CHAPTER 3

Execution Environment

Abstract

In a multiprocessor environment with parallel applications concurrently running, the Operating System is responsible for optimizing the system utilization and the individual application execution. The system utilization depends on several factors such as the number of processors assigned to each application, or which particular processors are assigned to each application.

In this work, we present the particular characteristics of the three elements that constitute our execution environment: the long-term scheduler or queuing system (Launcher), the medium-term scheduler or simply the scheduler (CPUManager), and the runtime parallel library (NthLib). These three elements provide us a total control of how applications are scheduled. Having the control of these elements, we have a powerful tool to analyze the effect of the different scheduling issues of both the performance of the applications and the system utilization.

3.1 Introduction

In this Thesis, we use a practical approach. To do that, we have created our execution environment to implement the coordinated scheduling and the processor scheduling policies proposed in this Thesis. Figure 3.1 shows the elements that compound our execution environment: the queuing system (Launcher), the processor scheduler (CPUManager), and the run-time parallel library (NthLib [63][64]). In this Chapter, we give a lot of details about the implementation of these elements. If the reader is not interested in these details, the reading can be continued in Chapter 4.

The Launcher is the queueing system. It implements the job scheduling policy that decides which particular application must be executed at any moment. The Launcher controls the multiprogramming level, that can be defined by the administrator (if the Launcher works uncoordinated with the CPUManager), or by the processor scheduling policy (if it works coordinated with the CPUManager). The Launcher has been implemented to introduce repeatability in the submission of workloads of parallel applications with the aim of evaluating them under different execution environment configurations.

The CPUManager is the user-level processor scheduler. Once the Launcher starts the execution of a queued application, it enters under the CPUManager control. It implements the processor scheduling policy, which (1) decides how many processors to allocate to each application, and (2) enforces the processor scheduling policy decisions. The CPUManager uses the native operating system interface to manage processes and processors and provides the interface to communicate with the Launcher, see Figure 3.1.

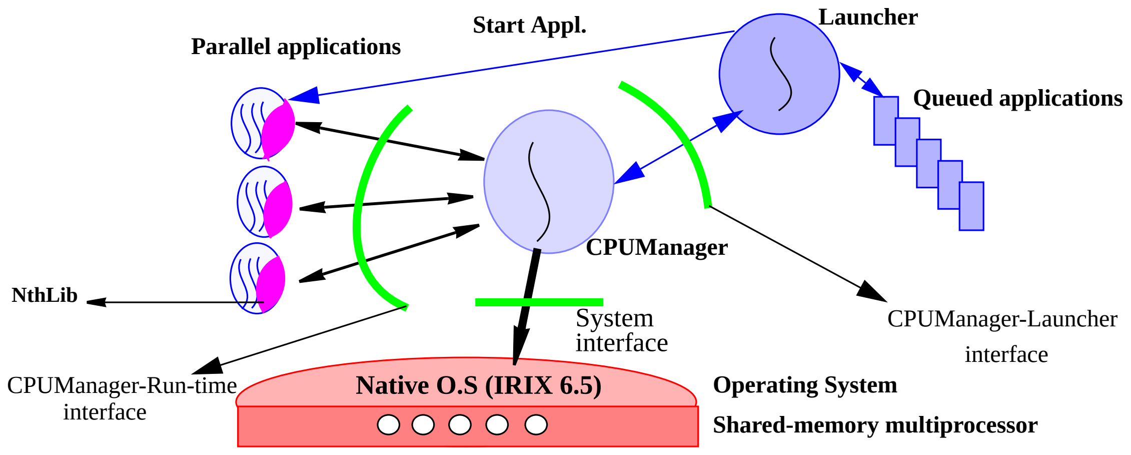

Figure 3.1: Execution environment

The NthLib [63][64] is the run-time library used in this Thesis to implement the policies and mechanism needed at the loop scheduling level. The NthLib was developed in the NANOS project [62][63][64] and modified in this Thesis. The NthLib supports the parallelism specified by users through OpenMP directives. It requests for processors using the CPUManager interface and reacts to changes in the number of processors allocated to the application.

The remainder of this Chapter is organized as follows: Section 3.3 presents the queueing system implemented in this work, the Launcher. Section 3.4 describes the internal structure of the CPUManager and the execution environment offered to parallel applications. Section 3.5 presents the characteristics that a parallel library should have to cooperate with the CPUManager, and a particular implementation of these features in NthLib. Finally, Section 3.6 presents the summary of this Chapter.

3.2 Related work: Resource Managers

The motivation to implement our own resource manager is clear, we need to have a complete control of this element. However, we comment in this Section some of the commercial resource managers that we can found.

Several computing industries are releasing operating system-based resource managers (RM). These RM have been designed and implemented under different operating systems and with different goals but with common features. The main characteristic of a RM is that it is designed to provide the administrator or the final user a major control over the architectural resources: CPUS's, virtual memory, I/O bandwidth, etc.

RM typically provide users or administrators an API to specify user requirements. User requirements can be as simple as *"to set aside specific CPUs to specific applications"*[101], or to *"guarantee a minimum entitlement of CPU, memory, and disk bandwidth available to a group of processes."*[100].

RM are usually implemented at user-level and they use the native operating system tools to allocate and manage the resources. Most of them allow a great configurability but they require a high knowledge about the system or the particular process to specify either the resource allocation to a workload or the resource requirements of a process. For instance, in the HP-UX Workload Manager [100], the administrator should specify performance goals and priorities for the workloads, and assign a performance monitor to the workload to measure its performance. The AIX WLM [45] allows the administrator to define different classes of jobs and assign different level of resources to each one. The Solaris RM [101] works in a similar way. These RM require that an administrator defines the different classes of jobs, priorities among them and the amount of resources that each one needs. They are oriented to guarantee a certain resource reservation from the point of view of the user, to ensure a certain medium-term resource distribution.

Other RM attack the problem of the resource reservation from the point of view of the thread. They are mainly RM oriented to real-time and multimedia applications [21][71][72]. In this case, the resources are specified as a quality of service. This kind of RM allow the co-existence of real-time and multimedia with time-sharing applications ensuring the reservation of the resource.

The CPUManager differs from the previously proposed RM in two main points. First, the CPUManager is application oriented. The unit of resource allocation and management is the parallel application. Other RM (like the Solaris RM or the AIX WLM) work at a workload or user granularity. Alternatively, RM oriented to real-time and multimedia applications work at thread granularity. Second, the CPUManager works at a different level than the rest of proposed RM. The CPUManager works at a similar level than a traditional operating system does. The CPUManager not only reserve CPUS to a particular application, but also performs the mapping between processes and processors and controls the initial memory placement.

3.3 The queueing system: The Launcher

The Launcher is the user-level queuing system used in our execution environment. It implements the job scheduling policy, that decides which application to execute from a list of queued applications. The aim of the Launcher is to be able to execute a workload of parallel applications several times under different system conditions and processor scheduling policies.

The Launcher executes a workload of parallel applications specified through a workload trace file. It receives as parameters the workload trace file, the maximum multiprogramming level, and a file with a list of applications. Using the workload trace file, we are able to (1) execute the same set of applications under specified conditions (submit time, initial request, etc.), (2) measure the system performance, and (3) compare results achieved under the different conditions.

The workload trace file follows the *Standard Workload Format* (SWF) proposed by Feitelson in [102]. Figure 3.2 shows a portion of a workload trace file. Columns different from -1 are *job_number, submit_time,* and *application_number*. In our case, the application number refers to the list of applications received by the Launcher. First application is application 0, second application is application 1, and so on.

```
#job_number submit_time application_number
0 12 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 3 -1 -1 -1 -1
1 20 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 2 -1 -1 -1 -1
2 21 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1-1 -1 -1 -1
```

Figure 3.2: Standard Workload Format

If the CPUManager is running, the Launcher will detect it and will work coordinated with it. In this case, the multiprogramming level received as parameter is not used at all, and it is decided by the multiprogramming level policy implemented by the CPUManager. If the CPUManager implements a processor scheduling policy that does not include a multiprogramming level policy, such as the Equipartition, the CPUManager implements a policy that decides a fixed multiprogramming level.

If the CPUManager is not running, such as when executing the native SGI-MP scheduling policy, the Launcher uses the multiprogramming level received as parameter.

3.3.1 Workloads used in this Thesis

We have used workloads that represent the execution of a system where applications arrive following a Poisson inter-arrival process. Figure 3.3 shows the equation used to compute the inter-arrival rate of applications. P is the number of processors in the system

(64 in our case), U is the load of the system that we want to generate. It ranges from 0 to 1. $T^1{}_i$ is the execution time of the application *i* in sequential. And FRAC is the maximum fraction of machine that we want this application uses (from 0 to 1).

For instance, if we execute four different applications and we want each one to demand about the same amount of cpu time, FRAC will be equal to 0.25. With the last equation in Figure 3.3, we calculate the inter-arrival frequency per application.

$$\lambda_i \times T_i^1 = P \times U_i \longrightarrow \lambda_i = \frac{P \times U_i}{T_i^1} \longrightarrow \lambda_i = \frac{P \times U_i \times FRAC}{T^1{}_i}$$

Figure 3.3: Equation used to generate the inter-arrival rate of application i.

