Operating Systems 2

Process Scheduling, Part 1

Arkadiusz Chrobot

Department of Information Systems

March 11, 2024

Outline

Process Scheduling

Scheduling in operating systems is making a decision about allocating a resource to a process. Usually, but not always when we talk about scheduling we mean *process scheduling* i.e. deciding on allocating the CPU to a process. The part of the kernel that decides which process should use the CPU as next is called a *process scheduler* or simply a *scheduler*.

Most of modern operating systems are a *multitasking* systems, which means that they interleave execution of processes. There are two types of such systems: *cooperative* and *preemptive*. In the former the process voluntarily decides when to stop using the allocated CPU. The main drawback of that approach is that a misbehaving process may block the entire system. The preemptive system may regain the CPU from the currently running process at any time. Linux, like other popular operating systems is a preemptive system.

Linux Scheduler

The main topic of this lecture is the *O(1) Scheduler* which was used in the kernel from the version 2.6.0 until 2.6.22. Its replacement will be discussed in the next lecture. Although the scheduler is no longer used it had many interesting elements that are worth to learn. The *O(1) Scheduler* is based on the original Unix scheduler, so the latter will be described first.

Original Unix Scheduler

The original Unix scheduler used the *Multiple Queues* scheduling. The processes in this schema are divided into two groups: regular processes with SCHED_OTHER scheduling policy and real-time processes with two scheduling polices SCHED_RR and SCHED_FIFO. The latter can be run only by privileged users. The SCHED_FIFO processes are not preemptive. The SCHED_RR processes are scheduled round-robin with long time intervals, which are called in Unix terminology *time slices*. The real-time processes have priorities ranging from 99 (highest) to 1 (lowest). The priority of each real-time process is static (time-invariant). Which means that when the SCHED_RR process consumes its time slice or when the SCHED_FIFO process *yields* they are given the same priority in the next scheduling round.

Original Unix CPU Scheduler

The SCHED_OTHER processes can be run by any user. They always got lower priority than the real-time processes and they undergo the round-robin scheduling. The priority of the process consists of two parts. The first one is a static priority. It is set by the user and called a *nice level*, which ranges from -20 (highest) to 19 (lowest). The second part is a dynamic component which is added to the static priority (or *base priority*) after each scheduling round and which can effectively decrease or increase the total priority of the process. This means that the processes can change the scheduling queue in the next scheduling round (see Fig. 1). The default nice level is 0. According to the POSIX standard with this level has to be associated a time slice equal or longer than $20ms$. Unix awards the I/O-*bound processes*, because those are usually the *interactive processes*. On the other hand it also tries to be fair for the CPU-*bound processes*.

Original Unix CPU Scheduler

Multi-Queue Process Scheduling

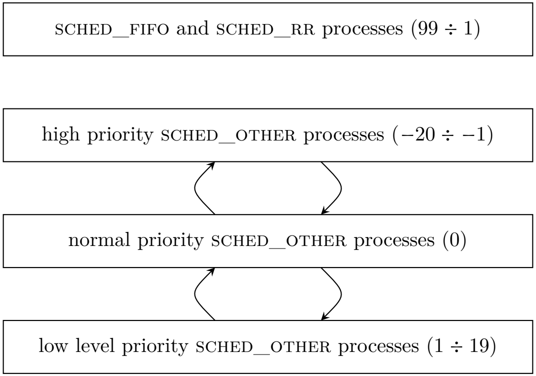
SCHED__FIFO and SCHED__RR processes (99 ÷ 1)
high priority SCHED__OTHER processes (−20 ÷ −1)
normal priority SCHED__OTHER processes (0)
low level priority SCHED__OTHER processes (1 ÷ 19)

Figure 1: Multi-Queue process scheduling scheme (for reference only)

O(1) Scheduler

The O(1) Scheduler is a modification of the original Unix scheduler authored by Hungarian programmer Ingo Molnár. Its features are as follows:

1. implements O(1) scheduling,
2. implements (almost) ideal SMP scaling,
3. implements thread-CPU binding in the SMP mode,
4. promotes interactive processes,
5. is optimized for the common case where more than one process is ready to run.

O(1) Scheduler

The O(1) Scheduler binds with each CPU in the system a single queue of processes that are waiting for allocation of the processor. The queue is a data structure of the type `struct runqueue` defined in the `kernel/sched.c` file, together with some useful macros. The access to the queues is synchronized with the use of spin-locks. Those variables are always locked in the same sequence to prevent deadlocks. The `runqueue` consists of two pointers to *priority arrays*, active and expired.

