Vincent Danjean
École Normale Supérieure de Lyon
Magistère d'Informatique et Modélisation
Second year period

# Extending the Linux Kernel with Activations for Better Support of Multithreaded Programs and Integration in PM²

Internship made at the University of the New Hampshire
with Robert D. Russell and Philip J. Hatcher.
September 15, 1999

**Abstract**

Nowadays, cluster of SMP machines are used a lot to perform heavy parallel computations. The new concepts around multithreading have been prooved suitable for the SMP architecture. Generally, the programmer uses a thread library to write this kind of programs. Such a library schedules the threads or asks the OS to do it, but both of these approaches still have problems. We introduce here another approach which relies on cooperation between the OS scheduler and the user application using *activations* and *upcalls*. This approach has been presented in an article[2] written in 1989. We implemented this model with LINUX and adapted the thread library MARCEL (from the programming environment PM$^2$) to use the activations. The performance observed was improved: there was no loss in speed but the problems caused by the blocking system calls were removed.

**Key words:** thread, multiprocessor, operating system, Linux, PM$^2$, MARCEL, activation, scheduler.

# Contents

# Introduction

Nowadays, there is a great deal of interest in concurrent programming. Indeed, this kind of programming can improve the structure and performance of computer programs on both uniprocessor and multiprocessors systems.

Several possibilities for concurrent programming exist, but the most useful is threads. These threads can be managed either by the OS (*Operating System*) or by the application, but both of these approaches have some inconveniences.

In this internship, we propose another solution that solves the previous problems, but needs specific support from the OS. The main idea is to let the kernel (the OS) report all its scheduling decisions to the application, so that the application can correctly managed its threads.

We have implemented these improvements in the LINUX kernel, and have adapted a thread library to take advantage of this new service. There is no need to modify the code of the threads. They run perfectly with the new library, and can use the new functionalities. The results that we obtained are really good.

# Chapter 1

# Concurrent programming

## 1.1 Interests

### 1.1.1 History

With old operating systems (like DOS), there was no support for concurrent programming. All operations had to be handled by one program, so that the design and the execution of an intrinsically parallel program (a server for example) were sometimes difficult to understand.

Dijkstra [5] and Hoare [8][9] showed that these programs can be simplified if they are structured in different threads communicating at discrete points within the program. Thus, the programmer can consider its threads separately, and rely on the system to manage automatically the scheduling of its threads.

### 1.1.2 Uniprocessor and Multiprocessor

The possibility of concurrent programming on a uniprocessor leads mainly to a better design of the applications. They are easier to write and understand.

With multiprocessors, in particular SMP (Symmetric Multi Processor) with shared memory machines, this kind of programming allows the program to take advantage of all the available CPUs. Thus, adding CPUs to the computer increases the performance.

However, using parallelism has a cost. It can be found in the overhead, in terms of processor cycles, required for creating, controling and synchronizing the threads, and in the programming overhead due to the effort needed to write an efficient parallel program.

## 1.2 The classic solutions for the thread libraries

Nowadays, lots of thread libraries exist. With Linux, we can use, for example, the LinuxThread[11] library, the Marcel[13] library, the T-Thread library of the Tulip[1] project, etc. These libraries can be classified in two fundamental categories, even if some of them are hybrid and mix the two approaches.

### 1.2.1 Kernel Threads

In one approach, the threads can be directly managed by the OS. This situation has many advantages. The application can use several processors on a SMP machine because the kernel can assign

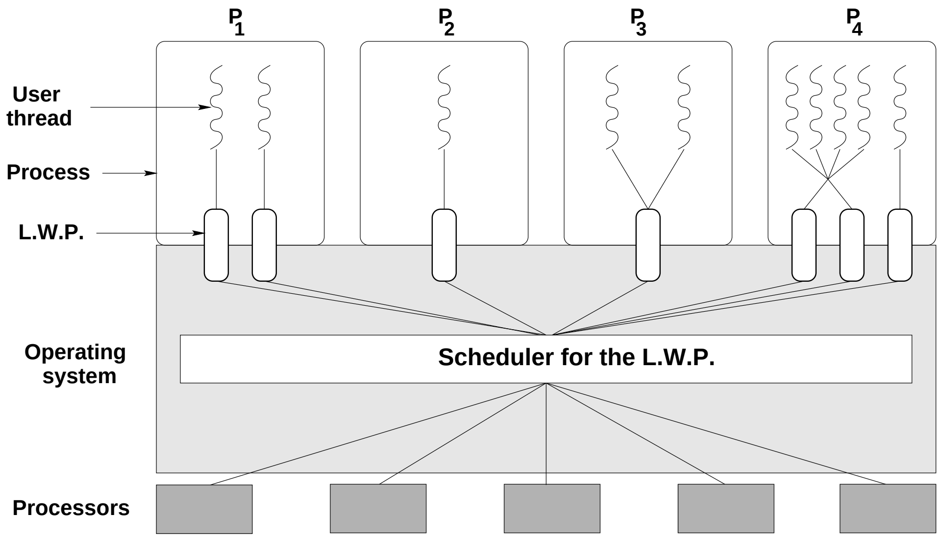



Figure 1.1: Different possibilities with SOLARIS.

different threads to each processor. When a thread makes a blocking system call (like a read on an empty socket, for example), the kernel can give control to another thread of the application.

The inconveniences of these threads are that they have only the properties allowed by the system. Often, projects like PM$^2$[12, 14], NEXUS[6] or ATHAPASCAN[7] need nonstandard properties for their threads, that cannot be added in such a system. The other objections that can be made to kernel threads are that they are often less efficient than user threads (because they need lots of kernel context swap, system calls, etc.), and they heavily use kernel resources (each as many as a process generally). So, with LINUX we cannot have more than about one thousand threads (shared between all the processes).

### 1.2.2 User Threads

In another technique, the threads of an application can be managed by the application itself. These user threads are often very efficient, do not use additional kernel resources, and can be tailored as we want. We can easily have thousands of threads **per** process.

Their problems are that the OS does not know anything about them. It does not even know that they exist or that the process is a multithreaded process. This has several consequences. The application cannot use several processors on a SMP machine, because the OS always gives only one processor to a monothreaded process. And when a thread makes a blocking system call, then all the threads of the process are blocked: the kernel is waiting for the end of the system call before giving control back to the application.

## 1.3 Hybrid thread libraries

To try to have the best of the two kinds of threads, some libraries mix them together: a few kernel threads are used to run user threads. Thus, they have the performance of the user threads, but are also able to take advantage of SMP machines. SOLARIS and MARCEL in PM$^2$ are two examples of such libraries.

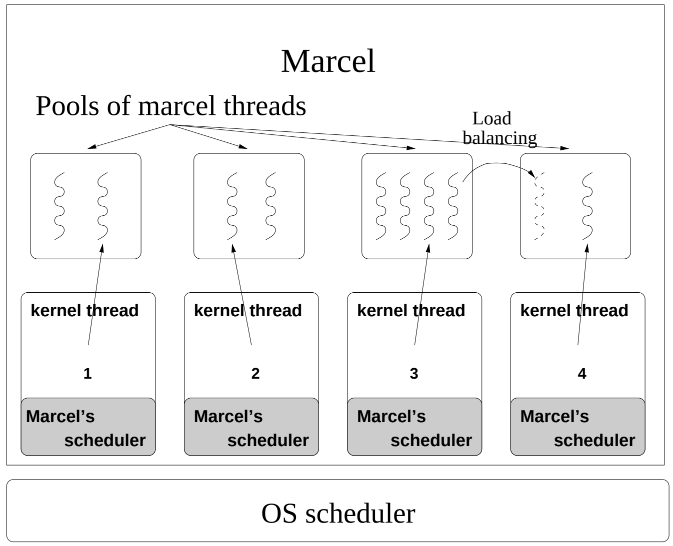



Figure 1.2: Each kernel thread manages a pool of MARCEL threads

### 1.3.1 SOLARIS

SOLARIS is an OS developed by SUN that currently works on sparc and i386 architectures. SOLARIS is able to mix the two kinds of threads: each process has some kernel threads called LWP (*Lightweight Process*) that in turn manage the user threads (if any) of the process. So, in figure 1.1, we can see :

- $P_1$ has 2 kernel threads;
- $P_2$ is a monothreaded process;
- $P_3$ has 2 user threads;
- $P_4$ shows the improvement allowed by SUN: there are two kernel threads that manage four user threads, and another kernel thread.