To generate the workload trace file, we have implemented an application that receives a list of applications, a $1/\lambda_i$ per application, and a maximum time per workload, *max_time*. This application generates a workload trace file that represents a system where each $application_i$ is submitted following a $1/\lambda_i$ and during *max_time* seconds. Workloads generated in this Thesis have been limited to 300 seconds. However, it is important to note that *max_time* only limits the maximum submission time. The Launcher waits for the complete finalization of all the jobs submitted, and all the jobs submitted are considered in our evaluation.

We have generated five workloads using four applications. We have used the swim, hydro2d, and apsi from the SPECFp95 [99] and the bt from the NASPB [48]. Table 3.1 shows the sequential execution time and the speedups of each one with 8, 16, and 32 processors. We have selected these applications because they have different speedup characteristics. Swim has a super-linear speedup, bt has a high speedup, hydro2d has low speedup, and apsi has very bad speedup. The complete performance analysis of these applications and their speedup curves can be found in Chapter 4.

Table 3.1: Parallel applications

Characteristic/Application(input)	swim(ref)	bt.A	hydro2d(train)	apsi(ref)
Exec.Time. in Sequential	212.2 sec.	1066.21 sec.	223.7 sec.	99 sec.
Speedup with 8/16/32/48 cpu´s.	14.5/26.5/32.7/26.2	5.9/12.1/20.5/24.1	6.7/7.4/5.5/3.6	0.93/0.93/0.92/0.9

Workload 1 is composed by swim's and bt's. Each one fills the 50% of the system. Workload 2 is composed by bt's (50%) and hydro2d's (50%). Workload 3 is composed by bt's (50%) and apsi's (50%). Workload 4 is composed by swim's (25%), bt's (25%), hydro2d's (25%), and apsi's (25%). Workload 5 is composed by only bt´s.

We have selected these workloads because each one is composed by applications with different speedup characteristics. In workload 1, 100% of the applications are scalable, swim´s with super-linear speedup and bt´s with good scalability. Workload 2 has a 50% of scalable applications (bt´s) and a 50% of applications with a medium speedup (hydro2d´s). Workload 3 is composed by a 50% of applications with good scalability and a 50% of applications with very bad scalability (apsi´s). Workload 4 is a mix of 25% of applications of each type: 25% of super-linear applications, 25% of scalable applications, 25% of applications with medium speedup, and 25% of no scalable applications. In workload 5, 100% of the applications are scalable but they are not super-linear.

3.3.2 Interface between the CPUManager and the Launcher

In this Thesis, we propose to establish an interface between the processor scheduler and the queueing system, coordinating the two levels. This coordination consists of sharing information and to consider this information to take scheduling decisions. The portion of the interface implemented by the CPUManager is presented in the CPUManager Section.

The Launcher informs the processor scheduler when a new application starts and when an application finishes. And the processor scheduler informs the queueing system when it can start a new application. Other information could be added such as the number of free processors, or the expected execution time of applications.

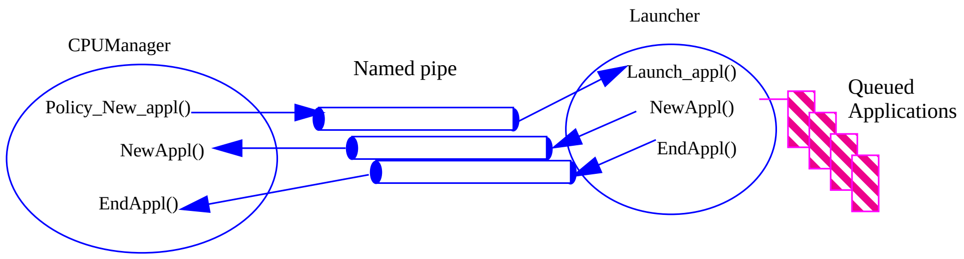

Figure 3.4: CPUManager-Launcher interface

In the current implementation, the CPUManager and the Launcher communicate through named pipes. A more complete version of the interface could be implemented using shared-memory but in this case it is efficient enough.

When the Launcher starts a new application, it sends the process identifier of the main thread of the application to the CPUManager. When the application finishes, the Launcher sends the process identifier of the main kernel thread to he application by other named pipe.

The CPUManager sends a byte to the Launcher each time the multiprogramming level policy decides that a new application can be started. Figure 3.4 shows the mechanism that implements the CPUManager interface.

3.4 The processor scheduler: The CPUManager

The CPUManager is a user-level scheduler. It implements the processor scheduling policy and enforces its decisions. It also implements the interface to coordinate with the Launcher and with the run-time library. The CPUManager uses the native operating system interface to enforce the processor scheduling decisions. In particular, it has been implemented on top of IRIX 6.5.

In order to implement a coordinated scheduler, the CPUManager and the other scheduling levels that communicate with it must agree in several rules related to the scheduling. The rules between CPUManager and the run-time library are the following:

- The run-time library has a list of work queues numbered from 0 to (maximum parallelism-1). However, the run-time library generates only work in the first P queues, where P is the number of processors available.
- The run-time library creates as many kernel threads as work queues. Therefore, it associates each kernel thread with a single work queue. The kernel thread executes the work inserted in the queue.
- When the CPUManager assigns one new processor to an application, the run-time library associates this processor with the first unallocated work queue.

Based on these three points, the CPUManager takes scheduling decisions such as deciding which kernel thread to associate to each processor, or deciding which kernel thread is more convenient to suspend when the CPUManager reduces the processor allocation of a running application. In the next subsections, we describe the CPUManager implementation.

The CPUManager was initially designed to implement space-sharing policies. In this Section, we explain the CPUManager under this kind of policies. Gang scheduling policies were implemented adding a time-sharing mechanism among jobs. Particular characteristics of Gang scheduling implementation are presented in Chapter 7.

3.4.1 CPUManager internal structure

The CPUManager wakes up periodically, at each quantum[1] expiration, and applies the processor allocation policy:

- It decides the **processor allocation** for the next quantum.
  - It decides **how many** processors to provide to each application.
  - It decides **which** processors to assign to each application.
  - It decides **which** kernel threads will run from each application.
  - It **maps** kernel threads with physical processors.
- It **communicates** its decisions to the applications.
- It **enforces** the processor allocation.

1. A typical quantum value is 100ms

Each one of these phases have several possibilities. The decisions concerning of which physical processors and which kernel threads will run are quite related, but we will consider them as independent phases.

Details about data used by the CPUManager are explained in Section 3.4.6. However, to understand the different phases we present the main data used by the CPUManager in Figure 3.5. The CPUManager has a *physical processor table*, with one entry per physical processor. Each entry records the application to which the physical processor is allocated (or NULL if it is free), and the last application to which it was allocated. The CPUManager also has a *job table*, with one entry per job. Some of the data associated to each job are the number of processors allocated, the number of processors requested, and a *kernel thread table*. This *kernel thread table* corresponds with the work queues managed by the run-time: the first entry corresponds to the first work queue, the second entry corresponds to the second work queue, and so on. The CPUManager records, per kernel thread, the physical processor identifier (if it is currently running), the last processor where it ran, the kernel thread status (RUNNING, PREEMPTED, etc), and the (process identifier/thread identifier). The per job table is allocated in a memory region shared by the run-time library and the CPUManager, and used to implement the interface between the CPUManager and the run-time library.

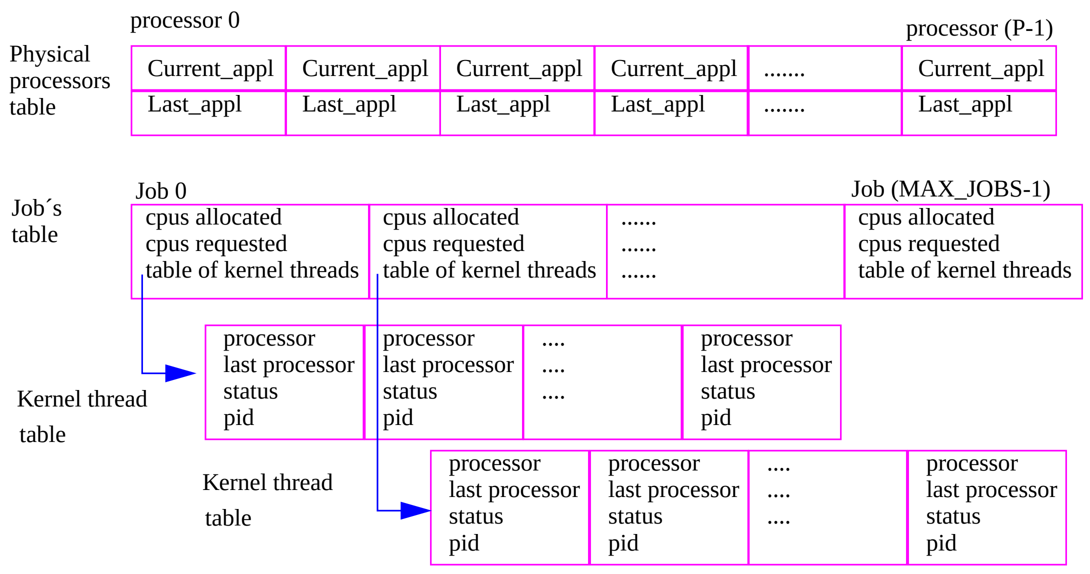

Figure 3.5: Main data structures managed by the CPUManager

3.4.2 Processor allocation

Processor scheduling policy decisions (how many processors)

The CPUManager implements the processor scheduling policy which is in charge of deciding how many processors will be allocated to each job in the next quantum.

