Lernmodul Process Scheduling

Vorlesung Betriebssysteme UniBe

März 2012

Inhaltsverzeichnis

Vorwort

Dieses Dokument soll eine Kopie des Kurses *Lernmodul Process Scheduling* bieten. Der Inhalt ist nicht von mir, sondern aus dem erwähnten Modul.
Minimale Korrekturen betreffen die folgenden Punkte:

- Tests, welche nicht gemacht werden sollen, wie "My Goals" und die im Kapitel 4 ("Personal Synthesis", "Survey") wurden gemäss Besprechung entfernt.
- Um der Gliederung in Folien Rechnung zu tragen, wurden zusätzliche Überschriften eingefügt. Diese wurden kursiv markiert.
- Formatierungsfehler wurden korrigiert (z.B. \222 zu ', fehlende Leerzeichen), leere Seiten und Abschnitte wurden entfernt.
- Die Formeln wurden neu gesetzt.

Kurzzusammenfassung Übungsstunde

Process Scheduling ist eines der beiden Module, die im Kurs Betriebssysteme anfallen (das andere wird später *File Systems* sein). Der Kurs ist auf **ILIAS** zugänglich. Geschätzter Aufwand ca. 22.5h.
Abgabe: Bis 16.04.2012, praktischer Teil per Email an Gerald Wagenknecht.

Zu erledigende Quizzes und Tests:

- Self Test (0p), Einschätzung
- Quiz (30p), Theorie
- "Final Quiz" (40p), Abgabe der Programmieraufgabe

Implementierung in Java mit Hilfe des PSSimulators, mit Template `ps_template.zip`.

1 Introduction

Process scheduling is the primary task of any modern operating system to handle multi tasking. Early operating systems did not offer multi tasking though, but the outstanding growth of CPU power as well as the development of thousands of applications and tools possibly used concurrently rapidly underlined the need of efficient process scheduling by the operating system. In this module, students learn about processes and scheduling processes under limited resources (CPU, memory, disks and other IO devices, ...). They also acquire knowledge on how theprocess scheduler of the operating system actually achieve its task. In order to apply their knowledge, the students implement severalscheduling algorithms inside a simulator.

1.1 Special Requirements

- Basic knowledge of the Java programming language.

1.2 Author

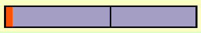
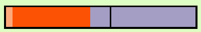
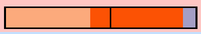
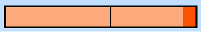
The module "Process Scheduling" has been created by *Frederic Aubert* and *Randoald Corfu*, University of Neuchâtel.

For further conceptual information regarding the module "Process Scheduling" please contact *frederic.aubert@unine.ch*.

1.3 Goals

- You study the concepts of process and process states.
- You learn about process scheduling and different algorithms in use.You have to pass the Self-test with no wrong answer, in order to prepare for the Quiz. The Quiz will be scored by your tutor.
- You apply your knowledge by implementing three scheduling algorithms insidea simulator. You perform actual simulation with given input, in order to compute specific statistics that you will be asked to comparatively discuss. Your written code as well as your comparative discussion will be evaluated by your tutor.

1.4 Schedule

1 Introduction		1:00
2 Theory		9:20
3 Knowledge Application/Exploration		10:50
4 Prove Your Knowledge and Skills		1:20
	overall duration	22:30

Legend:

🟥	time required for the current chapter of the module
🟧	time already passed in the module
🟪	time still available in the module

2 Theorie

2.1 Introduction

When using any modern computer, we often have the feeling that more than one program is executed at the same time. However this is not the case, since the processor, the key unit that executes the instructions of a program, can only handle one instruction at a time. In the reality, the processor swaps between all programs, allocating in turns a time slot to each of them during which some but not necessarily all instructions are executed. These time slots are small enough to give to the user an illusion of parallelism. We say that such Operating System handle multi tasking.

True parallelism can of course be achieved on multiprocessor computers. But even on such architectures, we often face the challenge that we have more programs in execution than we have processors in the machine. We again are in need of multi tasking. Furthermore since the memory is usually shared by all processors, the management of all parallel activities become even more complex.

To address the multi tasking issue, the early developers of operating systems formulated a model based on independent *sequential processes* operating in a *multiprocessor environment*. *Sequential process*, also simply called **process**, consists of a program in execution to which is associated a program counter, some registers, and a pool of memory which holds the content of program variables. Conceptually each **process** has its own *virtual processor*. It shouldn't have to worry about whether it is going to be executed on a single processor or on a multiprocessor computer.

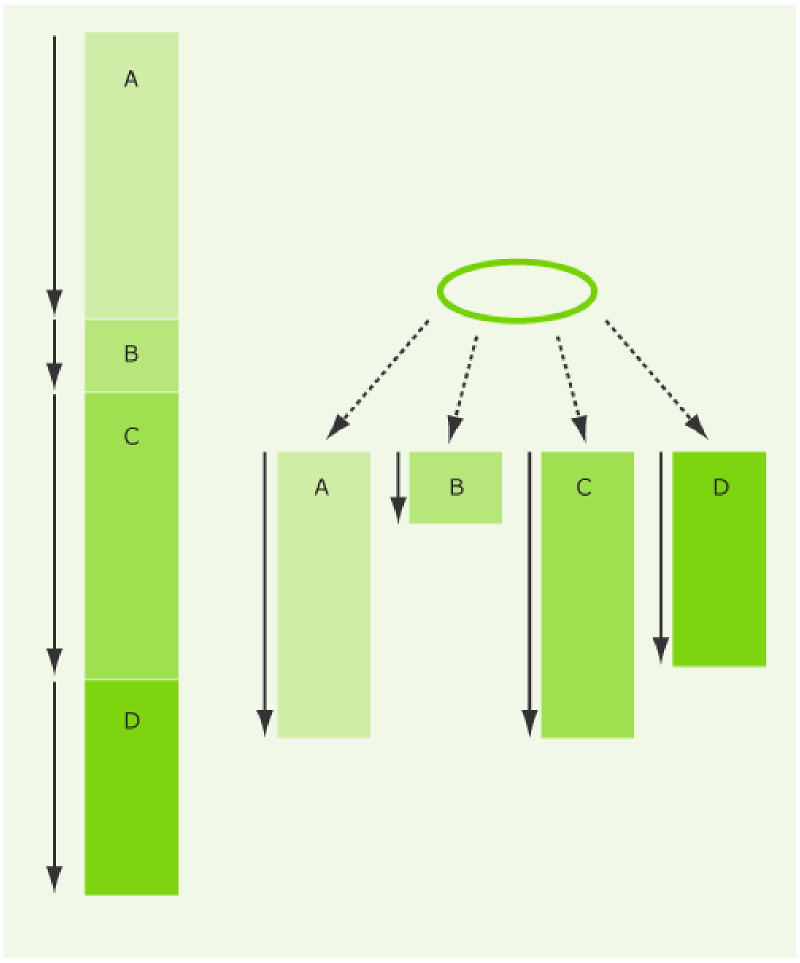


Abbildung 2.1: Figure 1

Let's consider that we have four processes, A, B, C, and D as shown in figure 2.1. For a process to be executed, all instructions need to be executed, from the first one to the last one. On the left side, the four processes are executed one after the other, sequentially, on a single processor computer. There is no multi tasking. On the right side, the four processes are executed simultaneously, on a multiprocessor computer that has at least four processors. The multiprocessor architecture provides multi tasking by itself.

