Chapter:
Context switching by the OS

In this chapter we are going to analyze the way the context switching, which is the main part of the OS, is implemented.

Approaching the problem of context switching

The term “context switching” in reality and in every day applications means the periodic switching of a context, which the processor is currently executing. For example, let the processor be in a specific state, from which we continue its operation and let it function. The steps followed by the processor will be determined by the current state of the memory and the instructions which are going to be executed one-by-one. Our task is to “recruit a scheduler”, a program, in other words, which will be executed for a short time interval in a periodic manner. This scheduler will schedule the next available context, aka process. And this will go on forever…

The main problem which we have to tackle is the fact that the scheduler must be “triggered” in a periodic manner. Hence, we need to search for a way make the ARM processor fire an interrupt-like response every specified amount of time. This can be done by using the internal peripheral offered by all ARM Cortex-M3 processors under the name “SYSTICK”

Some words about SYSTICK

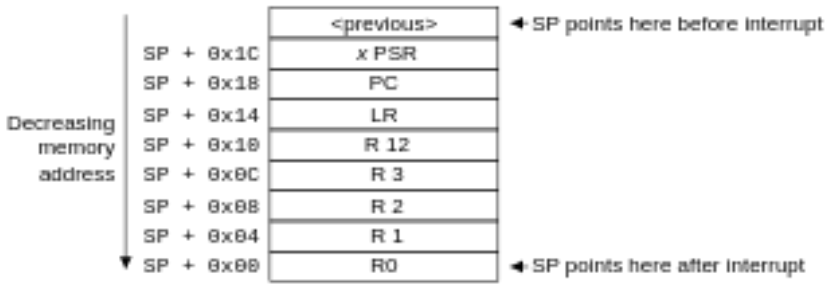
SYSTICK is a special 24-bit timer offered as a peripheral to any ARM Cortex-M3 processor. It counts down to zero, from where it reloads a special value set by the programmer. It has some memory mapped registers associated with it, from which the main ones are related to the current value(SYST_CVR) and the reload value(SYST_RVR) to reload when the timer reaches zero.

The main property offered by this special timer is the fact that it triggers a prioritized interrupt to the processor, which we are going to use in order to implement the scheduler. The period of the scheduler will be set by setting the associated register for the reload value, for which we must know the CPU clock to determine with accuracy the binary value. In the interrupt handler we are going to put the scheduler implementation, but first we must define what steps must be executed while scheduling the next process.

Exception handling: a brief overview

In order to implement the interrupt handler associated with the scheduler, we must understand first what happens whenever an exception occurs.

When an exception of any kind is triggered, the processor pushes to the current stack the following data



These data are poped from the when the exception returns.

Process scheduling; Top-level description

As we decided till now, the way we are going to implement the scheduler is through the interrupt of the SYSTICK timer. While scheduling the next process, some basic data about the state of execution of the previous process must be stored somewhere in memory, like the stack pointer and, generally, the value of all the registers as well as the address where the execution was interrupted.

Suppose that currently a process(let it be P1) is running and the SYSTICK triggers its corresponding interrupt. Then, as we said earlier, some data are been pushed immidiately to the stack automatically from the processor. Each process, of course, has its own memory area dedicated for the process stack, where those automatically pushed data will be stored. At the same time, for the process to continue its execution normally after the processor switches back to it, we ought to store the state of all the registers. These will be stored in a special memory place for each process, which we call PCB.

After having stored the vital data associated with the execution of the process, the main part of the scheduler is taking place, which is related with the selection of the next process(let it be P2). Many algorithms can be implemented, but the one used here is the next available process in the space of process. After the PCB of this newly selected process is stored to the corresponding registers, the stack pointer has taken the value of the stack space of the new process.

