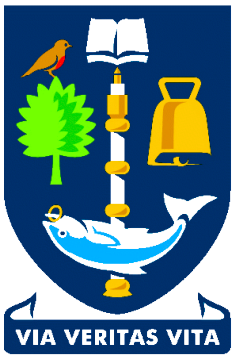


University of Glasgow

SCHOOL OF ENGINEERING

MSc PROJECT REPORT

AI-Linux

Implementing an AI with the kernel of an operating system

Author:
Nathan LORETAN 2348246

Supervisor:
Dr. Nicholas BAILEY

A thesis submitted in fulfilment of the requirements for the degree of
MSc Computer System Engineering

August 20, 2018

Abstract

AI-Linux is a research project that tries to implement an Artificial Intelligence within the kernel of an operating system, In this case, the Linux kernel.

The objective of this project is to design artificial agents that would change their behaviour regarding the utilization of the kernel. The design is focus on three parts of the kernel (process scheduler, load balance and page frame reclaiming algorithm).

All the agents use reinforcement learning (action, state, rewards) to learn the best way to complete their tasks during the execution of the kernel. For this project, the environment is considered to be dynamic. Hence, the number of processes running, their priority, the pages used are unknown and the state space and action space are constantly changing. Therefore, the design is focus on developing value function approximation to determine the Q-Value function of a state and an action based on different features to select the best move to do.

The results show that the agents can adapt its behaviour differently regarding the type of environment, resources used and processes running on the Operating Systems by giving different weights to the features selected. However, the measurements where performed in a controlled environment with a well-defined number of processes and action to execute. Further work has now to be done to test the design in a more real-world environment and to finally measure the actual performances of the design to fully evaluate it.

Acknowledgements

I would like to express my thanks towards my advisor for accepting to supervise my MSc project proposal. I would also like to thank him for the guidance and advises he provided.

Furthermore, I would also like to express my gratitude to the school of engineering that accepted my MSc project proposal and give me the opportunity to explore a subject that interested me.

Contents

List of Figures

List of Tables

List of Abbreviations

OS	**O**perating **S**ystem
CPU	**C**entral **P**rocessing **U**nit
FPU	**F**loating **P**oint **U**nit
PFRA	**P**age **F**rame **R**eclamation **A**lgorithm
GLP	GNU **G**eneral **P**ublic **L**icense
AI	**A**rtificial **I**ntelligence
RL	**R**einforcement **L**earning
MDP	**M**arkov **D**ecision **P**rocess
CFS	**C**ompletely **F**air **S**cheduler
SFS	**S**martly **F**air **S**cheduler
SB	**S**mart **B**alance
LRU	**L**east **R**ecent **U**sed
SPFRA	**S**mart **P**age **F**rame **R**eclaiming **A**lgorithm

Nomenclature

Reinforcement Learning

a	action
s	state
a'	next action
s'	next state
$V(s)$	Value Function of s
$V^*(s)$	Optimal Value Function of s
$Q(s,a)$	Action-Value Function of s and a
$Q^*(s,a)$	Optimal Action-Value Function of s and a
π	Policy
π^*	Optimal Policy
G_t	Return
$R(s)$	Reward in s
$R(s,a)$	Reward by taking a in s
α	Learning factor
γ	Discount factor
E	Error

Smartly Fair Scheduler

T_i	ith task
s_i	state of T_i
ta_i	utilization target of T_i
$F(s)$	fairness of state s
$f_i(s)$	fairness of state s regarding T_i
$D(s)$	CPU distribution of state s
$d_i(s)$	CPU distribution of state s regarding T_i

Smart Balance

T_i	ith task
C_y	yth CPU
t_i	state of T_i
c_y	state of C_y
$l_i(s)$	load added by T_i
$L_x(s)$	load on C_y

Smart Page Frame Reclaiming Algorithm

p_i	state of the ith page
q	state of the page queue

1 Introduction

Nowadays, almost all applications using a μController or μProcessor, from a car to a space rocket, implement an **Operating Systems (OS)** or **Real-time Operating Systems (RTOS)**. OS and RTOS are employed for a large variety of uses (embedded systems, servers, desktop computers) and are implemented on a lot of different hardware (architecture, number of CPUs, memory design, size of memory available) which represents the environments on which they are evolving.

However, most of the operating systems, like Linux, use the same algorithms to handle the resources (CPU, primary memory) when running on different environments or require to be customized to be employed for specific applications. But as the number of specific applications is constantly increasing because of the ubiquity of computer systems, the number of specific OS keeps increasing as well.

Furthermore, it is difficult for an OS to consider how the different running programs could influence each other regarding the environment by, for example, acquiring locks and blocking the others, modifying cache, or swapping out pages used by other programs. But, how could they?

Of course, considering the different programs and how they influence each other would imply to hard code solutions for too many different cases which is almost impossible. Fortunately, during the last decades, **Artificial Intelligences (AI)** have showed good opportunities to create adaptable systems that learn the best way to execute jobs in different environments regarding features present in each of them.

Hence, the idea of the project is to research the possibility of using artificial intelligence, agents, that learn how to attribute the resources among the running programs to create an adaptative operating system regarding the environment in which it is evolving. Unfortunately, creating a usable OS from scratch would take too much time that what is allocated for this project and it is interesting to have a usable one at the end of this project. Therefore, the AI is implemented within the well-known Linux kernel. The artificial intelligence uses **Reinforcement Learning (RL)** technics that allow the AI to adapt and learn directly when running in the environment without any prior examples and training.

The design of the artificial intelligence is focus on three specific parts. Process scheduling that determines the order of execution of the processes on the CPU. Process migration that determines on which CPU should be executed a process. Page reclaiming that determines which pages in the RAM to swap out to the disk.

The project is mostly focus on the two first parts, process scheduling and process migration, where the design of the artificial intelligence for each part has been implemented and tested. For the part page frame reclaiming, only an idea of design is presented.

The interesting part of this project is not in the performance that could be obtained by the artificial intelligence but mainly its behaviour to see if it can learn differently regarding the environment. Developing an operating system is difficult and involves a lot of elements to consider. Therefore, it is impossible to get good performance with the small time allocated for this project. Most operating systems spend years to improve their performance and to implement completely stable versions by team of hundreds of people. Furthermore, learning technics use complex calculation, real numbers, and a large amount of memory which already bring a lot of complexity as kernel programming requires:

- Implementing learning methods without floating point unit.
- Implementing learning methods with as less memory as possible.
- Implementing learning methods without long calculation time.

Another important point of this project is to keep the development of the artificial intelligence considering a dynamic environment. This increases the difficulty of the design of the artificial intelligence but corresponds more to the real-world of the operating systems where programs are started and stopped constantly, or the amount of memory used in memory is always changing.

The three first chapters of this project brings basis theory about operating systems, Linux, and Artificial Intelligence to help understand the development of this project. Chapter fourth explains the design of the artificial intelligence. Chapter fifth explains the implementation of the artificial intelligence in the Linux kernel, where they are acting, and the chapter sixth shows the result of their behaviour. Finally, the two last chapters give an overview of the future work and a conclusion to this project.

For this project, a GitHub repository is created at `https://github.com/nathanLoretan/MSc_Project` which include a kernel patch with the modifications, measurement scripts, and the report.

2 What is an Operating System ?

Operating systems are used for a large variety of applications such embedded systems, desktop computers or servers. But what do exactly an operating system?

The purpose of an operating system is to manage the computer's hardware and to provides an interface to allow user applications to interact with it. This interaction is performed by the kernel. Therefore, the operating system is built with two distinct parts the **User space** and the **Kernel**, figure 2.1.

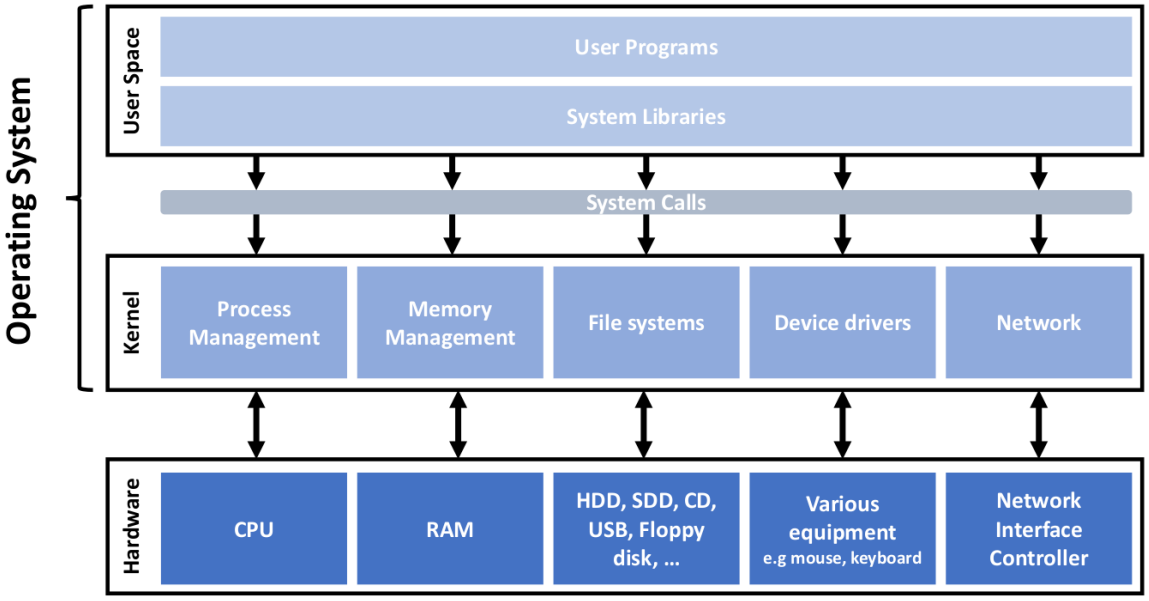


FIGURE 2.1: Simplify view of an operating system architecture

The user space includes the programs used by the user and the system libraries. The system libraries consist to set of functions and tools that allow programs to use the operating system. The user space can't perform direct actions on the hardware. Instead, it uses system calls, provided by the system libraries, which send requests to the kernel.

The kernel manages the different resources of the hardware. For example, it decides where in memory should a user program be placed, on which CPU it should run and how the files and folders should be organized in the memory. The kernel can be separated in five different parts, **Process Management**, **Memory Management**, **File systems**, **Device drivers** and **Network**. In this project, only the parts process management and memory management will be modified and explained in the following sections.

2.1 Process Management

The purpose of the process management is to create, delete and attribute the resources to different **processes**. The process is a program or a part of a program that is executed and is also called **task**. Both terms, task and process, are going to be used in this report, task is generally the term used in the Linux kernel.

Usually, a program is composed of several processes that will be executed together. Each process corresponds to a specific work, a set of instructions to execute when the program must perform a specific job. When the program requires to do this specific job, a new process is created and when the process has finished to work, the process is deleted or killed.

The process is executed on a **Central processing unit (CPU)** which executes the different instructions. The CPU can only execute one process at a time. However, modern computers use several CPUs or a CPU with several **Cores** which means that the CPU has several computing units. Only the term CPU is going to be used in this report.

A simple CPU can only execute one process at a time. But the user wants to see them executing in the same time. To do so, each process can be executed one after the others for a defined interval of time called **timeslice**. When a process finishes its timeslice, another one is executed on the same and the previous process waits its next turn to continue its job. Therefore, the running processes are executed concurrently one after the others, figure 2.2.

In an operating system, the **scheduler** is responsible to handle the alternance of execution among the processes that require to use the CPU. Each process that executes on a CPU is placed in a **runqueue** , or simply queue, and are selected on after the others. The scheduler must be capable to execute each process regarding its priority, if it should run more often, but must avoid a process to starve, which happens when the process never accesses to the CPU. The time a process has been executed is called **runtime**. Each time the scheduler selects a new process, the CPU must do a context switch where it saves the context of the previous process, which instruction it was executing, and load the context of the new one. Context switches take time and reducing them can increase the performance.

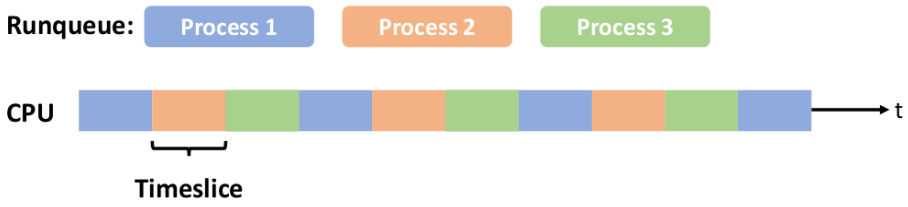


FIGURE 2.2: Execution of the processes on a CPU

On a multi-CPUs and multi-Cores systems, processes can be distributed among the different computing unit. In this case, several processes run fully parallelly. However, it appends sometimes that a CPU have more processes running on it or the

CPU is idling, when no process is running on it. In this case, a CPU migrates some of its processes to another one to have a fair distribution of the work load.

2.1.1 Process life-cycle

During its lifecycle, figure 2.3, a process passes through several states:

- **Create**, the process is created.
- **Running**, the process is currently executed on the CPU.
- **Ready**, the process is waiting to be executing and still in the runqueue.
- **Waiting**, the process can't be executed and is removed from the runqueue.
- **Deleted**, the process is killed.

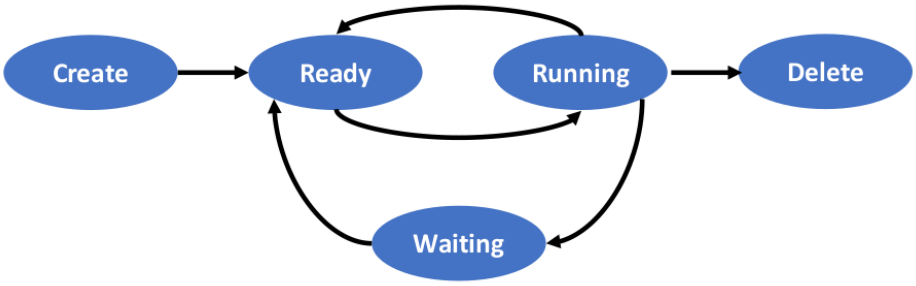
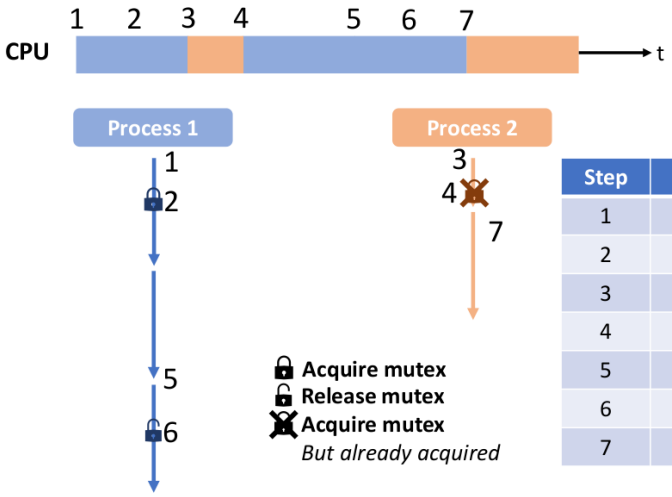


FIGURE 2.3: Life-cycle of a process

A process enters in waiting state in three different cases. First, it can put itself in waiting state and is going to be waked up after a timeout to execute a job at a specific time. Secondly, it enters in waiting state because it must wait on an external device to finish its job before moving forward. Finally, the process can be blocked by another one because both want to access the same memory address exactly in the same time.

The last case happens to avoid problematic situations when, for example, one process tries to read a value when the other one is writing on it. To prevent this, processes use **semaphores** and **mutexes** they can acquire and release when accessing memory to prevent another process to access the same memory in the same time. Semaphores and mutexes work as locks and when a process can't acquire a lock, it is blocked and put in waiting state, figure 2.4. Unfortunately, it is also possible for a process to be in waiting state with a lock acquired and this would make the other tasks waiting when they will try to acquire the lock in their turn.



Step	State P1	State P2	Mutex acquirer
1	Running	Ready	-
2	Running	Ready	P1
3	Ready	Running	P1
4	Running	Waiting	P1
5	Running	Waiting	P1
6	Running	Ready	-
7	Ready	Running	P2

FIGURE 2.4: Basic view of semaphores and mutexes functioning

2.2 Memory Management

The memory management is responsible of the memory used by a program. Usually in a computer system, two memories are used, the **primary memory** (RAM) and the **secondary memory** (e.g. disk, USB stick, ...). The primary memory is the memory where the CPU fetches the instructions to execute and the secondary memory is where programs and files are saved when not used. When a program is running, it is never completely moved in the primary memory, but only parts of it. Furthermore, a program is never placed concurrently in primary memory. Therefore, the operating system implements what it is called, **virtual memory** to allow a program thinking that it is placed concurrently and entirely in primary memory, figure 2.5.