That way, we have good performance, but the user threads are not very flexible in the case of SOLARIS.

### 1.3.2 PM$^2$

PM$^2$ (*Parallel Multithreaded Machine*) is a distributed programming environment that relies on a specific thread library called MARCEL which allows nonstandard functionalities such as thread migration over a network.

MARCEL was at first a user level thread library, but is recently becoming a two level thread library to take advantage of SMP machines [4]. It works now with some kernel threads, each managing a pool of MARCEL threads as you can see in figure 1.2.

With this solution, MARCEL can manage lots of its specific MARCEL threads, and has always good performance.

### 1.3.3 Not really the good solution

As we can see, the use of a two-level thread library improves programs: better performance and efficient use of SMP machines. But there still are some problems not solved.

First of all, when a user (or MARCEL) thread makes a blocking system call, the LWP (or the kernel thread in MARCEL) is stopped too. So, with a few blocking user threads, we can block all the LWP or kernel threads, thereby block the whole application, even if some other user threads are ready to run.

Another problem is that, even if we can choose the schedule of the MARCEL threads in each pool, we cannot do anything between the different pools. So, if a thread A in pool 1 has a lock and is preempted by the system, when a thread B in another pool wants the lock, it has to wait for the OS to give control back to pool 1, so that thread A can release the lock.

As we will see, these problems can be avoided if the OS scheduler reports what it is doing to the application. This cooperation between the OS scheduler and the application has been developed in this internship and MARCEL has been modified to support it.

# Chapter 2

# A new solution: kernel support for activations

This idea was first proposed in an article[2] written in 1989. Its authors have implemented this mechanism with the FastThread library on the Topaz system which does not run any more, and the sources have never been released. All the terms (*activation*, *upcall*, etc.) used in this document come from this article.

## 2.1 Principles

### 2.1.1 The activations

Activations are a kind of kernel thread. The main difference is that when an application uses a classical kernel thread, it generally designates a function that the OS will execute within a thread. This is the opposite for an activation: this is the OS that decides when an activation is created, executing a specific user function.

The second point specific to the activation is the fact that each time an activation blocks in the kernel, unblocks, or is preempted by the kernel, then the application receives a report of this fact.

### 2.1.2 Upcalls

An upcall is a mechanism by which the kernel calls a user function. Generally, it is the application that makes calls into the kernel with the system calls. In fact, an upcall is very similar to the signal handler used by the kernel to tell an application about UNIX signals.

The upcalls are used by the kernel to report some scheduling events to the application. The different kinds of upcall are presented there:

**upcall new** This upcall is the first one made by an activation. Each time an activation is created, the kernel uses this upcall to report this event to the application, so that the application can use the activation and run the code it wants.

**upcall block** When an activation blocks (within a system call for example), the kernel uses another activation of the process and makes this upcall to report to the user process the fact that one of its activations blocked.

**upcall unblock** This upcall is similar to the previous, but is used to report to the application that one of its blocked activations unblocked. In this upcall, the kernel gives to the application the state of the unblocked activation, so that the application can resume the user thread that was running on that activation.

**upcall preempt** This time, we report to the application that one of its activations was preempted by the scheduler. Here again, the kernel gives the state of the preempted activation, so that the thread running on it can be resumed.

**upcall restart** This upcall does not report anything. But it can be used by the application to synchronize its activations: an activation can ask the kernel to make an upcall in another activation. If the kernel has nothing to report to the application (so that none of the previous upcalls is useful), then it will use this upcall.

### 2.1.3 Interesting properties

There are several very interesting points about activations. First, there cannot be more running activations than processors on the machine. Indeed, to create more activations than processors, the kernel must preempt at least one other activation.

The other point is what happens when the last activation is preempted or blocks. If it blocks, there is no problem: another activation is created, launched with a `new` upcall, and then a `block` upcall is made. If it is preempted, then nothing is done at first. Indeed, if the kernel does not want to give any processor to the application, then we cannot make any upcall. But this is not a problem: the next time the application will have a processor, either this activation will continue (as it had never stopped), or, if another activation is launched instead of the preempted activation, then the kernel will use the other activation to make an `preempt` upcall. This allows the other activation to continue the thread preempted (instead of starting a new one), which can be very useful if, for example, the preempted thread owns a mutex that prevents other threads from running.

After each upcall, the application can choose itself which user thread to run in the activation. So, the application has complete control about which user thread to run in which activation, and receives a report each time one of its threads stops.

### 2.1.4 Example

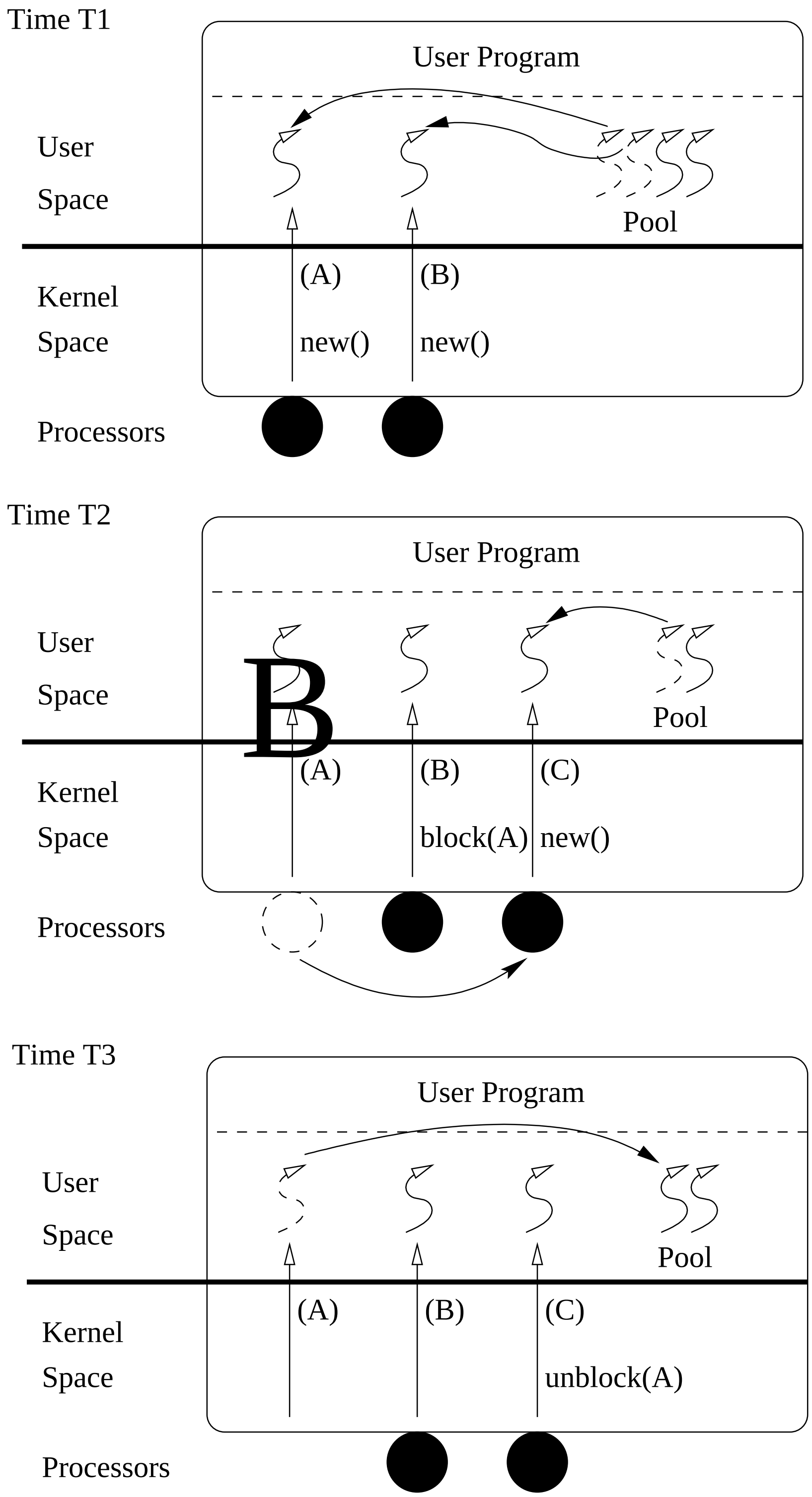
Let's see what happens on a dual processor machine when an application using activations makes a blocking I/O request. You can refer to the figure 2.1.