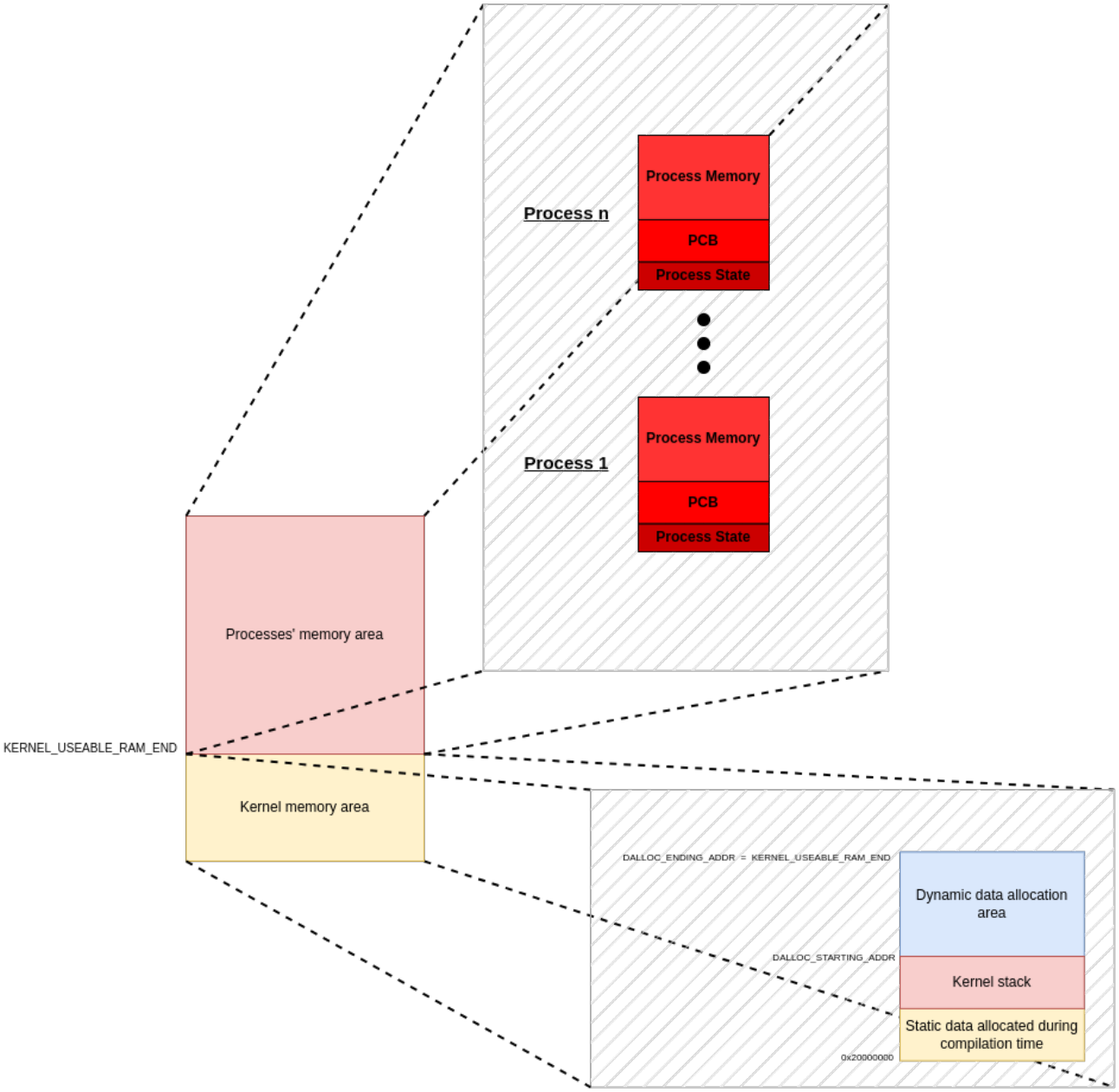
In the end, we return from the interrupt. This interrupt return causes the automatically pushed data during the start of the SYSTICK interrupt when P2 was executing to be popped, leading to the normal continuation of process P2.

