr>
</tbody>
</table>

Table 14.2: Salaries for the company’s workers.

14.2.3 Development phases

The following list details all the development phases involved in the project. These are quite generic in the sense that form part of any software development project. As regards the design and programming methodology, I have supposed a traditional iterative process with two different iterations.
14.2. PLAN MADE BY A FICTITIOUS SOFTWARE COMPANY

14.2.4 Task list

After the inception, the manager develops a development plan that lists all the tasks to reach the final goal, the relationships among them, their expected length and assigns them to specific resources. This plan keeps changing during the whole development as it has to be adjusted to changes requested by the client or due to the impossibility of delivering the results on time. For simplicity reasons, I have assumed that the plan does not change.

Tables 14.3 and 14.4 show the complete list of tasks defined for the project. Each task is accompanied by an identifier, the expected amount of time it takes and the resources assigned to it. Tasks also have a start and end date, but these fields have been left out for simplicity; they can later be seen in the Gantt charts.

<table>
<thead>
<tr>
<th>ID</th>
<th>Task</th>
<th>Duration</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Inception</td>
<td>12 days</td>
<td>M;C</td>
</tr>
<tr>
<td>2</td>
<td>Gather requirements from client</td>
<td>1 week</td>
<td>M;C</td>
</tr>
<tr>
<td>3</td>
<td>Write projects descriptions</td>
<td>1 week</td>
<td>A1;A2</td>
</tr>
<tr>
<td>4</td>
<td>Discuss proposals with client</td>
<td>2 days</td>
<td>M;C</td>
</tr>
<tr>
<td>5</td>
<td>Analysis</td>
<td>23 days</td>
<td>A1;A2;P1;P2;T</td>
</tr>
<tr>
<td>6</td>
<td>Learn PowerPC architecture basics</td>
<td>1 week</td>
<td>A1;A2;P1;P2;T</td>
</tr>
<tr>
<td>7</td>
<td>Learn Cell architecture basics</td>
<td>1 week</td>
<td>A1;A2;P1;P2;T</td>
</tr>
<tr>
<td>8</td>
<td>Study and document how CPUs are represented</td>
<td>1 week</td>
<td>A1;A2;P1;P2;T</td>
</tr>
<tr>
<td>9</td>
<td>Study and document the spufs interface</td>
<td>1 week</td>
<td>A2</td>
</tr>
<tr>
<td>10</td>
<td>Brainstorm characteristics of threads</td>
<td>2 days</td>
<td>A1;A2</td>
</tr>
<tr>
<td>11</td>
<td>Brainstorm characteristics of PEs</td>
<td>2 days</td>
<td>A1;A2</td>
</tr>
<tr>
<td>12</td>
<td>Set up development environment</td>
<td>7 days</td>
<td>A1;A2;P1;P2;T</td>
</tr>
<tr>
<td>13</td>
<td>Install Fedora Core 6</td>
<td>1 day</td>
<td>A1;A2;P1;P2;T</td>
</tr>
<tr>
<td>14</td>
<td>Install Cell SDK 2.1</td>
<td>1 day</td>
<td>A1;A2;P1;P2;T</td>
</tr>
<tr>
<td>15</td>
<td>Learn how Cogito works</td>
<td>5 days</td>
<td>A1;A2;P1;P2;T</td>
</tr>
<tr>
<td>16</td>
<td>Learn how the IBM SystemSim works</td>
<td>2 days</td>
<td>P1;P2;T</td>
</tr>
<tr>
<td>17</td>
<td>Implement some sample Cell Programs</td>
<td>3 days</td>
<td>P1;P2;T</td>
</tr>
<tr>
<td>18</td>
<td>Design (1st iteration)</td>
<td>13 days</td>
<td>A1;A2;P1;P2;T</td>
</tr>
<tr>
<td>19</td>
<td>Study the documents created during the analysis</td>
<td>3 days</td>
<td>A1;A2;P1;P2;T</td>
</tr>
<tr>
<td>20</td>
<td>Create abstract model for Project 1</td>
<td>1 week</td>
<td>A1</td>
</tr>
<tr>
<td>21</td>
<td>Create abstract model for Project 2</td>
<td>1 week</td>
<td>A2</td>
</tr>
<tr>
<td>22</td>
<td>Implementation (1st iteration)</td>
<td>8.89 days</td>
<td>P1</td>
</tr>
<tr>
<td>23</td>
<td>Implement model for Project 1</td>
<td>1 week</td>
<td>P1</td>
</tr>
<tr>
<td>24</td>
<td>Implement interface for Project 2</td>
<td>1 week</td>
<td>P2</td>
</tr>
<tr>
<td>25</td>
<td>Document problems during realization</td>
<td>1 day</td>
<td>P1;P2</td>
</tr>
</tbody>
</table>