Now let's imagine we have a dual processor computer. On the left side, the four processes are executed sequentially and independently on the two processors. When one process terminates, the next one takes its place. Here again the multiprocessor architecture provides multi tasking by itself. On the right side, the four processes are sliced into small fragments, which are then executed sequentially. When one fragment terminates, the next one takes it place. This rule allows for example a process to start on processor I, pass twice on processor II, and finish on processor I. Here, the time slot mechanism provides multi tasking. We can finally notice that multi tasking decreases global execution time.

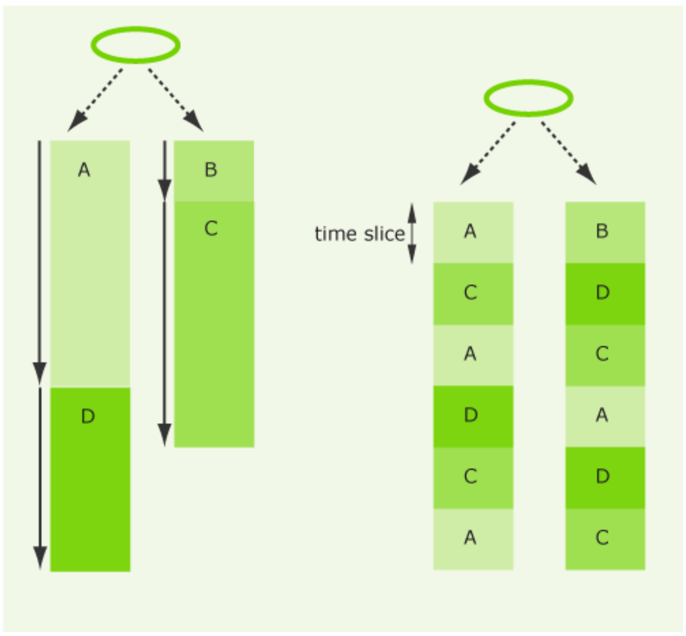


Abbildung 2.2: Figure 2

2.1.1 Single vs. Multi-Threading

The process model discussed so far implies that by itself a process, consisting of a program being executed, only does one task at a time. We say that such process has a single thread of execution, or simply that the program is single threaded.

This limitation may sometimes be very undesirable for a program: for example, when a user action on a graphical interface requires a complex calculation, the user may not be willing to wait for the answer before doing something else. Several threads of execution are needed to allow parallel work in the program. We therefore say that such program is **multi threaded**.

Multi threaded programs are obviously better adapted to run on multi processor computers, in the same way that multi process environments are much more comfortable for the user experience on multi processor computers. But once again, such programs can also be executed on a single processor, requiring each thread to share the processor. We will assume that the means and rules to give the processor to a process can also be used inside a multi threaded programs to give the processor to a thread.

2.1.2 Resources

To be executed, a process requires resources. There are different kinds of **resources**:

- hardware components of the computer, such as the processor, the memory, the hard drives, the mouse, the keyboard
- logical components of the operating system, such as file for data storing or socket for network access

In the classical computer architecture, the main resources are the **processor**, **the memory**, the **hard drives**, and the other **IO devices**. In this module, we fill focus on means and rules to grant a process access to processor for some time **(CPU time)**. We will talk about the access to the memory in the *Memory Management* module, the access to hard drives in the *File System* module, and access to other IO devices in the *Device Drivers and Input/Output module.*

2.2 Processes

A program becomes a **process** as soon as it gets executed. A program is a passive entity, whereas a *process* is an active entity, associated with a **program counter** specifying the next instruction to be executed and a set of resources that it requires. Multiple copies of the same program may be run simultaneously, each of them as a separate process. One user can invoke several copies of his/her web browser, or several users can open each their spreadsheet program.

An *Operating System* can be seen as a **collection of processes**. During its whole life time, some processes exist from the time the system starts through the time it shutdowns while others are spawned when needed. A user may spawn a new process when clicking on an icon representing an instant messenger program, or a process can spawn at any time one or more new processes.

In normal circumstances, a process terminates when it finishes its task. A compiler stops when the target *source code* is translated into *object code*. A word processor stops when the user exits it. When an unexpected error occurs during the execution of a program, the process also terminates.

2.2.1 Types and hierarchy

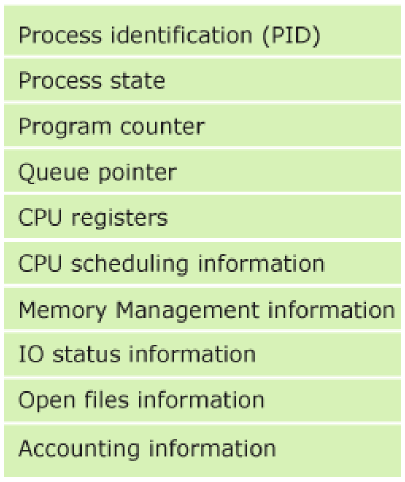
We can classify processes in two types:

- **Foreground processes**, essentially users initiated processes
- **Background processes**, essentially system processes devoted to common tasks, such as file printing, automated email retrieving. Such processes are often called ***daemons*** under UNIX and ***services*** under Windows

A process can spawn at any time a **child process**, which behaves like any other process. Under UNIX systems, the *parent* and *child* processes remains in a relational hierarchy, which implies some constraints. A parent process cannot terminate as long as any of its children are running. If a parent process is forced to terminate (killed), any of its children that are running are stopped at once. This is not the case under Windows, since a child process spawned by another process doesn't stay permanently in a hierarchical dependency with its parent.

2.2.2 Process Control Block

To manage the independent process model, it is necessary to have a special place in memory holding a **table of processes**. The table has an entry per process, called the **Process Control Block (PCB)**. Each entry keeps a number of information related to the process. We will discuss most of them in this module or in other modules.



Process identification (PID)
Process state
Program counter
Queue pointer
CPU registers
CPU scheduling information
Memory Management information
IO status information
Open files information
Accounting information

Abbildung 2.3: Figure 3

2.2.3 Process states (life cycle)

When a process is **created**, it is admitted into the table of processes with the state ***new***. At that time, there is no value for the program counter. As soon as the requested resources are identified and **allocated**, it is promoted to the state ***ready***.

The next step is the process to be granted access to the processor. This decision is made exclusively by a special program, the scheduler, which is implementing a scheduling algorithm as we will see later on. When access is granted, the process reaches the state ***running***. It is important to note, that at any given time only one process per processor can have the state running. The scheduler will take the process from the processor at the expiration of its time slot, making it fall back to the state ***ready***.

A running process might sometimes request some other resource or be blocked by another process. In those cases, it leaves the processor by itself and enters the state ***waiting***. It is now dependent on an event external to the processor. When that event occurs, the process will be promoted again to the state ***ready***. Finally a process that has completed its task goes to the state ***terminated***. Its associated PCB is deleted from the table of processes.