**At time T1** the kernel allocates the two processors to the application. Each activation A and B begins with a `new` upcall, and chooses a user thread to run.

**At time T2** the activation A makes a blocking I/O request. Hence, a `new` upcall is done and the activation C created. The kernel uses one of the activations B or C to make a `block` upcall, so that this application now knows that one of its threads is blocked.

**At time T3** the I/O request completes. The kernel again uses one of the two activations B or C to report to the application that the activation A unblocked. Because the `unblock` upcall has the state of the unblocked thread, the activation doing the upcall can choose to continue its thread, or to immediately resume the thread that was blocked.

Figure 2.1: A Blocking System Call with Activations



## 2.2 Interests

### 2.2.1 SMP compliance

The first good point about activations is that this model uses all the available CPUs, but not more. With kernel threads, you must have at least as many kernel threads as physical processors if you want to be able to use all of them. But, if some CPUs are used by other processes, then your kernel threads must share their CPU, which can lead to delays when we are waiting for a lock that is owned by another kernel thread for example.

### 2.2.2 Finite kernel resources

This model allows any user thread to block in the kernel without blocking the other user threads. This was already the case with the kernel threads, but with them, each user thread used kernel resources (the kernel had to know about each created thread). With activations, the kernel only knows about the running threads (no more than the number of processors) and the blocked threads.

### 2.2.3 Flexibility

All thread scheduling decisions are made by the user program. The OS does not have any constraint or restriction about the properties of the user threads. So, we have been able to use activations with the Marcel thread library which has some very unusual support, such as migration over the network.

# Chapter 3

# Implementation

The implementation of activations consists of two parts. At first, there is the support of activations in the OS. Secondly, we must have applications (or a thread library) that use these improvements.

## 3.1 In the OS

We have to choose an OS for this implementation. It has been LINUX because we have the sources of this OS, and lots of documentation in books [3, 15] or on the Internet exist about it.

The description which follows is just an overview of what has been done. For more precise documentation, the reader can refer to two annexes:

- Appendix A: *A New Application Programming Interface for Scheduler Activation Support in the Linux Kernel* which describes everything one needs to know to use activations in a user program;
- Appendix B: *Internals of Activation Support in the Linux Kernel* which describes the modifications made in the kernel, and the mechanisms used in the kernel.

### 3.1.1 Modifications in the LINUX kernel

The work has been done with version 2.2.10 of the LINUX kernel, which was the most up-to-date release of the stable LINUX kernel at the time of writing. However, the work (and the patch) can be easily ported to other versions of the kernel.

The main modifications consist of:

- add several new structures to handle activations' states;
- modify the `do_fork()` and `do_exit()` functions to be able to create and delete activations;
- modify the `schedule()` function to report an event each time an activation is preempted/blocked/unblocked;
- add some new system calls so that the application can request activations from the kernel;
- add the upcall mechanism for the activations.

### 3.1.2 Organization

The implementation uses one *task* or `kernel thread` (as it is named in the kernel) per activation. Each task uses some new private variables to save its current state as an activation. There are also variables shared between all the activations of a process.

### 3.1.3 The new system calls

Some new system calls are needed for the activations. There are four new system calls. For an accurate description of their parameters, see the appendix A.

**act_init** This system call is the first of these four that must be used if an application would like to use activations. With this system call, the application gives to the kernel some information needed, such as the address of the different functions for the upcalls, the number of activations we want (not necessarily as many as the number of processors), etc. Once the kernel has verified the validity of these parameters, it begins to use the activations.

**act_cntl** This system call is used by the application for two reasons:

- to request information about activations (like how many processors there are, how many activations are running);
- to modify some kernel variables related to the activations, such as the number of processors that the application wants to use.

**act_send** This system call is used by the application to request the kernel to make an upcall in one or several other activations. This is useful if an activation wants to stop a thread running on another activation for example.

**act_resume** This system call must be used after each upcall. It tells the kernel that another upcall can be made (there can be only one upcall running at a time).

### 3.1.4 The manager activation

This activation is a special activation. It never makes upcalls, but is used internally to manage the creation and the end of the other activations.

The manager activation is the *task* that was the UNIX process before the `act_init` system call was performed. All the other *tasks* of the process, which will be the normal activations, have this *task* as parent. Thus, only this one has to wait for the ending of the other *tasks*. (In LINUX, a *task* is necessarily informed about the ending of each of its children.)

The manager is also the *task* that owns the variables shared between all the activations of the process. This is useful because the manager exists from the beginning of the activation, and is ended when the process ends, so that there are no other activations running any more.

## 3.2 In MARCEL

After implementing the support for activations in the OS, it would be useful to have an application that uses it. We could have written a specific application, but it is better to adapt an existing thread library to the activations, and see what happens with the programs that use it.

Due to the good design of MARCEL, very few adaptations had to be made. MARCEL had a lock to protect itself from problems with reentrancy. This lock is used by the activations too. And if an activation owning the lock is preempted, then its thread is resumed as soon as possible.

# Chapter 4

# Validation

The adaptation of a existing library allows us to easily see the improvement due to the use of activations. There are actually two versions of MARCEL. One is a two-level thread library that we will call MARCEL-SMP, and the other is a single user thread library. The latter can be configurated to have automatic preemption (we will call it MARCEL-USER-PREEMPTION) or not (called MARCEL-USER). The MARCEL library with activation support will be called MARCEL-ACT. To have all the kinds of thread libraries, the PTHREAD library which is a kernel threads library, has been used too.

## 4.1 Activations with a uniprocessor machine

The first tests have been made on a uniprocessor machine: a Pentium II 266MHz with 96 MB of memory.

The first thing to say is that the samples of MARCEL still work with MARCEL-ACT. Then let's see more precisely some results.

### 4.1.1 The `sumtime2` program

In MARCEL, there is a sample program called `sumtime`. It is very useful because it is able to calculate the sum of the integers from 1 to a limit given in an argument to the program. What is really interesting is the fact that the process uses a lot of threads to calculate this sum. It splits recursively the sum into two parts and launches one thread for each of these two parts. Then these threads split again the sum they have to do and launch other threads. So, lots of threads are created and this program needs lots of synchronization between all its threads.

`sumtime2` is an adaptation of this program. It has been written so that it can run with all the versions of MARCEL and with the PTHREAD library. There is also another parameter for the program: we can choose that the threads have to execute a longer computation, so that the program spends time doing computation and not only synchronisation between threads.

**Without computation** In this case, the threads only launch two other threads, wait for their ending, and end themselves giving back the result. The results can be found in figure 4.1.

The first remark is that the PTHREAD library cannot handle a sum of more than about 250. With a greater value, there are too many threads and some of them cannot be created. The different versions of MARCEL can go until about 500. But whereas we can run simultaneously several versions of MARCEL that calculate until 500, we cannot do this with the PTHREAD library because in this case each thread uses a global resource (an entry in the task table of the OS).

Figure 4.1: Results of `Sumtime2` without calculation in each thread.

**pthrd** PThread library

**MaAct** Marcel-Act

**MaSMP** Marcel-SMP

**Ma** Marcel-User

**MaPre** Marcel-User-Preemption

Figure 4.2: Results of `Sumtime2` with calculation in each thread.

**pthrd** PThread library

**MaAct** Marcel-Act

**MaSMP** Marcel-SMP

**Ma** Marcel-User

**MaPre** Marcel-User-Preemption

The second remark is that, as we supposed, kernel threads are less efficient than user threads. The two-level library is a little less good than the other version of Marcel: there is more synchronisation to do (between the few kernel threads). Both the user level thread library and the one with activation support have similar good performance. Marcel-Act is perhaps a little slower than the user level Marcel libraries, due to the overhead needed to manage activations, but this is really not significant.

**With computation** Here, we had a computation in each thread, so that the synchronization becomes a smaller part of the execution of the threads. The results can be found in figure 4.2.