Chapter:
Memory organization by the OS

In this chapter we are going to analyze the way the memory is manipulated by the Operating System not only from the perspective of the kernel, but also from the perspective of the process.

General memory organization picture

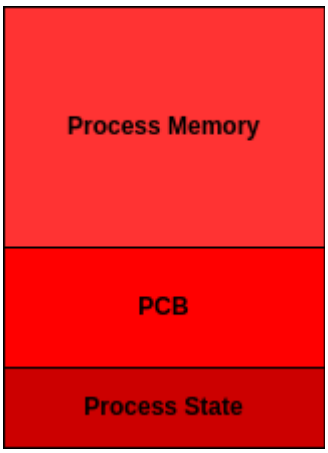
The following picture shows how the memory is treated by the Operating System



Those structures depicted in the figure are going to be clarified soon one-by-one.

How process-pages work

The total memory has a whole area dedicated for posible process that might get attached to the scheduler. This area is divided into fixed-size pages, called process-pages. Each process page has the following format:

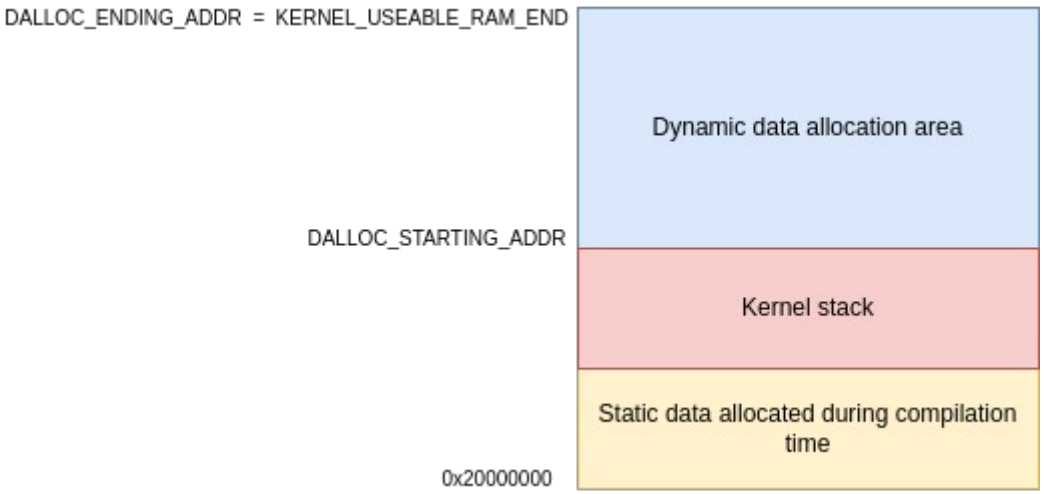


where

- **Process State**: Indicates whether this process-page is in use or not
- **PCB**: holds useful data like register values; needed for the context-switching
- **Process Memory**: the memory area handled by the process

How kernel-specific memory is organized

The term “kernel-specific memory” refers to the memory area that the kernel uses to implement its basic tasks e.g. hold a queue for the running processes or hold current running process usesful data. This area is depicted below, with the labels to the left having the same names as the macros used in source code for this purpose



How dynamic allocation works

In order to have the ability of dynamically allocating data and to avoid problems like fragmentation, we accompany the data to be dynamically allocated with meta-data, which store the size of the data-area to be allocated. In fact, the data that we store has the following picture:

In order for dynamic allocation to work, the dynamic data allocation area must be initialized to zeroes everywhere. Whenever malloc() (which is implemented from scratch) is called, it tries to find an empty slot where it can store the above frame. It is obvious that, when data is to be allocated, the first two bits are going to be non-zero because the size of the data can never be zero. That's why we are sure that in a memory area with only zeroes is likely to host the data. A final check must be made to assure that the zeroes are sufficient and the data fit-in there.