From a logical point of view, it has to be noted that the two states ready and run-

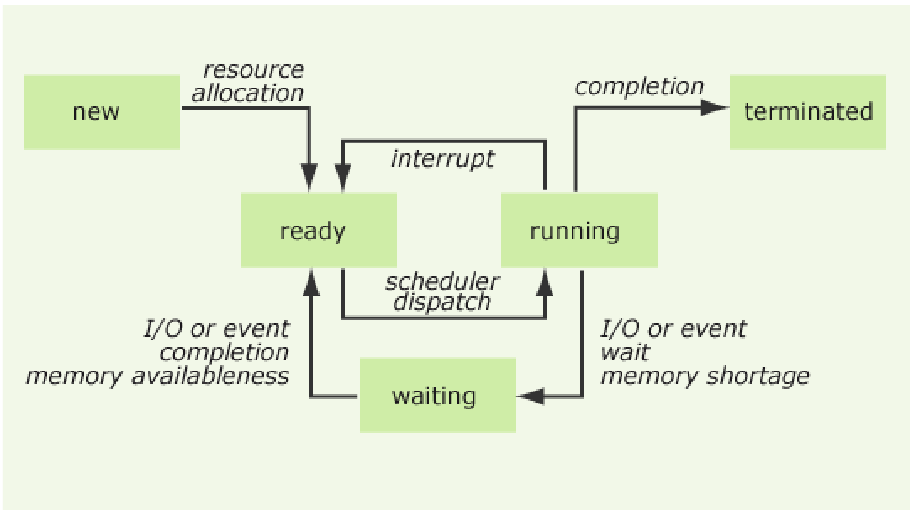


Abbildung 2.4: Figure 4

ning are semantically equivalent. In both cases the process can be executed, however in the former case, the processor is temporarily unavailable.

On the other hand, the state ***waiting*** implies that the process cannot be executed, even if the processor has nothing to do. Depending on the resource that is requested by the process, it finds itself in one of several kinds of waiting situation:

- ***IO waiting (short term)***. The process has to wait for an IO device to execute an action (i.e. reading from keyboard, writing to file) and subsequently producing an interrupt.
- ***Memory waiting (medium term)***. The process has to wait for memory allocation, possibly requiring the memory management unit to initiate a page replacement.

2.2.4 Context switching

We saw that sometimes the scheduler might take a process from the processor, in particular when its time slot expires. In order for the process to restart from where it lefts the next time it is granted access to the processor, all necessary information must be saved. This is done by storing the program counter and all CPU registers in the PCB of the process. At the time the process is granted access to the processor again, the scheduler restores the program counter and the CPU registers in the processor, so that the process can continue its task where it stopped. This mechanism, called **context switching**, is illustrated in the next figure. It has to be noted that context switching is a time consuming mechanism.

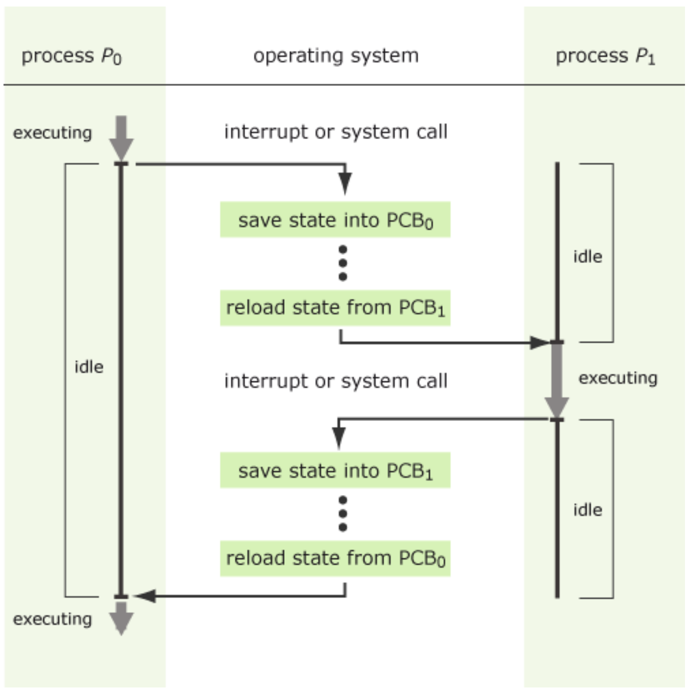


Abbildung 2.5: Figure 5

During the operation, the processor doesn't do any meaningful computation, it ist **idle**. Thus, special attention must be given to the frequency of context switching. In particular if time slots get too small, the processor will spend most of its time switching from one process to another.

2.2.5 Interruption

An **interruption** is an unexpected event, breaking the sequential progression of a program. It can be emitted by a hardware component or even by the processor itself when an error occurs during the execution of an instruction. Normally, the running process is taken from the processor for the interrupt to be handled. The interruption mechanism will be covered in more details in the **Device Drivers and I/O module**.

2.3 Multi Processor Architectures

As mentioned earlier, the general problematic is to grant m simultaneous processes access to n processors. In this chapter and in the next one, we will discuss solutions to achieve this goal restricting us to the case where we only have ***one*** processor. These solutions can later be extended to *multi processor* architectures.

2.3.1 Objectives

At this point we need to point to the difference between *scheduling policies* and *scheduling algorithms*. **Scheduling policies** are tightly tied to the objectives of a system and therefore can be arbitral; **scheduling algorithms** on the other hand are means to apply a policy and are defined as strict mathematical rules.

We can now differentiate two kinds of objectives. General objectives that apply in all situations and system dependent objectives that apply in specific situations, in which the system has particular requirements.

General Objectives The idea is to maximize the global usage of the computer system and its components; different policies can be taken into consideration:

- **Optimizing CPU usage**, minimizing its idle time, i.e. periods during which the CPU has nothing to do.
- **Maximizing the process throughput**, i.e. the number of processes that are granted access to the processor.
- **Minimizing the turnaround time**, i.e. the time it takes for a process to go through all its states, from its creation (created state) to its death (terminated state).
- **Minimizing the waiting time**, i.e. the time a process remains idle in the ready queue.

System dependent objectives These objectives can be strongly dependent on the characteristics of the system:

- **Real-time systems**. On such system, the policy is to *guaranty* access to the processor each time a process needs it.

- **Timesharing systems**. On such system, the policy is to *optimize* the response time to the user.

2.3.2 Approaches

We have to distinguish between three scheduling approaches: **long term**, **medium term** and **short term**. The mechanisms are different for each of them; in particular the choice of a policy is driven by the objectives that need to be achieved.

Long term scheduling applies to programs, mostly daemons also known as background services, which need to run once at specified time or repeatedly with given recurrence (every hour, every day, every month, and so on). It is its responsibility to promote programs to the new state of a process when the system is not overloaded. This scheduler is not going to be discussed in this course since it does not present any special interest in the context of operating systems.

Medium term scheduling applies to processes, which require large amount of memory that cannot be granted at present time. It is its responsibility to swap out of the memory some processes to allow process in the *waiting state* to be promoted to the *ready state*. This scheduler is going to be discussed in the *Memory Management* module.

Short time scheduling applies to processes that are in the *ready state*. It is its responsibility to grant processor time to processes that need it. This scheduler is going to be discussed in the next chapter through the end of this module.

2.4 Short time scheduler

The **short time scheduler** is a special program that selects among all ready processes the process that will be granted access to the processor. In case the time slot mechanism is in use, it also calculates the length of the time slot it will be given, depending on the number of other ready processes.