The priority array is a structure of the `struct prio_array` type. Inside this structure are two fields. The first one is an array which has 140 elements, which are pointers to circular doubly lists of processes (actually, elements that represents them for the scheduler). Priorities of the processes are recalculated inside kernel in such a way that the real-time processes get priorities ranging from 0 (highest) to 98 (lowest; 99 is unused) and the regular processes get priorities ranging from 100 (highest) and 139 (lowest). In that way the priorities can be used directly as indices in the arrays.

O(1) Scheduler

The other member of the `struct prio_array` structure is a bitmap (see Fig. 3) used for quickly locating a non-empty queue of highest priority in the array. The bit associated with the queue is set (its value is 1) and its position in the bitmap is the index of the array. As it was stated in the previous slide the `runqueue` has two pointers to such arrays. The first one points to the array of *active* priorities. It gathers processes that haven't yet used their time slices (the SCHED_RR and SCHED_OTHER processes) or are awaiting allocation of the CPU (SCHED_FIFO processes). The other pointer points to an array of *expired* priorities. A process that yields or consumes completely its time slice expires and goes to that array, awaiting there for the next scheduling round. Linux calculates a new priority for a SCHED_OTHER process as soon as it expires, unlike the original Unix, which calculates them after all processes have expired. After a while the array of active priorities becomes empty and all the processes are in the array of expired priorities. The kernel switches their roles efficiently by swapping their pointers (see Fig. 2).