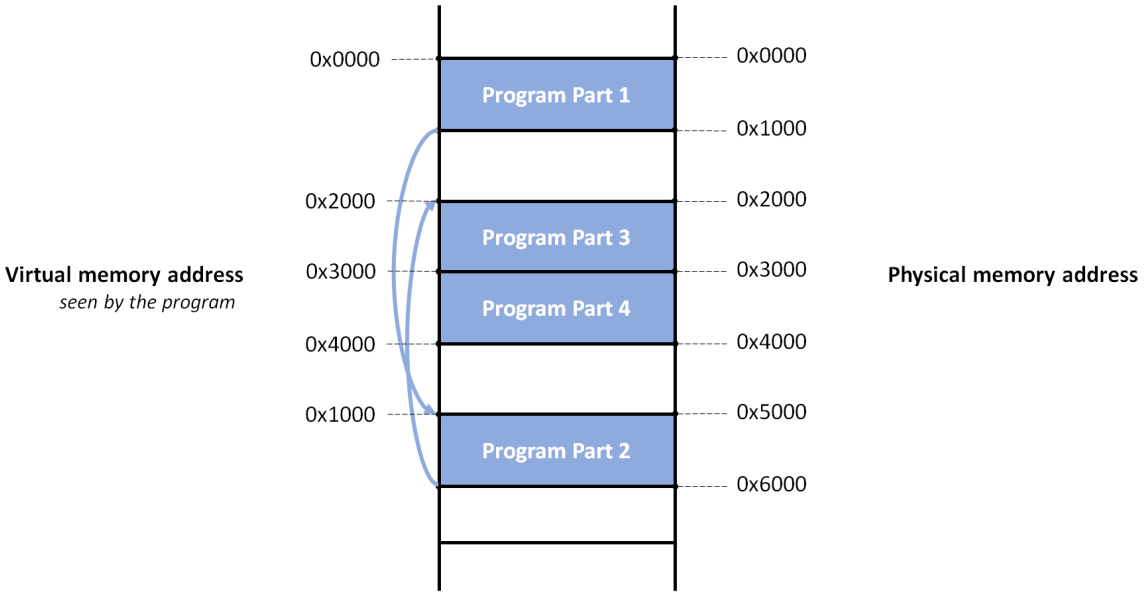
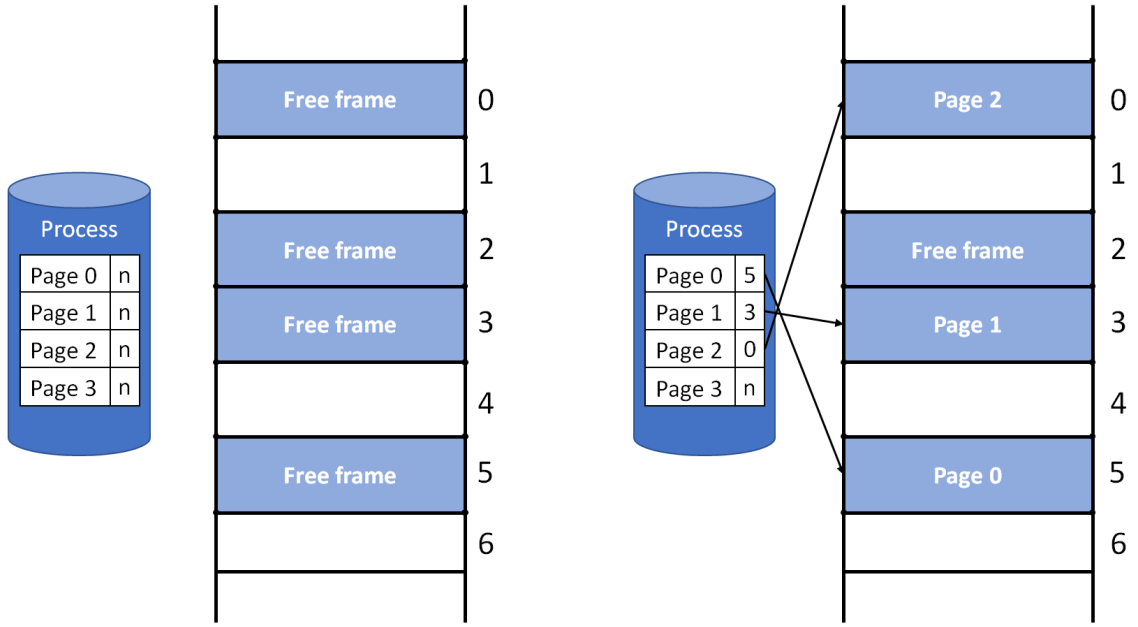


FIGURE 2.5: View of the memory from physical point of view and from the program point of view

To handle the virtual memory, the primary memory is divided in blocks called **frames** and a program is divided in blocks called **pages**, figure 2.6. When a program starts, one or several of its pages are allocated in different frames of the primary memory and get a virtual address that the program uses. As not all the program is loaded in the primary memory, the program sometimes tries to access to virtual memory corresponding to a page not present in memory. Therefore, the memory manager is responsible to move the page from secondary memory to primary memory and to give it a virtual address to allow the program to use it. When a program has finished, the pages are re-moved in the secondary memory. The action to move pages from secondary memory to primary memory is called **swap in** and the action to move pages from primary memory to secondary memory is called **swap out**.



n: no frame allocated for the page

FIGURE 2.6: Pages and frames

Sometime, a program required to swap in pages but no more frames are available in memory. Therefore, the operating system uses a **Page Frame Reclamation Algorithm (PFRA)** to determine which pages should be swapped out to free the memory and to allow the required pages to be swapped in. Unfortunately, swapping out and in pages takes time and the kernel always tries to select the best pages to replace. The kernel keeps the pages used in a queue and information about when a program accessed to them.

Another part of the memory management consists of the memory allocation, but this part is not considered during the project and therefore, not explained.

3 What is Linux?

The previous chapter explained a subset of components that composed an operating system. Unfortunately, writing an entire operating system involves a large amount of work and would be impossible for this project. Therefore, the project uses an existing operating system. In our case, the well-known Linux.

FIGURE 3.1: Tux the penguin, mascot of Linux [17]

3.1 History and philosophy

Linux was developed by Linus Torvald in 1991 and has been really appreciated and has attracted many developers that participate to the project by adding, changing, and improving the code. Linux is licensed under the **GNU General Public license (GLP)** which makes all the code available and everybody free to download the source code or to modify it. However, if you distribute your modifications, those would inherit from the license.

For this project, the last stable version of Linux released at the beginning of this project was used, version **v4.17.2**. It is important to notice that Linux is a kernel. The operating systems using Linux are called **Linux distribution** (e.g. Raspbian, Debian, CentOS, Ubuntu, Red-hat, Fedora or Slackware). As they use Linux, it makes them easy to modify and to use with a custom version of the kernel. For this project, the kernel is used with a Debian base system created with the help of debootstrap.

3.2 Why using it?

Linux is used to have the possibility to run the design of the artificial intelligence on a complete kernel instead of just writing a code for simulating the design. Furthermore, Linux is selected among other possible OS because it brings several advantages.

First, the code source is available, and the license allows to be free of modifications. Secondly, Linux has a large community of users and an easy access to documentation about the kernel. Moreover, Linux kernel is used with a large variety of distributions for various applications (Ubuntu for desktop computer, openWRT for router, Raspbian for Raspberry Pi). Therefore, if the modified kernel works with one distribution, it is easy to port it to another one for another kind of application. Then, using Linux would allow to have a usable kernel and to be focus on the modifications. Finally, the kernel can be used with several architectures (ARM, x86, Intel IA-64) and the implementation of the AI can be done regardless the processor architecture.

However, using Linux brings some disadvantages as well. First, the purpose of this project is to implement an artificial intelligence within a kernel and implementing an AI in a kernel would be more performant with a kernel specially designed for

it, which is not the case of Linux. Secondly, it is not possible to use the floating-point unit (FPU) in kernel part of Linux. Thus, there is no possible calculation with real numbers and Fixed point has to be used. The last disadvantage is that using an existing kernel limits the parts where the Artificial Intelligence can act inside the kernel.

4 What is an Artificial Intelligence ?

The previous chapters explain what an operating system is and which one is going to be used to implement an artificial intelligence. But what is an artificial intelligence and what does it do?

4.1 Agent

In the world, an agent is anything that can perceive its environment and acting (e.g. humans, robots, thermostats). The idea behind Artificial Intelligence is to construct **intelligent agents** that are rational. An agent is rational if it does the 'right thing' given its knowledge and its beliefs. The agent evolves in an environment in which it can perceive and act with the help of actuators and sensors, figure 4.1. Each action taken by the agent is done regarding the state of the environment at each point in time.

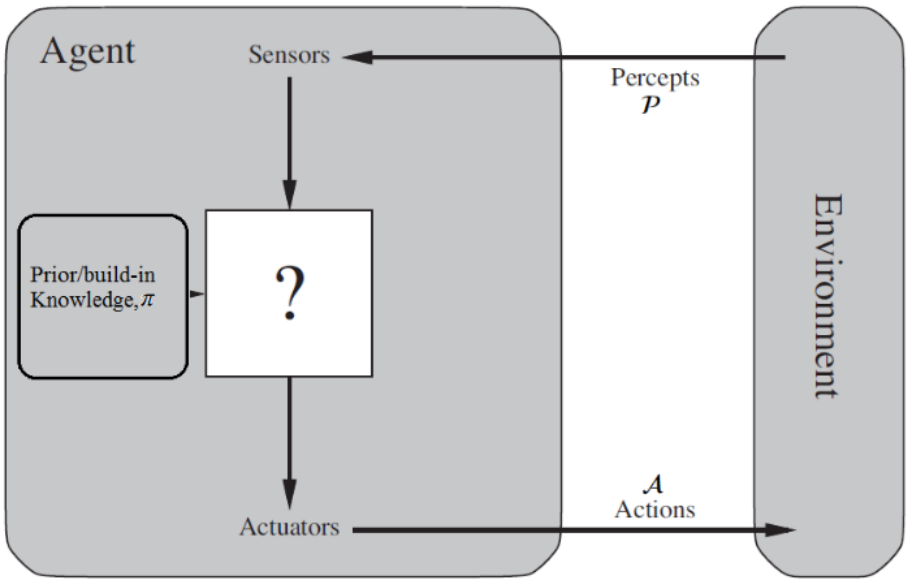


FIGURE 4.1: Agents interact with environments through sensors and actuators [13]

There are different characteristics of the environment and each of them adds complexities in the way to construct an agent. The characteristics of an environment are:

- **Partially - Fully observable**: The environment is fully observable if the agent accesses to the complete state of the environment at each point in time. Otherwise, it is partially observable.
- **Deterministic - Stochastic**: The environment is deterministic if the next state of the environment is fully determined by the current state and the action taken by the agent. Otherwise, it is stochastic.
- **Episodic - Sequential**: The environment is episodic if the future action does not depend on the previous one. Otherwise, it is sequential.
- **Static - Dynamic**: The environment is static if it does not change and the rules are always the same. Otherwise, it is dynamic.
- **Discrete - Continuous**: The environment is discrete if the set of states and actions have a finite set of possibility. Otherwise, it is continuous.
- **Single - Multi agent**: The environment is single agent if only one agent is involving in the environment. Otherwise, it is multi-agent.
- **Known - Unknown**: The environment is known if the agent has access to all the information, possible outcomes for all possible actions. Otherwise, it is unknown.

In the case of this project, the environment in which an operating system evolved is considered as:

- **Partially observable**: It is impossible to measure all characteristics of the kernel.
- **Stochastic**: The agent may select a process that blocks another one or requires to swap out important pages.
- **Episodic**: Processes are selected one after the others and pages are swapped when required.
- **Dynamic**: Processes and pages are constantly created and deleted. Hence, there is always new states and actions.
- **Discrete**: There is a finite set of processes to run/migrate and a finite set of pages to swap in/out.
- **Multi-agent**: Several CPUs schedule various processes that required different pages.
- **Unknown**: The resources that are going to be used at time t are impossible to predict.

4.2 Learning

It is interesting to note that an artificial intelligence does not necessarily involve learning. Sometimes, it only implies research algorithms to find the best path to a goal (e.g. find the shortest way between London to Glasgow). In this kind of problem, the environment is fully observable, static, and deterministic. Unfortunately, in our case, the world is not fully observable, static, and deterministic but dominated by uncertainty where the AI has to learn "How much good could it be to execute this action given my knowledge?". It is possible for an agent to learn from three different approaches:

- **Supervised Learning**, where we give examples to the agent which returns us an answer and we tell it if it is wrong or right.
- **Unsupervised Learning**, where we give examples to the agent which tries to determine the right answer by itself.
- **Reinforcement Learning**, where the agent runs in an environment and after each action, the agent gets a reward positive or negative.

It is reinforcement learning that is used in this project because the approach allows to directly learn during the runtime of the agent. The agent can be installed in any kind of environment and starts to learn and to progress without previous training.

4.3 Reinforcement Learning

As explained above, the idea of Reinforcement Learning is to learn when interacting with the environment, figure 4.2. To do so, RL is modelled as a **Markov Decision Process (MDP)** which is composed of:

- S: Set of possible states, called state space
- A: Set of possible actions, called action space
- P: State transition model
- R: Reward function
- γ: Discount factor $\gamma \in [0, 1]$

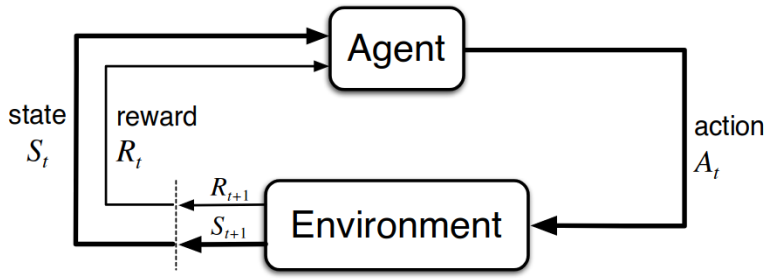


FIGURE 4.2: The agent–environment interaction in a Markov decision process.[15]

At each time step t, the agent takes an action a to go from state s to the next state s' with the transition $P(s'|s,a)$ that is the probability to arrive at state s' after taking action a in state s. After taking the action a, the agent gets a reward $R(s)$ or $R(s,a)$ regarding if the reward depends only on the state or the state and the action. The **Return** G_t, is the total discounted future reward from time-step t. The goal of the agent is to maximize G_t.

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{4.1}$$

4.3.1 Value function and Policy

Each state has an indication of "How good is it to be in state s?", called **Value function** $V(s)$, also called **Utility**. This value is the expected Return G_t from state s and is represented by the **Bellman Equation**, (4.2).

$$\begin{aligned} V(s) &= E[G_t|S_t = s] \\ &= E[R_t + \gamma G_{t+1}|S_t = s] \\ &= E[R_t + \gamma V(S_{t+1} = s')|S_t = s] \\ &= R(s) + \gamma \sum_{s' \in S} P(s'|s,a)V(s') \end{aligned} \tag{4.2}$$

To select to next move, the agent follows a **policy** π that gives an action a recommended for each state s. The goal of the agent is to find the **optimal policy** π^* that maximizes the expected Return G_t, and therefore, is based on the optimal value function $V^*(s)$.

$$a \leftarrow \pi(s) \tag{4.3}$$

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}} \sum_{s' \in S} P(s'|s,a)V^*(s') \tag{4.4}$$

$V^*(s)$ is the maximum value function from all the possible policy $V^\pi(s)$ and can be defined using the **Bellman Optimality Equation**

$$V^*(s) = max_\pi V^\pi(s) \tag{4.5}$$

$$V^*(s) = R(s) + \gamma \max_{a \in A} \sum_{s' \in S} P(s'|s,a)V^*(s') \tag{4.6}$$

Another value used in reinforcement learning is the **action-value function**, also called **Q-Value**, $Q(s,a)$ that represents "How good is it to take action a in state s?".

In this case, the optimal policy is based on the optimal Q-Value $Q^*(s,a)$ regarding the possible actions.

$$V(s) = \max_a Q(s,a) \tag{4.7}$$

$$Q(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)V(s') \tag{4.8}$$

$$\pi^*(s) = \underset{a \in A}{\operatorname{argmax}}\, Q^*(s,a) \tag{4.9}$$

The problem when an agent starts is that this agent knows nothing about the environment. It has no idea about a policy, value functions or transition model and it must learn them. Several technics exists but for this project Q-Learning is going to be used because it provides a performant solution not too complex than can be used in an operating system's kernel.

4.3.2 Q-Learning

Q-Learning is a model-free and off-policy learning method based on temporal difference. It is model-free because it does not require to know previously $R(s)$ and $P(s'|s,a)$ and off-policy because it does not follow a policy. As Q represents the value of taking action a in state s, using the maximum Q allows to directly choose the best action without any transition model.

At the beginning, the agent initialized the Q-Value of each state/action randomly. Then, the agent moves among the state by selecting the action using Q-Value. After each move, the agent evaluate the Q-Value used and update it with the reward returned and what it thinks to be the Q-Value of the next state. The agent continues to update the Q-Values of each state until they converge and don't change anymore. At this moment, it is considered that the agent has finished to learn.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{4.10}$$

$$a \leftarrow \underset{a \in A}{\operatorname{argmax}}\, Q(s,a) \tag{4.11}$$

α represents the learning factor which is used as parameter to indicate how much the old Q-Value should be updated.

4.3.3 Value function approximation

One problem with the technic described above is that it considers a small, finite and constant set of states/actions that the agent can explore quickly several times.