Table 14.3: Task list, part 1 of 2.
14.2. PLAN MADE BY A FICTITIOUS SOFTWARE COMPANY

Each bar in the diagram is annotated on the left with their task identifier and on the right with the list of resources assigned to that task.

14.2.6 Monetary cost

At last, I calculate an approximation of the project’s total cost. This is done based on the worker’s salary rates (shown in table 14.2), their assignments to specific tasks and the length of these. Table 14.5 summarizes the cost of each development phase and sums up the total for the project.

<table>
<thead>
<tr>
<th>Task group</th>
<th>Monetary cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>5.920,00 €</td>
</tr>
<tr>
<td>Analysis</td>
<td>32.840,00 €</td>
</tr>
<tr>
<td>Design (1st iteration)</td>
<td>8.800,00 €</td>
</tr>
<tr>
<td>Implementation (1st iteration)</td>
<td>2.880,00 €</td>
</tr>
<tr>
<td>Preliminary assessment</td>
<td>1.560,00 €</td>
</tr>
<tr>
<td>Design (2nd iteration)</td>
<td>4.000,00 €</td>
</tr>
<tr>
<td>Implementation (2nd iteration)</td>
<td>2.400,00 €</td>
</tr>
<tr>
<td>Testing</td>
<td>14.080,00 €</td>
</tr>
<tr>
<td>Final assessment and delivery</td>
<td>3.520,00 €</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>76.000,00 €</strong></td>
</tr>
</tbody>
</table>

Table 14.5: Monetary cost separated per development phases.
Figure 14.2: Gantt chart, detailed view 1.
Chapter 15

Future work

This chapter describes several system areas which deserve further research. Most of these ideas were born during the development of the project, specially when analyzing the current state of the art. Others have appeared as a result of our own work.

15.1 Extend usage of the PE system

In chapters 5, 6 and 7, we implemented an API to generalize the way PEs are represented. This API is currently in use by some areas of the kernel, but many subsystems have not yet been converted. A logical future step would be to analyze the requirements of those subsystems and rewrite them to use the PE system, being able to keep track of all PEs in the system or simply to be able to obsolete the old CPU-management API.

A specific example of this could be to have a single process scheduler in the system that was able to schedule threads for any kind of PE. This would require the generalization of tasks detailed in section 15.2.

15.2 Generalization of tasks

As described in section 8.1.1, Linux's struct task_struct is only able to represent threads running on the CPUs in the system. However, it cannot currently keep track of those threads running on more generic PEs.
machine load meter among them. It could be interesting to analyze this fact and come up with a way that calculates the machine load on heterogeneous multiprocessor system.

15.6 Scheduling policies

Chapter 9 has outlined many different ideas to improve the process scheduling on heterogeneous multiprocessor system. Unfortunately, many of them have not been implemented and evaluated on a real system, so we cannot predict how well they would behave. All these ideas should be further inspected, implemented and tested to obtain some empirical results of their utility.
Chapter 16

Conclusions

During the development of this project, we have studied how recent heterogeneous multiprocessor system are managed by current operating systems. In special, we have analyzed how the Linux operating system manages the Cell processor.

Based on what we found, we proposed changes in three different areas and implemented them to see how they behaved in the real world. The first of these areas was a redesign of how PEs are currently represented inside the kernel because, before of our work, there was no centralized and abstract representation of them. The second one was a naive attempt to reduce the overall system load by lowering the amount of PPE threads to manage SPE tasks. The third was the modification of current scheduling policies to evaluate how a different scheduler behaved under heavy system load.

