Mentoring **O**perating **S**ystem (**MentOS**)

Process management

Created by
Enrico Fraccaroli
enrico.fraccaroli@gmail.com

Table of Contents

Process descriptor

Process descriptor

The `task_struct` is a data structure used by the Kernel to represent a process and store information about it[1].

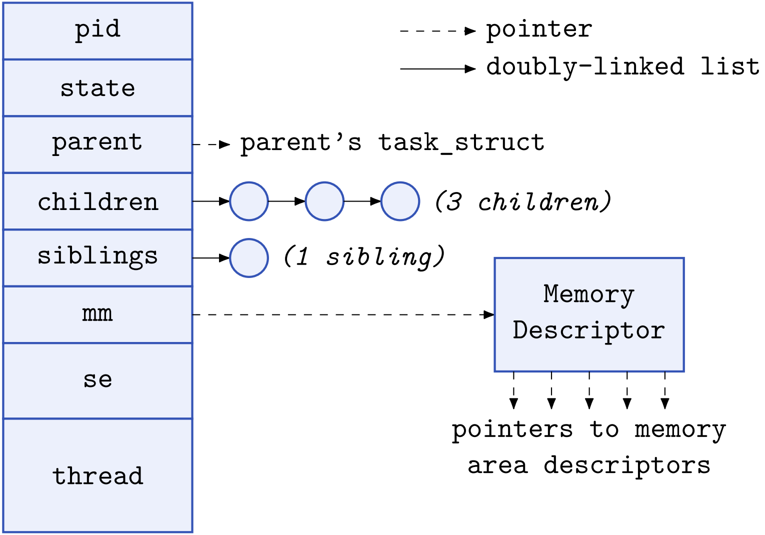
```
struct task_struct {
    pid_t pid;                      // the process identifier
    unsigned long state;            // the current process's state
    struct task_struct *parent;     // pointer to parent process
    struct list_head children;      // list of children process
    struct list_head siblings;      // list of siblings process
    struct mm_struct *mm;           // memory descriptor
    struct sched_entity se;         // time accounting (aka schedule entity)
    struct thread_struct thread;    // context of process
    struct list_head run_list ;     // pointer to the process into the scheduler
}
```

N.B.: The memory descriptor of a process is only reported here for completeness. It will be explained in detail in the Memory Management section.

---

[1] In Linux, it is quite big, 1.7KB on 32-bit machine (include/linux/sched.h)

task_struct memory representation



- - -► pointer
──► doubly-linked list

pid
state
parent - -► parent's task_struct
children ──► ○ ──► ○ ──► ○ *(3 children)*
siblings ──► ○ *(1 sibling)*
mm - - - - - - - - - - - - - - - -► Memory Descriptor
se
thread

pointers to memory area descriptors

Process descriptor

Process identifier

Process identifier

Process Identifier (PID) is numeric value identifying a process. When a new process is created a new PID is generated by summing 1 to the last assigned PID.

In Linux, the maximum value for a PID is 32768. When the PID maximum value is reached, the last assigned PID is reset to 0 before searching for a new PID.

The macro `RESERVED_PID` (usually set to 300) is defined to reserved PIDs to system processes and daemons, namely processes proving a service (*e.g.* a web server). All user's processes have PID greater than `RESERVED_PID`.

Process descriptor

State of a process

State of a process (1/3)

Process state is a numeric value describing the current state of the process. A process can be in one of the following state:

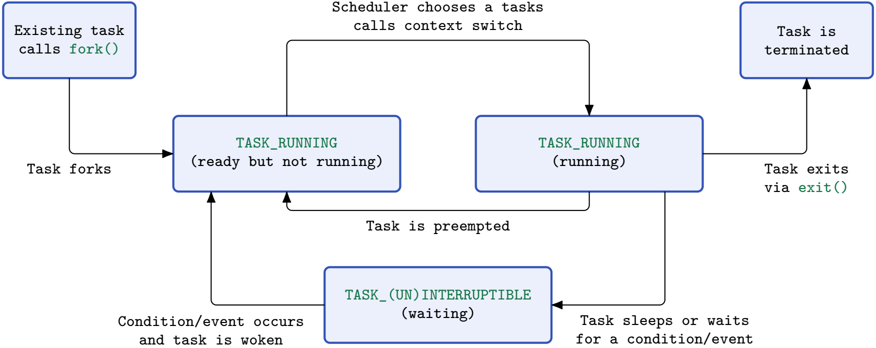
- `TASK_RUNNING`: either the process is currently in execution, or it has all the resources to be executed except the CPU.
- `TASK_INTERRUPTIBLE`: the process is blocked (sleep), waiting for some condition to run. When this condition exists, the kernel sets the process's state to `TASK_RUNNING`. The process also awakes and becomes runnable if it receives a signal (e.g., interrupt, signal, released resources).
- `TASK_UNINTERRUPTIBLE`: this state is identical to `TASK_INTERRUPTIBLE` but it does not depend on specific signal, it must wait without interruption for a specific weak-up call (e.g., task waiting for data transferred from block dev to buffer).

State of a process (2/3)

- `TASK_STOPPED`: process execution has stopped; the task is not running nor is it eligible to run.
- `EXIT_ZOMBIE`: Process execution is terminated, but the parent process has not yet issued a `wait4(0)` or `waitpid()` system call to return information about the dead process.
- `EXIT_DIED`: The final state: the process is being removed by the system because the parent process has just issued a `wait4()` or `waitpid()` system call for it.