O(1) Scheduler

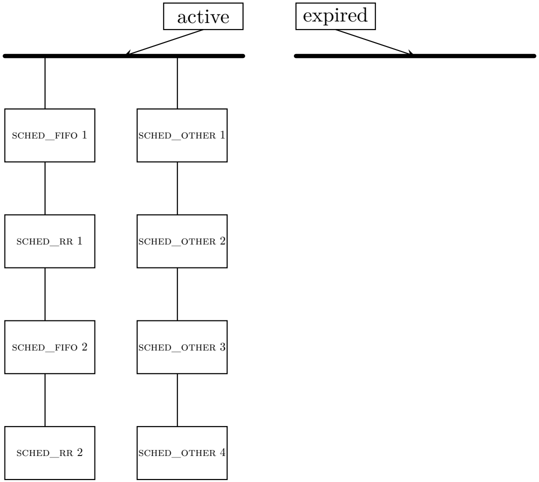

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

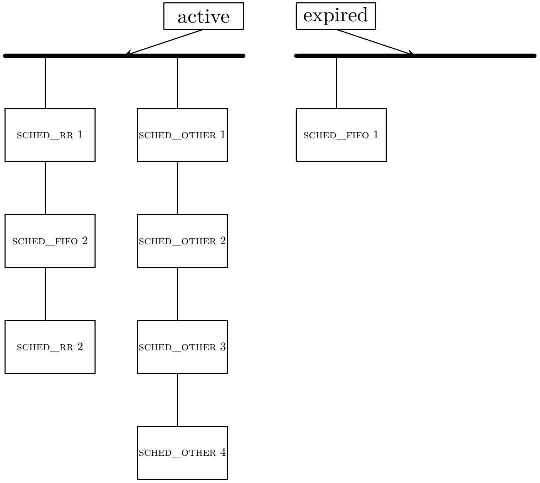

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

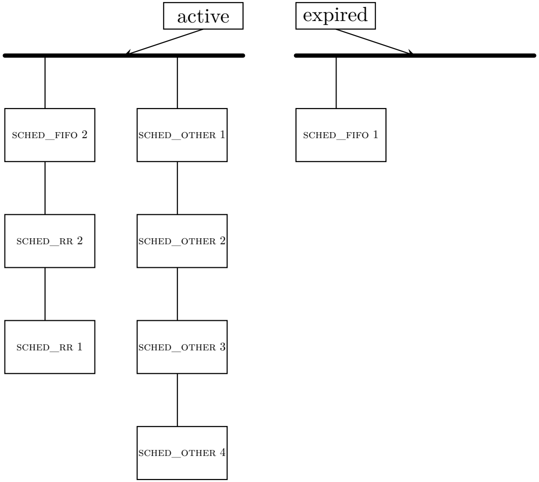

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

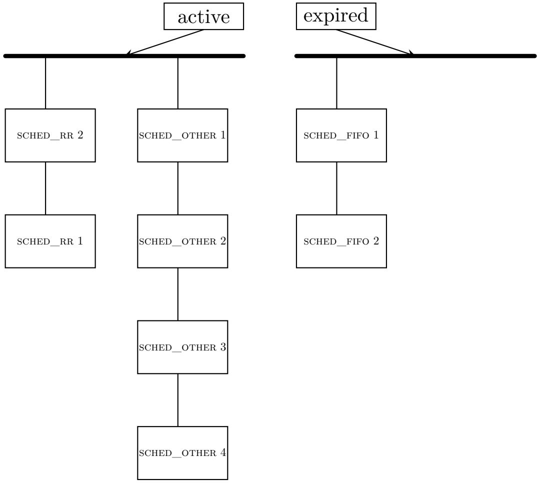

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

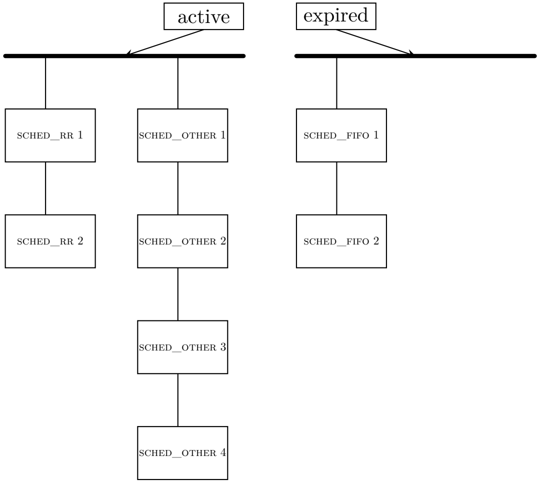

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

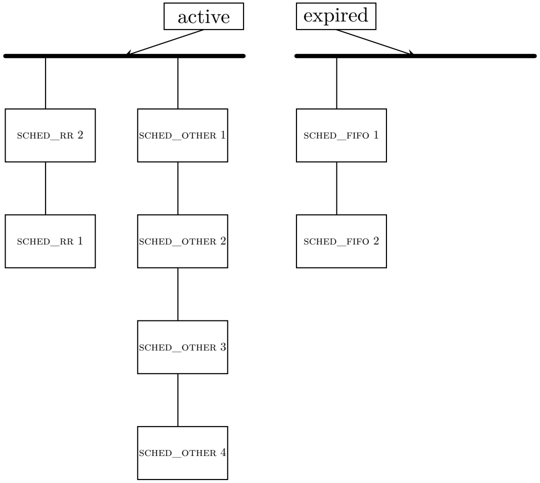

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

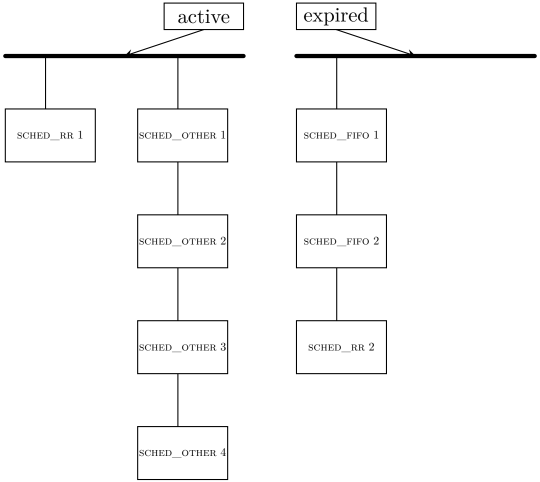

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

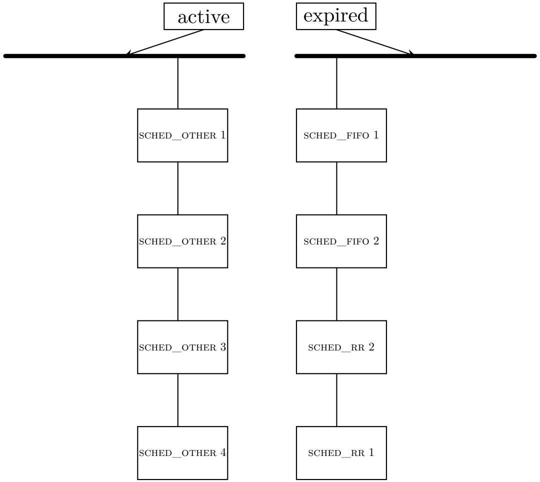

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

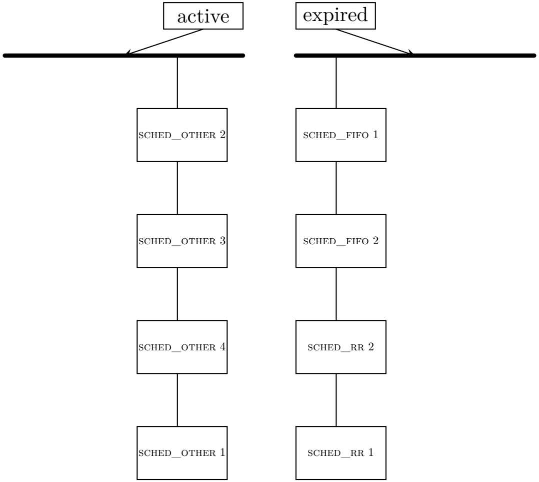