As we can see, on a uniprocessor machine, all the times are similar. However, Marcel-SMP and the PThread library are a little bit slower than the user level libraries and Marcel-Act.

### 4.1.2 Blocking system calls

A great interest of activations is the facility to correctly handle the blocking system calls within the user threads. Two programs show these results.

**The program `block`** This program launches two kind of threads. One performs a calculation and prints some information from time to time. The other blocks reading a file descriptor.

As we might suppose, with the PThread library and Marcel-Act, there is no problem. With Marcel-User, the program blocks immediately and the calculation threads do not do any computation. With Marcel-SMP and Marcel-User-Preemption, the program works. This is due to the fact that in these cases, Marcel uses signals to have automatic preemption. These signals interrupt the blocking system calls, so that the Marcel scheduler can launch another thread. Fortunately, the `read()` system call can be automatically resumed by the OS when the thread takes control back and returns from the signal handler.

**The program `semaphore`** This program is similar to the previous, but instead of having threads blocking when reading file descriptors, they block on a semaphore.

Marcel has implemented its own semaphores. But, they can be used only to synchronize the Marcel threads together, and not with other processes running on the machine. To do that, we need to use the semaphores of the OS. This works very well with the PThread library, because each thread is similar to a process from the point of view of the OS.

With the classic Marcel libraries (Marcel-User, Marcel-User-Preemption and Marcel-SMP), this does not work at all. The process makes a segmentation fault immediately. This happens because the kernel manages the IPC with processes or kernel threads. It can not handle the case when there are several user threads within the same process or kernel thread and each of these user threads uses the IPC calls.

However, with Marcel-Act, the program runs very well. There is no need to add additional support because this is directly managed by the activation. Each time a user thread blocks on a semaphore in the kernel, another activation is created, so that the other threads can run.

### 4.1.3 Summary

As we can see, the use of activations for a thread library on a uniprocessor is very useful. It keeps the performance of a user level thread library, but solves all the problems caused by blocking system calls.

## 4.2 Activations with a symmetric multiprocessor machine

With an SMP machine, a program using activations should be as fast as programs written with the PThread library or Marcel-SMP. There are still a few bugs that prevent us from using it with programs with a lot of synchronisation like the `sumtime2` program. These few bugs should be quickly fixed. However, we are able to have a few results with simple programs.

The first tests have been made with a program that launches four threads that perform some computation. The results can be seen in the figure 4.3. The machine used has two processors. As we could expect, the program computes nearly twice as long with Marcel-User and Marcel-User-Preemption which can use only one processor on the machine. The performance with the activations is as good as the PThread library or Marcel-SMP which use the two available processors.

Figure 4.3: Four threads doing computation on SMP.

| | time (s) |
|---|---|
| PThread library | 3.60 |
| Marcel-Act | 3.61 |
| Marcel-SMP | 3.62 |
| Marcel-User | 7.15 |
| Marcel-User-Preemption | 7.16 |

## Conclusion and future work

This work designed and tested the use of activations to manage threads. This approach is a completely new way to handle thread support for an OS. We had to write the support for the activations in the LINUX OS and to adapt a thread library to take advantage of this support.

Both have been written, so the tests could be made. We did not change the MARCEL interface, so that the old MARCEL programs still work with this new model. The performance on a uniprocessor is as good as the user library, but without the problems of the latter. With a multiprocessor, the first results seems to be good, even if there are still few bugs. These should be quickly removed so that we can see the performance with more complex programs.

The interesting things are that we can now use some features that were not possible before. The user threads can use blocking system calls like `read()` or even use semaphores. These calls are handled correctly and do not block the other threads. And the thread library is a user one: we do not need to change the kernel to add specific features to our threads.

Several improvements to the activations' support in the kernel can be made now. They are not already done, because the time was too short, or because other required parts are not yet ready. We can list some of them.

- add support for POSIX signals. For now, the POSIX signals are not usable with the activations. A support as written in the appendix *A New Application Programming Interface for Scheduler Activation Support in the Linux Kernel* can be added.

- add support for automatic preemption. Until now, support for automatic preemption in MARCEL was done with the `SIGALRM` signal. So, to handle automatic preemption in MARCEL with activations, we can either support signals, or add a new upcall, which will be probably more efficient.

- add support for other architectures that support LINUX. Indeed, the i386 architecture is the only one that can use the activation support. This can be improved because both MARCEL and LINUX can run on other architectures.

- use new kernel features. In an upcoming LINUX kernel release, there will be support to allow a *task* to create a new *task* as if yet another *task* was the parent. This feature will allow an activation to create itself new activations when needed, instead of having to wake up the manager activation.

- use shared locks between the user space and the kernel. If the kernel directly knows about the locks held by the thread in an activation, it can choose more efficiently whether it has to make an upcall or if it can avoid it.

- write a POSIX thread library[10]. Such a library with the use of the activations would be very useful.

Nonetheless, activations are already working well with MARCEL on a i386 LINUX system and seem to be a very interesting new way to manage user threads. This work shows that this model is a valid one, very useful for all people that need threads management, in particular when the threads are blocking. And this often happens within threads in a communication library for example.

# Appendix A

# A New Application Programming Interface for Scheduler Activation Support in the Linux Kernel

## A.1 Semantics

### A.1.1 Activation

All activations are launched by upcalls. If one activation finishes (call to exit, signal, ...), then the whole process finishes (other activations are simply preempted).

An activation is either running or blocked. It has an ACTIVATION IDENTITY or AID from 0 to the maximum of activations allowed by the kernel and requested by the application (see `act_init` A.2.1).

There exists a lock for the activations called ACT_LOCK. When an upcall is done, the activation automatically locks the lock except for signals (see below). This means that until the activation calls `act_resume`, no other upcall will be done. This is a way to protect the data about application context during an upcall. If an activation with this lock is preempted, then another activation will be preempted and this one will be rescheduled without an upcall.

### A.1.2 Signals

There exists only one mask/handler for the whole process. The stack used for the first signal is the one declared at the initialization. Then, the current stack pointer is kept and reused.

A signal is executed in an activation but it does not lock the lock and it makes its upcall directly in the signal handler (if it exists). Only one activation can handle a signal at a time (there is not simultaneous execution of signal handlers) except for synchronous signals (see below). This behavior allows the use of only one stack for signals.

The signals `SIGKILL`, `SIGSTOP` and `SIGCONT` affect all the current running activations.

The synchronous signals (such as `SIGSEGV`, `SIGILL`, `SIGBUS`, etc.) are sent to the current activation. In this case, the activation performs as if it has the lock. That is, if it is stopped, no upcall is done to the application, but it is rescheduled as soon as possible without upcall. In this case, we can have several signal handler executions at the same time (one asynchronous on the signal stack, the others synchronous on the activation stack).

## A.2 System calls

### A.2.1 act_init

*int* `act_init`(*act_param_t * param*);

This call is made only one time, at the beginning, to ask the kernel to use activations for this process. If all is good, we never return : an upcall is made to `act_restart` (with the indication of the current state, that allows to continue just after this system call).

The parameter *param* is a pointer to a data structure that contains:

- *int* `nb_max_activations` This is the maximum number of activations that the application can manage. 3 is the minimum : the kernel needs one for signals and 2 at least so it can make an upcall when an activation is stopped. 0 means no limit.
- *int* `nb_act_wanted` This is the number of activations that can run in parallel on the computer. It must be between 1 and the number of processors of the computer. The value 0 is accepted and stands for the number of processors of the computer.
- *void ** `act_sp` This value will be used for the stack pointer for each upcall.
- *void ** `sig_sp` This value will be used for the stack pointer for each signal.
- *act_buf_t ** `context1`
- *act_buf_t ** `context2` These two pointers designate areas where the kernel can store the application context for an upcall. These areas are architecture dependent.
- *int *()* `act_new`
- *int *()* `act_preempt`
- *int *()* `act_block`
- *int *()* `act_unblock`
- *int *()* `act_restart`

  These are the addresses of the functions to call for the activations.

**Errors:** Activations are used only if none of these errors occur.

`EBUSY` This system call has already been done.

`EFAULT` Some access permissions (write to buffers, read to functions, etc.) are wrong.