There is a room for improvement in the current design of all three areas and, specially, in the implementation. However, we did as well as we could in the given time frame of less than five months, which also included a long study period.

The study we made and the changes we proposed have shown us that there is still a lot of work to do to achieve a smooth integration of foreign PEs in a traditional system. We have only scratched the surface of this problem, and future research work – outlined in chapter 15 – should focus on continuing this work.
16.2 Personal comments

Back in January when I chose to work on this project, I barely knew anything about heterogeneous multiprocessor systems and the Cell processor architecture. Since then, I have spent a lot of time reading documentation and code, which has taught me a lot in this area and, still, I am a complete novice. I do not regret this at all, since this is a very interesting topic and it opens many different research possibilities in the near future.

Furthermore, I have achieved my personal goals of learning about Linux kernel development and the Cell processor. On the first hand, I have understood how the Git tool works and how it is used to maintain independent source trees for different Linux-based projects. I have also made my way through some of the Linux code, learning about its internal structure and coding practices.

On the other hand, I have learned how to work with the Cell processor at the programmer level. Thanks to this, I have been able to send multiple bug reports – shown in chapter 12, accompanied by patches, to the mainstream developers. Some of them have been already picked up and integrated into the source code, which means that these fixes will be part of future versions of their software.

I think it would be interesting to get the PE system commented and reviewed by the main Linux kernel developers and the maintainers of the Cell code. Doing so would raise a lot of ideas to improve the current APIs and maybe, eventually, get the code integrated into the mainstream kernel sources.
Appendix A

Fedora Core for Cell development

Fedora Core\(^1\) is a completely free, open-source based GNU/Linux distribution sponsored by Red Hat\(^2\). It provides the necessary packages to set up a machine that ranges from a typical desktop system with Internet communication abilities to a network server, passing through development and document publishing tasks. In some sense it is the free version of the commercial Red Hat system.

Fedora Core is available for several different platforms\(^3\) and, as outlined above, provides all the necessary tools to work on software development. It is the primary distribution in which Cell development happens. This chapter explains what the Cell SDK is, its relation to Fedora Core and how to install a functional Fedora Core system on the PlayStation 3, ready to aid in Cell development.

A.1 The Cell SDK

The Cell SDK is a set of utilities to develop software for the Cell platform, either directly on the native platform\(^4\) or on non-native ones. The kit includes:

**GNU GCC for the PPEs** A free C/C++ compiler to generate native code for the PPEs. If the SDK is installed on a non-native platform, a cross-compiler is provided instead.

---

\(^1\)http://fedoraproject.org/
\(^2\)http://www.redhat.com/
\(^3\)At the time of this writing, Fedora Core 6 was available for Intel i386, x64 and PowerPC machines.
\(^4\)The *native platform* refers to a Cell-based machine such as the IBM Cell Blade or the Sony PlayStation 3.
A.2. INSTALLING FEDORA CORE ON THE PLAYSTATION 3

# yum install freeglut freeglut-devel netpbm netpbm-devel \
   libICE libSM qt libX11-devel libXext-devel \
   libXfixes-devel libXmu-devel mesa-libGL-devel \
   mesa-libGLU-devel rsync perl

Then install the SDK itself by using the provided script. You will need to have the Cell SDK 2.1 ISO image (CellSDK21.iso) on your hard disk:

# mount -o loop CellSDK21.iso /mnt/cdrom
# cd /mnt/cdrom/software
# ./cellsdk install --nosim
# umount /mnt/cdrom

A.1.2 Building the examples

To ensure that the Cell SDK works fine, build the example programs provided by it. Some of these examples may require additional libraries that are not yet installed on your system. If they are not present, the build will fail; in that case, just install the prerequisites and relaunch the build process:

# opt/ibm/cell-sdk/prototype/cellsdk build --gcc

Note that this takes a long while on the PlayStation 3.

A.2 Installing Fedora Core on the PlayStation 3

The PlayStation 3 is a gaming machine that allows the execution of a Linux system out of the box, without any illegal hacking required. The main goal of installing Linux on this machine is to have a relatively cheap computer on which to do Cell development, because it is powered by one of these powerful processors.