The processor scheduling policy uses the *job table*. The processor allocation policy generates a new processor allocation, a temporal table, with an entry per job. This table has the number of processors allocated to each job.

Figure 3.6 shows a possible processor allocation generated by a processor allocation policy in a machine with 16 processors. The only information that this phase generates is a number of processors that the job will receive during the next quantum.

Job identifier	1	2	3	4	5
allocation	2	1	8	1	4

Processors=16

Figure 3.6: List of processor allocation

The only condition that the CPUManager imposes to the processor allocation policy is that the total number of processors allocated must be less or equal than the number of physical processors of the machine.

This is the only phase of the CPUManager that we will modify and evaluate in this Thesis. The particular processor scheduling policies implemented and evaluated are presented in Chapters 5, 6, and 7. Other aspects of the CPUManager have been fixed because they are out of the focus of this Thesis. However, to have a complete processor scheduler it has been necessary to implement all the CPUManager phases. In next sections we describe the different choices made at each one.

Allocating processors to jobs (which processors)

Once the scheduling policy has decided how many processors will assign to each application for the next quantum, the following step is to decide which physical processors will be allocated to each job. We will refer to the algorithm followed to decide which processors are assigned to each application as the processor placement policy, see Figure 3.7. This phase receives the processor allocation table generated in the previous phase (*alloc*), the *job table (jobs)*, and the *physical processors table (phys_proc)*. Based on this information, it generates a new *physical processor table (next_phys_proc)*. Figure 3.7 presents the algorithm used for this purpose. In this phase, the *physical processor table* is not still modified, only a temporal version is generated. The *physical processor table* is effectively updated in the last phase of the CPUManager.

```
input:  physical_processor_table  (phys_proc),  job_table  (jobs),  table  with  number  of
processors per job (alloc)
output: temporal physical processor table (next_phys_proc)
placement_policy()
{
  for(cpu=0;cpu<MAX_CPUS;cpu++)next_phys_proc[cpu]=NULL;
  for (current_job=0;current_job<active_jobs;current_job++){
    if (jobs[current_job].current<=alloc[current_job]){
      maintain_cpus=jobs[current_job].current; // receives equal or more processors
    }else{
      maintain_cpus=alloc[current_job]; // receives less processors
    }
    maintain_n_first_processors(current_job,maintain_cpus);
  }
  for (current_job=0;current_job<active_jobs;current_job++){
    if  (alloc[current_job]>jobs[current_job].current){  //  The  application  receives  more
processors
      alloc_last_n_processors(current_job,alloc[current_job]-jobs[current_job].current);
    }
  }
}

input:  physical_processor_table  (phys_proc),  job_table  (jobs),  table  with  number  of
processors per job (alloc)
output: temporal physical processor table (next_phys_proc)
maintain_n_first_processors(int job, int cpus)
{
  for (kthread=0;kthread<cpus;kthread++){
    curr_cpu=job_table[job].kernel_threads[kthread].cpu;
    next_phys_proc[curr_cpu]=job;
  }
}
input:  physical_processor_table  (phys_proc),  job_table  (jobs),  table  with  number  of
processors per job (alloc)
output: temporal physical processor table (next_phys_proc)
alloc_last_n_processors(int job, int cpus)
{
  for(kthread=job_table[cpu].current;kthread<job_table[cpu].current+cpus;kthread++){
    curr_cpu=job_table[job].kernel_threads[kthread].last_cpu;
    // if the job has never run previously, we look for a free cpu near the rest of cpus
    // allocated to the job
    if(curr_cpu==NULL)curr_cpu=select_new_cpu(job);
    next_phys_proc[curr_cpu]=job;
  }
}
```

Figure 3.7: CPUManager processor placement algorithm

In a CC-NUMA machine, like the Origin 2000, the placement of processors has a significant influence in the execution time of parallel applications. For instance, if we assign separated processors to a parallel application, it will pay the cost of accessing remote pages. The aim of this phase is to select the more *convenient* set of processors per application.

With the aim of exploiting, as much as possible, the data locality, the CPUManager implements a placement policy oriented to maintain the processor affinity. It tries to execute the job in the same set of processors that in the last quantum.

The *placement_policy* function calculates how many processors from the previous quantum are kept for the next quantum per application. The *maintain_n_first_processors* function looks into the kernel thread table of the job and selects the cpus allocated to the first P work queues (from 0 to *alloc*[job]-1).

To those applications that will receive more processors, the *alloc_last_n_processors* function selects those cpus where kernel threads, from current to the new allocation, ran the last time. In the case that these kernel threads had never run, the function *select_new_cpu* selects new cpus to run them. This function tries to allocate a new cpu following three criteria: (1) a free cpu where the job has run previously, (2) a free cpu near the rest of cpus of the job, and (3) any free cpu.

After this phase, The CPUManager will have a *physical_processor_table* with the previous quantum distribution and a temporal *physical_processor_table* with the new processor distribution. Next phases will suspend and resume kernel threads to enforce the new processor distribution.

Selecting the set of kernel threads to execute each job (which kernel threads)

In the previous phase, the CPUManager has selected the set of physical processors that will run on each application. In this phase, it selects the set of kernel threads that will run from each application.

We have decided that, at any moment, each application will have as many kernel threads running as physical processors assigned, trying to execute in a efficient *operating point*, this is the "*process control*" approach proposed by Tucker and Gupta in [105]. Following the criteria commented in the introduction, the CPUManager will select the kernel threads associated with the first N work queues.

Figure 3.8 shows the algorithm used by the CPUManager to decide which kernel threads will run in the next quantum.

```
input:  job_table  (jobs),  temporal  physical  processor  table  (next_phys_proc),  table  with
number of processors per job (alloc)
output: job_table (jobs)
select_kernel_threads()
{
  for(current_job=0;current_job<active_jobs;current_job++){
    for(kthread=0;kthreads<alloc[job];kthreads++)
      jobs[current_job].kernel_threads[kthread].tmp_status=SELECTED_TO_RUN;
    for(kthread=alloc[job];kthreads<MAX_KTHREADS;kthreads++)
      jobs[current_job].kernel_threads[kthread].tmp_status=SELECTED_TO_SUSPEND
  }
}
```

Figure 3.8: Kernel thread selection algorithm

Mapping the kernel threads to physical processors (map kernel threads)

Once decided which physical processors and which kernel threads will run in the next quantum, the CPUManager must establish the mapping among them. This phase receives the temporal physical processors table and the *job_table*, and modifies the kernel thread table of each job to decide the mapping between each kernel thread and the physical processor. The difference from the second phase is that in the second phase we select a set of processors to run the application and in this phase we assign one processor to each kernel thread. The two phases are very related but we have implemented it separately.

Figure 3.9 shows the algorithm used by the CPUManager to map kernel threads to physical processors. The CPUManager will maintain in the same cpu those kernel threads that were currently running. If the kernel thread is not currently running, The CPUManager selects the last cpu where the kernel thread ran. In the last case (it is a new kernel thread), the *select_new_cpu* function selects a cpu from the temporal *physical_processors_table* not yet allocated to any kernel thread.

Once finished this phase, the CPUManager has completely decided the new processor distribution for the next quantum, but not yet enforced it. In the next Section, we will describe the different CPUManager options to decide the moment at which the processor allocation decisions are enforced.

```
input:job_table (jobs), temporal physical processors (next_phys_proc), table with number of
processors per job (alloc)
output: job_table (jobs)
map_kernel_threads()
{
  for (current_job=0;current_job<active_jobs;current_job++){
    for(kthread=0;kthread<alloc[current_job];current_job++){
      cpu=jobs[current_job].kernel_threads[kthread].cpu;
      if (cpu==NULL){
        cpu=jobs[current_job].kernel_threads[kthread].last_cpu;
        if (cpu==NULL) cpu=select_new_cpu();
      }
      jobs[current_job].kernel_threads[kthread].cpu=cpu;
      }
    }
  }
}
```

Figure 3.9: CPUManager mapping algorithm

3.4.3 Enforcing the CPUManager decisions

Before the end of each activation, the CPUManager should enforce the processor allocation, which may involve suspending some *running* threads and resuming some *suspended* threads. This enforcement is done by using the native operating system calls. Table 3.2 shows the main system calls [46] provided by IRIX to manage processes. They are part of the system interface shown in Figure 3.1.