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

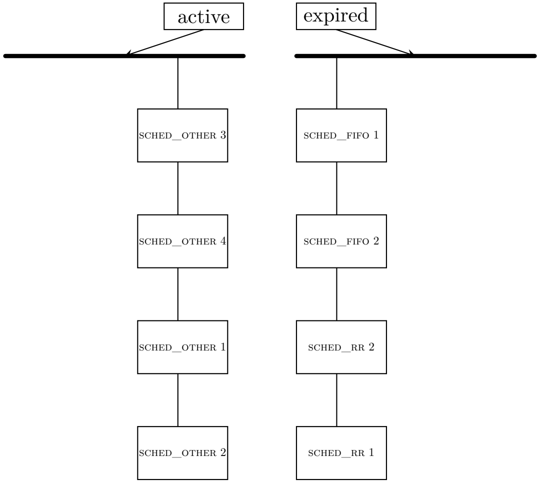

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

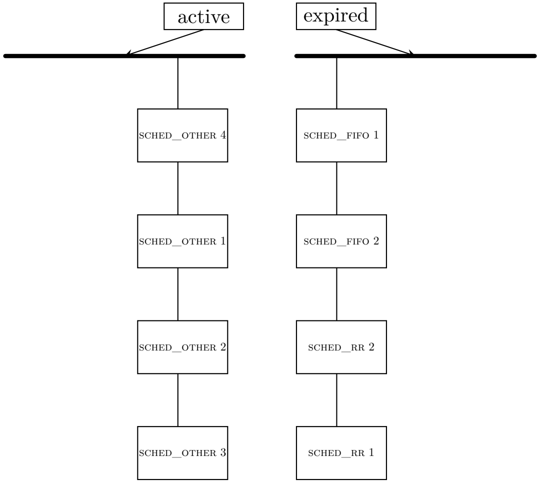

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

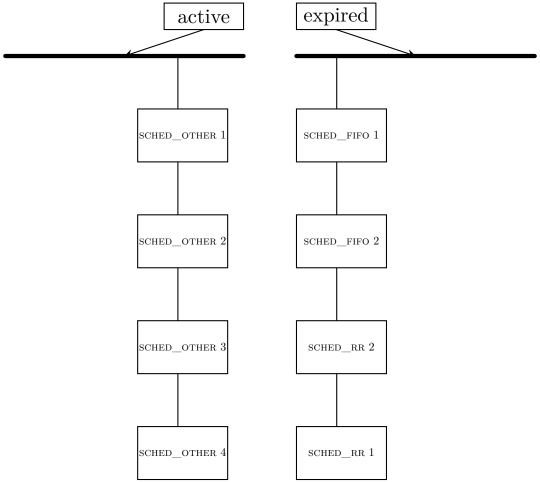

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

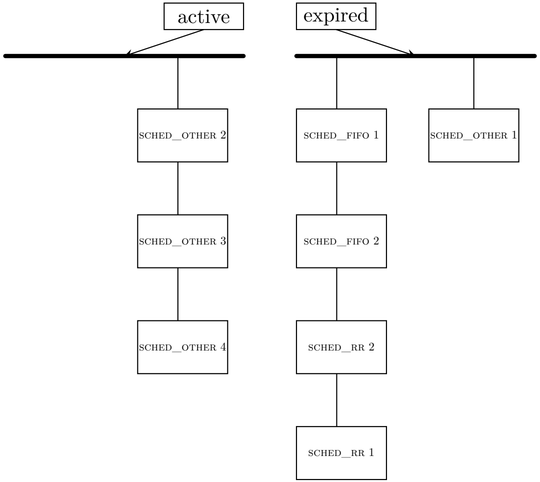

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

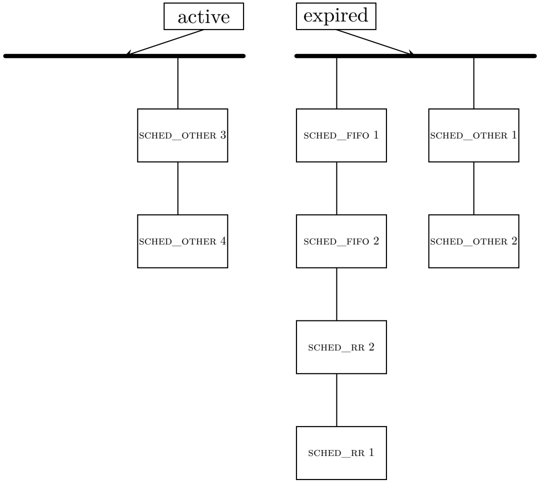

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

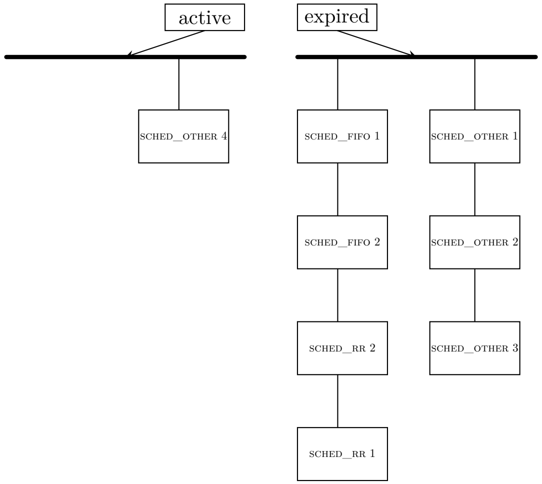

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

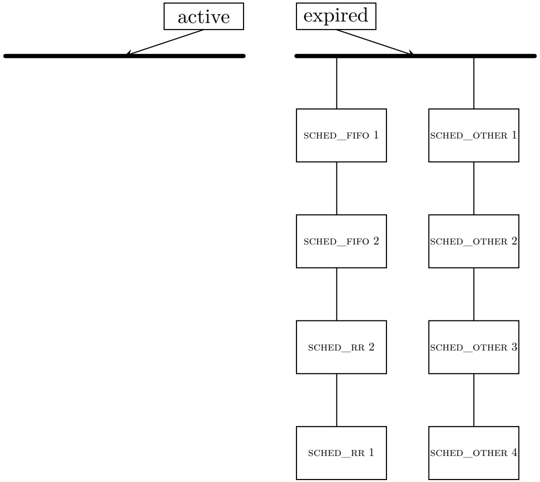

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

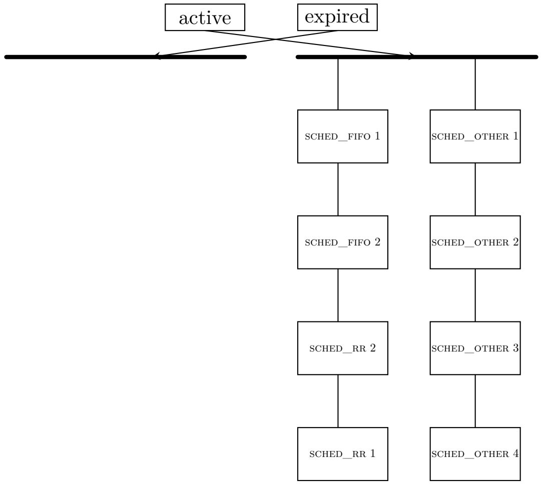

Figure 2: O(1) Scheduler priority arrays (for reference only)

O(1) Scheduler

0	1	2	3	4	5	6	7
0	1	1	0	0	0	1	1

Figure 3: Bitmap Example (for reference only)

O(1) Scheduler

Context Switching

The O(1) Scheduler algorithm is implemented in the `schedule()` function. Its main responsibility is only to choose the next process that should run from the ones that wait for allocation of the CPU, but it also calls the `context_switch()` function which switches the processes. The code of the `schedule()` function is simple, CPU architecture independent and effective. The time of choosing next process to run doesn't depend on the total number of ready to run processes. The code of the `context_switch()` function is on the other hand CPU architecture dependent. Its efficiency depends on the organization of the processor. Usually it takes longer to switch context in the RISC than in the CISC processors.