---

5http://alphaworks.ibm.com/tech/cellsw/download
6This is in contrast to the PlayStation 2, which could also run Linux but required the physical modification of the machine and the reverse-engineering of hardware specifications.
A.2. INSTALLING FEDORA CORE ON THE PLAYSTATION 3

Once the firmware is up to date, reboot the machine and partition the hard disk. To do this, go to Settings, System Settings and open the Format Utility. Once there, select the Format Hard Disk option and confirm this operation twice. Later on, ask for a Custom partitioning scheme and make your selection; we used Allot 10GB to the PS3 System to leave plenty of space (around 50 GB) for GNU/Linux as the machine is not used for gaming.

At last, we have to install the OtherOS boot loader, which is a tricky step. First of all, grab your USB flash memory and hook it to a PC. Once there, create a PS3 folder in its root, followed by a otheros folder inside it, which will contain the otheros.self\(^7\) and otheros.bld\(^8\) files.

After configuring your USB flash memory, connect it to the PlayStation 3 and go to Settings, System Settings, Install Other OS. The system should find the files placed in the flash disk and offer to install them; confirm the operation. At the end of the process, the system will ask you to hit the X button to start the installer; do so.

At this point we are ready to start the installation of Fedora Core 6.

Keep in mind, though, that the machine is still configured to boot straight into the Game OS by default. Once you have completed Fedora Core’s installation, reboot the machine into the Game OS and go to Settings, System Settings, Default System. There, select Other OS to make the GNU/Linux system be the one to start by default on all further boots.

A.2.2 Starting the installation

To install Fedora Core 6 on the PlayStation 3, we need to have two disks at hand:

- The Fedora Core 6 installation DVD\(^9\). Installation from independent CDs will not work. Be sure to burn this at the lowest possible speed or the machine can have problems reading it.

- The add-ons CD\(^10\), a disk that includes several components required to make the system work on the machine.

---
\(^7\)http://www.playstation.com/ps3-openplatform/index.html
\(^8\)http://www.kernel.org/pub/linux/kernel/people/geoff/cell/ps3-linux-distro-kit-20061208/kboot/otheros.bld
\(^9\)ftp://ftp.rediris.es/mirror/fedora.redhat6/ppc/iso/FC-6-ppc-DVD.iso
A.3 The Yum package manager

Yum, the Yellowdog Updater Modified, is an interactive, automated update program which is used to maintain systems using the RPM packaging system. Fedora Core is one of such systems, and thus uses Yum as the primary tool to install new software and painlessly maintain the installed packages up to date.

Yum is built on top of the rpm utility. The latter provides a low-level way to interact with the packaging system, allowing the installation, upgrading and removal of individual packages. The former makes all these tasks easier by combining installations and updates of complex software sets, tracking dependencies as needed. Let’s now see the most common operations performed with Yum.

In order to install new packages, use the following syntax:

```bash
# yum install pkg1 [.. pkgN]
```

And to upgrade the entire system with the latest versions of all available packages, simply do:

```bash
# yum update
```

It is worth to note that the above two commands will connect to remote servers to download the most up-to-date packages, assuming we have configured the tool to do that. Therefore, this is the easiest way to maintain the software running on a Fedora Core system.

A.4 Initial system configuration

After booting the minimum Fedora Core system installed in section A.2, we will find that it lacks a lot of functionality, including the most basic utilities and development tools. Let’s log in as root to get the remaining bits set up.
A.4.3 Yum configuration

As mentioned earlier, Yum typically uses remote repositories to fetch software packages. However, after the minimal set up it is configured to use the installation DVD as the main and only repository. This causes some problems and prevents us from benefiting from the vast amount of software available in the on-line repositories. To enable these, follow the following instructions.

Edit the /etc/yum.repos.d/fedora-core.repo file and comment out the lines that look like baseurl=file:///mnt/cdrom/. Then, uncomment all other baseurl lines that were commented. This effectively turns on the on-line repositories and disables the use of the installation DVD.

Edit the /etc/yum.repos.d/fedora-extras.repo file and replace the first occurrence of enabled=0 by enabled=1. This enables the Fedora Extras repositories, which include a lot of software that is not yet part of the Fedora Core ones\(^\text{12}\).