`ENOSYS` Activations not currently implemented.

**return from `clone()`** The system call clone is called to create the first activation. It can return an error.

### A.2.2 act_resume

*int* `act_resume`(*act_buf_t * buffer*, *int fast_restart*);

This call releases the lock ACT_LOCK if the activation had it. Then, if *buffer* is not `NULL`, the activation continues with the state saved in *buffer**, if it is `NULL`, the activation continues after this system call.

**Errors:** With bad access permission to buffer, a SEGV is generated.

`EACCES` The initialisation of activations was not done yet.

If `fast_restart` is not null, and if we are ending an upcall which has the `fast_restart` flag set, then the kernel restarts the preempted or unblocked activation at the place where it was.

### A.2.3 act_cntl

*int* `act_cntl`(*int cmd*);
*int* `act_cntl`(*int cmd*, *long arg*);
*int* `act_cntl`(*int cmd*, *long * arg*);

These calls are used to manage/read the activation's behavior and state. The operation in question is determined by *cmd*:

ACT_CNTL_INC_PROC ask the kernel to allocate one more processor for the activations.

ACT_CNTL_DEC_PROC ask the kernel to allocate one less processor for the activations.

ACT_CNTL_GET_PROC return the current number of processors that the application has requested from the kernel.

ACT_CNTL_GET_MAX_PROC return the number of processors on this hardware platform.

ACT_CNTL_GET_MAX_ACT return the greatest AID that the kernel can use. It was a parameter of `act_init`. 0 means no limit.

ACT_CNTL_WAIT_UPCALL the activation will block in the kernel until an upcall (other than `act_preempted`) is about to be made. No upcall will be done to report this blocked activation, and no new activation will be created.

**Errors:** With bad access permission to buffer, a SEGV is generated.

`EACCES` The initialisation of activations was not done yet.

`EINVAL` The parameter *cmd* is out of range.

### A.2.4 act_send

*int* `act_send`(*int act*);

This call asks the kernel to stop and relaunch the activation with AID*act* if it runs with an upcall to `act_restart`. Else, nothing is done.

If *act*=`ACT_SEND_ALL`, all running activations will be stopped and upcalled.

If *act*=`ACT_SEND_MYSELF`, the current activation will be stopped and upcalled.

**Errors:** With bad access permission to buffer, a SEGV is generated.

`EACCES` The initialisation of activations was not done yet.

`EINVAL` The parameter *act* is out of range.

## A.3 Upcalls

When an upcall is done, no other upcall can be done until the activation calls `act_resume`: an upcall always acquires the lock ACT_LOCK which can only be released during the system call `act_resume`.

We do not return from these upcalls. If we do, the system generates a `SEGV`.

### A.3.1 act_new

*void* `act_new`(*int num*);

A new activation is launched. Its AID is *num*.

### A.3.2 act_preempt

*void* `act_preempt`(*int num*, *int preempt*, *int fast_restart*);

The activation with AID *num* has been stopped, then upcalled to advertise that the activation with AID *preempt* has been preempted. The state of the two activations can be found in the buffers given in the parameter to `act_init`.

If `fast_restart` is set, then we can use this flag in resume, so that the preempted activation will restart where it was when the scheduler gives it control back.

### A.3.3 act_block

*void* `act_block`(*int num*, *int block*);

The activation with AID *num* has been stopped if running, then upcalled to advertise that the activation with AID *block* has been blocked. The state of the activation with AID *num* can be found in the first buffer given in parameter to `act_init`.

### A.3.4 act_unblock

*void* `act_unblock`(*int num*, *int unblock*, *int fast_restart*);

The activation with AID *num* has been stopped if running, then upcalled to advertise that the activation with AID *unblock* has been unblocked. The state of the two activations can be found in the buffers given in the parameter to `act_init`.

If `fast_restart` is set, then we can use this flag in resume, so that the preempted activation will restart where it was when the scheduler gives it control back.

### A.3.5 act_restart

*void* `act_restart`(*int num*);

The activation has an upcall during its execution. Its AID is *num*. An activation receives this upcall after a call to `act_send` or `act_cntl` with `DO_UPCALL`. The state of the activation can be found in the first buffer given in the parameter to `act_init`.

## A.4 Remarks

### A.4.1 Current implementation

The signal management has not yet been implemented. For now, signals and activations cannot be used together in a program. This must be fixed in a future release.

### A.4.2 Perspective

Several points can be dealt with later, such as :

- `act_cntl` can be extended so can we can read/write more parameters like stacks used, addresses of upcalls, etc.
- We can make upcalls without the lock ACT_LOCK for a function like `act_send` or every time that the kernel reschedules an activation so that it can switch between user thread contexts iff it wants.

  This will need another upcall (another address in `act_init` and an extension to `act_cntl`).
- No behavior has been defined yet if an activation calls the system calls `fork`, `clone`, `exec`, etc.
- No semantics has been defined for the signals `SIGVTALARM` and `SIGPROF`.

# Appendix B

# Internals of Activation Support in the Linux Kernel

## B.1 Files

The current implementation has been done with the kernel 2.2.10 of Linux. Here is the list of files concerned with the activations:

### B.1.1 *include/asm-i386/unistd.h*

- Add the `__NR_act_*` constants for the new system calls.

### B.1.2 *include/asm-i386/act.h*

- New file

- Main header file for the activations. All constants and structures are defined in this file.

### B.1.3 *include/asm-i386/act_sched.h*

- New file

This header file is included by the previous. It is a separate file because we have there all that is necessary to include in the following. So modifying *act_sched.h* or *sched.h* causes recompilation of most (all?) the files of the kernel, whereas modifying *act.h* causes recompilation of only few files when building a new kernel.

*act_sched.h* and *act.h* could be merged in later release.

### B.1.4 *include/linux/sched.h*

- Add a function `schedule_real()` (see B.4.2).

- Add several variables in `struct task_struct` (see B.2.1).

### B.1.5 *arch/i386/kernel/entry.S*

- Add the new system calls' entries.
- Add some assembly code around the `schedule` call (see B.4.2).
- Add some assembly code to call the upcall (see B.4.2).

### B.1.6 *arch/i386/config.in*

- Add menu entries for `CONFIG_ACT` and `CONFIG_ACT_DEBUG`.

### B.1.7 *kernel/Makefile*

- Add *act.o* to the list of object files.

### B.1.8 *kernel/exit.c*

- Add code in `do_exit` (see B.4.4).

### B.1.9 *kernel/fork.c*

- Add code in `do_fork` (see B.4.4).

### B.1.10 *kernel/act.c*

- New file
- Main file to manage the activations.

Note: This file has some architecture dependencies. It could be better to put it in *arch/i386/kernel* in a future release. (Not done because the directory *kernel* is one of the first to be compiled when building a new kernel, so it's quicker to see errors.)

### B.1.11 *kernel/sched.c*

- Add code when switching between tasks (see B.3.1).
- Add the function `schedule_real()` (see B.4.2).

## B.2 Data structures

### B.2.1 *struct task_struct*

Some fields have been added to the structure *struct task_struct* to manage the activations.

**need_upcall**

*long* `need_upcall` ;

**Values:**

0 We do not go through `do_upcall` (B.4.2) before returning from a schedule or a system call. A process that does not used activations always has the value 0 there.

1 We will go through `do_upcall` (B.4.2) next time we return from a schedule or a system call.

The offset of this field from the beginning of the structure is hard-coded in the assembly file *arch/i386/kernel/entry.S*. The hard-coded value is verified when executing `sys_act_init` in *kernel/act.c*.

### act_in_use

*int* `act_in_use` ;

**Values:**

`ACT_NOTUSED` This kernel task does not used activations. In this case, the following fields have no significance for the task.

`ACT_USED` **or** `ACT_USED_WAITING_EVENT` This task is an activation. See the activation's states (B.3) for more information.

`ACT_MANAGER` This task is the manager activation of the process. See the manager activation (B.4.4) for more information.

### act_want_upcall

*volatile int* `act_want_upcall` ;

**Values:**

0 The activation does not require an upcall.

1 The activation does require an upcall. If the kernel has no task that needs an upcall, then an `act_restart` upcall will be generated when it becomes possible (lock `upcall_lock` free). Otherwise the kernel will generate the upcall that is needed.