O(1) Scheduler

The SCHED_OTHER Processes Scheduling

The kernel can change the priority of a regular process basing on its interaction level in the previous round of scheduling. The priority of less interactive processes is decreased by $+5$ and the priority of more interactive processes is increased by -5. Linux uses some heuristics to evaluate the level of interaction of each of the processes. The evaluation is based on the ratio of the total length of the time when the process was active to the total length of the time that process spend waiting. The often the process waits the more interactive it is. The interactive processes get longer time slices. The default time slice (nice level 0) is $100ms$, the longest available time slice for the regular processes is $200ms$ (nice level -20) and the shortest is $10ms$ (nice level 19). If a process is highly interactive the kernel do not expires it after it consumes its time slice, but awards it with the same priority in the active priorities array. This however may cause starvation of the already expired processes. To prevent this the kernel runs periodically the `EXPIRED_STARVING()` macro.

O(1) Scheduler

The New Process Scheduling

A child process gets half of its parent remaining time slice. This amount is also granted to the parent. After both processes expire a new priority and hence new time slice is allocated for both of them.

O(1) Scheduler

Awaiting Processes

A process doesn't have to consume its time slice in one turn. It may for example invoke a system call and await for its result. In that case it is not eligible to the TASK__RUNNING state and cannot be in the `runqueue`. Hence, it is moved to an appropriate queue of waiting processes, of the `wait_queue_head_t` type (see the 5^{th} laboratory instruction for details) and the `schedule()` function is called which chooses another process for running. The sleeping process can be awaken with the use of, for example, the `wake_up()` function when the event it is waiting for occurs. Then it is moved to the `runqueue` with the remaining time slice it has. If this process if of higher priority than the current one then the `need_resched` flag is set.

O(1) Scheduler

Processes Preemption

The process is preempted when the `need_resched` flag is set. It's one of the bits in the `flags` member of the `thread_info` structure. As a consequence of setting this flag the `schedule()` function is called which chooses a next process and calls the `context_switch()` function which swaps the current and the next process by changing the virtual memory mapping (it invokes the `switch_mm()` function for this task) and setting the context of the CPU. The latter task is performed by the `switch_to()` function (called by the `context_switch()` function), which also preserves the kernel stack and the registers for the current process.
The user process can be preempted when the control returns to the user space, after a system call or an interrupt handling. A kernel thread may be preempted when the control returns from an interrupt handling, when its `preempt_count` counter (another member of the `task_info` structure) is set to zero, when it calls directly the `schedule()` function or when it starts waiting for some event.

O(1) Scheduler

Scheduling in Multiprocessor Systems

The O(1) Scheduler also handles the multiprocessor scheduling. In such systems some processes may be associated with one and only one CPU, but usually most of them run on any available processor. In that case it is sometimes necessary to balance the workload of the CPUs, which means that some of the processes can be moved from one `runqueue` to the other. This is accomplished by the `load_balance()` function, which is activated by the kernel when one of the `runqueues` is empty. It is also invoked periodically by the timer interrupt. In that case it moves processes when one of the `runqueues` is 25% longer than the others.

Scheduler User-Space API

There are user-space functions that allow the processes to impact the scheduling policy (some privileges for some of them are required):

- `nice()` — sets the nice level,
- `sched_setscheduler()` — sets the scheduling policy,
- `sched_setparam()` — sets parameters for scheduling policy,
- `sched_get_priority_max()` — returns the maximal priority for a given scheduling policy,
- `sched_get_priority_min()` — returns the minimal priority for a given scheduling policy,
- `sched_rr_get_interval()` — returns the time slice length of a SCHED_RR process,
- `sched_setaffinity()` — bounds a process or thread to specified CPUs,
- `sched_getaffinity()` — returns a bitmap that specifies on which CPUs the process is allowed to run,
- `sched_yield()` — makes the process to relinquish the CPU.

Scheduler User-Space API

If the O(1) Scheduler is used then calling the `sched_yield()` function is expensive for the process because, it has to wait until the next scheduling round to get the CPU. The process becomes immediately expired. This function in unavailable for the kernel threads. They should use the `yield()` function instead.

Questions

?

The End

Thank You for Your attention!