2.4.1 Types and properties

We have seen so far that at any given time aprocess is either using/requiring the processor, or waiting on some IOresources which when allocated will resume the process. Thus, a processexecution can also be defined as the strict alternating of CPU execution and IOwait, called respectively **CPU burst** and **IO burst**. Processes withlong CPU bursts and few IO bursts are said to be **CPU bound** while processes with a high rate of IO bursts and usually short CPU bursts are said to be **IO bound**.

2.4.2 Pitfalls

When developing a short time scheduler, we need to be aware of some common pitfalls and take them into consideration.

- **Starvation**. Starvation occurs when a process is never granted access to the processor. Several reasons could be source of starvation. Algorithms must be evaluated and proved against starvation.
- **Context switching**. We can't avoid this mechanism totally because it is the fundamental requirement to share the processor between processes. A process state transition to or from running involves context switching. It has to be considered as expansive as some CPU time is lost to non process computing activities. Algorithms have to take care of keeping context switching at minimum level.

2.4.3 Queues

The overall process scheduling is regulated by the use of **queues**. There are three types of queues.

- **Job queue**. Each newly created process is placed in the *job queue*, associated with the state new. This queue contains all the processes in the system.
- **Ready queue**. Each time a process is granted access to the memory and have obtained all necessary resources, it is placed in the *ready queue*, associated with state *ready.*
- **Waiting queues**. When a process is not terminated and leaves the processor, it is placed in a *waiting queue*, associated with the state *waiting.* There are many kinds of waiting queues: *wait for interrupt handling queues*, *wait for child process termination queues* and *device queues.* Each device has its own queue, with a name related to the type of the device: keyboard, screen, disk# 0, disk# 1, microphone, headset, terminal unit, and so on. Each time a process requires such a device, it is placed in the relevant **device queue**.

The *ready queue* and the *device queues* are implemented internally as *linked lists.* Each element of the list is a Process ControlBlock (PCB) entry. The queue header holds pointers to the first and last PCBin the list. Each PCB holds a pointer to the next PCB in the queue, as shown in the figure 2.6.

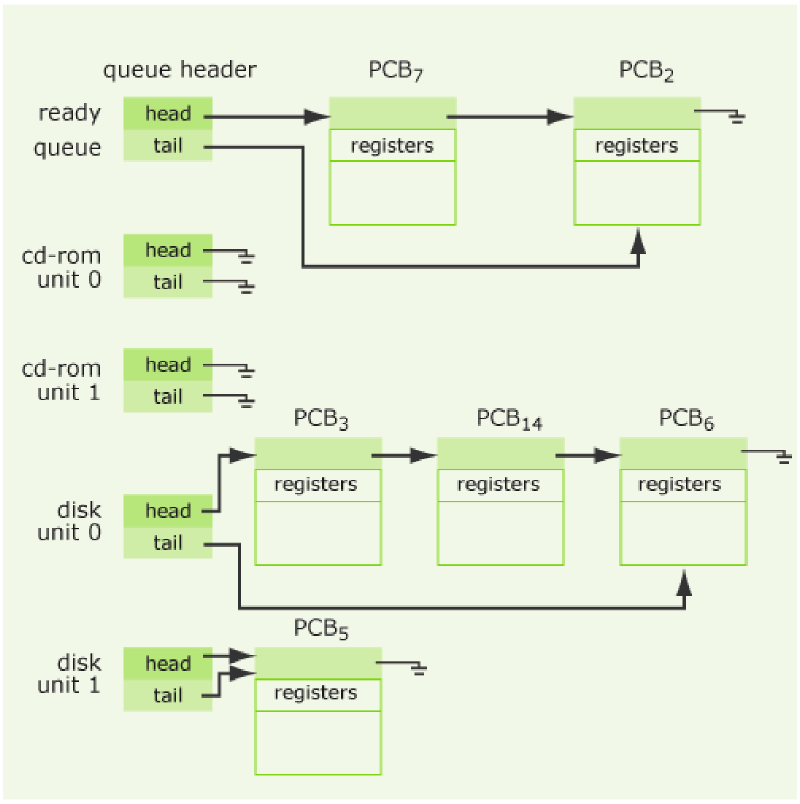


Abbildung 2.6: Figure 6

Queues are events driven. When a process releases a resource the first process in line in the corresponding queue isgranted access to it. If the later process now has been granted access to allrequested resources, it is placed back in the *ready queue*. The *processor* is associated with the *ready queue*, and the *hard drives* and other *IO devices* with their own *device queue*. In the remainder of this module, we will focus on the *processor* and the *ready queue*, leaving specific device queues mechanisms to be discussed in the *Device Drivers and I/O* module.

When a process is granted access to the processor, because it has all requested resources, it is associated with the state *running*. Such process doesn't belong to the ready queue or to any device queue anymore. The process can now either finish, and be associated with the *state terminated*, or leave the processor for numerous other reasons:

- The allocated time is over. The process is taken from the processor and is placed back in the ready queue, this mechanism is called **preemption**.
- The process is interrupted. The process is taken from the processor and is placed back in the wait for *interrupt handling queue*.
- The process might request access to an IO device. The process leaves the processor

and is placed in the *device queue* of the corresponding device.

- The process might spawn a new child process and wait for its termination. The process leaves the processor and is placed in the *wait for child process termination queue*.

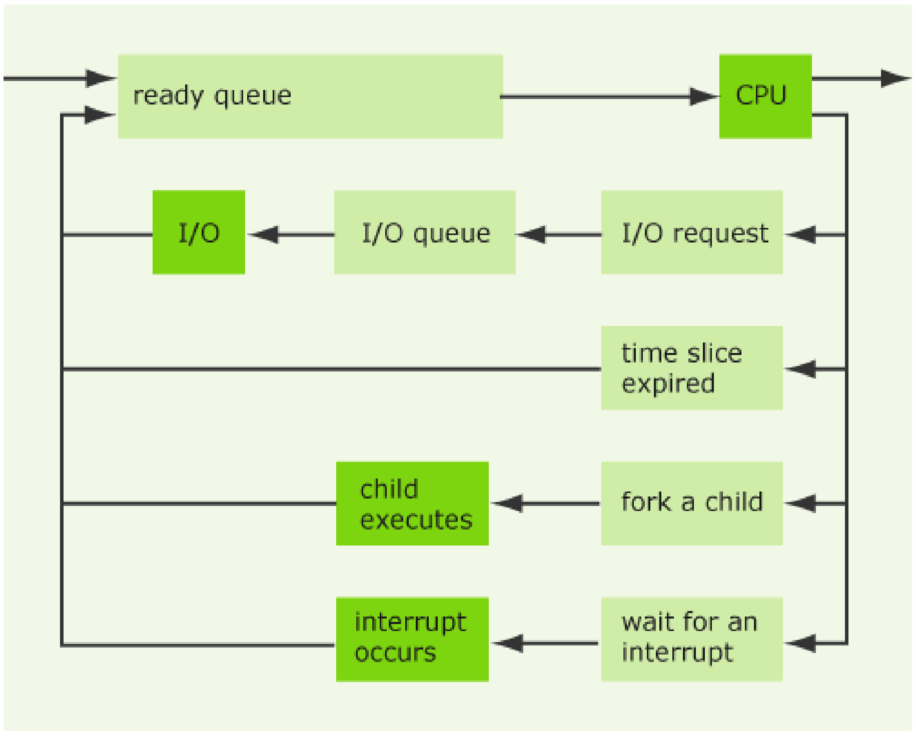


Abbildung 2.7: Figure 7

On *time-sharing system*, there is also a medium term scheduler, responsible of sharing access to the memory. A process in the *ready queue* can be swapped out to free memory and to allow another process to join the *ready queue*. This approach can be viewed as an additional event, as shown in the next figure 2.8.