Unfortunately, in the case of an operating system, there is constantly new processes created and memory space needed. Hence, this make the state space and action space really huge even if the possible state and action to select for the next iteration is small (only those present in the queues).

To solve this problem, Q-Value can be calculated using **Value Function Approximation**. The idea is to represent the Q-value using a function. The function takes the state and/or the action as input parameters and uses it to return an estimate Q-Value corresponding to those parameters.

The state is represented by a feature vector, $\mathbf{x}$, and each feature has a weight, $\mathbf{w}$. For example, a typical value function approximation is a linear combination of features, (4.12). Another example of function approximation is a neural network. In this project, the design of an agent, chapter 5, consists to find a good function approximation to estimate the Q-Value of a state.

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix}$$

$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ ... \\ w_n \end{pmatrix}$$

$$\hat{Q}(s,a) = \mathbf{x}(\mathbf{s})^\mathsf{T}\mathbf{w} = \sum_i x_i(s) w_i \tag{4.12}$$

After each step, an error E is calculated based on the approximated value and the actual value returned $R(s,a) + \gamma \max_{a'} Q(s',a')$. Then, the error E is derivate regarding the weight vector and each weight is updated using what is called **Gradient Descent** method, (4.14).

$$\begin{aligned} E &= (Q(s,a) - \hat{Q}(s,a))^2 \\ &= (G_t - \hat{Q}(s,a))^2 \\ &= (R(s) + \gamma \max_{a'} Q(s',a') - \hat{Q}(s,a))^2 \end{aligned} \tag{4.13}$$

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{1}{2}\alpha \frac{dE}{d\mathbf{w}} \\ &\leftarrow \mathbf{w} - \frac{1}{2}\alpha(-2)[R(s) + \gamma \max_{a'} Q(s',a') - \hat{Q}(s,a)] \frac{d\hat{Q}(s,a)}{d\mathbf{w}} \\ &\leftarrow \mathbf{w} + \alpha[R(s) + \gamma \max_{a'} Q(s',a') - \hat{Q}(s,a)] \frac{d\hat{Q}(s,a)}{d\mathbf{w}} \end{aligned} \tag{4.14}$$

5 Design

In this project, the artificial intelligence is implemented in the part process management and memory management of the kernel. Three agents are designed, one for the process scheduling, one for the process migration and one for the page frame reclamation algorithm. Each agent is designed and inspired by the current algorithm implemented in Linux and tries to consider what could be improved by using learning. As Linux uses mostly the idea of task in the kernel instead of process, the term task is going to be used all the long of this chapter. The design of the agent using reinforcement learning is done based on the following idea:

- What actions can do the agent?
- How to represent the state of the agent?
- How to select the best action?
- What could influence the agent's behaviour, in order to choose the feature vector?
- What do we want to maximize in order to choose the reward?

One goal for this project is to keep a design for a dynamic environment by assuming that nothing is known before starting and the number of tasks and pages are constantly changing. This means that the agent ignores the number of processes, the kind of processes and the memory used. This approach is chosen to comply with the real environment of most of the operating systems where most of the programs that are going to be used by an OS are unknown. This brings a higher level of complexity in the design and brings a different approach, nearer to the real-world than some previous researches explained below. Therefore, main part of the design is to created functions to approximate the Q-Value of a state and an action.

5.1 Process Scheduling

5.1.1 Linux Scheduler

Completely Fair Scheduler

Completely Fair Scheduler (CFS) is the current default scheduler used by Linux. The idea of the scheduler is to be fair regarding the tasks. Therefore, considering an ideal multi-tasking CPU that has 100% physical power, the same amount of CPU

power should be allocated to each task. For example, if there are two tasks running, each of them will run 50% of the CPU power.

Unfortunately, an ideal multi-tasking CPU doesn't exist as tasks have to run one after the others on a CPU. To do so, CFS give to each task a **vruntime** that keeps track how much a task has used the CPU. When selecting a task, the scheduler look to the "expected CPU time" a task should have gotten and picks the task with the smallest vruntime, which is the task that got the less opportunity to run.

Another reason to use a vruntime instead of the real runtime, is that each task has a priority, called **nice** value in Linux. As a task with a higher priority should get more CPU time than one with a lower priority, vruntime is calculated corresponding to the priority of the task to make run more often a task with a higher priority.

Imperfections

It is important to note that CFS scheduler is an efficient scheduler. However, it only considers the priority of a task and how much CPU time it got. But what if a task is always directly blocked by another one when starting? What if a task always goes to sleep directly after running ? And what if a task finishes its timeslice with semaphores or mutexes acquired?

All of these could influence the runtime of the other tasks. For example, if a task is often blocked by the other one, it could be less blocked if it runs before the other tasks. Unfortunately, those elements are difficult to consider when implementing a scheduler because it represents too many conditions that must be taken care of. It is why learning could help to improve those elements.

5.1.2 Smartly Fair Scheduler

Smartly Fair Scheduler (SFS) is the new scheduler developed for this project. SFS tries to find the best way to distribute fairly the CPU among the tasks to increase their runtime according to their priority and reduce the number of context switches based on their influence on the others.

Difficulties

The difficulty of learning with a scheduler is that the state space and action space are constantly changing because there are constantly new tasks starting or finishing their jobs. Therefore, the set of tasks to choose is always changing and it is important to deal with this dynamic space. It is why function approximation must be used.

Related work

Some research papers have been made to implement learning with Linux scheduler. Most of them, [7][10][1], where trying to classify the type of processes to determine different timeslices. The technics used is supervised learning which involves having a large set of examples and training the agent offline. In the case of the project, the timeslice is considered to be static and the interest is in selecting a task regarding how much it has run not to adapt its timeslice to its type.

An interesting research paper developed a real-time scheduler via reinforcement learning[12]. This scheduler learns the best way to schedule tasks to keep them finishing their job before a given deadline. It defines the state using the runtime of the task and the action to be the next task to run. However, the scheduler learns with a predefined number of tasks and used temporal difference to directly learns the value function of each task. But in our case, state space and action are considered has to be dynamic. However, the main idea of this scheduler is used and adapted to be used with value function approximation.

Design

The design of SFS is inspired by the paper real-time via reinforcement learning. The idea is to define a utilization target that determines how the CPU is distributed amount the tasks (e.g. regarding the priority of the tasks), figure 5.1. Then, the scheduler follows the utilization target to select the task to run.

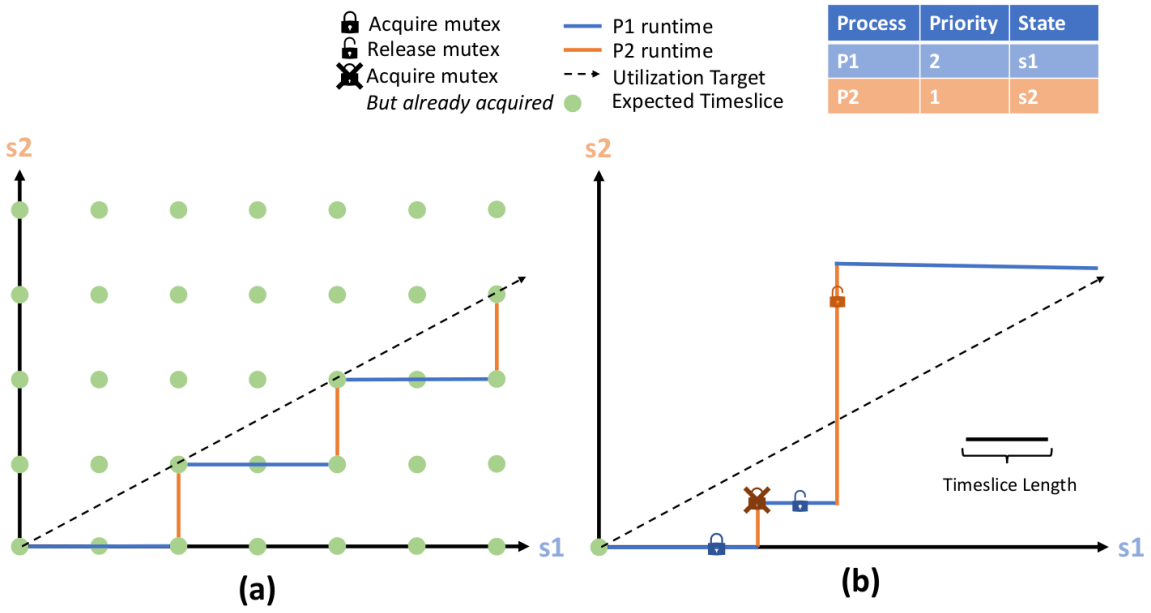


FIGURE 5.1: (a) shows the idea of utilization target in a perfect world where each process can always finish its timeslice. (b) shows the idea of utilization target in a real-world where processes block each other to access to the same memory address.

The goal of the agent is to learn how to determine the best utilization target regarding the states of the tasks. With this idea, the concept of fairness of CFS is kept because the utilization target represents the perfect fairness among the tasks.

The expected utilization target ta_i of the ith task T_i is calculated with a m-vector of features $\mathbf{x_i}$ and averaged regarding all the other tasks. In this way, ta_i indicates how much a task i should have run compared to the n others.

$$\mathbf{x_i} = \begin{pmatrix} x_{i1} \\ x_{i2} \\ ... \\ x_{im} \end{pmatrix}$$

$$y_i = \mathbf{x_i}^\intercal \mathbf{w} \tag{5.1}$$

$$ta_i = \frac{y_i}{\sum_{j=1}^{n} y_j} \tag{5.2}$$

The action a_i consists to select the task T_i that will run on the CPU. The state s_i of the task T_i is defined by its runtime and the next state s_i' of a task correspond to the state after running the task from s_i. The ideal value of s_i' is the previous state s_i plus a timeslice t. This happens when the task has running without any interruption.

The state of the runqueue s is defined as the sum of the states of all the tasks within it. To select the next action, the scheduler must determine the Q-Value of each task and select the one with the highest value. The Q-Value of a task is based on its state and how much it should has run regarding its utilization target. This indicates how much the task is fair compared to the others, called fairness $F(s)$. Lower is $F(s)$, more fair is the scheduler and zero represents the perfect fairness.

$$s = \sum_{i=1}^{n} s_i \tag{5.3}$$

$$F(s) = \sum_{i=1}^{n} |s_i - ta_i s| \tag{5.4}$$

To select the best task, the scheduler must consider how much selecting it would improve the fairness. For this, it needs to estimate the fairness of the next state of the task s_i'. To do so, it needs to estimate s_i' and therefore, uses the average delta execution avg_{dt} added to the current state of the task.

$$s_i' = s_i + avg_{dt} \tag{5.5}$$

Then, one way of selecting a task would be to pick the task giving the best $F(s')$. But this method involves calculating n times the sum of n elements if n tasks are in the queue. And it is not computationally optimal. Fortunately, between each time a

task runs, only its state has changes. Therefore, it is possible to consider the Q-Value by the difference of fairness between state s and s'. In this way, the Q-Value consider how much fairer the scheduler is going to be by selecting a specific task.

$$f_i(s) = |s_i - ta_i s| \tag{5.6}$$

$$f_i(s') = |(s_i + avg_{dt}) - ta_i(s + avg_{dt})| \tag{5.7}$$

$$F(s') = F(s) - f_i(s) + f_i(s') \tag{5.8}$$

$$Q(s,a) = F(s) - F(s') = f_i(s) - f_i(s') \tag{5.9}$$

The derivate of Q is needed to update the different weights of each features used to calculate the utilization target. The complete calculation can be found in appendix A.

$$\frac{dQ(s,a_i)}{dw_j} = x_{ij}\frac{\sum_{z=1}^{n} y_z - y_i^2}{(\sum_{z=1}^{n} y_z)^2}[\frac{s_i - ta_i s}{|s_i - ta_i s|}(-s) - \frac{s'_i - ta_i s'}{|s'_i - ta_i s'|}(-s')] \tag{5.10}$$

The features used to calculate the utilization target are selected to influence the choices of the agent based on maximizing the fairness between the tasks, maximize the runtime of each tasks and reducing the number of context switches. The features used are:

- **Task's priority**. The agent must keep in mind the priority of a task to do the selection as a higher priority requires more privilege.
- **Average number of locks acquired**. The agent must consider that a task acquiring a lot of mutexes or semaphores could risk sleeping with them and avoid the other one to acquire them in their turn.
- **Average number of times blocked**. The agent must consider that it would be better to let a task that is often blocked runs more often or that letting it runs at first could help it to get access to the resources without being blocked.
- **Average number of times going to sleep**. The agent must consider that if a task always go to sleep, it would be better to let it run at first to not have to worry about anymore.

Finally, in order to indicates to the agent how much fair was its decision, the reward $R(s,a)$ considers the difference of CPU distribution $D(s)$ among the task using task's priority p_i. Lower is $D(s)$, more fair is the scheduler and zero represent the perfect distribution.

$$D(s) = \sum_{i=1}^{n-1} \sum_{j=2}^{n} |s_i p_j - s_j p_i| \tag{5.11}$$

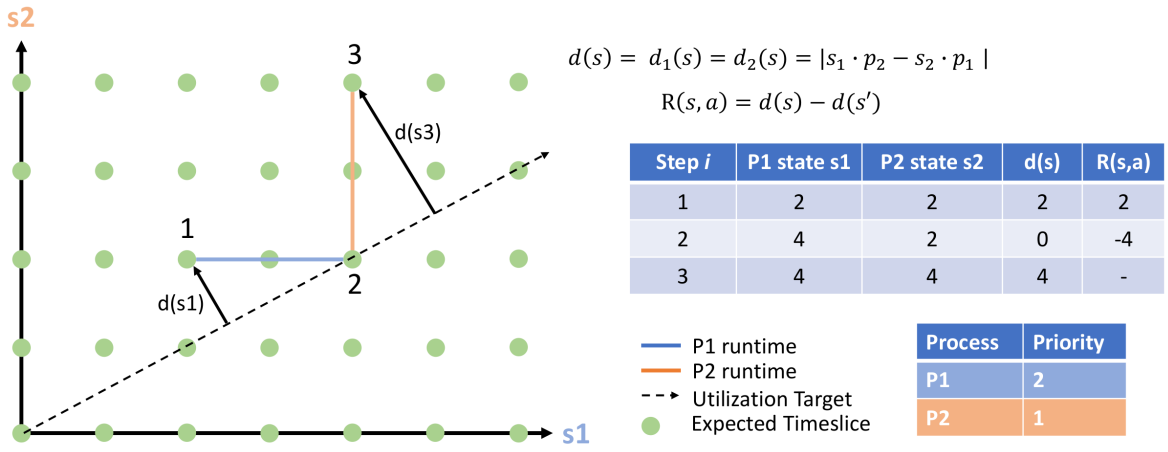
As for the value function, the computation is not great and only one task has run between each calculation. Hence, the reward can be calculated based on the difference of distribution between state s and s', figure 5.2.

$$d_i(s) = \sum_{j \neq i}^{n} |s_i p_j - s_j p_i| \tag{5.12}$$

$$D(s') = D(s) - d_i(s) + d_i(s') \tag{5.13}$$

$$R(s, a_i) = D(s) - D(s') = d_i(s) - d_i(s') \tag{5.14}$$

In this way, if the distribution was better in state s, the reward is negative (bad move) and opposite in the other case. Priority is used to calculate the reward as it is an important element that the agent must always keep considering.



Step *i*	P1 state s1	P2 state s2	d(s)	R(s,a)
1	2	2	2	2
2	4	2	0	-4
3	4	4	4	-

Process	Priority
P1	2
P2	1

FIGURE 5.2: calculation of the reward based on the utilization target

5.2 Process Migration

5.2.1 Linux migration

Load Balance

Linux groups the different CPUs in a hierarchically way composed of **scheduling domains**, **scheduling groups** and CPUs, figure 5.3. Each domain can contain one or more scheduling groups and each group can contain one or more CPUs.

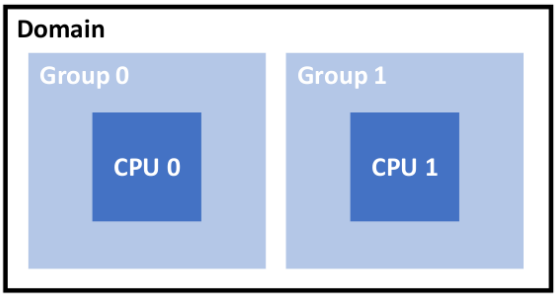


FIGURE 5.3: Linux CPUs hierarchy

Load balancing, moving a task to another CPU, is always done between groups of a scheduling domain. And a task is moved from one CPU to another one only if the total workload of a groups in a scheduling domain is significantly lower than the workload of another group in the same scheduling domain[4].

The workload of a group is calculated based on the average load in a group. Each task has a load represented by a weight based on its priority, appendix B, and the average in a group is calculated based on the number of tasks running in this group.

Imperfections

Linux's load balance considers only the workload of the queue. But what if a task runs with bad CPU-mates? What if all tasks in the CPU are always blocking each others?

All of these could influence the runtime of the tasks. For example, it would be really difficult for a task to run if each time all the other tasks in the CPU acquire locks it requires. As for the Completely Fair Scheduler, those elements are difficult to consider and it is why a learning approach could bring a solution.