Table 3.2: IRIX system calls used in the CPUManager to manage processes

Functionality	System call
Resume kernel thread	unblockproc(pid)
Suspend kernel thread	blockproc(pid)
Bind kernel thread to processor	sysmp (MP_MUSTRUN_PID, cpu, pid)
Unbind kernel thread	sysmp (MP_RUNANYWHERE_PID, pid)

When the CPUManager suspends a thread, it could be executing the application code or the run-time code. If the kernel thread is executing application code, the CPUManager may suspend a kernel thread when holding a lock or just before ending its work. Suspending a kernel thread in one of these points could be critical, since it may cause the remaining threads to be delayed.

In order to avoid such problems, the CPUManager can work in two different modes, according to the moment when its decisions are enforced:

•*Immediate* mode: the allocation is effective before the end of the current activation of the CPUManager. CPUManager decisions are enforced in a synchronous way before sleeping for the next quantum.

•*Deferred* mode: the allocation is effective, at least, at the beginning of the next CPUManager activation. Changes have to be carried out by the applications themselves. At the beginning of each activation, the CPUManager checks that all their decisions made at the previous activation have been accomplished. This mode corresponds to the *two_minute_warning* mechanism proposed by Markatos et al. in [61].

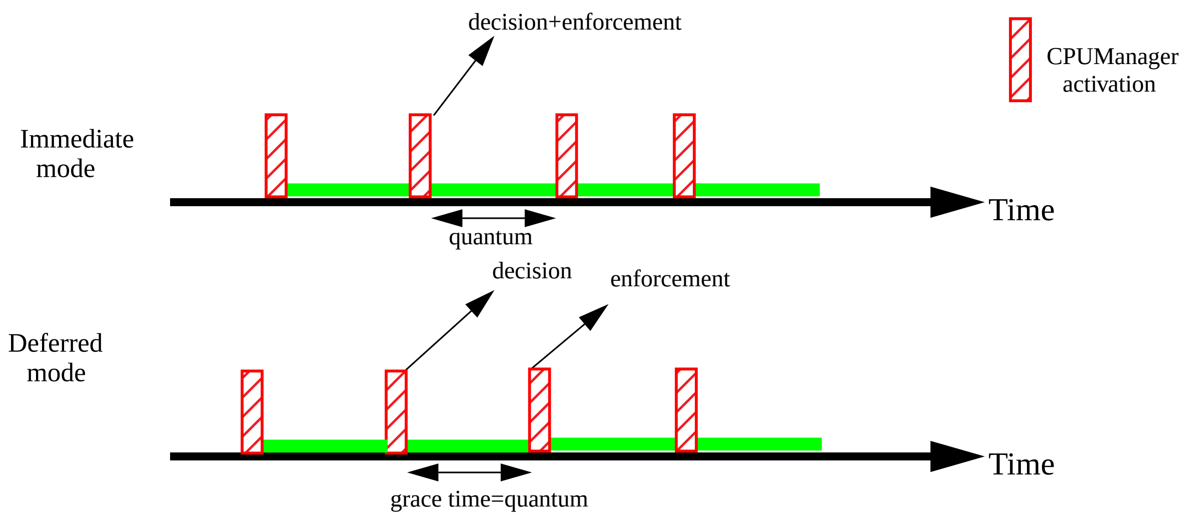

Figure 3.10: Immediate mode vs. Deferred mode

Figure 3.10 shows the different points at which the CPUManager decisions are enforced in the immediate mode and in the deferred mode.

Immediate mode

When the CPUManager works in immediate mode, it enforces the allocation decided just before the end of the current activation. When the CPUManager decides to take out some processors from an application, it changes the status of the selected threads from *running* to *suspended* (we refer to this situation as a *preemption*, and to these threads as *preempted*). Preempted threads must be recovered by the parallel library because they could be executing application code at the time of the preemption. To recover a preempted thread means that a *running* thread from the same application has to *handoff* its processor to the preempted thread until it finishes the work that it was executing. Figure 3.11 shows the function that enforces the CPUManager when executing in immediate mode.

```
input:     jobs_table(jobs),     physical_processor_table     (phys_procs),     temporal
physical_processor_table (next_phys_procs)
output: jobs_table(jobs), physical_processor_table (phys_procs)
enforce()
{
  for(cpu=0;cpu<MAX_CPUS;cpu++){
    phys_procs[cpu].last_appl=phys_procs[cpu].current_appl;
    phys_procs[cpu].current_appl=next_phys_procs[cpu];
  }
  for (current_job=0;current_job<active_jobs;current_job++){
    for(kthread=0;kthread<MAX_KTHREADS;kthread++){
      cpu=jobs[current_job].kernel_threads[kthread].cpu;
      pid=jobs[current_job].kernel_threads[kthread].pid;
      // One more processor
      if ((jobs[current_job].kernel_threads[kthread].tmp_status==SELECTED_TO_RUN) &&
           (jobs[current_job].kernel_threads[kthread].status==PREEMPTED)){
        jobs[current_job].kernel_threads[kthread].status=RUNNING;
        jobs[current_job].current++;
        sysmp (MP_MUSTRUN_PID, cpu,pid); //Binds the kernel thread to the cpu
        unblockproc(pid); // Resumes the kernel thread
      }
      // One less processor
      if ((jobs[current_job].kernel_threads[kthread].tmp_status==SELECTED_TO_SUSPEND) &&
           (jobs[current_job].kernel_threads[kthread].status==RUNNING)){
        jobs[current_job].kernel_threads[kthread].last_cpu=cpu;
        jobs[current_job].kernel_threads[kthread].cpu=NULL;
        jobs[current_job].current--;
        jobs[current_job].kernel_threads[kthread].status=PREEMPTED;
        sysmp (MP_RUNANYWHERE_PID, cpu,pid); //Unbinds the kernel threads
        blockproc(pid); // Suspends the kernel thread
      }
    }
  }
}
```

Figure 3.11: CPUManager enforcement function

If the number of processors assigned to an application changes frequently[2], the immediate mode can cause a significant number of thread migrations in order to recover preempted threads. To minimize these inopportune preemptions, we have implemented the deferred mode.

Deferred mode

The *deferred* mode is implemented through the *two_minute_warning* technique. It is a technique to minimize undesirable preemptions. The idea of this technique is to inform parallel applications that they have to release some processors, and give them the opportunity to release the processors while running in a safe state. The CPUManager informs applications setting a flag indicating that they have to release some of their processors, and applications have to check this flag.

2. This is not a desirable situation but depends on the scheduling policy

Applications are granted a *grace-time* by the CPUManager to release the processors. If this *grace-time* expires, and any application has not released its processors, the CPUManager will preempt its processors and will enforce the re-allocation.

Deciding which processors must be released is not a trivial task. The first option is that the CPUManager decide the set of processors to be released. The second option is that the parallel application selects the processors itself.

In our particular implementation, we have implemented the first approach. We have taken this decision because the CPUManager has a more global view of the system and its capacity to decide which processors have to migrate is better than the local view of the application.

The CPUManager provides applications with a list specifying the processors that they have to release, and a list with the applications that will receive each processor. With this information, applications can release a processor and allocate it to the new owner. In order to make easy the implementation, we have considered that the *grace-time* be equal to a quantum. We also assume that all the applications behave *friendly* and they will respond to the CPUManager request as soon as possible.

This technique involves cooperation among the CPUManager and parallel applications. The communication between the CPUManager and the run-time library has been implemented using shared-memory.

Since the *two_minute_warning* does not entirely avoid the inopportune preemptions, a recovery mechanism is still needed.

The *two_minute_warning* reduces the number of preemptions, which means less preempted threads and less thread migrations to recover the preempted threads. On the other hand, this technique works asynchronously. That means that the recovery mechanism must be very accurate. If the recovery mechanism is not very accurate, there is a risk that it enters in a cyclic phase, all the threads recovering all the threads. The part of the recovery mechanism implemented in the NthLib is explained in Section 3.5.1 .

We have selected the *two_minute_warning* mechanism because it is an accepted proposal in the bibliography. However, we have observed that in our execution environment, and with the processor scheduling policies implemented, the two-minutes warning mechanism does not introduce significant benefits. We have observed the behavior of some workload executions to see why this mechanism does not introduce significant benefits. The two-minutes warning mechanism was designed for an execution environment without coordination and where processor re-allocations where frequent. In our execution environment, processor scheduling policies try to maintain stable, as much as possible, the processor allocation and to coordinate the different scheduling levels. This observation demonstrate us, with a real example, that in an execution environment with coordination between levels and inside levels, some techniques that shown very important in other environments, in our case are not needed. This result, enforces our Thesis about the necessity of a coordinated execution environment and a global design of scheduling policies.

For these reasons, even we have implemented it, we have not used the *two_minute_warning* mechanism in the evaluation of this Thesis.