To conclude, clean all Yum cache information by doing:

```
# yum clean all
```

A.4.4 Build a new kernel

The add-ons CD installs a 2.6.16 kernel specifically built for the PlayStation 3 because the one included in the official Fedora Core 6 does not work in this machine. However, this is such an old version that causes some incompatibilities with HAL, a daemon required by many other programs that we will be installing later on.

To resolve this problem, you will need to build and install a new, updated kernel at this point. As the machine is not yet configured, you must build this kernel in another machine, using the cross-compiler shipped with the Cell SDK\(^\text{13}\). See the appendix C for more details on how to do this.

Once installed and enabled in kboot’s configuration file, reboot to use it.

---

\(^{12}\)Starting with Fedora 7, Fedora Extras has been merged into the main repositories and the Core name has been dropped from the distribution.

\(^{13}\)You can also try to build it on the machine by installing the GCC compiler and related utilities at this point, but this can be complex.
A.4. INITIAL SYSTEM CONFIGURATION

Follow this by installing some development utilities, required to build native software for the
PPUs:

```
# yum install make gcc g++ flex bison byacc m4 glibc-devel
```

At last, if you want to build a Linux kernel natively on the machine, add some more packages:

```
# yum install git cogito ncurses-devel rpm-build
```

Note that, so far, we have not installed any program that requires a graphical environment,
nor we have installed anything but a minimum set of tools. Therefore, the machine will boot
very quickly.

A.4.7 Add users

Now that the system is functional, proceed to add any necessary users and groups.
Appendix B

The GIT and Cogito tools

As almost any open source project, Linux development sources\(^1\) are public. They are made available through a Source Code Management (SCM) tool which allows cooperative and distributed development.

More specifically, Linux uses Git\(^2\) as its SCM tool. This utility was started by the Linux developers because none of the other free SCM systems available at the moment suited their needs, and they wanted to move away from the proprietary BitKeeper\(^3\). The key ideas behind Git are:

- Fast operation on large trees: The utility was designed with the Linux source tree in mind, which is very large. Therefore, the most common operations on it run as quickly as possible.

- Distributed development: There are lots of people working on the Linux project all around the world. Ideally, each of these could keep a custom managed tree in their machine, and only push back their changes to the main development sources after approval by the maintainers. Git makes this possible.

- Support for non-linear development: The main development sources are not suited for big code refactoring tasks nor the addition of new features that need to touch a lot of code.

\(^1\)With development sources we refer to those changes to the code tree that are not yet part of any public release; that is, the code that is under heavy development in preparation of the next public version.

\(^2\)http://git.or.cz/

\(^3\)http://www.bitkeeper.com/
After that, start inspecting the `git(7)` and `cogito(7)` manual pages to get a first impression on the functionality of the two programs.

### B.2 Branches

As already mentioned, Git is a distributed version control system. In order to achieve the distributed functionality goal, it makes heavy use of branches, so it is important to understand how they work before being able to use the system.

A branch is a set of related revisions that generally have something in common. Branches are used to work aside from the main development sources, to allow you try out different experiments without disturbing the stable code.

![Branch Diagram](image)

**Legend:**

- R:n - Revision n
- B:t - Branch t

**Figure B.1:** An example of branches in a repository.

The revision graph in figure B.1 shows an example of a repository with three different branches. There is a `master` branch which represents the principal development sources. After revision 3, someone decides to fork the development by creating another branch labelled as `test`; this developer will be able to work on his own set of revisions without distorting anyone using `master`. Once he has finished his work at revision 7, he merges his branch with the principal
Figure B.2: An example of the relationships between branches across repositories.

$ cg clone git://git.example.net/projects/foo foo-master
$ cg clone git://git.example.net/projects/foo#test foo-test

After the above two commands, we have created two new local repositories in the directories we explicitly specified: foo-master and foo-test. Each of these has a origin branch (the name used by default) that points to the remote branch we specified for each repository.