5.2.2 Smart Balance

Smart Balance (SB) is the load balance developed for this project. SB tries to find the best neighbours to a task by considering more the affinity that could have a task with its CPU-mates.

Difficulties

The first difficulty is the same than for the process scheduler in the way that the state space and action space are dynamic. Another difficulty is that it is important to consider the state of the CPUs and the state of the task to migrate to take the right decision. If the agent thinks that a task should be moved but the CPU where to move it cannot support more load, the agent should consider doing nothing, not to force the migration. Finally, each CPU migrates the tasks independently.

Related work

In the paper Operating System scheduling on Heterogeneous Core systems[3], reinforcement learning is used to learn a function B_i that approximates the expected future Instruction per cycle on a CPU. It defines the state per core and each state is defined by the instruction per cycle, average cache affinity and average cache miss rate which is used in the value function approximation. Then, it considers moving tasks to improve the instruction per cycle on each CPU.

The paper uses the idea of reinforcement learning with function approximation, as employed in this project. Unfortunately, using features such instruction per cycle and cache information required access to the **Hardware Performance Counter** which is specific to the processor architecture. Furthermore, Linux doesn't provide functions to directly access to this counter in the kernel space, only from the user space and it would take a lot of time to configure them. As well, this paper only considers the state of the CPUs but not the state of the task. However, the idea of this paper is going to be used with different definition of the state and the way to approximate value function.

Design

The design of SB is inspired by the paper Operating System Scheduling on Heterogeneous Core systems but with a slightly different function approximation and different features to define the state. The idea is to consider moving a task comparing the load when having the task on the source CPU, where the task is, and the destination CPU, where to move the task. As Linux, during a balance procedure, SB always takes the busiest CPU, the source, and moves tasks to the CPU that is executing the balance procedure, the destination.

The destination CPU performs the action a_i to migrate a task T_i among n others from a source CPU. The state of the ith task t_i is a m-vector of features and the state of the yth core c_y is the sum of the features of each of the tasks present on it. The global state s is defined by the state of the k CPUs available.

$$\mathbf{t_i} = \begin{pmatrix} x_{i1} \\ x_{i2} \\ ... \\ x_{im} \end{pmatrix}$$

$$\mathbf{c_y} = \sum_{i=1}^{n} \mathbf{t_i} \tag{5.15}$$

$$s = \{c_1, c_2, ..., c_k\} \tag{5.16}$$

To determine the busiest CPU, the agent calculates the load of each CPU L_y and picks the one with the highest value. The load of the CPU is the sum of the load of

each task l_i presents on it. The load of a task is represented its state and multiply by the weights corresponding to its state's features.

$$l_i(s) = \mathbf{t_i^T w} \tag{5.17}$$

$$L_y(s) = \sum_{i=1}^{n} l_i \tag{5.18}$$

The action a_i to move a task is based on the Q-Value that is calculated by value function approximation. The Q-Value of not moving the task uses the load of the source CPU and the Q-Value of moving the task uses the load of the destination more the load of the task considered for migration. In this way, the agent considers on which CPU would fit the most the task.

$$Q(s, a_i) = \begin{cases} L_y, & if\ yth\ CPU\ is\ the\ source\ CPU \\ L_y + l_i, & if\ yth\ CPU\ is\ the\ destination\ CPU \end{cases} \tag{5.19}$$

As the Q-Value is defined by the sum of features multiply by their weights, the derivate of Q used for updating the weight is simply the sum of features used to calculates the Q-Value.

$$\frac{dQ(s, a_i)}{dw_j} = \sum_{i=1}^{n} x_{ij} \tag{5.20}$$

The features used are selected to influence the choices of the agent based on improving the runtime of the tasks and the possibility for them to complete their timeslice. The features used are:

- **Average delta execution**. The agent must consider migrating a task with low average delta execution because it may be often blocked by bad neighbours.
- **Average number of times the task has been scheduled**. The agent must consider moving a task not often scheduled because either the others have higher priorities or either there are too many tasks on the CPU.
- **Average number of times blocked**. The agent must consider moving a task that is often waiting because of bad neighbours.
- **Average number of completions, finishing its timeslice without being blocked**. The agent must consider that a task is well placed if it can run without disruption from other tasks.

Features such Instruction Per Cycle and Cache miss would be interesting to use but unfortunately, they require more configuration and are not implemented. It is why other features are used that do not involve configuring the hardware performance counter. Furthermore, the hardware counter is dependents on the architecture of the

processor and one reason for this project to use Linux is to have an abstraction from the hardware.

Finally, in order to indicate to the agent how much right was the decision to migrate a task to another CPU, the reward is calculated with the difference between the average number of times blocked avg_b and the average number of completions avg_c before and after the action.

$$r(t_i) = avg_b - avg_c \tag{5.21}$$

$$R(s, a_i) = r(t_i') - r(t_i) \tag{5.22}$$

In this way, if the number of times blocked is bigger or the average number of completions is lower on the new CPU, the reward would be negative and indicates a bad move.

5.3 Page Frame Reclaiming

5.3.1 Linux Page Frame Reclaiming

Least Recent Used

Least Recent used (LRU) considers that the pages that should be removed should be the pages the least recently used. This consideration is made based on the fact that a page recently used is more likely to be re-used soon.

Linux LRU uses two lists, an inactive which contains the tasks least recently used and an active list that contains the others. When the kernel needs to free pages to allocate new ones, the kernel tries to remove pages from the inactive list per batch of 32 pages.

Imperfection

One problem when using LRU is that for example, a page recently used but only used each hour would be kept if the reclamation happens right after its utilization. Hence, a not really useful page would be kept and another one that would be accessed soon is removed. Each time the kernel tries to access to a page not present, a page fault occurs and handling a page fault takes time. Therefore, reducing the page fault is important. However, it is difficult to determine how much useful is a page. It is why learning could help to determine the best pages to free.

5.3.2 Smart Page Frame Reclaiming

Smart Page Frame Reclaiming Algorithm (SPFRA) is the page frame reclaiming algorithm developed for this project. It tries to reduce the number of page faults by removing the less useful pages with the least chance to be used.

Difficulties

As for the process scheduler, the first difficulty is that the state space and action space are dynamic because pages are constantly added and removed. Another difficulty is that choosing the best page to removed needs to consider the state of the actual page and the state of all the other pages presents in the list to determine how much useful is this page compared to the others.

Related work

For page reclaiming, previous researches where mainly focus on classifying the pages to determine which one should be removed or not. For example, a research, Machine learning feature selection for tuning memory page swapping [11], uses supervised learning to classify the data contained in the page and to select the pages to swap.

Another research, Better caching using reinforcement learning [9], uses reinforcement learning with the implementation of LRU in Linux kernel. Pages are judged if they should be moved in active/inactive list or swapped out to disk. However, it only considers the page itself but not how much is it useful to keep it regarding the others. Therefore, the approach used for this project is different because it considers the state of the page and the queue containing the pages and don't reuse the idea of active/inactive list as Linux.

Design

The design of the reclamation algorithm is slightly the same than for SB. The idea is to consider two states p and q. q represents the state of the page queue. p is the sum of the state of all the page in the queue with p_i the state of the ith page. Each q and p_i are defined with m-vectors of features. The global state s is defined by the state of the queue and the states of the n pages present on it.

$$\mathbf{p_i} = \begin{pmatrix} x_i 1 \\ x_i 2 \\ ... \\ x_i m \end{pmatrix}$$

$$\mathbf{q} = \begin{pmatrix} y_1 \\ y_2 \\ ... \\ y_m \end{pmatrix}$$

$$\mathbf{p} = \sum_{i=1}^{n} \mathbf{p_i} \tag{5.23}$$

$$s = \{p, q\} \tag{5.24}$$

The page queue performs the action a_i to keep a page or to swap it out. Each time the agent takes the decision, it calculates the Q-Value based on the features used by q and p multiply by their weights. The Q-Value of keeping a page uses p entirely and the Q-Value of swapping out a page consider p less the state of the page p_i considered for swapping.

$$Q(s, a_i) = \begin{cases} \mathbf{p}^\intercal \mathbf{w_p} - \mathbf{q}^\intercal \mathbf{w_q}, & if\ the\ page\ is\ kept \\ (\mathbf{p} - \mathbf{p_i})^\intercal \mathbf{w_p} - \mathbf{s}\mathbf{q}^\intercal \mathbf{w_q}, & if\ the\ page\ is\ swapped\ out \end{cases} \tag{5.25}$$

As the Q-Value is defined by the sum of features multiply by their weights, the derivate of Q used for updating the weights is simply the features used to calculates the Q-Value.

$$\frac{dQ(s, a_i)}{dw_{qj}} = y_j \tag{5.26}$$

$$\frac{dQ(s, a_i)}{dw_{pj}} = x_{ij} \tag{5.27}$$

p allows to compare the pages among them and q allows to increase the criterions for a page to stay in the queue for the cases when too many pages are in memory. The features used are either related to the page queue, denoted by Q, or to a page, denoted by P. The features are:

- **(Q) Number of pages to reclaim**. If there is a lot of page to reclaim, the agent must consider to be less indulgent and try to remove more pages.
- **(P) Average of time between each access**. The agent must consider keeping pages often access.
- **(P) Time since last access**. The agent must consider moving a page not accessed for a long time. However, regarding the average of time between each access, if the time since the last access is near to this value, it would be wise to keep it because it would be accessed soon.

Finally, in order to indicate to the agent how good was the decision to remove a page, the reward is calculated by considering the interval between the reclamation

procedure and when the page is accessed t. The idea is that if the page was kept in the queue and not accessed before its average interval between each access avg_t, it was a bad idea to keep it and in the case of swapping out a page, the bad move comes when the page is accessed before its average interval between each access.

$$R(s, a_i) = \begin{cases} avg_t - t, & if\ the\ page\ was\ kept \\ -(avg_t - t), & if\ the\ page\ was\ swapped\ out \end{cases} \tag{5.28}$$

6 Implementation

This chapter highlight the important parts where kernel has been modified. The explanations are generalized to give a simple overview of where the agents are acting in the kernel.

The explanations below are based on the Linux version 4.17.2. Hence, it is possible that previous or future versions use different names for some elements and functions listed. A kernel patch is created to add the modification to the initial kernel and can be found on the project repository.

The process scheduler and process migration code are implemented as a new Linux scheduler class in the file *"kernel/sched/smart.c"*. When the new scheduler is selected, the CFS Linux scheduler is replaced by the new one. Unfortunately, the page frame reclaiming algorithm is not implemented but a basic idea of the implementation, not tested, is presented. Each agent's implementation has to take care of several aspects:

- When to update the state?
- When to update the features?
- When to calculate the reward?
- When to select the next action?
- When to update the weights?

6.1 Configurable Parameters

Linux kernel can be configured to indicate the modules, architecture, debugging tools and other elements that must be used by the kernel. All configurations are defined in the file *.config*. Using the configuration file allows to set the kernel without rewriting the code. For this project, several configurable parameters are defined for the agents. A list of those parameters can be found in appendix D.

6.2 Process scheduler

For SFS, each CPU has its own scheduler and therefore, takes the decision independently about which task to schedule. Hence, there is an agent for process scheduling in each CPU. However, each of them uses the same weights. The figure 6.1 shows the basic algorithm for the scheduler. More detailed activity diagrams about functions listed below can be found in appendix C.

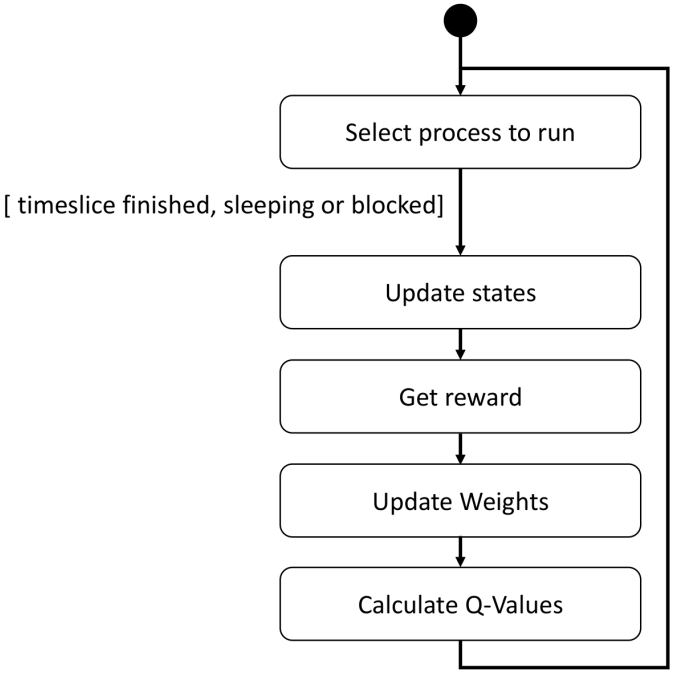


FIGURE 6.1: Activity Diagram of SFS

6.2.1 Update the state

The state is updated exactly in the same place that vruntime with CFS. It is done by the function *update_curr()* which is called:

- Periodically by *scheduler_tick()*
- When a task is removed from the queue, *enqueue_entity()*
- When a task is added to the queue, *dequeue_entity()*

This function updates the state of the queue and the current task scheduled. Each time this function is called, the time that the current task has been executed, the delta execution, is added to the state of the current task and the state of the runqueue. Only the state of the current task is updated because it is the only one that has run. When a task is removed from or added to the queue, its state is subtracted or added to the state of the queue.

CFS scheduler uses what is called **normalization** where the minimum vruntime of tasks present in the queue is subtracted or added to the vruntime of a task when

this one is removed from or added to a queue. In this way, when a task is created or moved to another CPU, its state stays consistent with the other tasks in the CPU. SFS scheduler keeps this idea and therefore, the task's state is normalized each time it is removed or added to the queue. Hence, the value of the minimum task's state is always maintain by the queue. When a new task is created, its state is set to zero and will be normalized when added to a queue.

Another part of the state is the average delta execution to estimate the next state of the task. This value is calculated based on the state of the task divided by the number of times it has been scheduled.

6.2.2 Update the features

The features used for SFS scheduler are task's priority, average number of locks acquired, average number of times blocked and average number of times going to sleep.

The priority is based on the load weight of a task defined by Linux. Those values are defined in the array *sched_prio_to_weight[]*, appendix B, and are static. The values are defined in order to give 10% more CPU to a task that has one priority higher than another. They are used by SFS to consider correctly the importance of the priority of a task.

The average number of locks acquired is calculated by using the sum of all the locks the task has acquired during its life and divided by the number of times a task has been scheduled. The sum of the locks acquired is incremented when the task acquires one of the synchronization methods not involving busy looping (semaphore, mutex, read/write semaphore).

The same calculation is used for the average number of times a task is blocked with the sum of the number of times being blocked divided by the number of times scheduled. The sum is incremented each time a task tries to acquire a semaphore, mutex, read/write semaphore, already acquired by another task and is removed from the runqueue to wait.

Finally, the average number of sleeping is calculated with the sum of time going to sleep divided by the number of times scheduled. The sum is incremented each time the task asks or required to sleep or is waiting for an event (e.g. disk access).

The different average listed above are calculated in the function *update_curr()* when updating the state of the task.

6.2.3 Calculate the reward

The reward is simply calculated each time the function *update_curr()* is called after updating the states and the features.

6.2.4 Select the next action

The next action, next task to schedule, is selected by the function *schedule()* which is called:

- Periodically when the previous task has finished its timeslice.
- When the previous task has been blocked by another one.
- When the previous task went to sleep or asked to change, idle.
- When a task with a higher priority has been added to the queue.

schedule() called the function *pick_next_entity()* which calculates the Q-Value and selects the task with the max Q-Value. It is this task that is going to be the next one scheduled.

6.2.5 Update the weights

In learning, batches are used to avoid updating each time the weights. Instead, the agent sums the delta weights calculated and when updating the weights, adds the average of delta weights over all the batches to the weights.

As the weights are the same for each agent, each agent has a sum of the delta weights and at each interval, a synchronization, *synch_point()*, is done to update the weights.

The delta weights are calculated before *pick_next_entity()* is called and a new task selected. However, after updating the weights, the current Q-Values of the agents are wrong because calculated with the previous value of weights. Hence, the values can't be used to calculate the delta weights. It is why a *"dirty_weight"* flag is used to avoid calculating wrong delta weights just after updating the new weights.

Another part of the update is to check for a weight convergence to see if the agent has finished to learn. Therefore, if the delta weight is several times lower than a threshold, this means that the error is low, the weights are considered to have converged and the agent must stop learning. In this case, the delta weights are no more calculated and the synchronization timer stopped.

6.3 Process migration

For SB, each CPU chooses to migrate a task or not independently. Hence, each CPU has an agent for the process migration. Each of them uses the same weights. The figure 6.2 shows the basic algorithm of the migration. More details activity diagrams about some functions listed below can be found in appendix C.