3.4.4 Interface between the CPUManager and the Launcher

The CPUManager implements five functions to control the multiprogramming level: *Init_Multiprogramming_Level*(), *Dynamic_Multiprogramming_Level*(), *StartAppl*(), *NewAppl*(), and *EndAppl*().

Figure 3.12 shows the functions implemented by the CPUManager. The *Init_Multiprogramming_level*() function initializes the multiprogramming level value (*ML*) to a *Default* value, defined by the administrator, sets the number of running applications to 0 (*TotalAppls*), and initializes the number of pending applications to *Default*. This function is executed only once. The *Dynamic_multiprogramming_level*() function is executed at each activation of the CPUManager. This function executes the multiprogramming level policy if there are not pending applications.

In this case, we have shown the example of a fixed multiprogramming level: *Fixed_New_appl*(). This function returns TRUE if the multiprogramming level is less than a given *Default* multiprogramming level. In that case, the *ML* is increased and the queueing system is notified. The interface also includes two functions *NewAppl*(), executed each time a new application is started, and *EndAppl*(), executed each time an application finishes. These functions maintains consistent the values of *ML* and *TotalAppls*.

This multiprogramming level policy, *fixed*, has been used with the Equipartition and the Equal_efficiency evaluated in this Thesis.

```
Init_Multiprogramming_level(int Default)
{
  ML=Default; pendings=Default;
  TotalAppls=0
}
Dynamic_multiprogramming_level()
{
  if (pending==0)
    if (Fixed_New_appl()) StartAppl();
}
StartAppl()
{
  pending++;
  ML++;
  NotifyLauncher(); /* writes in the pipe */
}
```

```
NewAppl()
{
  pending--
  TotalAppls++;
}
EndAppl()
{
  TotalAppls--;
  ML--;
}
int Fixed_New_Appl()
{
  if (ML<Default) return TRUE;
  else return FALSE;
}
```

Figure 3.12: CPUManager-Launcher interface

The *Fixed_New_appl*() function will be substituted by the particular multiprogramming level policy implemented by the processor scheduling policy.

3.4.5 Interface between the CPUManager and the Run-Time library

The CPUManager basically implements the interface specified in the NANOS project to communicate the run-time parallel library with the processor scheduler. The basic set of functions that this interface defines is the following [63]:

Table 3.3: CPUManager-NthLib interface

Function	Description
int cpus_request(int ncpus)	Sets the number of processors the application would like to run on
int cpus_requested()	Informs about the number of requested processors
int cpus_current()	Informs about the number of currently assigned physical processors
int cpus_askedfor()	Returns TRUE if the calling thread is marked to be released
int cpus_release_self()	Releases the current physical processor.
int cpus_preempted_work()	Informs about the number of preempted kernel threads
work_t cpus_get_preempted_work()	Returns the identifier of a previously preempted work
int cpus_processor_handoff(work_t work)	Attempts to transfer the current physical processor to the previously preempted work
int cpus_future()	Returns the number of processors the application will have at the end of the current quantum

This interface is offered by the CPUManager to the run-time library. It is implemented through shared-memory between the CPUManager and the run-time library. In the next sub-section, we present the data structures managed by the CPUManager that implement this interface. Functions written in bold font have been implemented or modified in this Thesis[3].

We have introduced two modifications respect to the original NANOS specification. The first change is the functionality of the *cpus_askedfor*(). In the original NANOS specification *cpus_askedfor()* returns true if the application has to release some thread. In the current implementation, it only refers to the calling thread. It returns true if the calling thread is assigned to a physical processor selected by the CPUManager to be re-allocated to another application. The second difference is that the CPUManager implements a new function: *cpus_future()*, which informs about the number of processors that the application will have at the end of the current quantum.

3.4.6 Shared Data Structures

The CPUManager manages three data structures: jobs, kernel threads, and physical processors. The main fields associated to these data have been commented previously. In this Section, we detail data associated with each one. Jobs and kernel threads are shared between CPUManager and run-time. Physical processors are private to the CPUManager.

Jobs

The job is the unit of processor allocation. The CPUManager has a table of jobs, and each job has the following information:

- Job identifier
- maximum parallelism
- requested number of processors, *request*
- number of *running* threads, *current*
- processors that the job will have at the end of the next quantum, *future*
- number of preempted[4] threads, *preempted*
- list of processors to be released by the application
- speedup and execution time tables
- kernel threads table

Each job has a unique identifier automatically generated by the CPUManager. When the application starts, it communicates its maximum parallelism to the CPUManager. The maximum parallelism is the maximum number of processors that the application will require. Therefore, the application can dynamically change its processor request from 1 to its maximum parallelism. Its dynamic processor request is set in the *request* field, and it is one of the parameters that the processor allocation policy takes into account when it distributes processors. For instance, the application can set to 1 the *request* when it enters in a sequential phase or when it starts an I/O or it can reduce its *request* when it executes a loop with only a few iterations.

Current is the number of assigned/running processors that the application has at any moment. *Future* is the number of processors that the application will have assigned/running, by the end of the current quantum. When the CPUManager runs in *immediate* mode, values of *current* and *future* are always equal because re-allocations are

3. Some of them had only been specified but not implemented.
4. A preempted thread is a thread that has been suspended by the CPUManager

synchronous. On the other hand, when the CPUManager runs in *deferred* mode, the number of assigned and running processors can be different during a *grace-time* after the execution of the CPUManager. This is because the processors are asynchronously re-allocated.

The CPUManager maintains a counter of preempted threads by application, set in *preempted*. It does not maintain a list of preempted threads since this information is available checking the status of each kernel thread. When the CPUManager preempts a thread, its work has to be recovered by the application. The applications use the *preempted* value to check whether they have any preempted thread. The CPUManager preempts a thread either when it works in *immediate* mode and an application will receive less processors in the next quantum, or when the CPUManager works in *deferred* mode and the application has not reacted to the CPUManager requirements to release a processor. Each application has associated a list of processors to be released. This list is empty when working in *immediate* mode.

The CPUManager has the speedup and execution times tables in the shared memory. This information is used by the processor allocation policies that consider performance information. This table is modified by the run-time that measures the application performance and read by the processor scheduling policy.

Finally, each application has a kernel threads table. As we pointed out in the introduction, each kernel thread is associated to a work queue, numbered from 0 to (maximum parallelism -1).

Kernel threads

A kernel thread is the unit of scheduling to which a CPU can be assigned to execute it. The CPUManager associates the following information to each kernel thread:

- pid
- status
- *cpu_id*, if RUNNING, identifier of the physical processor, otherwise a NULL value.
- last cpu where it run

Each kernel thread has a unique identifier. In our case, it is the identifier that the operating system assigns to the kernel thread (pid). If the kernel thread is RUNNING, the CPUManager updates *cpu_id* with the identifier of the physical processors where it is running.

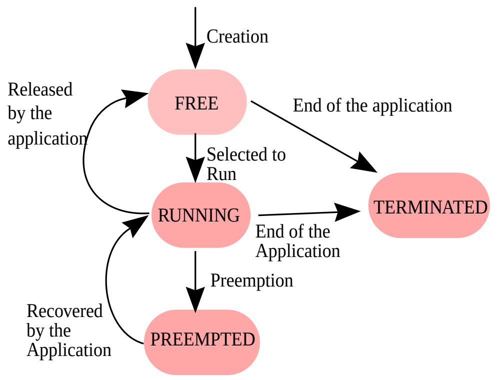

Figure 3.13: State diagram of a kernel thread

Kernel threads can be in four logical states: FREE[5], RUNNING, PREEMPTED or TERMINATED. Figure 3.13 shows the state diagram of a kernel thread. Kernel threads are created FREE when the application starts. When the CPUManager selects a kernel thread for execution, its status changes to RUNNING. If the application voluntarily releases the physical processor associated to a kernel thread and suspends it, its status changes to FREE. On the other hand, if the CPUManager suspends the thread, it becomes PREEMPTED. When an application ends its execution all its kernel threads become TERMINATED. A PREEMPTED thread must be recovered by a RUNNING thread from the same application because it could be executing application code when preempted. Finally, each kernel thread has a field that indicates the last cpu where it run.

Physical processors

The information associated to each physical processor is:

•current job, *current_appl*
•Identifier of the last application that runs in this processor, *last_appl*

If the physical processor is allocated to an application, the CPUManager will set in *current_appl* the identifier of the application. Otherwise, it will have a NULL value. The *last_appl* field contains the identifier of the last job that ran in this processor.

5. In both, *free* and *preempted* states, the kernel thread is suspended

3.5 Run-time library features

Parallel applications interact with the CPUManager through a run-time library. In this Section, we present the main features that a parallel library must include to coordinate with the CPUManager, and the main improvements introduced in this Thesis to NthLib [63][64], the run-time library used in this Thesis.

The parallel library communicates with the CPUManager in order to provide scheduling information: processor requirements, etc. And the CPUManager informs the parallel library about the number of processors assigned to it, and about the thread preemption.

The run-time library follows the rules specified in the introduction of Section 3.4.