### struct_act_info

*struct act_struct* `struct_act_info` ;
See B.2.2

### *act_info

*struct act_struct* `*act_info` ;
See B.2.2

### act_id

*int* `act_id` ;
The activation id. For now, the manager has always the id 0.

**act_state, act_known_state**

*int* `act_state, act_known_state` ;

**Values:**

`ACT_NEW`

`ACT_RUNNING`

`ACT_PREEMPTED`

`ACT_BLOCKED`

`ACT_UNBLOCKED`

`ACT_FREE`

`ACT_END`

`ACT_IN_UPCALL`

`ACT_READY`

For a description of the meaning of these values, see the activation's states (B.3). Basically, `act_state` is often the real state of the activation, whereas `act_known_state` is the last state of the activation known (through upcalls) by the process.

***act_next, *act_prev**

*struct task_struct* `*act_next, *act_prev` ;
Double linked list of all the activations of a process (including the manager).

**act_linked**

*int* `act_linked` ;

**Values:** This field shows the list in which the activation is. See Lists (B.4.3) for more information.

`ACT_LINK_NONE` The activation is in none of the lists.

`ACT_LINK_PREEMPTED` The activation is in the `preempted` list.

`ACT_LINK_BLOCKED` The activation is in the `blocked` list.

`ACT_LINK_UNBLOCKED` The activation is in the `unblocked` list.

`ACT_LINK_FREE` The activation is in the `free` list.

`ACT_LINK_RESTART` The activation is in the `restart` list.

***act_list_next, *act_list_prev**

*struct task_struct* `*act_list_next, *act_list_prev` ;
Fields for the double linked list in which the activation is. See Lists (B.4.3) for more information about the lists.

#### *act_wait_queue

*struct wait_queue* `*act_wait_queue` ;
An activation can use this wait queue when it is waiting for an event.

#### *act_regs

*struct pt_regs* `*act_regs` ;
Pointer to a structure with the registers of the activation. Used to have the state of a unblocked activation.

### B.2.2 *struct act_struct* or *act_info*

Although each activation has such a structure in its task_struct, only one is used per process. In fact, the structure used is the one of the manager activation, the field `act_info` in all the activations of this process is a pointer to the structure of the manager activation. This structure could be allocated and freed in the memory, but it is small and it's easier to do like this.

This structure is used for all the variables that must be shared between all the activations of a process.

#### param

*act_param_t* `param` ;
See B.2.3

#### act_lock

*spinlock_t* `act_lock` ;
This spinlock protects all the operations related to the upcall for the process. In a later release, it could be interesting to see if it's possible and useful to split this spinlock into several that protect fewer operations per lock.

#### current_upcall

*int* `current_upcall` ;

**Values:** This field shows the kind of upcall being done.

`ACT_UPCALL_NONE` No activation makes an upcall.

`ACT_UPCALL_NEW` An activation is executing an upcall `act_new`

`ACT_UPCALL_PREEMPTED` An activation is executing an upcall `act_preempt`

`ACT_UPCALL_BLOCKED` An activation is executing an upcall `act_block`

`ACT_UPCALL_UNBLOCKED` An activation is executing an upcall `act_unblock`

`ACT_UPCALL_RESTART` An activation is executing an upcall `act_restart`

***manager**

*struct task_struc* `*manager` ;
This is a pointer to the manager activation of the process.

***act_in_creation**

*struct task_struc* `*act_in_creation` ;
This is a pointer to the activation that is in creation (when there is one). See B.4.4 for more information.

***act_fast_restart**

*struct task_struc* `*act_fast_restart` ;
This is a pointer to the activation that could be fast restarted (when there is one). See also state PREEMPTED (B.3.2).

***act_running_slept**

*struct wait_queue* `*act_running_slept` ;
This wait queue is used for the activations calling `act_send(ACT_CNTL_WAIT_UPCALL)`. These activations are still considered as running (so no new ones are created) although they are sleeping in the kernel.

**nb_act_created**

*int* `nb_act_created` ;
This is the total number of all the created (and still existing as kernel task) activations including the manager, the activations in the `free` list, etc.

**nb_act_running**

*int* `nb_act_running` ;
This is the number of activations running. We want this number to be equal to the previous. We count as running all the activations really running, but also activations in an upcall (even if it's blocked !) and activations preempted by the kernel but not yet reported to the process by an upcall.

**nb_act_preempted, nb_act_blocked, nb_act_unblocked, nb_act_free, nb_act_restarted**

*int* `nb_act_preempted, nb_act_blocked, nb_act_unblocked, nb_act_free, nb_act_restarted` ;
The number of activations in each list. (See B.4.3)

***list_act_preempted, *list_act_blocked, *list_act_unblocked, *list_act_free, *list_act_restarted**

*struct task_struct* `*list_act_preempted, *list_act_blocked, *list_act_unblocked, *list_act_free, *list_act_restarted` ;

A pointer to an element of each list (or `NULL` if the list is empty). (See B.4.3)

***upcall_lock**

*volatile struct task_struct* `*upcall_lock` ;
This is a pointer to the activation doing an upcall. This field is `NULL` when no upcall is being done.

### B.2.3 *act_param_t*

This structure is given to the kernel during the initialization.

**nb_max_activations**

*int* `nb_max_activations` ;
This is the maximum number of activations that the kernel is allowed to create for this process.

**nb_act_wanted**

*int* `nb_act_wanted` ;
This is the number of activations that the application wants to run in parallel on the computer.

***kernel_act_struct**

*struct kernel_act_struct_t* `*kernel_act_struct` ;
This is a pointer to a small area in user space where the kernel will be able to write and execute some code. It is basically used to allow some activation (like the manager) to have a little section of code to return into the kernel after executing a signal handler for example.

**act_sp**

*void** `act_sp` ;
This will be the stack used for the upcalls.

**sig_sp**

*void** `sig_sp` ;
This will be the stack used for the signals (handled by the manager).

***cur_buf, *stopped_buf**

*act_buf_t* `*cur_buf, *stopped_buf` ;
These are two pointers to areas where the kernel will save the activations' state before doing the upcall.

***act_new**

*void (* `*act_new` *)(act_id_t aid)*;
This is a pointer to the function that will be used for the upcall `act_new`.

***act_preempt**

*void (* *`*act_preempt`* *)(act_id_t cur_aid, act_id_t stopped_aid, int fast_restart)*;
This is a pointer to the function that will be used for the upcall `act_preempt`.

***act_block**

*void (* *`*act_block`* *)(act_id_t cur_aid, act_id_t stopped_aid)*;
This is a pointer to the function that will be used for the upcall `act_block`.

***act_unblock**

*void (* *`*act_unblock`* *)(act_id_t cur_aid, act_id_t stopped_aid, int fast_restart)*;
This is a pointer to the function that will be used for the upcall `act_unblock`.

***act_restart**

*void (* *`*act_restart`* *)(act_id_t cur_aid)*;
This is a pointer to the function that will be used for the upcall `act_restart`.

## B.3 Activation's states

Each activation in a process is in a particular state, which determines what happens in the kernel. All of the changes of state are protected by the spinlock `act_lock` (B.2.2).

The main state of an activation is given by the field `act_in_use` (B.2.1). There are four possibilities:

`ACT_NOTUSED` This kernel task does not use activations. Not very interesting in this document.

`ACT_MANAGER` This task is the manager activation. See the manager activation (B.4.4) for more information.

`ACT_USED` The scheduler changes the state of this activation when it schedules or unschedules it.

`ACT_USED_WAITING_EVENT` This activation does not change its state when the scheduler schedules or unschedules this activation.

You can refer to figure B.1 for the states of a non-manager activation (`act_in_use` set to `ACT_USED` or `ACT_USED_WAITING_EVENT`).