If instead we want to keep the two remote branches into the same local repository, we have to first add a new remote branch to our repository and then use the fetch command to fetch it. It could go like this:

$ cg clone git://git.example.net/projects/foo foo
$ cd foo
$ cg branch-add remote-test git://git.example.net/projects/foo#test
$ cg fetch remote-test

B.3 Checking out a tree

Checking out a remote repository basically means creating a complete local copy of it. This is also known as cloning the repository. The syntax to do this is shown below, and has been detailed in B.2:

$ cg clone <remote-address> [local-directory]
Appendix C

Building a custom kernel for the Cell

The Linux kernel has support for Cell machines. It includes all the necessary features to interact with the SPUs and manage their resources. This port is based on the PowerPC one, as the two machine types share a lot of implementation details\(^1\).

This annex describes how to build and install a working kernel for the PlayStation 3 and the IBM SystemSim for Cell.

C.1 Fetching the sources

The official Linux sources have support for the Cell processor, but they only work on the IBM Cell Blades. Using these sources will result in a kernel that is not bootable neither on the PlayStation 3 nor on the IBM SystemSim for Cell, which also means that there are no up-to-date binary images available for these two machines.

Fortunately, there are some public development branches in the Git repositories\(^2\) that do support these two machines. These are:

```
linux/kernel/git/arnd/cell-2.6.git This branch is where generic Cell development happens.
```

Here is where improvements to the SPU management and other stuff that is specific to the Cell go before reaching the mainstream sources.

\(^1\)Basically, a Cell machine can be seen as a PowerPC one with some additional components (the SPUs).

\(^2\)http://git.kernel.org/
C.3. GENERIC BUILD PROCESS

C.3 Generic build process

The generic process of building a Linux kernel consists of:

1. Creating a configuration file for the target machine. We will use the example configuration files distributed along the kernel for simplicity.

2. Building the sources. This is as easy as typing make at the top-level directory of the source tree once it is configured.

3. Installing the resulting kernel image and modules. This step depends on the target machine.

C.4 PlayStation 3 details

C.4.1 Configuration

The ps3-linux tree comes with a sample configuration for the PlayStation 3 that is fairly accurate and complete. We will use it unmodified to build our own kernel.

In order to configure the sources to use this pre-configured file, do:

$ make ps3_defconfig

The above command will leave a .config configuration file in the current directory, ready to be used by the build framework.

C.4.2 Installation

Once the kernel is built, we must install it. The first step consists of copying the binary image to the /boot directory and installing the modules, so simply do:

$ cp vmlinux /boot/vmlinux-custom
$ make modules_install

Once this is done, add the following line to /etc/kboot.conf so that the kboot boot loader knows which kernel to execute:
to run the simulator. And the good thing is: as we did not use modules, we need not take any other extra step to get it to work.

If it doesn't work, you can edit the configuration file, look for the line that sets the `imagefile` variable and change it to directly point to your kernel image. For example:

```
set imagefile /path/to/my/kernel
```
Bibliography


Glossary

ABI The Application Binary Interface is the set of rules that define the conventions to transform a high-level language into a binary program and how this binary program interacts with external binary modules. 137

API The Application Programming Interface is the specific interface (function signatures, data types, etc.) exposed by a program or library to external source code so that the two can interact. 36, 44, 50, 52, 66, 97, 101, 102, 106, 108, 111, 122, 134, 136, 137, 145, 150, 151

ASMP Asymmetric MultiProcessing is a multiprocessor machine architecture in which each PE can have access to different memory sets. Not all PEs in the system share the same view of the available memory, nor can access them all. 7

BSC The Barcelona Supercomputing Center is a research center, located in Barcelona, Spain, that works on supercomputation. Its mission is to investigate, develop and manage information technology in order to facilitate scientific progress. 20

Cell SDK The development kit for the Cell platform that provides compilers and related tools to easily write programs that use both the PPEs and the SPEs in the architecture. 20, 40, 90, 95, 115, 121, 125, 126, 133, 150, 155, 155, 156, 157, 163, 174

Cell The Cell Broadband Engine is an advanced microprocessor designed by IBM, Toshiba and Sony that bundles a general-purpose PowerPC CPU and several calculus-only coprocessors known as the SPEs. 4, 10, 15, 16, 16, 17–20, 28, 31, 39, 51, 67, 84, 95, 105, 112–115, 119, 125–127, 132–134, 136, 149–151, 155–157, 173