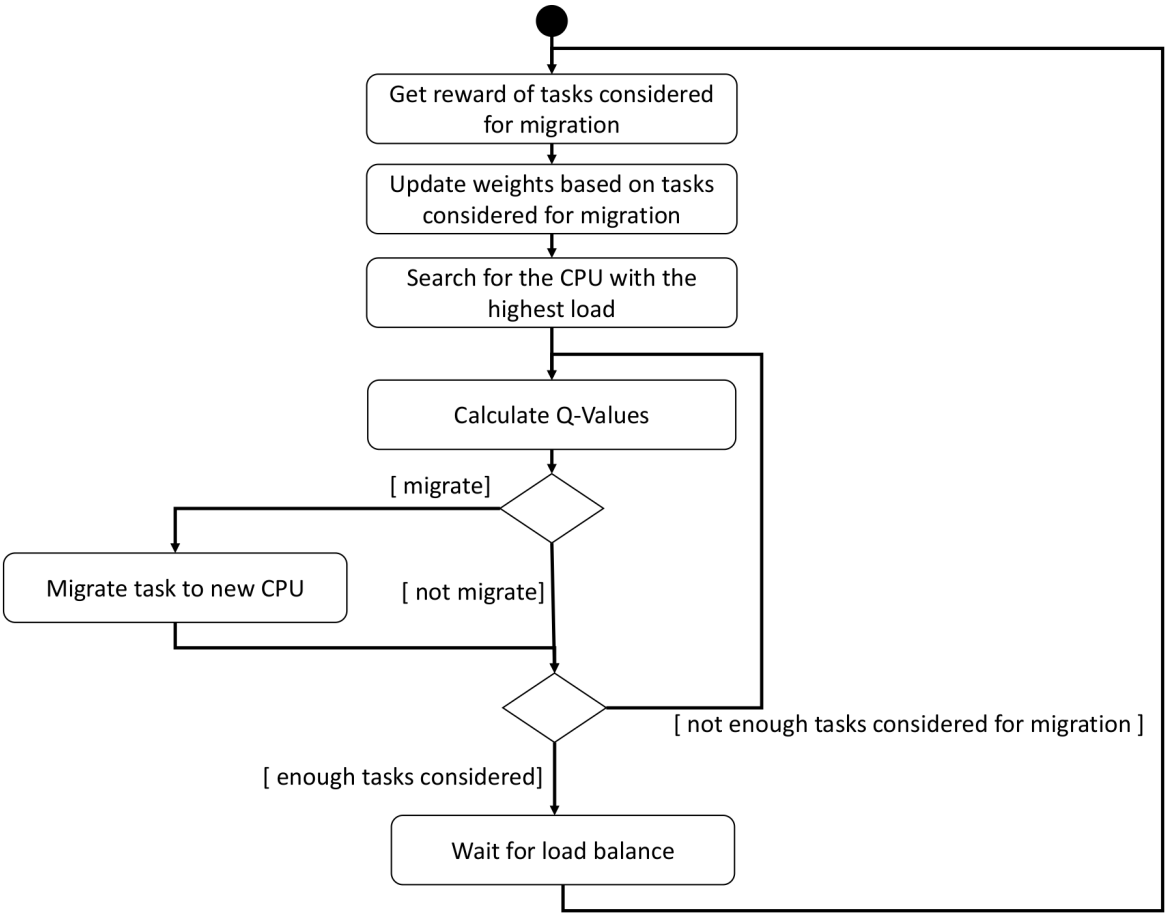


FIGURE 6.2: Activity Diagram of SB

6.3.1 Update the state

Generally, all the updates are done exactly in the same place than for the process scheduler. Hence, the state of the current task scheduled and the state of the CPU on which it runs are updated in the function *update_curr()*. Each time a task is removed from or added to a CPU, *dequeue_entity()* and *enqueue_entity*, its state is subtracted or added to the CPU's state exactly has for the process scheduler.

6.3.2 Update the features

The features used for SB are the average delta execution, average number of times blocked, the average number of times the task has been scheduled and the average number of completions. The averages are also calculated in the function *update_curr()*. In the case of the two first features, their values are the same than those calculated for the SFS.

The average number of completions is calculated by using the sum of the number of time that a task has been executed without being blocked divided by the number of time it has been scheduled.

The average number of times that task has been scheduled is calculated based on an a configurable timer. Then, the number of times that the task has been scheduled is divided by the number of times the timer occurred.

6.3.3 Calculate the reward

The reward is calculated in the function *load_balance()* before calculating the delta weights. Each time the function is called, the agents parse the tasks that have been considered for migration and calculate the reward based on the task.

6.3.4 Select the next action

The next action, migrating a task, is done in the function *load_balance()* called periodically. Each time, the queue with the higher load, source CPU, is selected and the current CPU, destination CPU, tries to migrate tasks from the source to the destination CPU.

The selection of the tasks to migrate is done in the function *detach_tasks()*. A task is evaluated and the agent decide to migrate it or not. If the task is moved, it is detached from the source CPU and attach to the destination CPU. When the task is detached, it is dequeue from the source CPU, *dequeue_entity()*, and enqueue to the destination entity, *enqueue_entity()*. Hence, the load of each CPU is updated before continuing the migration operation which is repeated with the other tasks present in the source CPU.

always several tasks are considered for migration because moving only one task, each time *load_balance()* is called, is not accurate and it could cause a CPU to stay overloaded with a lot of tasks for a long time. this way of doing is the same than what is used in Linux kernel.

6.3.5 Update the weights

As the process scheduler, the weights are the same for each agent. Therefore, a synchronization, *synch_point()*, occurred at a defined interval as for SFS. The delta weights are calculated in the function *load_balance()* before starting migrating any tasks. As for the reward, only the tasks that have been considering for migration, in the function *detach_tasks()*, are used to calculate the delta weight.

After each synchronization, a *"dirty_weight"* flag is set to avoid calculating the delta weight using Q-Value calculated with previous value of weights as for the process scheduler.

6.4 Page Frame Reclaiming

SPFRA is not implemented and tested during this project. However, it is still interesting to consider a base for starting a potential implementation. When reclamation is required, the agent always tries to swap out several pages at once. For each page the agent selects, it calculates the Q-Value and decide to keep it or to swap it out. If the agent has free enough space it stops, else it continues with the next pages. The figure 6.3 shows the basic algorithm of the page frame reclamation.

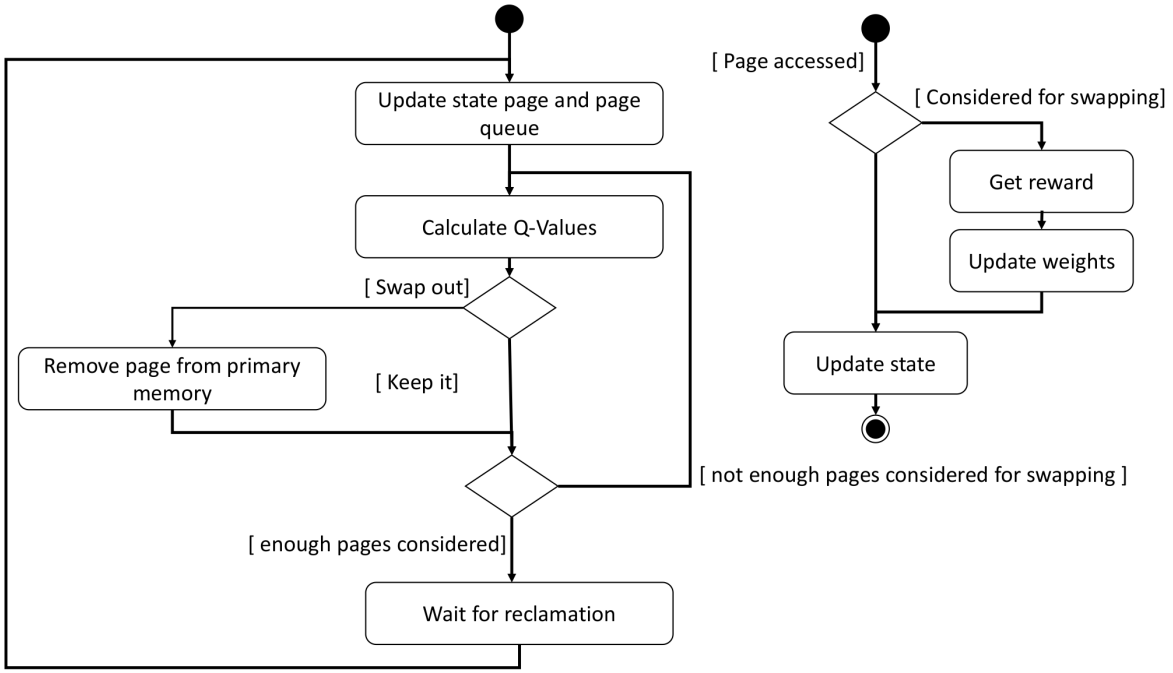


FIGURE 6.3: Activity Diagram of SPFRA

6.5 Debug file

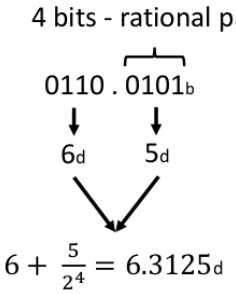
To evaluate the modifications, it is important to get access to the different variables, weights, features, Q-Values involved. To do so, the kernel used **Proc** which is a pseudo file system allowing to create **proc files** that provide system diagnostic. For example, it is possible to get any information about each task by the proc file corresponding to the task number. All those files are located in the directory *"/proc/"* of the operating system. For this project, two files are created *"schedai_debug"* for the process scheduler and *"schedsb_debug"* for the process migration. Those files indicate:

- The weights
- The sum of delta weights for each agent
- The states of each agent and task
- The rewards
- The Q-Values

- The value of the features

6.6 Fixed Point

One particularity of the kernel is that there is no possibility to use floating point unit. Hence, there is no possible operation with real-number. Instead, fixed point number with Q-format are used to calculate the different states, averages, Q-Values. Q-format defines a number of bits to represent the fractional part and the rest for the integer part.



Binary	Decimal	Fixed point
0000 . 1000	8	0.5
0010 . 0100	36	2.25
0011 . 1100	60	3.75

FIGURE 6.4: Q-format representation

The particularity of the Q-format is that addition and subtraction operations don't change. For multiplication operations, the result is divided by two to the q number of bits used for the fractional part and for division operations , the result is multiply by 2 to the q number of bits used or the fraction part.

$$a \cdot b \rightarrow \frac{a \cdot b}{2^q} \tag{6.1}$$

$$\frac{a}{b} \rightarrow \frac{a \cdot 2^q}{b} \tag{6.2}$$

7 Results and Measurements

The measurements are focus on the behaviour of the artificial intelligence rather than the performance. The behaviour is evaluated by looking which weight is updated and regarding the importance given depending on the kind of environment in which the agents are evolving. The behaviour of the agents gives a first overview of how the design makes the agents react to see if they are correctly influence by the

type of processes running and if they can learn. The agents are considered to have finished learning if the weights have converged.

The performances are not considered during the project because in order to start performance measurement, it is important to know if the agents can learn. And unfortunately, there is not enough time allocated for this project and therefore, there was not the possibility to start measuring performances. Therefore, only the behaviour is considered and performance considered as future work, chapter 8.

7.1 Environment

In order to keep control during the measurements, the operating system is running on the virtual machine **Qemu**. No graphic interface is used which makes the operating system only usable with command line terminal.

To test the kernel, a root directory is created to be the user space of the operating systems. This directory is created with **debootstrap** that installs a Debian base system. Therefore, the operating system uses is a tiny Debian distribution.

Finally, the behaviour of the processes created for the measurement must be known. To do so, the program **stress-ng** is used. This program allows to generate processes that add load to the kernel to stress it. For example, processes that constantly write and delete file, or constantly execute computation on the CPU. In this way, it is possible to estimate how the agent would behave and would update the weights.

7.2 Procedure

To measure the weights, a python script *"weight_read.py"* is created which parses the debug files *"schedai_debug"* and *"schedsb_debug"* each second and read the value of the weights. The values are written in share files between the tested OS (guest OS) and the OS running the virtual machine (host OS). The script is run automatically one the guest OS when the kernel boots.

Unfortunately, running the script to measure the weights involves creating processes that influence the measurement. To have as less perturbation on the measurement script as possible, the script is executed with the highest priority possible.

Another python script, *"weight_plot.py"*, is created, running on the host OS, to plot the values saved in the shared files between the guest and host OS. The plots can show in real-time the evolution of the weights and the script is run on the host OS.

7.3 Test cases

The main difficulty is to define the initial weights, the learning factor and the discount factor. Unfortunately, there is no rules to do it but only estimation of how the

TABLE 7.1: Test cases

Tests	AI	Stress-ng	Nbr. CPUs	Nbr. Processes	Priority
CPU-Bound	SFS	cpu	1	3	20
				3	10
				3	0
				3	-10
	SB		4	40	20
I/O-Bound	SFS	io	1	5	0
	SB		4	20	0
Blocking each others	SFS	dentry	1	10	0
	SB	switch	4	40	0

agents would behave. Therefore, the measurements are done with different initial values for the factors to see different results.

The process scheduler SFS and the process migration SB are going to be tested separately on three scenarios:

- CPU Bound processes
- I/O Bound processes
- Processes blocking each other

CPU bound processes are processes that fully use the CPU without sleeping or accessing to the disk. In the case of the tests, the processes are constantly doing an sqrt operation.

I/O bound processes are processes that are always accessing data in the disk. Those processes spend most of the time waiting for a disk access to finish. In the case of the tests, the processes are constantly asking to synchronize the data in primary memory to the secondary memory. Hence, this will write any buffered data in memory out to the disk.

To make the processes blocking each other, two different ways are used. In the case of process scheduling, processes constantly create or remove directory entries. A directory entry is the mapping of filename to its inode which contains the information about a file (type, permission, size, ...). In the case of process migration, processes send messages via pipe to a child process to force context switching. Different methods are used because each of them has a better influence on the operating system, regarding the number of CPU, to nicely see their influence on the weights.

The table 7.1 shows for the different scenarios the number of processes and their priority, number of CPU and the stress-ng command used. One difficulty in choosing the scenario is not to stress too much the operating system. As measurement script are as well processes, stressing too much the operating systems would make impossible to measure anything. Furthermore, even a common operating system could crash if the tests generated with stress-ng add to much load.

TABLE 7.2: Kernel Configuration (N.U = NOT USED)

Parameters	SFS		SB	
SCHED_SFS	X		X	
SCHED_SFS_TIMESLICE	1ms		1ms	
SCHED_SFS_SYNCH_TIMER	1s		1s	
SCHED_SFS_FIXP_SHIFT	14		14	
SCHED_SFS_ALPHA	0.05	0.01	0	
SCHED_SFS_GAMMA	0.5	0.01	0	
SCHED_SFS_W_PRIO	1		1	
SCHED_SFS_W_AGV_BLOCK	0		0	
SCHED_SFS_W_AGV_SLEEP	0		0	
SCHED_SFS_W_AGV_LOCK	0		0	
SCHED_SFS_CONV	0		0	
SCHED_SB	N.U		X	
SCHED_SB_SYNCH_TIMER	N.U		1s	
SCHED_SB_ALPHA	N.U		0.05	0.01
SCHED_SB_GAMMA	N.U		0.5	0.01
SCHED_SB_W_AVG_DEXEC	N.U		-1	
SCHED_SB_W_AVG_SCHED	N.U		-1	
SCHED_SB_W_AVG_BLOCK	N.U		-1	
SCHED_SB_W_AVG_COMP	N.U		-1	
SCHED_SB_CONV	N.U		0	
SCHED_SB_INTERVAL	N.U		10ms	

In Linux, the priority is represented by value between -20 and 20 where 20 is the lowest, -20 is the highest and 0 is the normal priority. The measurement scripts have the priority -20. Processes with different priority are only used in the case of CPU-Bound processes because it is in this case that the priority of a process would have an influence on how agents behave. In the case of the process migration, the idea is to add a large quantity of processes to have a high load. However, this could also make impossible to the measurement scripts to run. It is why in this case, the priority of the processes is the lowest one.

The table 7.2 shows how are set the configurable kernel parameters, described in appendix D, for the tests. For each test cases, four measurements are performance, one for each combination of α and γ. SFS is used during the test of SB but will not learn to not influence too much the decision of the agents performing the balance. The combination of both, SFS and SB, together is not tested due to a lake of time. The number of bits representing the fractional part for fixed point calculation is set to 14 bits to have a resolution of $61.035 \cdot 10^{-6}$.

For SFS, the initial weights make the agent start taking decision only based on the

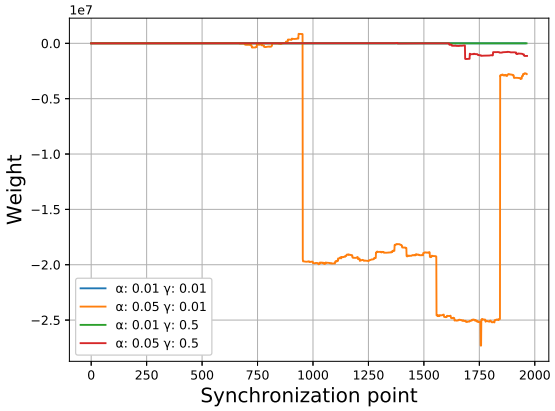
priority of the processes. During the boot process of the kernel, some processes with high priority run, and the agents have to absolutely let them run correctly otherwise the kernel could crash. It is why at the beginning, only the priority matter. In the case of SB, the initial weights are negative. In this way, following the equation 5.19, lower is the sum of features on CPU, higher is the Q-Value and thus, better it is to migrate on it.