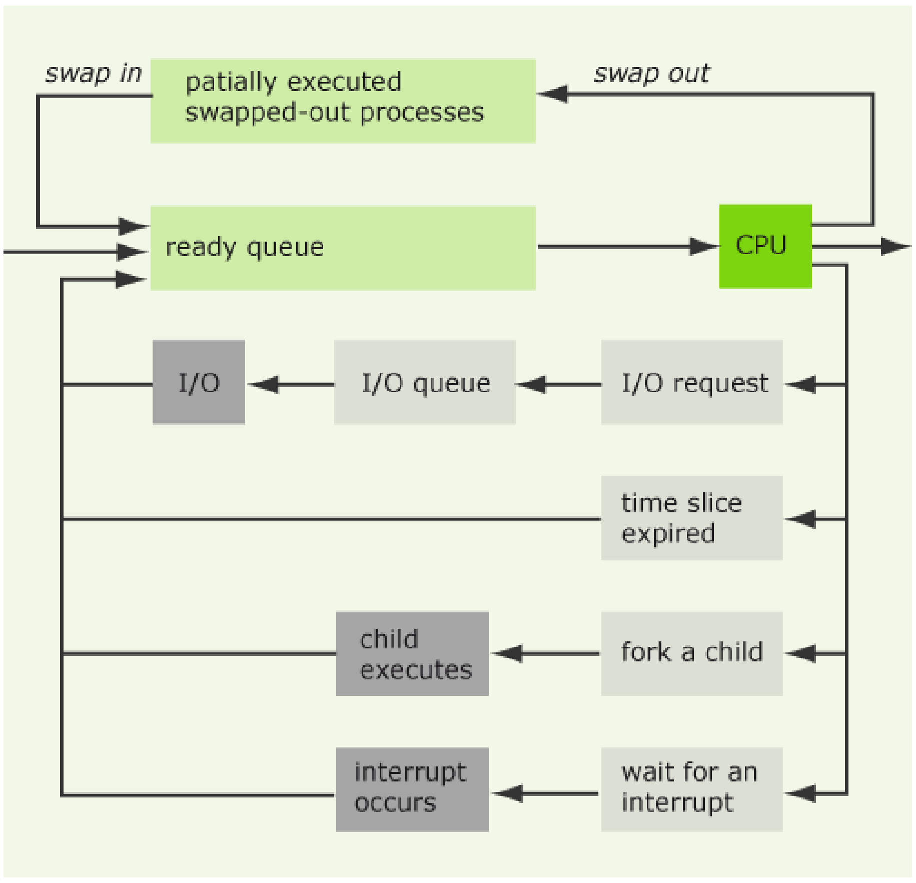


Abbildung 2.8: Figure 8

2.4.4 Algorithms

In this section, we will treat of algorithms that pick a process in the ready queue and give it to the processor. We will focus ourselves to the single processor case, since multi processors scheduling has the exact same problematic as soon as a process is dispatched to a processor. We will discuss the multi processor case later in this chapter.

Scheduling algorithms can be divided into two categories in respect to how they deal the allocation of processor time. A **non-preemptive** scheduling algorithm grants access to the processor for unlimited time. The process run until it finishes, blocks due to an IO request or blocks to wait for the termination of another process. A **preemptive** scheduling algorithm grants access to the processor for a limited time. If the process is still running when the allocated time is over, it is taken from the processor, placed back at the end of ready queue, and replaced by the next process in the ready queue.

Non-preemptive algorithms

- **First-Come-First-Served (FCFS)**
  First-Come-First-Served algorithm is the most standard non-preemptive algorithm. Processes that request the CPU are granted access to the processor in order of arrival. This is simply accomplished by organizing the ready queue as a FIFO queue. When a new process arrives, its PBC is linked to the tail of the queue.

  Let's consider that we have four processes arriving in the order P1, P2, P3 and P4 with run time of respectively 12, 6, 4 and 2 minutes. FCFS algorithm will run them in their order of arrival. We can calculate the waiting mean time (WMT) and

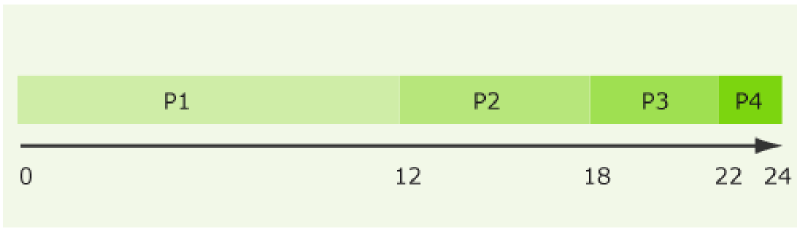


Abbildung 2.9: Figure 9a

  the average turnaround time (ATT).
  - WTM = (0 + 12 + 18 + 22)/4 = 13 minutes
  - ATT = (12 + 18 + 22 + 24)/4 = 19 minutes

- **Shortest-Job-First (SJF)**
  Shortest-Job-First algorithm is an alternative to FCFS. Processes that request the CPU are granted access to the processor starting with the shortest job. Let's go back to our previous example, we still have four processes arriving in the order P1, P2, P3 and P4 with run time of respectively 12, 6, 4 and 2 minutes. SJF algorithm will run them in the order P4, P3, P2 and P1. We can again calculate the waiting mean

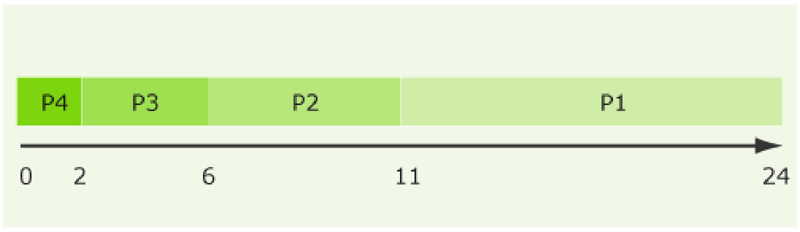


Abbildung 2.10: Figure 9b

  time and the average turnaround time.
  - WTM = (0 + 2 + 6 + 12) / 4 = 5 minutes
  - ATT = (2 + 6 + 12 + 24) / 4 = 11 minutes

We can see that SJF gives better results than FCFS and seems to be optimal. If the run time are a, b, c and d respectively, the average turnaround time is (4a + 3b + 2c+ d)/4. It is obvious that a contributes most, so to minimize the average turnaround time a should be the shortest job, b should be the next, then c and finally the longest job d as it affects only its own turnaround time.

However in the case the processes aren't arriving at the same time, and this is more realistic in common usage, SJF algorithm can give much worst results. It also requires the knowledge of the overall run time of each job, which also very often is not possible. Thus, SJF algorithm can hardly be used in modern systems.