CPU scheduler The algorithm that schedules threads for the system CPUs. 75
GLOSSARY

I/O Input/Output refers to the devices connected to a machine that permit interaction with
the user or other non-PE devices. xv, 3, 5, 11, 30, 31, 40, 84, 86, 88, 89, 95, 111

IBM Cell Blade A high-performance machine made by IBM that bundles two Cell processors
in a single board. 19, 20, 112, 155, 173

IBM SystemSim for Cell An application included in the Cell SDK that simulates a Cell
machine and allows the execution and monitoring of PPE and SPE applications in a
controlled environment. 19, 20, 156, 173, 174, 176

Intel i386 A 32-bit microprocessor architecture that evolved from the first 8-bit chips designed
by Intel. This is the most widely used PC architecture nowadays, even though it is quite
limited and too complex due to all the compatibility features it carries. 5, 15, 25, 26, 28,
30, 54, 155

interrupt An asynchronous notification raised by the hardware and captured by a PE. 25, 31

ISA The set of rules that define which instructions a PE supports and how they are encoded
to form a binary program. 25, 25, 26, 41, 55, 56, 58, 145

ISR The kernel-level routine that is executed as the result of the reception of an interrupt. 31

libspe The standard high-level library, included in the Cell SDK, to interact with and manage
the SPEs. 95, 97, 101, 106, 109, 111, 120–122, 125–128, 133, 146, 150, 156

Linux A free, Unix-like operating system kernel with support for multiple platforms. 9, 10,
19, 20, 23, 24, 28, 31, 32, 34, 36, 37, 39, 49, 51, 52, 56, 63, 65, 75–78, 84, 95, 103, 106,

LS The block of local memory found inside each SPE. 18, 18, 27, 28, 86

machine description The set of information and properties that describe all the PEs found
in a machine, their relationships and their capabilities. 44

main memory The memory accessible from all CPUs, and usually from all PEs in case of an
heterogeneous multiprocessor system. 3, 6, 7, 11, 12, 16, 18, 26–28, 86, 96
percpu A Linux kernel API used to define data fields that are specific to each possible CPU in the system. 34–36, 49, 50, 65

PID The Process IDentifier is a number that identifies a thread inside the Linux kernel or a process in user space. 76

PlayStation 3 The third generation of Sony’s gaming machine. This is equipped with a Cell processor. 18–20, 37, 85, 86, 89, 113, 115, 119, 121, 125–127, 133, 134, 150, 155–160, 163, 173–175

POSIX A standard that defines an API to manage system resources; usually found in Unix-like systems. 76

PowerPC A microprocessor architecture designed by IBM currently used in research areas and many hardware devices due to its versatility and power. Most commonly found in embedded devices, high-end servers or appliances (such as gaming machines). 5, 12, 15–18, 23–26, 30, 41, 54, 66, 67, 132, 155, 173

PPE The PowerPC Processing Element is a general-purpose processors found in the Cell, compatible with the PowerPC 970. 10, 16, 17, 17, 29, 31, 39, 45, 46, 51, 67, 68, 70, 81, 84–86, 89, 101, 103–106, 109, 112–114, 149, 155, 156

PPU The PowerPC Processing Unit is the processing core found inside a PPE. 17, 17, 78, 164

process Run-time representation of a program. 11

program Sequence of instructions that perform a transformation of main memory contents or deals with I/O devices. 11

RISC The Reduced Instruction Set Computing is an instruction architecture that defines few types of machine-level instructions in order to make them as fast as possible. 15

SIMD A Single-Instruction Multiple-Data instruction is an operation that operates on multiple data values at once. 18, 26
spufs The SPU File System is the low-level Linux interface used to export the management of SPEs to user space. 95, 95, 96–98

stop and signal An SPU event explicitly caused by a machine-level SPU instruction, used to request a service to the PPE. 101, 113

TGID The Thread Group IDentifier is a Linux-specific identifier used to group all threads that belong to a single process. This is the identifier that is exported to user-land applications as the PID. 76

thread An execution flow within a process. 11

x64 An extension to the Intel i386 architecture that adds 64-bit addressing and instruction support. This was originally designed by AMD and implemented in its Athlon 64 chips, and was later adopted by Intel. 155