Since the CPUManager can preempt application kernel threads, the run-time library has to provide a work recovery mechanism. The work recovery mechanism implies that some of the remaining threads must finish the pending work. This event also implies that the parallel library has to periodically check whether it has some preempted threads in order to recover it.

3.5.1 NthLib modifications

The nano-threads library, NthLib, is a user-level thread package mainly designed to support efficient parallel execution and good adaptability to the varying system conditions. It is designed primarily to support fine-grain parallelization and multiple levels of parallelism in shared-memory multiprocessors. It is further described in [63][62].

In this Thesis, we have modified the work recovery mechanism, implemented the two-minutes warning mechanism, the dynamic kernel thread creation/destruction, and activated the dynamic memory migration mechanism.

3.5.2 Work recovery mechanism

The work recovery mechanism is needed when a *running* thread is suspended by the CPUManager, becoming *preempted*. The work recovery mechanism has to guarantee that the pending work of the preempted thread is finished by another processor. This mechanism must be implemented "*with care*", avoiding entering in recovering cycles. In this Section, when we refer to kernel thread 0 (kt0), we mean the kernel thread associated to work queue 0, and so on.

,

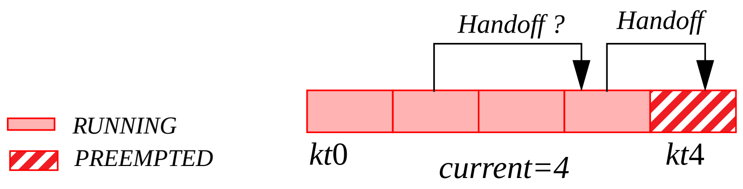

Figure 3.14: kt0 will handoff its processor to kt4 to finish its pending work.

Figure 3.14 shows an example about a possible problem related with the work-recovery mechanism. In this example, kt3 has detected that kt4 is preempted. It handoffs its processor to kt4 to finish its work. If kt1 detects that kt3 does not have processor (because it is recovering kt4), it could conclude that kt3 needs help to finish its work, when this is not the case.

This set of possibilities implies that the work recovery mechanism must specify:

- which thread can recover a preempted thread
- what recovering a thread means
- which threads need to be recovered

To specify the work recovery mechanism, we classify the kernel threads into two groups: those that are associated to work queues with identifier less than the *current* number of processors allocated, and those that are associated to work queues with identifier greater or equal than *current*. We will refer to the first set as the "*allocated zone*" and to the rest as the "*unallocated zone*".

The work recovery mechanism also introduces two new thread states: Recovering another thread and Being Recovered. The list of thread states then is the following: RUNNING, FREE, PREEMPTED, Recovering another thread (RECOVERING) and Being Recovered (BEING_RECOVERED).

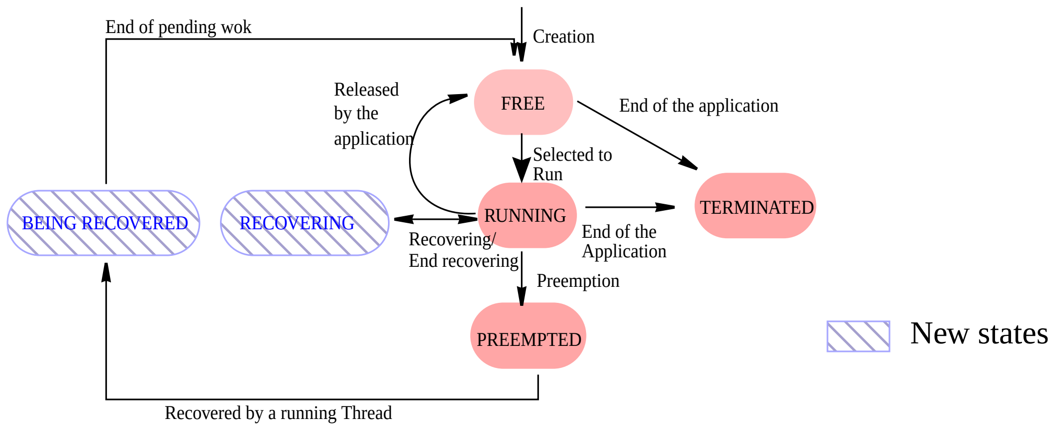

Figure 3.15: State diagram of a kernel thread with recovering mechanism modifications

Figure 3.15 shows the state diagram of a kernel thread after incorporating the new states resulting of the work recovering mechanism. We specify that only threads in the allocated zone and in state RUNNING can recover another thread. And the only threads that need to be recovered are threads in the unallocated zone in state PREEMPTED.

When a thread finishes its pending work, it executes a piece of code called *idle loop*. The *idle loop* continuously checks whether there is pending work and whether there are preempted threads.

When the kernel thread enters in the idle loop, it can be in four different situations (each kernel thread must check these situations and in this order):

- Marked as BEING_RECOVERED
- RUNNING, and there are PREEMPTED threads that need help
- RUNNING, and there are not PREEMPTED threads, (the normal / stable situation)

If the kernel thread is BEING_RECOVERED, it must handoff its processor to the kernel thread that previously helped it. To know that, the run-time has a per thread field (*return_to*) that specifies to which kernel thread it has to return.

If the kernel thread is RUNNING, it has to check if there are PREEMPTED threads that need help. In that case, if the work queue of the thread is in the allocated zone, it will get the first PREEMPTED thread, it will mark itself as RECOVERING, the PREEMPTED thread as BEING_RECOVERED (setting the *return_to* field), and finally, it will handoff its processor to the PREEMPTED thread.

If the thread is RUNNING and there is not PREEMPTED work, the thread has to iterate in the idle loop until it receives new work to execute.

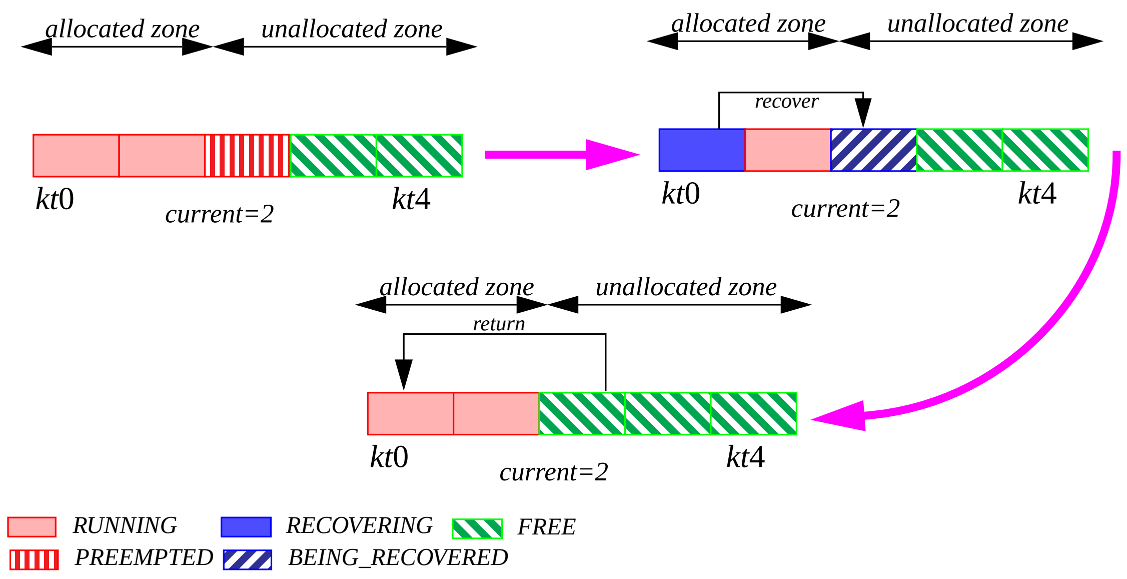

Figure 3.16: Recovery mechanism, kt0 recovers the work of the kt4 and returns to its queue.

Figure 3.16 shows an example about how the work recovery mechanism works. When kt0 detects that kt2 is preempted, it handoffs its processor and marks kt4 to return to work queue 0. When kt2 finishes its pending work and enters in the idle loop, it detects that it has to return to work queue 0, and then it handoffs its processor to kt0.

Figure 3.17 shows the work recovery algorithm: the *idle loop*, and the *cpus_get_preempted_work*() function.

```
idle_code()
{
  ...
  // If I was being_recovered I must return to the original work queue
  if (myself->status==BEING_RECOVERED){
    myself->status=FREE;
    my_job->kernel_threads[myself->return_to]->status=RUNNING;
    cpus_handoff(myself,my_job->kernel_threads[myself->return_to]);
  }
  // If there are preempted threads I must recover it,
  if ((cpus_preempted()>0) && (myself->wq_id<cpus_current())){
    // Only stable threads can recover preempted threads
      work_preempted=cpus_get_preempted_work();
      myself->status=RECOVERING;
      // The kernel thread must return to the original work queue
      work_preempted->return_to=myself->wq_id;
      work_preempted->status=BEING_RECOVERED;
      my_job->preempteds--
      // The kernel thread is lended
      cpus_handoff(myself,work_preempted);
  }
  ....
}
kthread * cpus_get_preempted_work()
{
  // The algorithm look for a thread in the unallocated zone and PREEMPTED
  for (kthread=cpus_current();kthread<MAX_THREADS;kthread++){
    if (my_job->kernel_threads[kthread]->status==PREEMPTED)
      return my_job->kernel_threads[kthread]
  }
}
```

Figure 3.17: Work recovery mechanism

3.5.3 Two_minute_warning mechanism

The two_minute_warning mechanism implies modifications in the CPUManager and in the run-time library: Decisions that in the immediate mode are executed synchronously by the CPUManager, in the deferred mode are executed asynchronously by the run-time library. These modifications introduce two main problems: in one hand the asynchronous behavior, and in the other hand the fact of giving responsibility to the run-time.