Preemptive algorithms

- **Round-Robin (RR)**
  Round-Robin algorithm is the most standard *preemptive* algorithm. It is based on FCFS algorithm, with the difference it artificially cuts the CPU time intoslices, called **time quanta**. As before, the *ready queue* is organizedas a FIFO queue and whenever a new process arrives, its PBC is linked to the tail of the queue.

  The processor is always granted to the first process in the queue but at most for a limited amount of time, and at this stage three scenarii can occur:
  - The process may finish in the given quantum, and thus leaving the processor willingly and being associated with the state *terminated.*
  - The process may, before the end of the quantum, be interrupted, request access to an IO device or spawn a new child process subsequently waiting for its termination, and thus leaving the processor willingly and being associated with the state *waiting.*
  - The process may still have instructions to execute at the end of the quantum. The processor is taken away, and is given to the next process in line. Thus the former process still associated with the state *ready* is placed back at the end of the *ready queue* to take another share of the processor when its turn comes again.

  As noted earlier, taking the processor from a running process at the end of a quantum and giving it to another is time consuming. The time used in context switching is lost in the sense the processor doesn't do any meaningful computation. The performance of Round-Robin algorithm therefore heavily depends on the ratio of context switching length versus quantum length. For example, if we measure the context switching as 20 time units and the quantum is 100 time units, we can estimate that about 20% of the CPU time is wasted. This could lead us to choose long quanta, so less time is wasted in context switching, but in this case the algorithm would becomes less and less preemptive degenerating into its base non-preemptive version FCFS.

The length of the *quantum* is obviously influencing the performance of the system. A given quantum could minimize the waiting mean time, while another could minimize the average turnaround time. If quanta tend to be small, the system will have shorter response time giving the user a greater feeling of parallelism. As a rule of thumb, we can say that in *time-sharing systems*, which widely make use of Round Robin algorithm, more than 50% of the CPU bursts should be shorter than the quantum.

- **Shortest-Remaining-Time-First (SRTF)**
  Shortest-Remaining-Time-First algorithm is an improvement of SJF algorithm, which we remember rely solely on the overall run time of the jobs. We have seen earlier that processes strictly alternatebetween CPU bursts during which it needs the processor and IO bursts during which it waits for resource allocation. When a process that has been waiting on a resource is ready to resume, it is reintroduced in the *ready queue*. At this stage, the remaining run time of the job has decreased. Furthermore if the scheduler is able to make an estimation of the length of the next CPU burst of such process, which is obviously less or equal than the remaining time of the job, it can also use this new information to elect the process with the shortest next CPU burst to be granted access to the processor. Note that so far this SRTF version is still non-preemptive letting processes run through the end of their CPU bursts.

  The biggest issue remains, like in SJF, that in realistic environment the processes aren't arriving at the same time and that early CPU bound processes can tie up the processor. SRTF is to reorder the ready queue each time a new process joins the queue or that a waiting process is resumed. If the first process in the *ready queue* now has a shorter next CPU burst than the currently running process, the later is preempted and the processor is granted to the former. Thus we have a preemptive SRTF version, and we can see with the next example that it is more efficient than the non-preemptive version.

  Let's consider that we have four processes. P1, arriving at 0 minute, has a run time of 7 minutes. P2, arriving at 2 minutes, has a run time of 4 minutes. P3, arriving at 4 minutes, has a run time of 1 minute. P4, arriving at 5 minutes, has a run time of 4 minutes.

  Non preemptive SRTF will run them as follow. We can calculate the waiting mean

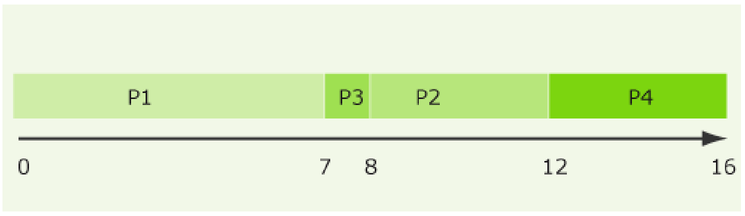


Abbildung 2.11: Figure 10a

time (WMT) and the average turnaround time (ATT).

- WTM = (0 + 6 + 3 + 7)/4 = 4 minutes
- ATT = (7 + 10 + 4 + 11)/4 = 8 minutes

Preemptive SRTF will run them as follow. We can again calculate the waiting mean

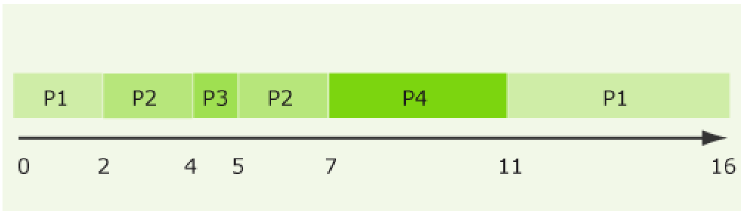


Abbildung 2.12: Figure 10b

time and the average turnaround time.

- WTM = ((0+9)+(0+1)+(0)+(2))/4 = 3 minutes
- ATT = (16 + 5 + 1 + 6)/4 = 7 minutes

- **Next CPU Burst Estimation**
  As we have seen earlier, SFJ algorithm, as well as its SRTF improvement, is worth consideration since it has been shown to give optimal average turnaround times. However in order to be applicable it relieson an estimation of the overall run time of each job or at least of the length of the next CPU burst.

  If we can make the assumption that the sequence of CPU burst and IO burst of a process is quite regular over its lifetime, we can approximate the length of the next CPU burst as being close to the previous ones. For the calculation, we use the following **exponential average formula** for the estimation of the length of the next CPU burst.

$$\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha)\tau_n \tag{2.1}$$

  where t_n is the length of the last CPU burst, τ_n is the average of the length of past CPU bursts, and α is a tuning parameter in the range $[0; 1]$.

  We can now discuss the value to give to α.

  - $\alpha = 0$, then $\tau_{n+1} = \tau_n = ... = \tau_0$. Only the first CPU burst is considered to be meaningful.
  - $\alpha = 1$, then $\tau_{n+1} = t_n$. Only the latest CPU burst is considered to be meaningful, we forget immediately about past history.

  - $\alpha = 0.5$. We average old and recent CPU bursts. As example, we estimate the sixth CPU burst as follow, $\tau_5 = 0.5t_4 + (0.5)^2t_3 + (0.5)^3t_2 + (0.5)^4t_1 + (0.5)^5(t_0 + \tau_0)$, where τ_0 is an arbitrary starting value statistically given by typical processes. We notice the decreasing importance of old CPU burst in the average over time. This is known as aging average, which often is used when prediction must be made on the basis of previous measurement.

  The formula becomes the following:

$$\tau_{n+1} = \frac{(t_n + \tau_n)}{2} \tag{2.2}$$

- **Priority Scheduling**
  We can also associate with each process a notion of priority. The implicit scheduling policy requires that a higher priority process has to be run before a lower priority process. So far FCFS and RR can be considered as equal priority scheduling, while SJF and SRTF can be considered as priority scheduling, with an associated priority defined as the inverse of the overall run time respectively the next CPU burst.