### B.3.1 Activation with `ACT_USED`

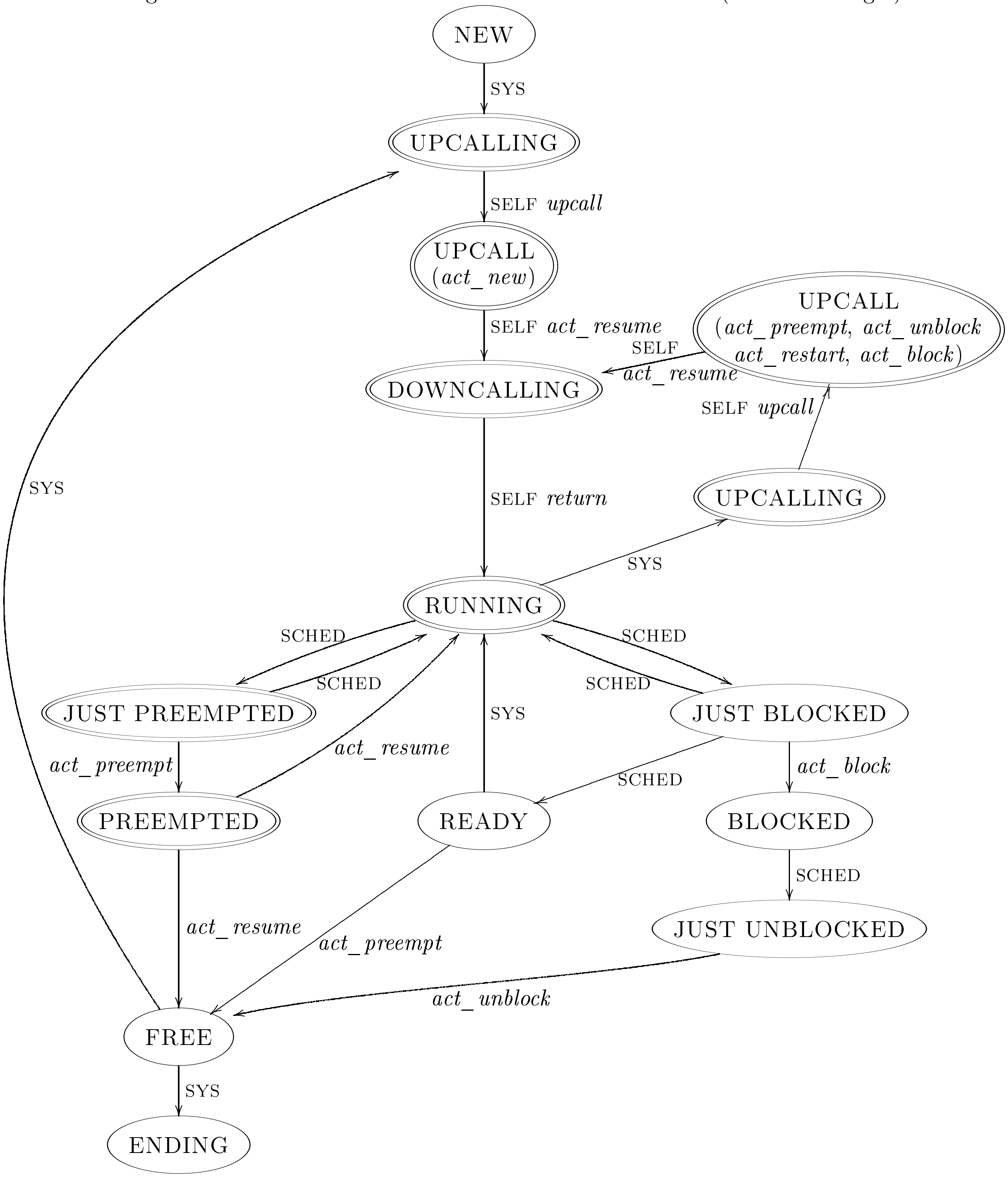
With `act_in_use` set to `ACT_USED`, an activation has its `act_known_state` set to `ACT_RUNNING` or `ACT_BLOCKED`.

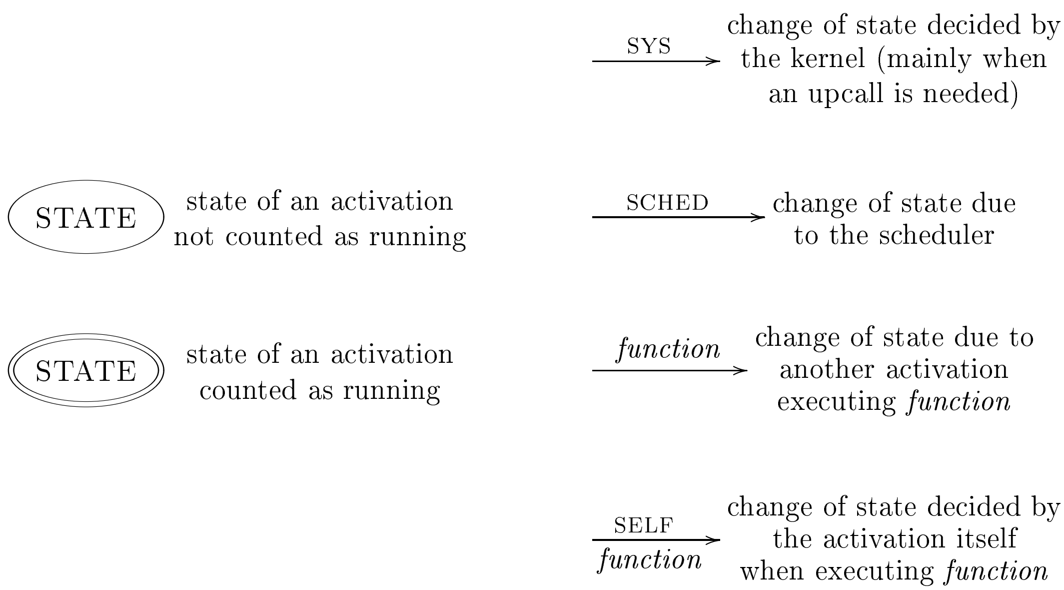
An activation can have its state changed in the scheduler when `act_in_use` is set to `ACT_USED`.

We often come into this mode after a `act_resume` system call. In this case, the activation goes into the state RUNNING (B.3.1).

There are several states depending on the fields `act_known_state` and then `act_state`.

Figure B.1: The Different States of an Activation (not the manager)






### State RUNNING

`act_known_state` and `act_state` are set to `ACT_RUNNING`.

The activation is running on a processor. Normally, we also have this state when the task is `TASK_UNINTERRUPTIBLE`, which means the kernel has preempted this task for a short moment.

### State JUST_PREEMPTED

`act_known_state` is set to `ACT_RUNNING` and `act_state` to `ACT_PREEMPTED`.

The activation has been preempted by the scheduler just before returning to user mode, but this has not yet been reported with an upcall. The activation is in the `preempted` list and is still considered as running.

### State JUST_BLOCKED

`act_known_state` is set to `ACT_RUNNING` and `act_state` to `ACT_BLOCKED`.

The activation has not been preempted by the scheduler just before returning to user mode but just before returning to kernel mode, and this has not yet been reported with an upcall. The activation is in the `blocked` list and is not considered as running any more.

### State BLOCKED

`act_known_state` is set to `ACT_BLOCKED`.

This means that the kernel is waiting for this activation to be unblocked. It has already been reported to the process as blocked. The first time the scheduler will give control to this activation just before returning to user mode, the activation will go to the state JUST_UNBLOCKED (B.3.2).

### B.3.2 Activation with ACT_USED_WAITING_EVENT

With act_in_use set to ACT_USED_WAITING_EVENT, an activation's state is no longer changed in the scheduler.

We often go to sleep using act_wait_queue in this mode or go back to a state with ACT_USED (B.3.1).

The state of the activation is mainly dependent on the value of act_state.

#### State NEW

act_state is set to ACT_NEW.

The activation will be waiting to be able to make an upcall to act_new. It is not yet considered as running. The field act_in_creation (B.2.2) is set to itself.

#### State UPCALL

act_state is set to ACT_IN_UPCALL.

The activation is the one doing an upcall.

#### State UPCALLING

act_state is set to ACT_IO.

The activation is performing a read/write in user space to set the parameters for an upcall and get/set the registers for the upcall. This state is only useful when swapping occurs.

#### State DOWNCALLING

act_state is set to ACT_IO.

This state is the same as the previous in fact. The only difference is that we are not doing an upcall, but in the act_resume function. The activation is performing read/write in user space to get and set the registers to resume the activation. This state is only useful when swapping occurs.