The deferred mode modifies the idle loop. In this mode, kernel threads have also to check if they are marked to be released. They will check this condition before checking if there are PREEMPTED threads.

The fact of releasing the processor to another application is much more complicated that it initially seems. There are several possible implementations of this technique, but we have decided that the run-time library will implement the same policy than the CPUManager. To do that, the run-time library needs access to the information of the rest of applications to have access to their kernel thread tables (pid, status, etc). The run-time also needs a new information: the CPUManager informs it about which specific kernel threads must be released and to which job it must be allocated. Since this is a research environment, we have not taken into account security issues and we have made accessible all the CPUManager data structures to the run-time library. Another approach could consist of allocating the application kernel thread to an idle thread of the CPUManager and that this thread performs the context switch to the new job. However, we have adopted the first approach: allocation decisions have been moved from the CPUManager to the run-time library.

```
while (application not finished){
...
 if (cpus_asked_for())
   cpus_release_self()
....
 if (pending_work())execute(work)
}
```

Figure 3.18: Modifications in the idle loop

If the physical processor is marked to be released, the executing thread has to allocate its processor to the new owner and deallocate itself, see Figure 3.18. The *cpus_asked_for*() function returns TRUE if the kernel thread has been marked to be released. In that case, it executes the *cpus_release_self*() function. This function looks into the temporal processor allocation table to see the target application associated with it and applies the same processor placement algorithm implemented by the CPUManager presented in Section 3.4.2.

The two_minute_warning mechanism also affects the work recovery mechanism. When working in deferred mode, the criteria to differentiate between the allocated and unallocated zone can not be the value of *cpus_current()* because there is a fraction of time in which this value changes frequently (when the application is releasing threads).

The solution is to use the value of the *cpus_future*() function to differentiate the allocated zone and the unallocated zone. The *cpus_future*() function returns the number of processors that will be allocated to the application at the end of the current quantum. In this case, threads that can recover PREEMPTED threads are those that are allocated to work queues with identifier less than *cpus_future*().

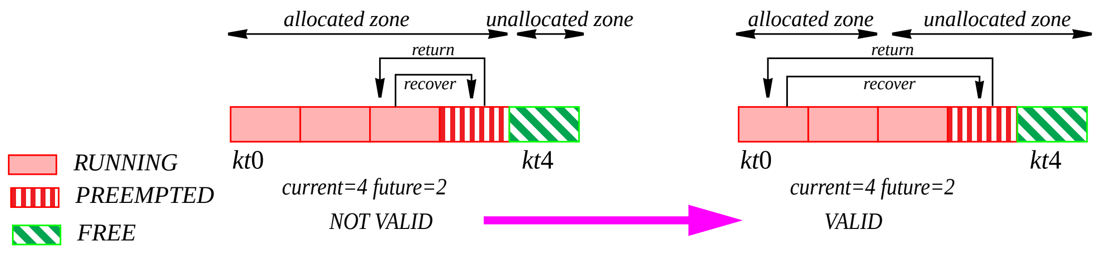

Figure 3.19: Modifications in the work recovery mechanism introduced by the deferred mode.

Figure 3.19 shows an example about the necessity of modifying the work recovery mechanism. In the left side, there is an example about a possible situation when executing in deferred mode and directly applying the work recovering mechanism explained in the previous Section. In this case, kt3 is in the allocated zone and it detects that there is a PREEMPTED thread. However, kt2 is a kernel thread that has to release its processor, and it can not recover PREEMPTED kernel threads.

On the right side of Figure 3.19, there is an example about how the work recovery mechanism works when executing in deferred mode with the modification introduced in this Section. The allocated zone is determined by cpus_future(), not cpus_current(). The cpus_future() value is fixed during the complete quantum. Using this value, the only kernel threads that can recover PREEMPTED threads are kt0 and kt1.

3.5.4 Demand based thread creation

We have also incorporated a mechanism to dynamically create and destroy the kernel threads in NthLib. The aim of this technique is to adjust, as much as possible, the number of kernel threads to the number of processors allocated.

This technique is motivated by the fact that it is a common practice to statically generate kernel threads. Usually, run-time libraries create as many kernel threads as the maximum parallelism specified by the user at the submission time. This implementation is efficient enough if the total number of requested processor by active applications is not very high. The problem is that in the case of dynamic processor scheduling policies, it is possible to reach situations such as a job executing in 4 processors and with 64 kernel threads created. In this kind of extreme situations, it can result in a system saturation due to the lack of resources, for instance filling the complete process table.

Our proposal is to dynamically create and destroy the kernel threads. We propose to create a few more kernel threads than processors allocated to the job, limited by the maximum parallelism specified by the job.

The CPUManager implementation is affected by this modification because we decided that the CPUManager performs the allocation of processors to processes. With this new mechanism, the CPUManager can try to allocate a processor to a work queue and it can find that the associated kernel thread is not yet created. To manage this new situation, we have modified the status diagram associated to kernel threads, and we have included more information in the interface between the CPUManager and the run-time library.

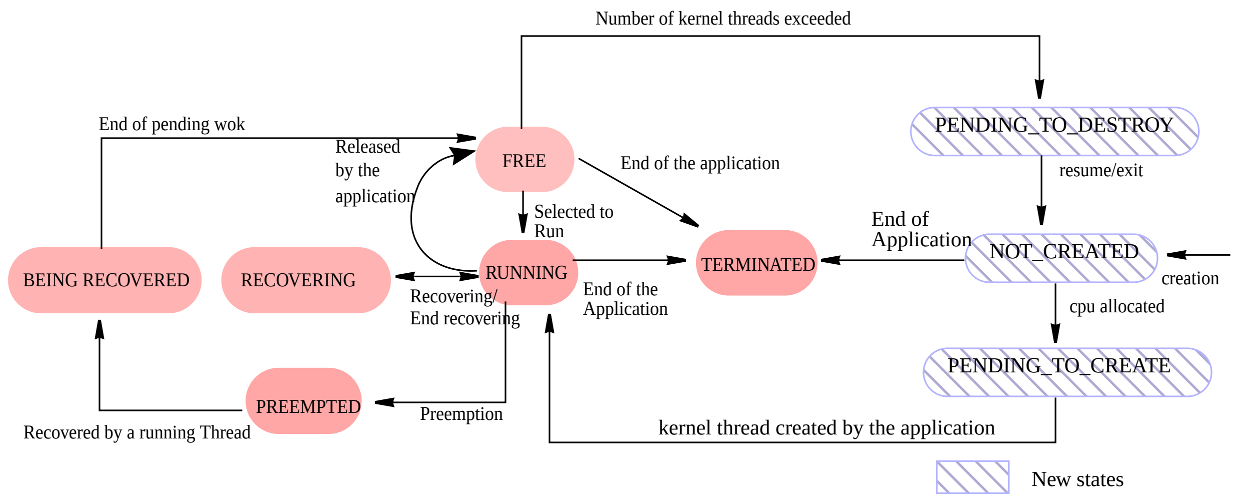

Figure 3.20: Kernel thread state diagram with dynamic thread creation

CPUManager modifications

We have included three new states: NOT_CREATED, PENDING_TO_CREATE, and PENDING_TO_DESTROY, see Figure 3.20.

We have also included two new fields per job in the job table, *cpus_pending* and *cpus_pending_to_release*. The *cpus_pending* is the number of kernel threads pending to be created by the application. This value will be consulted by the run-time through a new function in the CPUManager-run-time interface: *cpus_pending*(). The *cpus_pending_to_release* is a boolean that indicates that the application has an excessive number of kernel threads, compared with the number of cpus allocated, and that it must destroy some kernel threads. This value is consulted by the run-time library through the *cpus_pending_to_release*() function.

When the application starts, NthLib initializes the data associated with the first work queue with the information of the main process. The rest of work queues, do not have kernel threads associated to them. These work queues are initialized as NOT_CREATED.

When the CPUManager allocates more processors to the application, it follows the algorithm presented in Section 3.4.2. However, it checks, per work queue, if the associated kernel thread is created or not. If the kernel thread is not created, the CPUManager marks the kernel thread as PENDING_TO_CREATE and increments the *cpus_pending* value.