  But in most cases priorities are system or user specified. Processes can be categorized in groupsaccording to their importance in the system. For example, we can have systemprocesses which ensure the fundamental functionalities of the system, interactive processes which are IO driven, service processes which provide additional functionalities and computation processes which are CPU driven. Each group is assigned a priority level, with an associated quantum and its own *ready queue*, and at any given priority level processes are treated sequentially using Round-Robin algorithm. As long as a process stays in a higher level ready queue, processes in a lower level *ready queue* are not granted access to the processor. This has the major drawback that it can lead to **starvation** (also known as **indefinite blocking**).

  Solutions have been developed though to make this class of algorithms usable. One idea is to allow the priority of a process to dynamically change over time adjusting itself based on different criteria. In order to illustrate this, let's consider having *P* priorities, ranging from *0* being the lowest to *P-1* being the highest, each of them having their own *ready queue*. When a process is created, it is associated with a priority, its **nominative priority**, depending on its nature and placed in the corresponding level *ready queue*. During the entire lifetime of a process, its priority can be increased or decreased according to specific criteria through events, moving the process up or down within some part or even the whole queue hierarchy. At this stage it has to be kept in mind that system processes should always have a higher priority than user processes.

  Firstly preemption could force the process priority to be decreased by one or more units. Converselythe passing of time could slowly increase the priority of each ready by a given amount, this mechanism being known as an **aging**. Furthermore whenever a process re-enters the *ready queue*, for example after the completion of an IO request, it could do it at its nominative priority, at its latest running priority (the priority it had at the time it left the processor), or at a priority directly proportional to the unused percentage of its last running quantum making place to complex policies and optimizations. It has to be noted, as a final consideration, that process priorities could be used not only to order the *ready queue* but also other *device queues* making devices more easily available to higher level processes.

2.4.5 Multiprocessor

We have seen in the introduction that multi tasking can obviously take advantage of multiprocessor computers, which allows morethan one process to be executed simultaneously. However on such systems the improved efficiency comes at the cost of more complex scheduling decisions, especially in the case of some device belonging exclusively to a givenprocessor.

We distinguish two types of multiprocessor architectures: **symmetric** and **asymmetric**. In a symmetric multiprocessor (SMP), also known as a homogenous system, all processors areidentical. Each processor is said to be **self-scheduling**, which mean it alones makes all scheduling decisions. In an asymmetric multiprocessor (AMP), however, one processor, called the **master processor**, takes the responsibility of all system tasks and specifically of all scheduling decisions dispatching the processes to the otherprocessors, called the **slave processors**, which are solely devoted to execute user (non system) tasks. Now, as we remember our first general objective is to optimize CPU usage, let's observe more closely how it relates to the two types of architectures.

On a symmetric multiprocessor (SMP), each process whether system or user can be executed on any processor. There are different approaches to scheduling in such context. A *common ready queue* can be shared by all processors, or a common controller can dispatch, possibly depending on the current load on each processor, the process to the *private ready queue* of the processor on which it will stay during its entire lifetime.

On an asymmetric multiprocessor (AMP), each user process can be executed on any *slave processors*. The *master processor* manages a *single ready queue* similarly to what have been discussed in earliersections. Processes are granted access, in the order of the queue, to the *slave processors* as they become free. Thus the load on each processor is inherently **balanced**. If it occurs that a process must be executed on aprocessor having specific characteristics, the *master processor* additionally manage a *ready queue* for each subset of processors sharing the same characteristics.

In SMP architecture with a *common ready queue* or in AMP architecture, a process might be executed on one processor, leave it after being preempted or making an IO request, and resume later on another processor. In this situation, known as **process migration**, *context switching* gets more expansive, since the cache of the newly used processor needs to be populated with data that is to be invalidated in the cache of the former processor. To retain efficiency, scheduling algorithms aim to associate a process with its most recent used processor, possibly making itwait a little bit longer in the *ready queue* in order to minimize or even avoid *process migration*. This is known as **processor affinity**. We can observe however that this policy opposes to the **load balancing** policy, which we remember aim to distribute the work evenly among all processors. Thus there is no absolutely best solution, and a trade-off mainly depending on hardware considerations and on the nature of the processes in the system need to be set.

2.5 Readings section

You have now reached the readings section that is subdivided into required and recommended readings.

Required readings are articles taken out of real life, originating from the area of the theory section. Recommended readings are interesting and fit the theory but are not a must; they are intended for people more interested into the study matter.

With these readings you get used to the style of scientific writting.

It is not easy to read and understand scientific articles. If you start reading with the abstract you get an overview of the work presented. The introduction leads to the presented facts and helps understanding them. The most important section is the discussion. That is where the gained results are brought into context.

2.5.1 Books

- Abraham Silberschatz, Peter Baer Galvin, Greg Gagne: Operating System Concepts, 7th Edition 2005, John Wiley & Sons, Inc.; ISBN 0-471-69466-5
  Part two - Process Managment, Chapter 3 & Chapter 5
- Andrew S. Tanenbaum: Modern Operating Systems, 2nd Edition 2001, Prentice Hall, Inc.; ISBN 0-13-092641-8
  Chapter 2 - Porcesses and threads, Section 2.1 & Section 2.5

2.6 Self Test

The Self Test helps you to decide whether you are ready for the Quiz or not. In case of wrong answers you get links to additional information where you can improve your knowledge. The Self Test is not reviewed. It is for your personal use and benefit only.

You can start the Self Test by following this link and clicking on the *"Start the Test"* button. There are no time limitations to this task and you can always come back.

Evaluation criteria: No points can be earned for this task.

2.7 Quiz

If you feel ready for the hands-on session and you are satisfied with your Self Test results, solve the Quiz. The Quiz results will be reviewed and evaluated by your tutor. You must finish the Quiz before proceeding to the hands-on session!

You can start the Quiz by following this link and clicking on the *"Start the Test"* button. There are no time limitations to this task and you can always come back.

Evaluation criteria: You can earn a total of 20 points for this task and each question gives a certain amount of points.

3 Knowledge Application/Exploration

3.1 Simulator Overview

In this section, we will introduce you to the simulation engine that you will use for the hands-on session of this module. The UniNE Process Scheduling Simulator is a Java program that is designed to run virtual processes in an environement consisting of virtual devices. It also gathers information throughout its execution in order to allow the computation of pertinent statistics.

- **Virtual devices** - A virtual device actually represent an hardware device in a real system. This can be any IO device, such as an hard disk, or another media device such as a Floppy drive, a CD/DVD drive or a Network controller.
- **Virtual processes** - A virtual process actually represent a process in a real system. The virtual process starts its life upon its creation, which can occur at any time. The virtual process is then in turns either requiring the processor, for a variable amount of time, or requesting any IO device existing in the environement, also for a variable amount of time. The last stage of a virtual process is its termination, which occur after it has completed its last instruction.

3.2 Simulator Components

Here we will introduce you to the three interfaces that you will have to know in order to make use of our simulation engine.

The *Kernel* interface is the core interface of the UniNE Process Scheduling Simulator. It represents the scheduler consisting of system call handlers and interrupt handlers that are to be called by the simulation engine at specific times on the basis of the life cycles of all the processes in to the system. It is made of the following methods:

- void systemCallInitIODevice(String deviceID, Simulator simulator) - This system call is issued once for each device listed in the devices file specifiedon the command line when the Simulator is executed.
- void systemCallProcessCreation(String processID, long timer, Simulator simulator) - This system call is issued once for each process, at the process' arrival time as specified in its data file. This system call indicates the arrival of the process in the system.