#### State JUST_UNBLOCKED

act_state is set to ACT_UNBLOCKED.

The activation is coming back from the state BLOCKED (B.3.1). It will go in the unblocked list. It is not considered as running (in fact since the activation was in the state BLOCKED). The unblock was not yet reported with an upcall.

#### State READY

act_state is set to ACT_READY.

The activation will be waiting in the ready list. The activation is ready to resume but not known as running. Either an upcall to act_preempted will be done, or the activation will be restarted as it is if the process needs one more activation.

#### State PREEMPTED

act_state is set to ACT_PREEMTED.

An activation is waiting is this state during the upcall made to report its state. It leaves this state when the act_resume of the current upcall is being done. Depending on the second parameter of act_resume, this activation will restart immediately or change to state FREE (B.3.2). The pointer act_fast_restart (B.2.2) is set to this activation.

#### State FREE

act_state is set to ACT_FREE.

The activation will be waiting in the free list until a new upcall needs to be done. If there are enough free activations, then the activation can be destroyed.

#### State ENDING

act_state is set to ACT_END.

The activation will make a call to do_exit(0).

## B.4 Principles

### B.4.1 Flow control

Before returning from a system call or from a timer interruption (for scheduling), the kernel looks to see if the task has a pending signal, and if it has, calls the function do_signal that establishes the new context for the signal handler and then continues the execution. Now, we have another test: after looking for a pending signal, we look for a pending activation, that is, if need_upcall (see B.2.1) is true (= 1), the do_upcall is executed.

### B.4.2 do_upcall

This function first looks to see if we are returning to user mode or kernel mode (this happens if the timer interrupt occured during a system call for example). If we are in the latter case, then we return immediately.

The following depends on the field act_in_use (B.2.1).

If we are the manager (value ACT_MANAGER), then we execute the function activation_manager.

If the value is ACT_USED, then we look to see if we need an upcall and if it's possible to do one (upcall_lock free); if yes, we do one.

And if the value is ACT_USED_WAITING_UPCALL, then we wait for the correct event (we often sleep using act_wait_queue after setting some variables).

If the activation manager or an activation with ACT_USED_WAITING_UPCALL needs to return in few moments to user mode (to handle signals for example), then the code put in kernel_act_struct at the initialization is used. This is mainly a call to act_cntl(ACT_CNTL_RET_IN_KERNEL). (See B.4.2)

### Upcall

Only one activation can make an upcall at any time. It has to have the `upcall_lock` set to itself. The field `current_upcall` is set to the kind of pending upcall. The lock and this field are freed in the system call `act_resume` if this is the activation that owned the lock that called `act_resume`.

Each time an upcall is done, a few instructions are put on the stack, so that the application calls `act_cntl(ACT_CNTL_RET_UPCALL)` if it returns from the upcall function. (See B.4.2)

### Special uses of `act_cntl`

The system call `act_cntl` is used with special values for its parameter to handle special cases.

With the parameter `ACT_CNTL_RET_UPCALL`, a `SEGV` is generated. This happens when an activation returns from an upcall function.

With the parameter `ACT_CNTL_RET_IN_KERNEL`, we verify that `act_in_use` is either `ACT_MANAGER` or `ACT_USED_WAITING_UPCALL` and then just return. But the return will be intercepted and the correct function called just before the return to user mode (see B.4.2).

### Calls to `schedule`

In the scheduler, we need to known if we are preempted or blocked, that is, if we called `schedule` just before returning to user space (so that we can give the activation state in an upcall), or elsewhere in the kernel.

When compiling the kernel with `CONFIG_ACT`, the function `schedule` is renamed to `schedule_real`.

Now, the function `schedule` calls `schedule_real` with the `NULL` parameter.

And instead of calling `schedule` in the file *entry.S*, we call `act_schedule`. This function calls `schedule_real` with a `NULL` parameter if we are ready to return in kernel mode, or with a pointer to the user registers if we are ready to return to user mode.

## B.4.3 Lists

Each activation can be put in one of several lists. These lists are used to sort the activations and to be able to easily find one of them with a particulate state.

The lists existing are:

`preempted` The activations in this list are the ones with `act_state`=`ACT_PREEMPTED` and `act_known_state`=`ACT_RUNNING`. They are still considered as running.

`blocked` The activations in this list are the ones with `act_state`=`ACT_BLOCKED` and `act_known_state`=`ACT_RUNNING`. They are no longer considered as running.

`unblocked` The activations in this list are the ones with `act_state`=`ACT_UNBLOCKED` and `act_known_state`=`ACT_BLOCKED`. They are no longer considered as running.

`free` The activations in this list are the ones free to be reused for an upcall `act_new`.

`restart` The activations in this list are the ones with `act_state`=`ACT_READY` and `act_known_state`=`ACT_RUNNING`. They are no longer considered as running. They are either reported as preempted, or directly launched instead of creating a new activation if we need an other activation.

### B.4.4 Activation manager

The activation manager has `act_id`= 0 and `act_in_use`=`ACT_MANAGER`. It's field `act_state` is mainly `ACT_RUNNING`. The only exception is at the beginning during the creation of the first activation, where its state is `ACT_NEW` (so that the first activation makes an upcall `act_resume` and not `act_new`). The other exception is for the end where its state is `ACT_END` (see B.4.4).

#### Creation

When a new activation is needed, we first look at the `ready` list. If there is an activation there, then we resume it immediately.

If there is no `ready` activation, we look at the `free` list. If there is a kernel task there, then we use it to make the upcall `act_new`.

If there is no `free` kernel task, then we create a new one if we would not go over the limit of the user (see `nb_max_activations` B.2.3). When creating a new activation (calling `do_fork`), the manager has its `act_known_state` set to `ACT_NEW` instead of `ACT_RUNNING`.

#### Ending

When an activation comes to `do_exit`, it looks at its `act_state`. If its set to `ACT_END`, then the activation just ends, removing itself from the double linked list of the activation of the process and decreasing the number of activations launched (see B.2.1 and B.2.2).

## B.5 Limits of the current implementation

- Not fully debugged and tested on SMP.
- Not tested at all with swapping. Must be good now.
- Signal catching completely broken, even in the manager. I will work on it later.
- Unknown bugs...

# Bibliography


[1] Advanced Computing Laboratory. Tulip: A Portable Parallel Run-time Class Library. http://www.acl.lanl.gov/tulip.

[2] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[3] Rémy Card, Éric Dumas, and Franck Mével. *Linux 2.0: Api Système et Fontionnement du Noyau.* Eyrolles, 1997.

[4] V. Danjean. Introduction d'un Ordonnanceur de Processus Légers Mixte dans $PM^2$ pour l'Exploitation Efficace des Architectures Multiprocesseurs. Rapport de stage de première anné de MIM, 1998.

[5] E.W. Dijkstra. Cooperating Sequential Process. In *Programming Languages*, pages 43–112. Academic Press, 1968.

[6] I. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proc. 1st Intl Workshop on Parallel Processing.* Tata Mc Graw Hill, 1994.

[7] I. Ginzburg. *Athapascan-0b: Intégration efficace et portable de multiprogrammation légère et de communications.* Thèse de doctorat, Institut National Polytechnique de Grenoble, LMC, Sep 1997.

[8] C.A.R Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[9] C.A.R Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[10] IEEE Standards Department. *1003.4d8 POSIX System Application Program Interface: Threads Extensions [C language]*, 1994.

[11] Xavier Leroy. The LinuxThreads library. http://pauillac.inria.fr/~xleroy/linuxthreads.

[12] R. Namyst. *PM2 : un environnement pour une conception portable et une exécution efficace des applications parallèles irrégulières.* Thèse de doctorat, Univ. de Lille 1, Janvier 1997.

[13] R. Namyst and J.-F. Méhaut. MARCEL *: Une bibliothèque de processus légers.* Laboratoire d'Informatique Fondamentale de Lille, Lille, 1995.

[14] R. Namyst and J.-F. Méhaut. *$PM^2$ Parallel Multithreaded Machine : Guide d'utilisation.* Laboratoire d'Informatique Fondamentale de Lille, Lille, 1995.

[15] Alessandro Rubini. *Linux Device Drivers.* O'Reilly, first edition, February 1998.