The different factors are mostly influencing how fast the agent learns. However, even if bigger value makes the agents learn faster, they could make impossible to converge by modifying too much the weights.

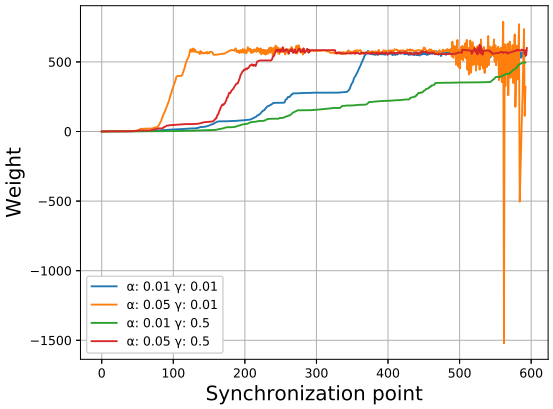
Finally, it is important to note that in the case of the measurements, the environment was controlled and the number of processes running is static. Hence, the convergence of the weights is faster.

7.4 Tests and Results

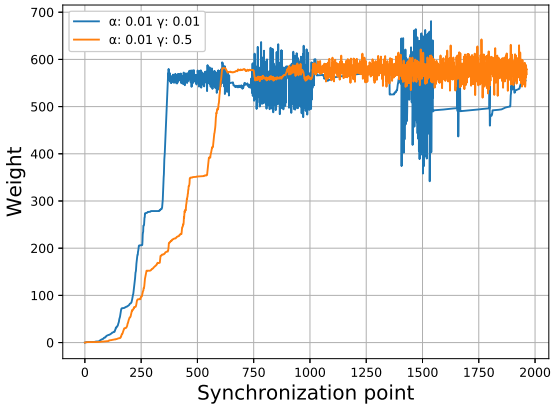
7.4.1 SFS - CPU bound processes



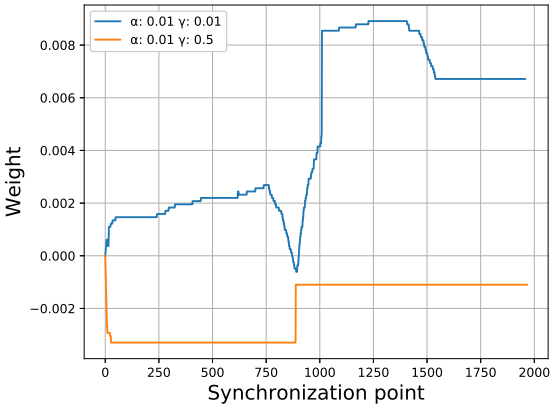
(A) the task's priority



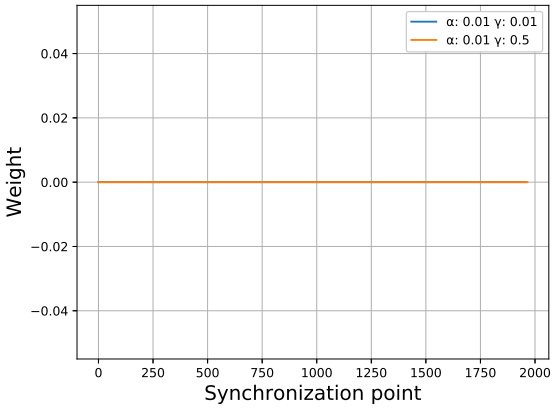
(B) the task's priority, at the beginning



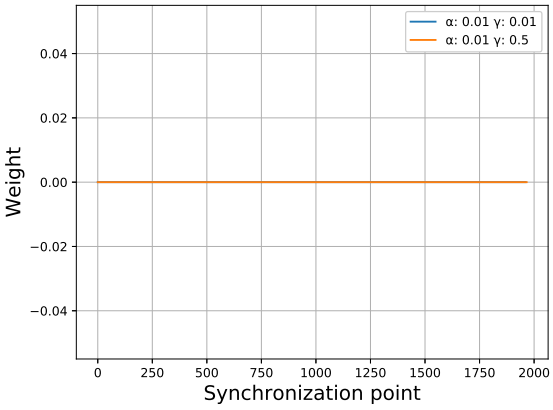
(C) the task's priority, $\alpha = 0.01$



(D) average number of times sleeping



(E) average number of locks acquired



(F) average number of times blocked

FIGURE 7.1: SFS - test with CPU Bound processes with $\alpha = 0.01$

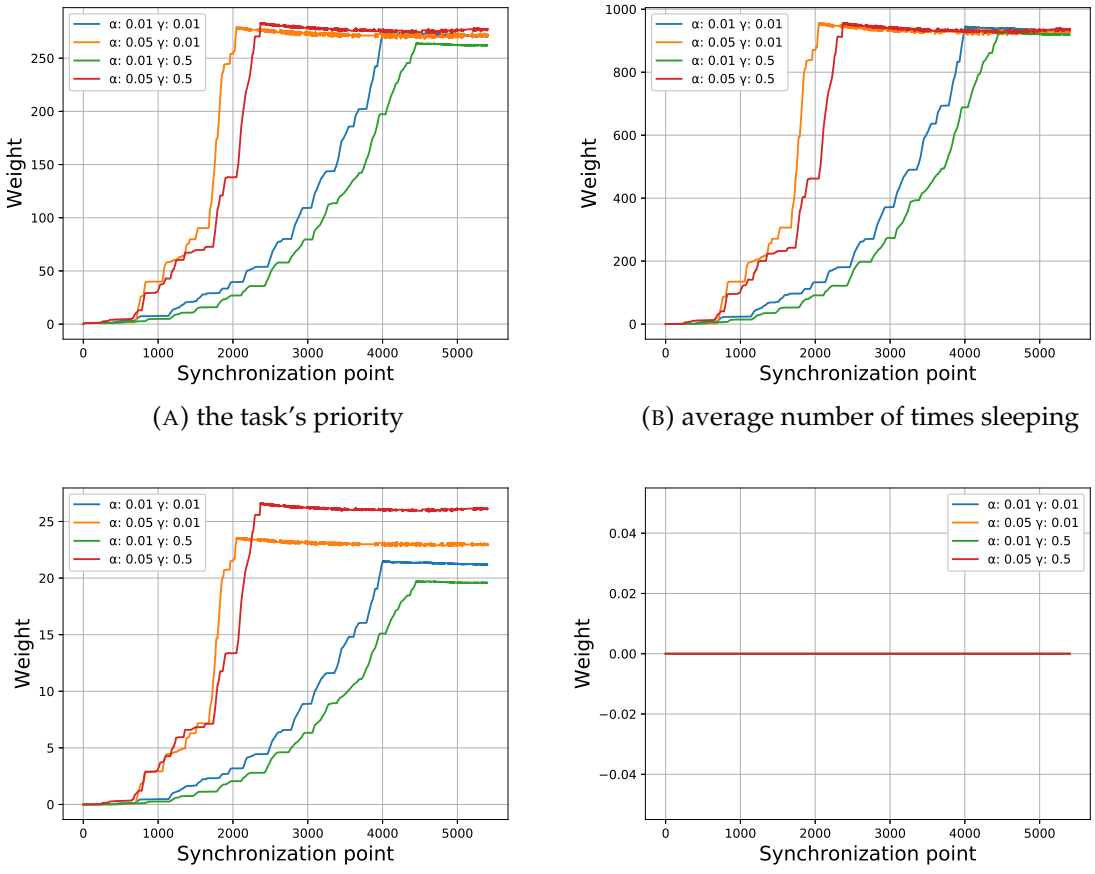
The figure 7.1a shows that the weights for the task's priority with a learning factor $\alpha = 0.05$ are unstable and finish with a huge value $> 1 \cdot 10^7$. However, the figure 7.1c

shows that for a learning factor $\alpha = 0.01$ the weights can correctly converge even if it keeps oscillating. This shows that learning factor of $\alpha = 0.05$ is not adequate.

Furthermore, figure 7.1b shows that at the beginning of the measurement, all the priority weights converge but due to the oscillation, the convergence is broken near $synchronization = 550$. A solution to this issue is to use a learning factor that decrease over the time. In this way, the oscillation of the weights would progressively decrease and let them stabilize.

The results show that the agents consider only the priority as a factor to define the utilization target of a task. As well, the agents give no importance to the weight for the number of times blocked, average number of locks acquired. This is the normal behaviour as CPU-Bound processes are always executing their timeslice without any interruption. As well, it is possible to see that the weights given to the average number of times sleeping looks more like noise as they don't converge and that their value is too low < 0.01 to have an influence. This noise is likely coming from the measurement script that is used to collect the weights.

7.4.2 SFS - I/O bound processes



(A) the task's priority

(B) average number of times sleeping

(C) average number of locks acquired

(D) average number of times blocked

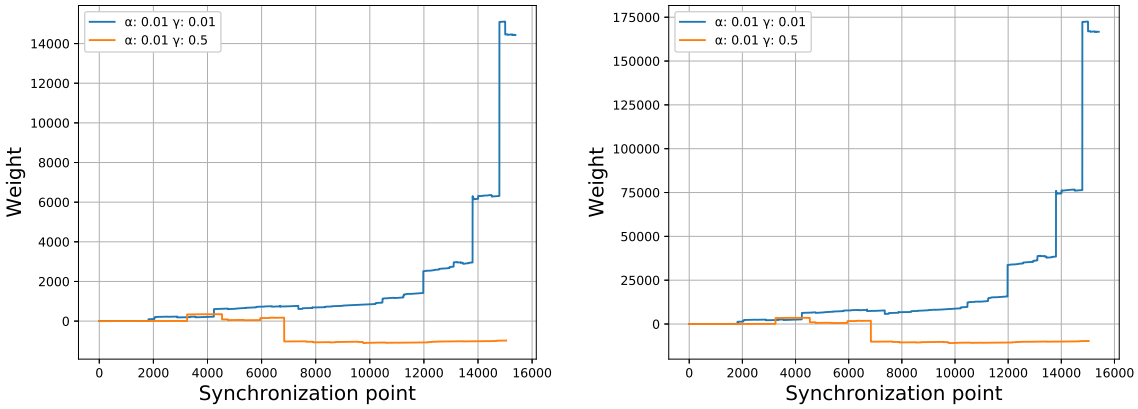
FIGURE 7.2: SFS - test with I/O Bound processes

For this test, the results show that the weights nicely converge to the same value. An interesting part is to see that compared to the first test, the weights given to the priority is higher in the CPU-Bound test, about 600 than in this test, about 275. Then, the agents give large importance to the weights for the number of times sleeping in this test, about 900, compared to the CPU-Bound test where it is more like noise. This is the expected behaviour since most of the processes are waiting for disk access instead of constantly executing instructions as during the first test. Hence, the agents give more importance to a task that sleeps a lot and try to make it run first.

Another interesting part is to see that with a higher value of learning factor $\alpha = 0.05$, the weights converge faster. It is normal as the update value added to the weight is bigger.

Finally, it is still possible to see that the weight given to the number of times blocked is null. This is normal due to the fact that the tasks are not blocking each other.

7.4.3 SFS - Processes blocking each other

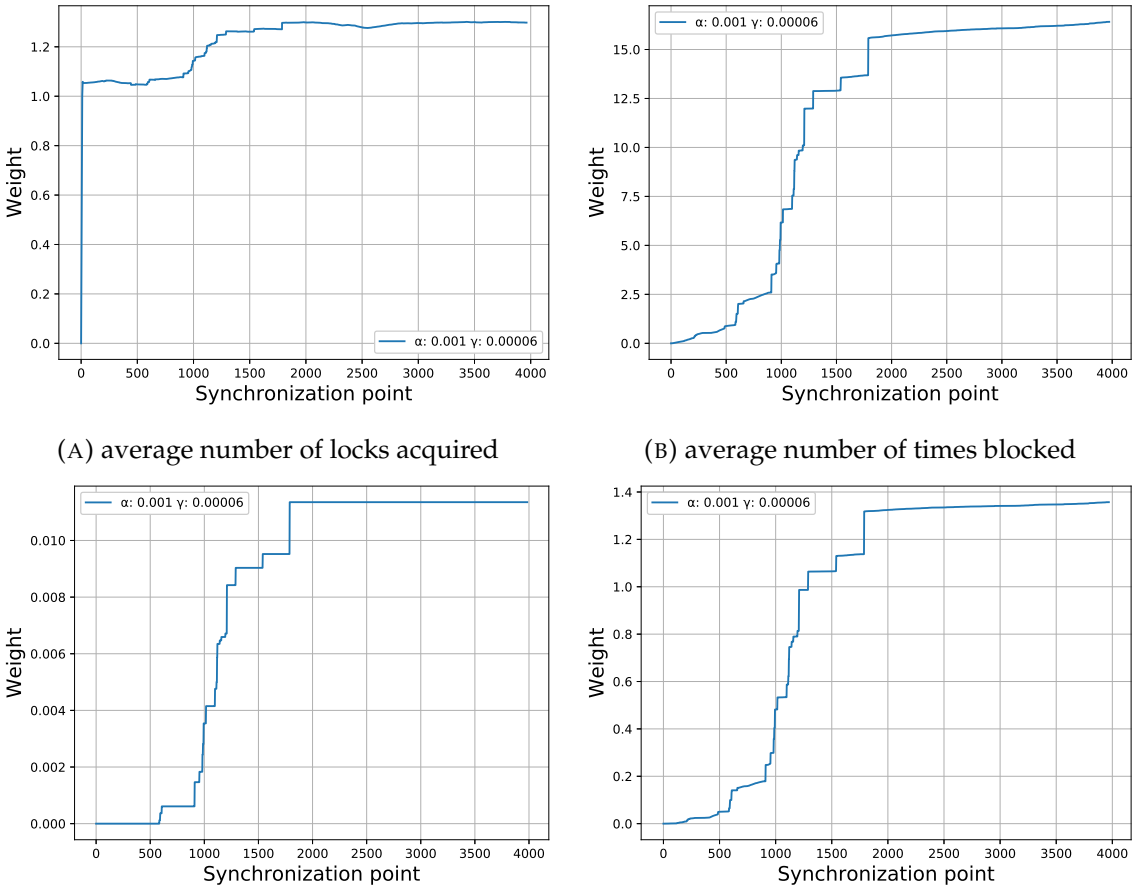


(A) average number of times blocked

(B) average number of times sleeping

FIGURE 7.3: SFS - test with processes blocking each other, $\alpha = 0.01$

For this test, the figure 7.3 shows the weights obtain with $\alpha = 0.01$ for the average number of times blocked and the average number of times sleeping where it is possible to see that the weights can't converge. Therefore, another test, figure 7.4, is performed using a learning factor $\alpha = 0.001$ and discount factor $\gamma = 61.035 \cdot 10^{-6}$ where the weights converge normally. This shows that the agent can't learn but needs a lower learning factor as for the first test.



(A) average number of locks acquired

(B) average number of times blocked

(C) average number of locks acquired

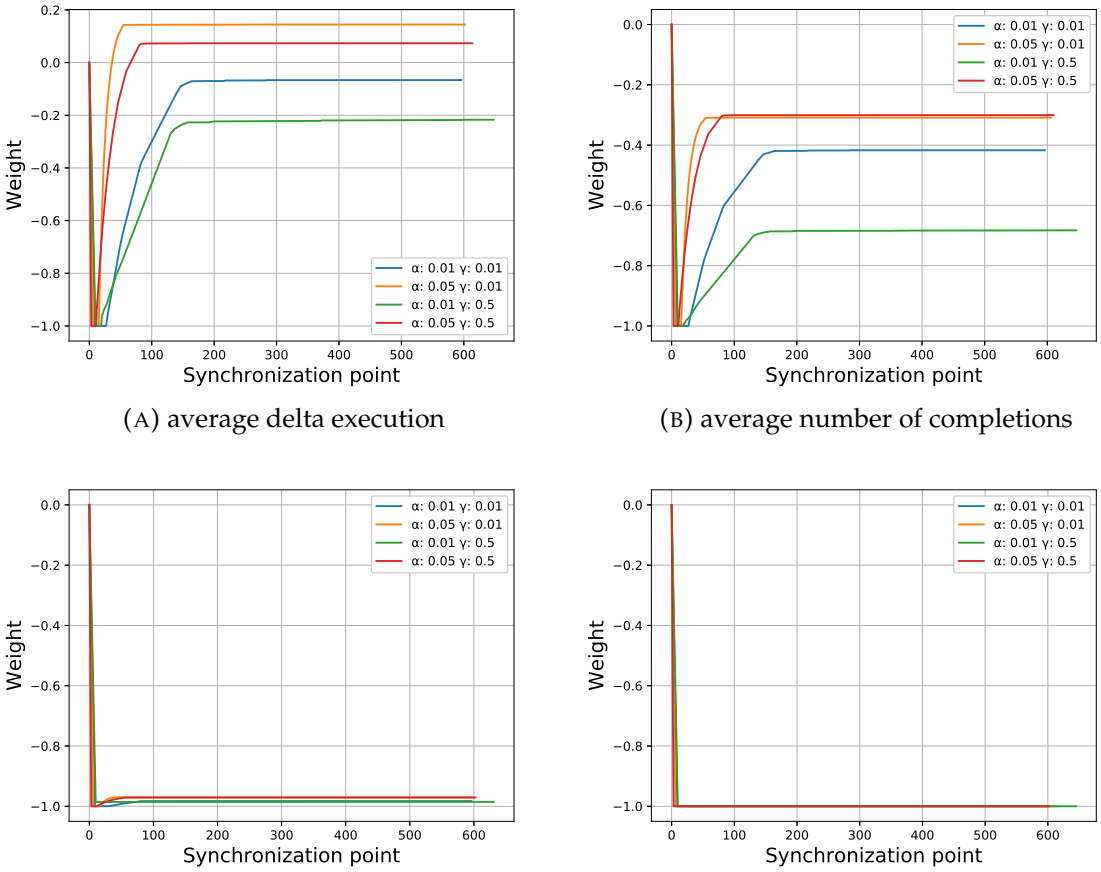
(D) average number of locks acquired

FIGURE 7.4: SFS - test with processes blocking each other, $\alpha = 0.001$

An interesting part is that even if the weights don't converge with $\alpha = 0.01$, the agents still give a lot of importance to the weights for the average number of times blocked and sleeping but give really low importance to the priority comparing to the previous tests. A large value of weight for the average number of times sleeping is normal in this case because the instructions executed by the processes required disk accesses like the I/O-Bound test. As well the agents really consider now the number of times a task is blocked and thus, give more importance to a task that is often blocked and goes often sleeping.