The last state, PENDING_TO_DESTROY, is set by the NthLib. When the CPUManager considers that the application has an excessive number of kernel threads (compared with the number of cpus allocated). In that case, the CPUManager sets the *cpus_pending_to_release* field. Then, when the NthLib detects that it has to release some kernel threads, it selects some of them and marks them as PENDING_TO_DESTROY.

We have implemented the decision of what is an excessive number of kernel threads using two different thresholds: a global and a local threshold. If the ratio between the total number of processes created and the number of physical processors is greater than the global threshold, the CPUManager will consider that the system is heavily loaded. In that case, it calculates the ratio of (processes created / processors allocated) per application. If this second ratio is greater than the local threshold, the CPUManager sets the *cpus_pending_to_release* of this application. In the current implementation, the global threshold has been set to 2 and the local threshold to 1.25.

Moreover, we use a third level of system load control. The CPUManager implements a *security threshold* to avoid a system crash due to lack of resources (kernel threads in this case). The CPUManager calculates the relationship between number of kernel threads and number of cpus. If this ratio is greater than the *security threshold*, no new applications are allowed to execute. This *security threshold* works coordinated with the multiprogramming level and modifies any multiprogramming level decision. In the current implementation, the *security threshold* has been set to 6.

Run-time library modifications

We have also modified the run-time library to periodically check if the job has kernel threads pending to create, or if it has to destroy kernel threads. This code has been introduced in the idle loop. If *cpus_pending*() returns a value greater than zero, the run-time will look for all the kernel threads in PENDING_TO_CREATE state, and it will create one kernel thread per work queue. Once created, the run-time decrements the counter *cpus_pending*, changes the thread state from PENDING_TO_CREATE to RUNNING, and resumes the kernel thread.

More complicated is the dynamic destruction of kernel threads. As in the previous case, we have introduced some code in the idle loop to detect if the job has to reduce the number of kernel threads (checking the *cpus_pending_to_release*() function). In that case, kernel threads associated with work queues identified with the highest numbers are selected to be destroyed. The number of kernel threads selected are calculated using the local threshold. Each kernel must destroy itself: the run-time marks selected threads as PENDING_TO_DESTROY, and resumes it. When kernel threads enter in the idle loop, they detect that they must exit.

Table 3.4 shows the IRIX system calls used to create and destroy kernel threads.

Table 3.4: IRIX system calls

Functionality	System call
Create kernel thread	sproc(...)
Delete kernel thread	exit(..)

3.5.5 Memory management

In CC-NUMA architectures, processes should be scheduled in processors near their memory pages to achieve a good system performance. There are two ways to enforce that: doing a good initial memory mapping, and using memory page migrations. The first choice is only valid if the application memory accesses follow a static pattern. This technique needs an individual analysis of each parallel application, and the processor allocation must be fixed during the complete execution of the application. The second choice, memory migrations, will allow us to work without any knowledge about the application memory behavior.

The native operating system used in this Thesis, IRIX 6.5, has a *dynamic page migration* mechanism. Dynamic page migration is a mechanism that provides adaptive memory locality to applications that execute in a NUMA machine. The migration mechanism checks the memory pages and decides whether a page must be migrated, depending on a migration policy. In this Section, we present the modification that we have introduced in the run-time to activate the dynamic page migration mechanism.

Policy Modules

Users are allowed to select a policy from a set of available policies for each virtual memory operation. Virtual memory operations are: Initial allocation (Placement policy, page size policy, and fallback policy), Dynamic relocation (migration policy and replication policy), and paging policy. Any portion of a virtual address space, down to the level of a page, may be connected to a specific policy via a Policy Module.

When the operating system needs to execute an operation to manage a Section of a process address space, it uses the methods specified by the Policy Module connected (attached) to that Section.

In this Thesis, we have modified the placement policy and the migration policy associated with all the code, data, and stacks of parallel applications.

Table 3.5: Policy modules functions

Function name	Description
pm_filldefault(...)	Fills a policy_set with predefined default values
pm_create(...)	Creates a policy module
pm_setdefault(...)	Selects a new default policy for stack, text, or heap.
pm_attach(...)	Connects a policy module to a virtual address space range

Table 3.5 shows the Policy Module functions used in the run-time library to modify the placement policy and the migration policy. The *pm_filldefault*() function fills a policy_set_t structure with the predefined values in the system. A policy_set_t structure has fields to

specify the following policies: placement policy name and arguments, fallback policy name and arguments, replication policy name and arguments, migration policy name and arguments, paging policy name and arguments, and page size.

Once we have a default Policy Module we can modify it. In this Thesis, we have modified the placement policy fields and the migration policy fields. We have set the placement policy to *PlacementFirstTouch* and the migration policy to *MigrationControl*. *PlacementFirstTouch* indicates that memory will be allocated in the node where creation happened. *MigrationControl* indicates that users can specify different migration parameters. The *MigrationControl* policy receives as argument a data structure that defines these user parameters. Once modified the Policy Module, we have to create the handle that we will use to associate to the memory regions. The Policy Module creation is done using the *pm_create*(..) system call, that receives as a parameter the Policy Module previously filled up.

Once created, it only remains to attach the Policy Module with the memory regions. In this case, we have created only one Policy Module because we want to apply the same policy to all the application memory regions. The association of a Policy Module with a memory region is done through the *pm_attach*( ) function. This function receives as parameter the Policy Module, the address of the memory region, and the size of the memory region.

The *pm_setdefault*() function associates a policy module to the stack, text, or heap memory regions. This function has been used to modify the default policy associated to these memory regions.

Migration parameters

The SGI Origin2000 hardware implements a competitive algorithm based on comparing the remote memory access counters to the local memory access counters. When the difference between remote and local accesses is greater than a tunable threshold, an interrupt is generated to inform the Operating System than the physical memory page is suffering an excessive number of remote references. The interrupt handler decides whether the page has to be migrated or not. The final decision depends on several controls that can limit the page migration.

Figure 3.21 shows the sequence of code used in the NthLib to create a PM with the dynamic memory migration mechanism activated. We first fill the Policy Module with the default values, and modify the placement and migration policies. Once filled up, we create it with the *pm_create*() function. The Policy Module created is used to modify the policy associated to code, data, and stacks. This default policy is inherited at *fork* or *sproc* time, and a new one is created at exec time.

The last memory regions that we have to modify are user stacks. These stacks are not allocated from the heap, then we have to attach them explicitly using the *pm_attach*() function. The *pm_attach*() function must be executed once per allocated user stack.

```
migr_policy_uparms migr_args;
policy_set polset;
pmo_handle_t PM;
pm_filldefault (&polset);

migr_args.migr_base_enabled = 1; // The rest of fields are default values
migr_args.migr_base_threshold = 50;
migr_args.migr_freeze_enabled = 0x0;
migr_args.migr_freeze_threshold = 0x14;
migr_args.migr_melt_enabled = 0;
migr_args.migr_melt_threshold = 0x32;
migr_args.migr_enqonfail_enabled = 0;
migr_args.migr_dampening_enabled = 0;
migr_args.migr_dampening_factor = 0x5A;
migr_args.migr_refcnt_enabled = 0;
polset.placement_policy_name=”PlacementFirstTouch";
polset.placement_policy_args =NULL;
polset.migration_policy_name = “MigrationControl”;
polset.migration_policy_args = &migr_args;
PM = pm_create (&polset);
pm_setdefault(PM,MEM_STACK);
pm_setdefault(PM,MEM_TEXT);
pm_setdefault(PM,MEM_DATA);
....
pm_attach(PM,stack_address,page_size);
....
```

Figure 3.21: Policy Module creation and migration mechanism activation

Activating the dynamic page migration mechanism, memory pages can be automatically migrated by the system. Experiments done in this Thesis have been performed activating this mechanism because it has been shown useful. Our experience also shows that this mechanism is useful in those environments where the process reallocation frequency is not very high. Otherwise, the mechanism is not able to migrate memory pages to follow the movements of the processors.

3.6 Summary

In this Chapter, we have described the three main elements in our execution environment: The queueing system (Launcher), the scheduler (CPUManager), and the run-time library (NthLib).

The Launcher controls the arrival of parallel applications and implements the job scheduling policy. The job scheduling policy decides the order of application execution. The Launcher has allowed us to execute workloads in a controlled way, that means under certain load conditions and with a specified job scheduling policy. Using the Launcher we have evaluated the processor scheduling policies presented in this Thesis under the same execution conditions.

The CPUManager is a user-level scheduler that implements the processor allocation policy and enforces its decisions. In this Chapter, we described the different phases that implements the CPUManager and the decisions adopted on each one.

Finally, we presented characteristics that a run-time library must accomplish to interact with the CPUManager. We present the particular case of the NthLib, the run-time library used in this Thesis. We have introduced several modification such as the implementation of the two_minute_warning mechanism, the work recovering mechanism, the memory page migration, or the dynamic thread creation. We have give a lot of details about our processor scheduler implementation and about issues that usually are not described, such as the processor placement function, but that in fact, have a great influence in the performance of the scheduling policies.