- void systemCallProcessTermination(long timer, Simulator simulator) - This system call is issued once for each process, once the process has completed its execution.
- void systemCallIORequest(String deviceID, long timer, Simulator simulator) - This system call is issued each time a user process requests an I/O Operation. Note that any requests for operations on devices that do not exist should cause the program to terminate with an error message.
- void interruptIODevice(String deviceID, long timer, Simulator simulator) - This interrupt may arrive from any I/O device created by a InitIODevice system call. If an interrupt arrives from any other device the scheduler should cause the program to terminate with an error message.
- void interruptPreemption(long timer, Simulator simulator) - This interrupt may arrive when preemption occurs.
- String running(long timer, Simulator simulator) - This method returns the name of the process currently in the running state. When no other process is ready to run the Kernel must report that the Ïdlepprocess is running.
- void terminate(long timer, SimulatorStatistics simulator) - This method is called when the simulation has completed. The code in this method should compute statistics and display the results of the simulation.

The *Simulator* interface provides you simple means to interact with the simulation engine. The following methods are designed to be used in any system call handler or interrupt handler:

- boolean schedulePreemptionInterrupt(long delay) - Schedule a timer interrupt to occur after the specified number of time units. The timer is set to begin when the system call that started it completes. Thus a process started to run when a timer is set will be allowed to run for a full ”delay” time units. Only one timer interrupt can be scheduled at a time. If the event queue already contains a timer interrupt that timer interrupt will be canceled and a new one created.
- boolean cancelPreemptionInterrupt() - Cancel a timer interrupt if one has been set. If a timer interrupt has been set, it is removed from the event queue. If no timer interrupt has been set this method does nothing.
- long queryOverallTime(String processID) - Query the overall time of a given process. If the process exists, its overal time is returned. If the process does not exist, the value -1 is returned.
- long queryBurstRemainingTime(String processID) - Query the remaining time of the current burst of a given process. If the process exists, its remaining time of the current burst is returned. If the process does not exist, the value -1 is returned.

The *SimulatorStatistics* interface provides you simple means to retrieve information from the simulation engine.The following methods aredesigned to beused specifically in Kernel's terminate method:

- long getSystemTime() - Return the current system time. This is the total number of virtual time units that have elapsed since the system has started. It should agree with the value obtained from getUserTime() + getIdleTime()
- long getUserTime() - Return the current user time. This is the number of virtual time units that the system has spent performing user operations.
- long getIdleTime() - Return the current idle time. This is the number of virtual time units that the system has spent executing the idle process.
- int getSystemCallsCount() - Return the current system calls count. This is the number of time the simulator has called a Kernel method.
- void formatStatistics(PrintStream stream, long system, long user, long idle, int calls, int saves, long wmt, long att) - Print statistics. This is the method to call with yourvalues to nicely print out the statistics. *System.out* should be used as the first formal parameter.

3.3 First Come/First Serve

3.3.1 Tasks

- You have to implement the *Kernel* interface for a First Come/First Serve scheduler.
- You have to print the overall systemTime, userTime and idleTime and compute the CPU usage.
- You have to print the overall system call count.
- You have to calculate the register saves count.
- You have to compute the average turnaround time (ATT).
- You are to compute the waiting mean time (WMT).

3.3.2 Indication

In Kernel's terminate method, you are to call SimulatorStatistics' formatStatistics method with appropriate parameters in orderto print all statistics in a standard way. The output ought to be similar to the following:

```
System Time: 24 [100.00%]
User Time: 24 [100.00%]
Idle Time: 0 [0.00%]
```

System Calls: 8
Registers Saves: 0

Av. Turnaround Time: 19.0
Wait Mean Time: 13.0

HINT: Some parameters can be received from the SimulatorStatistics class.

3.3.3 Submission

In order to complete this task, you will be using the OSLab Hands-On applet. You will use this applet to write your code, compile it, debug it and run an automated evaluation procedure. You will be required to successfully pass the automated evaluation procedure in order to receive a fingerprint that you will be asked to copy and paste to the appropriate field in the Final Quiz of the module.

3.3.4 Evaluation

The OSLab Hands-On applet will run your implementation of the scheduler with a randomized set of virtual devices and processes and automatically check your output for correctness against our reference implementation. Your tutor will also evaluate your coding style.

3.4 Round Robin

3.4.1 Tasks

- You have to implement the Kernel interface for a Round Robin scheduler with a time slice of 2.
- You have to print the overall systemTime, userTime and idleTime and compute the CPU usage.
- You have to print the overall system call count.
- You have to calculate the register saves count.
- You have to compute the average turnaround time (ATT).
- You have to compute the waiting mean time (WMT).

3.4.2 Indication

We consider that a preemption occurs a full time slice delay after a process started running. If the process leaves the CPU before the expiration of the time slice, this preemption is canceled.

In Kernel's terminate method, you are to call SimulatorStatistics' formatStatistics method with appropriate parameters in orderto print all statistics in a standard way. The output ought to be similar to the following:

```
System Time: 24 [100.00%]
User Time: 24 [100.00%]
Idle Time: 0 [0.00%]

System Calls: 14
Registers Saves: 6

Av. Turnaround Time: 16.0
Wait Mean Time: 10.0
```

HINT: Some parameters can be received from the SimulatorStatistics class.

3.4.3 Submission

In order to complete this task, you will be using the OSLab Hands-On applet. You will use this applet to write your code, compile it, debug it and run an automated evaluation procedure. You will be required to successfully pass the automated evaluation procedure in order to receive a fingerprint that you will be asked to copy and paste to the appropriate field in the Final Quiz of the module.

3.4.4 Evaluation

The OSLab Hands-On applet will run your implementation of the scheduler with a randomized set of virtual devices and processes and automatically check your output for correctness against our reference implementation. Your tutor will also evaluate your coding style.

3.5 Shortest Remaining Job First (preemptive)

3.5.1 Tasks

- You have to implement the Kernel interface for a **preemptive** Shortest Remaining Job First scheduler.
- You have to print the overall systemTime, userTime and idleTime and compute the CPU usage.
- You have to print the overall system call count.
- You are to calculate the register saves count.

- You have to compute the average turnaround time (ATT).
- You have to compute the waiting mean time (WMT).

3.5.2 Indication

In Kernel's terminate method, you are to call SimulatorStatistics' formatStatistics method with appropriate parameters in orderto print all statistics in a standard way. The output ought to be similar to the following:

```
System Time: 24 [100.00%]
User Time: 24 [100.00%]
Idle Time: 0 [0.00%]

System Calls: 8
Registers Saves: 3

Av. Turnaround Time: 11.0
Wait Mean Time: 5.0
```

HINT: Some parameters can be received from the SimulatorStatistics and the Simulator classes.

3.5.3 Submission

In order to complete this task, you will be using the OSLab Hands-On applet. You will use this applet to write your code, compile it, debug it and run an automated evaluation procedure. You will be required to successfully pass the automated evaluation procedure in order to receive a fingerprint that you will be asked to copy and paste to the appropriate field in the Final Quiz of the module.

3.5.4 Evaluation

The OSLab Hands-On applet will run your implementation of the scheduler with a randomized set of virtual devices and processes and automatically check your output for correctness against our reference implementation. Your tutor will also evaluate your coding style.