Finally, the results show an important problematic in reinforcement learning which is to find the best factors and their influence on the final weights. Testing different factors to see the possible influence on the results is part of future work explains in chapter 8.

7.4.4 SB - CPU bound



(A) average delta execution

(B) average number of completions

(C) average number of times scheduled

(D) average number of times blocked

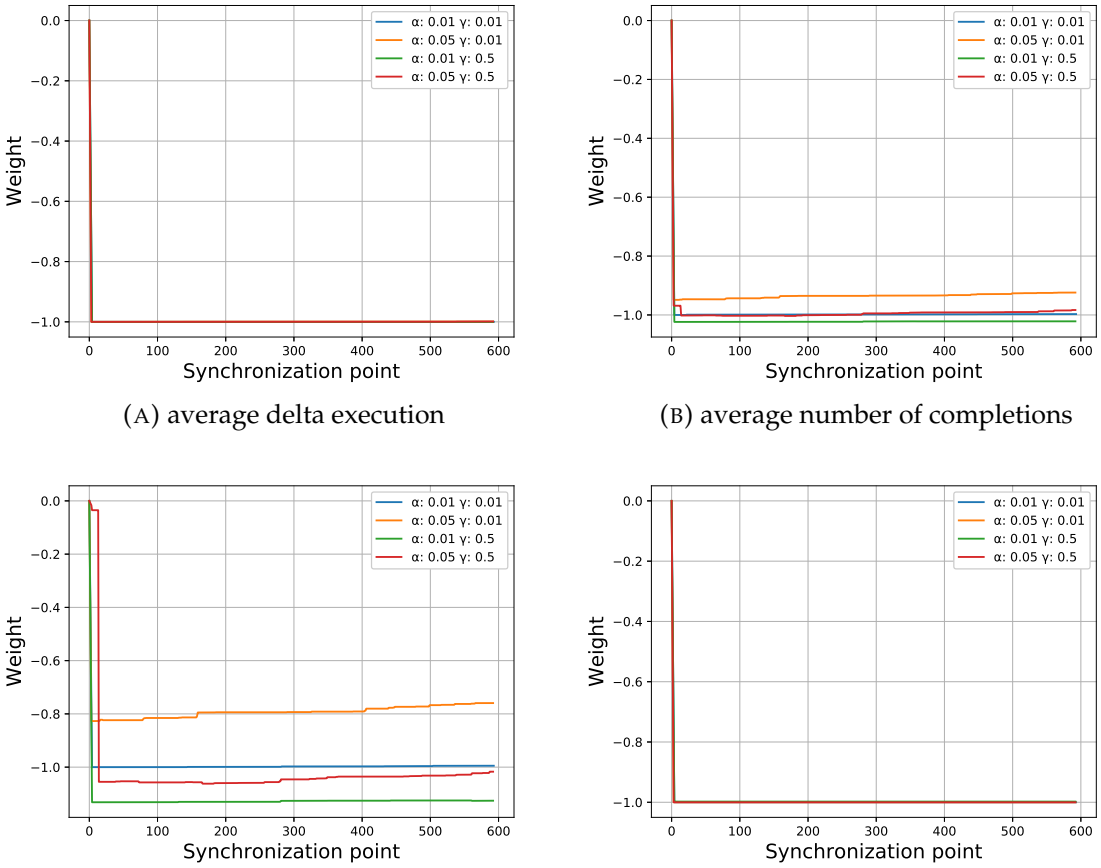
FIGURE 7.5: SB - test with CPU Bound processes

For this test, the results show that the weights converge quickly to their final value. All CPU-Bound processes have the same delta execution and always finish their timeslice without any interruption. This implies that the agents learn how to balance processes that bring the same and constant load. Therefore, it is likely why the weights converge quickly because the agent can quickly find a good balance.

It is possible to see that the curves are smooth. This is because the number of processes running was too large and blocked at the beginning of the test the measurement script. Therefore, some measurements were missed but it is not a problem as it is still possible to see that the weights converge and that the agents have learned how to distribute the load to allow the measurement script to run.

Finally, it is possible to see that the value of the number of times scheduled is low. This imply that the interval used to calculate the value is probably too short, 10ms, and could be increased. However, this would not change how the agents learn but mostly if this feature gets more importance due to a higher value and thus, make the delta weight of this features bigger.

7.4.5 SB - I/O bound



(A) average delta execution

(B) average number of completions

(C) average number of times scheduled

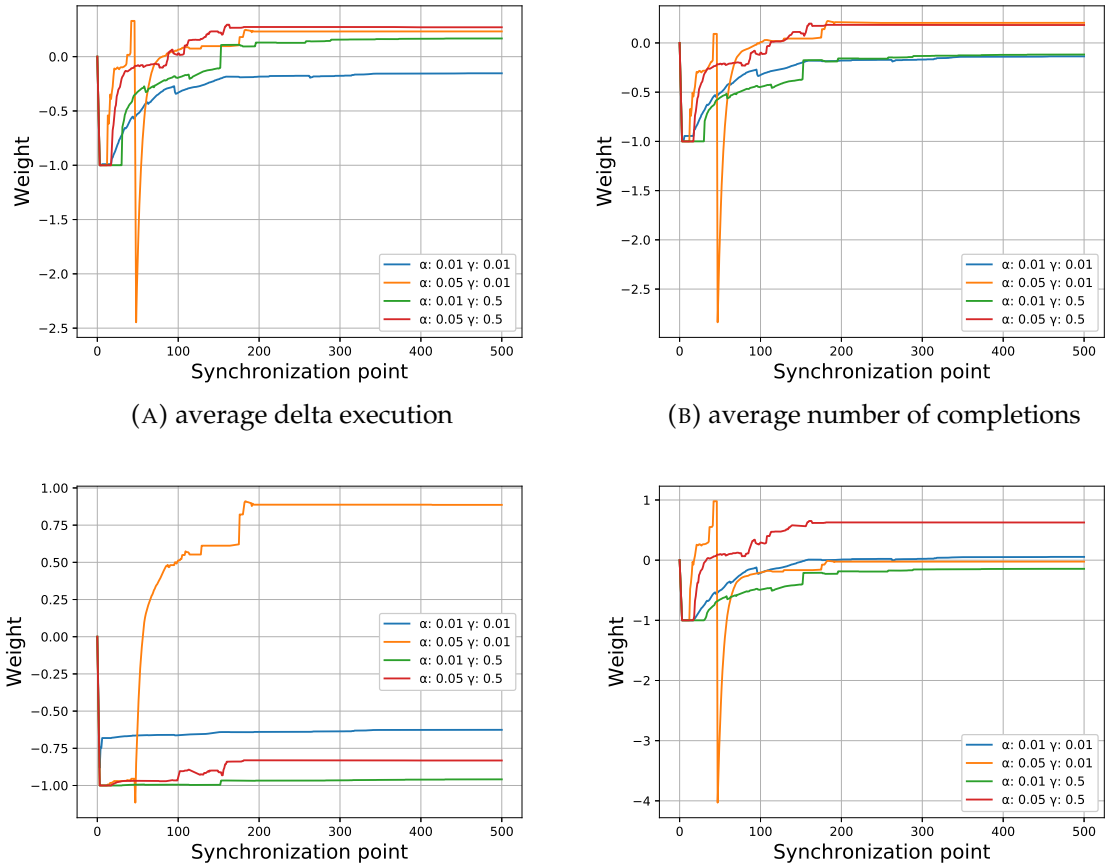
(D) average number of times blocked

FIGURE 7.6: SB - test with I/O Bound processes

For this test, the results show that the weights are constant all the long of the test and don't change. This behaviour is normal since the processes are mostly sleeping. As they spend most of the time waiting, they don't use the CPU and don't add any load. Therefore, the agent doesn't really have to learn anything as the load added by the processes is low.

Finally, it is possible to see that for some features, figure 7.6b and 7.6c, the weights slightly change at the start of the test. This is mainly due to the measurement script during the boot procedure.

7.4.6 SB - Processes blocking each other



(A) average delta execution

(B) average number of completions

(C) average number of times scheduled

(D) average number of times blocked

FIGURE 7.7: SB - test with processes blocking each other

For this test, it is possible to see that the agents give as much importance to the weights for average delta execution in this test than for the CPU-Bound test, -0.2 to 0.2, and they give more importance to the average number of completions in this test, -0.2 to 0.2, than for the CPU-Bound test, -0.7 to -0.3. This is the expected behaviour as making the processes blocking each other reduces the number of completions. Therefore, the agents response by increasing the corresponding weights to give them more influence.

Furthermore, it is possible to see that as expected, the weights for the average number of times blocked has increased as well. This is the expected behaviour as the agent would consider that more blocked processes involve more load on a CPU.

Then, as for the scenario with CPU-Bound processes, the weights converge quickly. This is also likely due to the static environment defined for the test where exactly the same type of processes doing exactly the same thing.

Finally, A interesting part of those results is that it is possible to see that in the case $\alpha = 0.05$ and $\gamma = 0.01$, at around 50 synchronizations, the weights of each feature

sharply changes. It is likely that the agents did really bad actions and increased the load on one CPU. The consequence is that the agents received a large bad reward and corrected the weights to fix the problem. It is possible to see that it results a large influence on the weight for the number of times scheduled which is much higher that for the other cases.

7.5 Discussion

First, it is possible to see that using the right factors, the weights can converge. This shows that the design allows the agents to learn correctly to arrive to final values that they think maximizing the expected reward. It is also possible to see that sometimes, the weights are not completely stabilized but keep oscillating, figure 7.1a. This could be avoided by using a learning factor that would decrease overtime. However, one complexity of this method is that it is difficult to know exactly how to reduce the learning factor to make the agents stop learning at a certain point in time.

Secondly, the result shows that the agents can learn weights regarding the type of scenario. For example, it is interesting to see that for process scheduling, in the case of CPU-bound processes, most of the importance is given to the priority but in the case I/O-bound processes, most of the importance is given to the average number of times sleeping.

It is possible to see that during the measurements, the agents give not a lot importance to the average number of locks acquired, for SFS, and the average number of times scheduled, for SB. This can be because the actual values of the features are too low. Thus, the solution is either to scale up the values or to ignore them and consider that those features are not interesting.

Finally, the main focus of tests performed is to see if the agents can learn and to give more important to some features in specific contexts. Now, it would be interesting to perform further tests to see if the design allows to really reduce the context switches, increases the runtime of the agent, or influence the performance of the operating systems. All of this are part of future work explains in the next chapter.

8 Future Work

The results give a first view of what the agents are capable to do and that they can learn and modify their behaviours regarding the environment. However, the project only starts digging in the possibility of implementing an artificial intelligence in the kernel but a lot of works still need to be done.

First, the page frame reclamation algorithm has not been implemented due to a lake of time. Thus, it would be interesting to start implementing the presented design to

test it and to evaluate the possibility of using artificial intelligence in another part of a kernel.

Then, further tests must be performed to evaluate the agents in a more dynamic environment with number of processes not constant, always changing what they are doing and mixing CPU-Bound and I/O-Bound. The project only analyses if the agent changes its behaviour regarding different scenarios and different factors are used to see the different learning curves of the agent. But in a future work, it would be interesting to measure the performance with different factors and on a larger scale to finally arrive at the main goal that is to be able to use the same operating sytem on different environment without considering a large range of modifications.

Another aspect for future work concerns the features used. The features selected for this project has the particularity to be simple to measure. However, other relevant features, such instruction per cycle or cache misses, could give more precise information about the state of the processor. Unfortunately, this kind of features involves configuring the hardware performance counter of the processor and each counter is different for each type of processor. It is why more general features as been used. Therefore, it would be interesting to bring the opportunity to have both general features and architecture specific features.

Another concern is the optimization of the code. The main problem when calculating the Q-Value is that it requires to be calculated for each element and this may take time. Even if the design already tries to reduce the overhead by removing some sum operations, some calculations still imply max operations that require to parse each element to find the maximum one. Hence, the overhead of computation could be distributed to reduce the time complexity and the computation time.

Then, one aspect of this project is that it uses Linux kernel to be able to have a usable operating system on which implementing the artificial agents. But, further development would be better on a kernel written from scratch to avoid legacy code and to have artificial intelligences being the heart of the operating systems. Hence, Linux was great as first trial but to have a better use of artificial intelligence, the kernel must be fully implemented regarding the need of the agents and not the agents implemented regarding what the kernel allows them to do, as with Linux.

9 Conclusion

The project shows an approach to design artificial agents acting in the parts process scheduling, process migration and page reclamation of the kernel. The design of the agents is created using reinforcement learning and value function approximation. For each kernel's part considered, a state space, action space, a reward function and a set of features are defined to make the agents learn and behave regarding the environment in which they evolve.

For this project, only the part process scheduling and process migration are implemented using the Linux kernel as a new Linux scheduler class. The new kernel is tested using tiny Debian distribution generated with debootstrap and virtualized with qemu. The result shows that the agents adapt they behaviour regarding the type of processes running (CPU-Bound, I/O-Bound, blocking each other) by giving more importance to specific features depending on the test. However, the results show as well that some selected features are less responsive and thus, not necessarily relevant.

Only the behaviour of the agents using the developed design is tested. But to fully test the design, further measurements must be performed to evaluate the performances with different value of learning and discount factors and with more complex scenarios.

Finally, the project only starts digging in the possibility of implementing AI in an operating systems and only present a design for three small parts of the kernel. But the complete idea on which is based this project is to create a complete kernel where each part would be built to be adaptive. In this way, all the kernel would be built using a bunch of different agents acting separately or in cooperation to allow the operating systems to understand and behave correctly regarding the type of environment on which it is running.

Bibliography

[1] "A Machine Learning Approach for Improving Process Scheduling: A Survey". In: *International Journal of Computer Trends and Technology (IJCTT)* 43 (2017).

[2] Greg Gagne Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. 9th Edition. John Wiley and Sons, Incorporated, 2014.

[3] David Vengerov Alexandra Fedorova and Daniel Doucette. "Operating system scheduling on heterogeneous core systems". In: *In Proceedings of 2007 Operating System Support for Heterogeneous Multicore Architectures* (2007).

[4] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel*. 3rd Edition. O'Reilly, 2006.

[5] Dr. Bjørn Jensen. "Artificial Intelligence". University Lecture. University of Glasgow. 2015.

[6] Robert Love. *Linux Kernel Development*. 2nd Edition. Pearson Education Incorporated, 2005.

[7] Atul Negi and Kishore Kumar P. "Applying Machine Learning Techniques to Improve Linux Process Scheduling". In: *TENCON 2005 - 2005 IEEE Region 10 Conference* (2005), pp. 1–6.

[8] Chandandeep Singh Pabla. "Completely Fair Scheduler". In: *Linux journal* (2009). URL: https://www.linuxjournal.com/node/10267.

[9] Surbhi Palande. "Better caching using reinforcement learning". In: (2018). URL: http://wiki.ubc.ca/Better_caching_using_reinforcement_learning.

[10] Vani M Prakhar Ojha Siddhartha R Thota and Mohit P Tahilianni. "Learning Scheduler Parameters For Adaptive Preemption". In: *Fourth International Conference on Advanced Information Technologies and Applications* 5 (2015), pp. 149–162.

[11] Battle Rick. "Machine learning feature selection for tuning memory page swapping". MA thesis. naval postgraduate school, 2013.

[12] Christopher Gill Robert Glaubius Terry Tidwell and William D. Smart. "Real-Time Scheduling via Reinforcement Learning". In: *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence* (2012), pp. 201–209.

[13] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd Edition. Prentice Hall, 2009.

[14] Dr. David Silver. "Reinforcement Learning". University Lecture. University College London. 2015.

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd Edition. MIT Press, 2017.

[16] Linus Torvalds and thousands of collaborators. *Linux kernel 4.17.2 Documentation*. 2018.

[17] Vexels. *Linux logo*. 2016.

A $\frac{dQ}{dw}$ for process scheduler

This appendix shows the calculation of $\frac{dQ}{dw}$ used by SFS scheduler to update the weight.

$$\mathbf{x_i} = \begin{pmatrix} x_{i1} \\ x_{i2} \\ ... \\ x_{in} \end{pmatrix}$$

$$y_i = \mathbf{x_i}^\intercal \mathbf{w}$$

$$ta_i = \frac{y_i}{\sum_{j=1}^{n} y_j}$$

$$f_i(s) = |s_i - ta_i s|$$

$$Q(s, a_i) = f_i(s) - f_i(s')$$

$$\frac{dQ(s, a_i)}{dw_j} = [\frac{df_i(s)}{dw_j} - \frac{df_i(s')}{dw_j}]$$

$$\begin{aligned} \frac{df_i(s)}{dw_j} &= \frac{d|s_i - ta_i s|}{dw_j} \\ &= \frac{(s_i - ta_i s)}{|s_i - ta_i s|}(-s)\frac{dta_i}{dw_j} \end{aligned}$$

$$\begin{aligned} \frac{dta_i}{dw_j} &= \frac{dta_i}{dy_i}\frac{dyi}{dwj} = \frac{dta_i}{dy_i}x_{ij} \\ &= x_{ij}\frac{d\frac{y_i}{\sum_{z=1} y_z}}{dy_i} \\ &= x_{ij}\frac{\sum_{z=1} y_z - y_i^2}{(\sum_{z=1} y_z)^2} \end{aligned}$$

$$\frac{dQ(s,a_i)}{dw_j} = x_{ij}\frac{\sum_{z=1} y_z - y_i^2}{\sum_{z=1} y_z)^2}[\frac{s_i - ta_i s}{|s_i - ta_i s|}(-s) - \frac{s_i' - ta_i s'}{|s_i' - ta_i s'|}(-s')]$$

B Linux *sched_prio_to_weight*

This appendix shows the weight given by Linux to a task regarding its priority. The values are used by SFS as value for the priority feature.

TABLE B.1: Linux scheduler priority to weight

prio	weight	prio	weight	prio	weight	prio	weight	prio	weight
-20	88761	-19	71755	-18	56483	-17	46273	-16	36291
-15	29154	-14	23254	-13	18705	-12	14949	-11	11916
-10	9548	-9	7620	-8	6100	-7	4904	-6	3906
-5	3121	-4	2501	-3	1991	-2	1586	-1	1277
0	1024	1	820	2	655	3	526	4	423
5	335	6	272	7	215	8	172	9	137
10	110	11	87	12	70	13	56	14	45
15	36	16	29	17	23	18	18	19	15

C Activity Diagrams

This appendix shows the activity diagram of the functions:

- update_curr()
- schedule_tick()
- schedule()
- dequeue_entity()
- enqueue_entity()
- synch_point()
- detach_tasks()
- attach_tasks()
- load_balance()

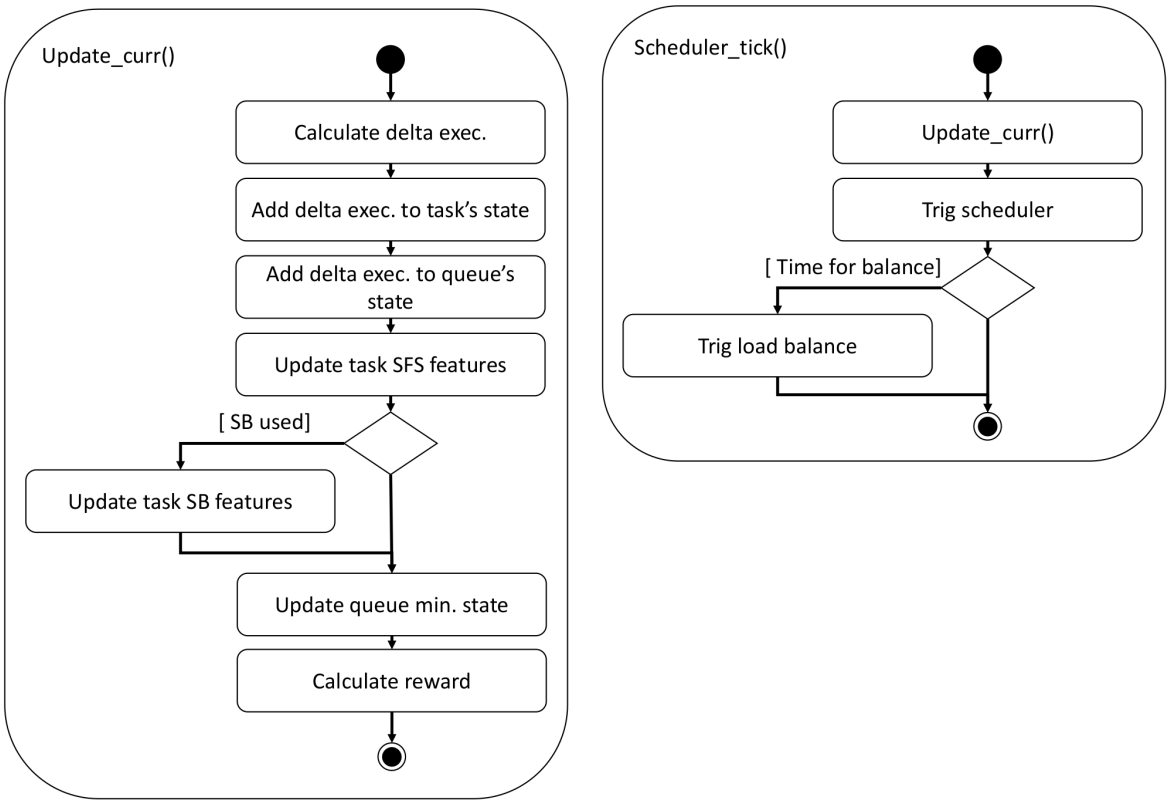


FIGURE C.1: Activity Diagrams of the functions update_curr() and schedule_tick()

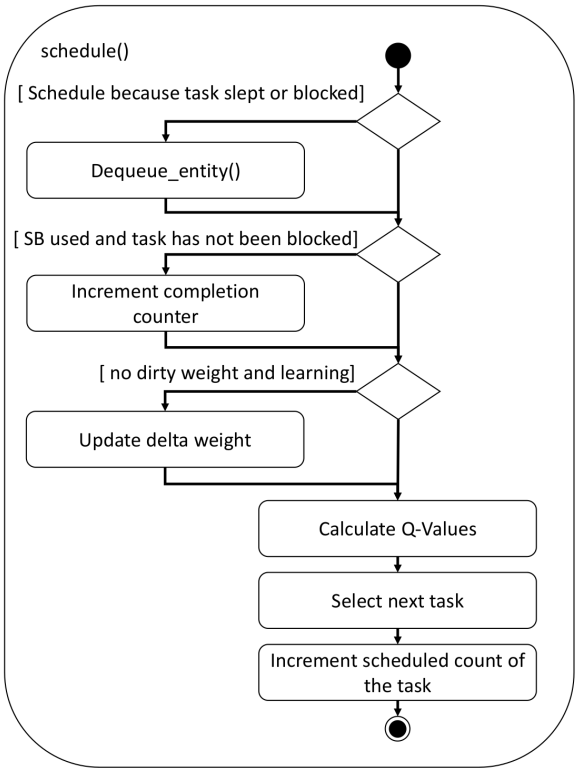


FIGURE C.2: Activity Diagram of the function schedule()

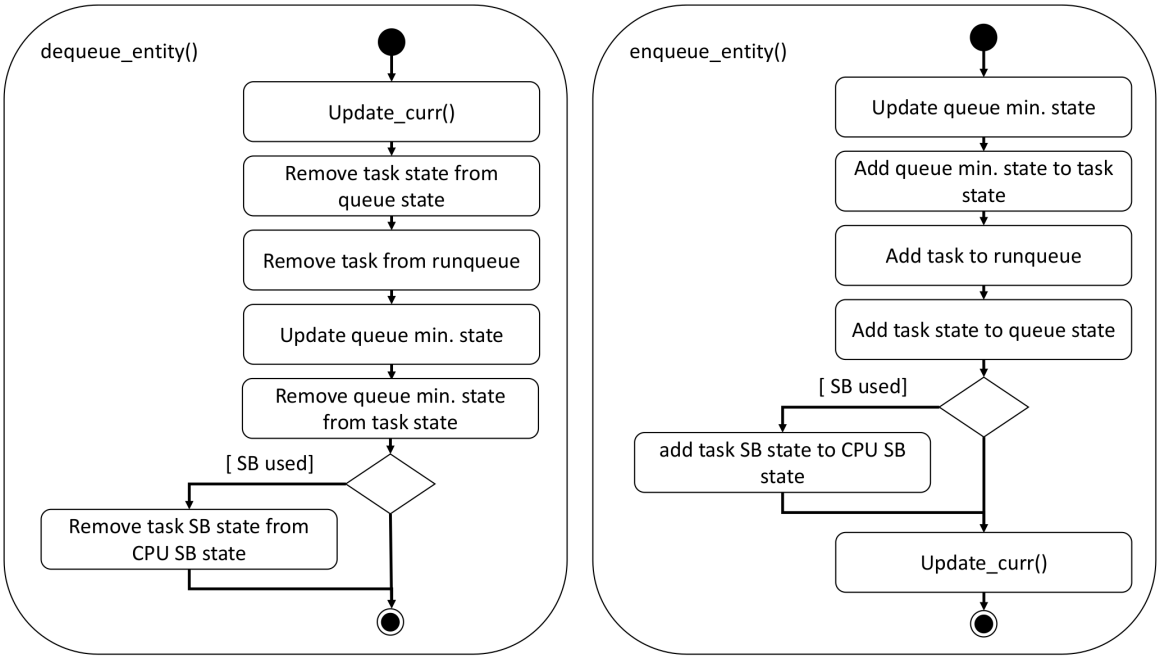


FIGURE C.3: Activity Diagrams of the functions dequeue_entity() and enqueue_entity()

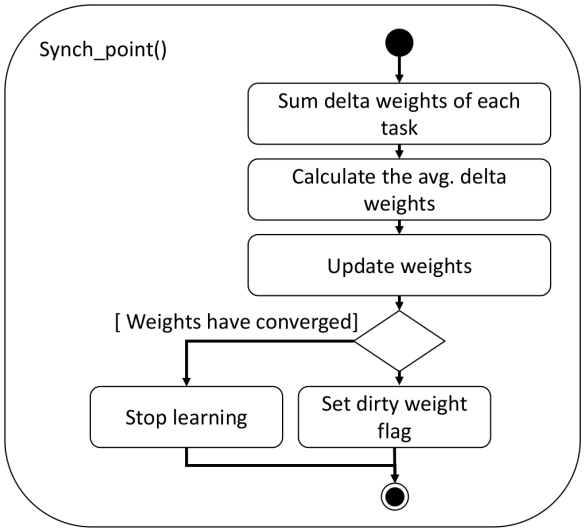


FIGURE C.4: Activity Diagram of the function synch_point()

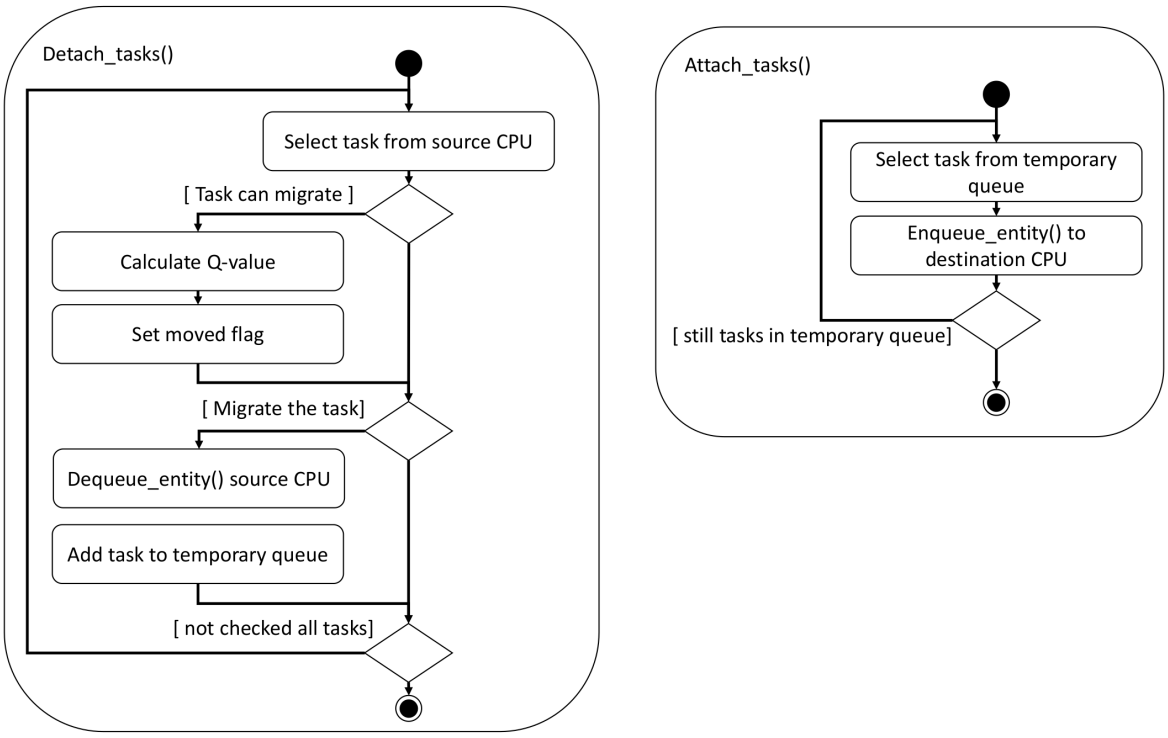


FIGURE C.5: Activity Diagrams of the functions detach_tasks() and attach_tasks()

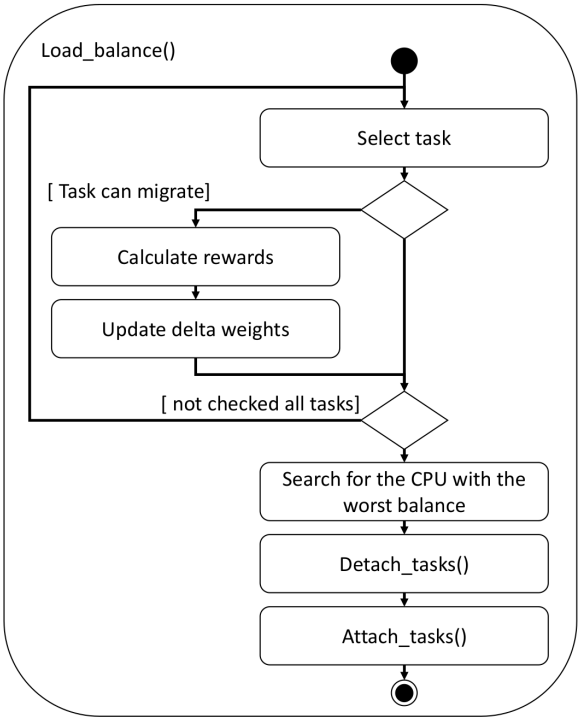


FIGURE C.6: Activity Diagram of the function load_balance()

D Kernel Configurable parameters

This appendix shows the possible configurable parameters defined for the artificial intelligence implemented in Linux. The parameters are:

- **SCHED_SFS**: activate SFS scheduler.
- **SCHED_SFS_TIMESLICE**: value of the timeslice used by SFS scheduler in ns
- **SCHED_SFS_SYNCH_TIMER**: interval between each weights' update in ms
- **SCHED_SFS_FIXP_SHIFT**: number of bits representing the fractional part
- **SCHED_SFS_ALPHA**: SFS learning factor α
- **SCHED_SFS_GAMMA**: SFS discount factor γ
- **SCHED_SFS_W_PRIO**: initial weight for task's priority
- **SCHED_SFS_W_AGV_BLOCK**: initial weight for the average number of times
- **SCHED_SFS_W_AGV_SLEEP**: initial weight for the average number of times going to sleep
- **SCHED_SFS_W_AGV_LOCK**: initial weight for the average number of locks acquired
- **SCHED_SFS_CONV**: convergence threshold for the weights of SFS
- **SCHED_SB**: activate SB balance
- **SCHED_SB_SYNCH_TIMER**: interval between each weights' update in ms
- **SCHED_SB_ALPHA**: SB learning factor α
- **SCHED_SB_GAMMA**: SB discount factor γ
- **SCHED_SB_W_AVG_DEXEC**: initial weight for the average delta execution
- **SCHED_SB_W_AVG_SCHED**: initial weight for the average number of times the task has been scheduled
- **SCHED_SB_W_AVG_BLOCK**: initial weight for the average number of times blocked
- **SCHED_SB_W_AVG_COMP**: initial weight for the average number of completions
- **SCHED_SB_CONV**: Convergence threshold for the weights of SB
- **SCHED_SB_INTERVAL**: interval from which is based the average number of times the task has been scheduled