Remember init (PID $= 1$) process.

State of a process (3/3)



Existing task calls `fork()`

Task forks

Scheduler chooses a tasks calls context switch

`TASK_RUNNING` (ready but not running)

`TASK_RUNNING` (running)

Task is terminated

Task exits via `exit()`

Task is preempted

`TASK_(UN)INTERRUPTIBLE` (waiting)

Condition/event occurs and task is woken

Task sleeps or waits for a condition/event

Figure: Flow chart of process states

Relationships among processes (1/2)

Processes created by a program have a parent/child relationship. When a process creates multiple children, these children have sibling relationships.

```c
struct task_struct {
    // ...
    pid_t pid;                      // the process identifier
    struct task_struct *parent;     // pointer to parent process
    struct list_head children;      // list of children process
    struct list_head siblings;      // list of siblings process
    // ...
}
```

Fields of `task_struct` describing the relations among processes:

- parent: pointer to the process's parent;
- children: The head of the list containing all children created by the process.
- **sibling**: The head of the list containing all children created by the process's parent.

Relationships among processes (2/2)

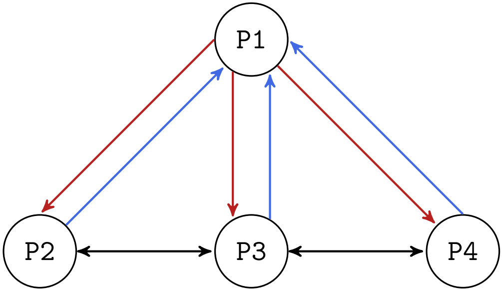


Figure: Parenthood relationships among four processes.

Red lines go from **parent** to **child**.
Blue lines go from **child** to **parent**.
Black lines show relations between **siblings**.

Process descriptor

Time accounting

Time accounting (1/3)

The field **se** of our `task_struct` is a structure called `sched_entity`, which holds all the information about scheduling activities.

```
struct task_struct {
    //..
    struct sched_entity se;      // time accounting (aka schedule entity)
    //..
}
```

It contains the **priority** and **execution times** of a process.

```
struct sched_entity {
    int      prio;             // priority
    time_t   start_runtime;    // start execution time
    time_t   exec_start;       // last context switch time
    time_t   sum_exec_runtime; // overall execution time
    time_t   vruntime;         // weighted execution time
}
```

Time accounting (2/3)

- **prio**
  Defines the execution priority of a process. It has a value in the range [100, 139], where 100 means the highest priority, and 139 means the lowest priority.
  By default, the priority of a new generated process is 120.
  A process can increment/decrement its `prio` value by using the system call *nice(inc)*, which takes as input parameter a value in the range [-20, 19].
  Examples:
  - nice(1) (increment `prio` value of calling process by 1 unit)
    120 ⇒ 121
  - nice(-5) (decrement `prio` value of calling process by -5 units)
    120 ⇒ 115

Time accounting (3/3)

- **start_runtime**
  The system execution time reporting when the process was first executed in the CPU.
- **exec_start**
  The system execution time reporting when the process was last executed in the CPU.
- **sum_exec_runtime**
  The overall execution time spent by the process in CPU.
- **vruntime**
  The virtual runtime, namely the weighted overall execution time spent by the process in CPU (see CFS).

Process descriptor

Context of a process

Context of a process

The field **thread** of our `task_struct` is a structure called `thread_struct`, which holds all the information about the execution of a process.

```
struct task_struct {
    //..
    struct thread_struct thread; // context of process
}
```

It is called the **context** of a process, and whenever a process is **not running**, it contains all the vital information required to **resume** it.

```
struct thread_struct {
    uint32_t ebp;         // base pointer register
    uint32_t esp;         // stack pointer register
    uint32_t ebx;         // base register
    uint32_t edx;         // data register
    uint32_t ecx;         // counter
    uint32_t eax;         // accumulator register
    uint32_t eip;         // Instruction Pointer Register
    uint32_t eflags;      // flag register
    bool_t fpu_enabled;   // is FPU enabled?
    savefpu fpu_register; // FPU context
}
```

Scheduler

Scheduler

Data structures

Scheduler data structures

The **runqueue** data structure is the most important data structure of the scheduler. It collects all system processes in running state.

```
struct runqueue {
    unsigned long nr_running; // number of processes in running state
    struct task_struct *curr; // pointer to current running process
    struct list_head_t queue; // list of processes in running state
}
```

Pay attention!

The `queue` field is the *head* of a circular, doubly-linked list collecting all system processes in running state. Consequently, a field **run_list** of type *struct list_head* is added in the *struct task_struct*.
(see slides fundamental concepts for more details).

Scheduler execution flow

The scheduler is called after the handle of an interrupt/exception. In detail, the following operations are performed by the scheduler:

1. updates the time accounting variables of the current process;
2. tries to wake up a waiting process. Whether a waiting condition is met, a process is woken by setting its state to running, and inserting it into the runqueue (topic not faced in current slides);
3. run scheduling algorithm to pick the next process to be executed by CPU from the runqueue;
4. performs context switch.

Scheduler

Scheduling algorithms

Scheduler selection (1/3)

- MentOS supports different types of scheduling algorithms, which are selected during compilation via **cmake**, and **called** by the `scheduler_pick_next_task` function;
- Furthermore, MentOS supports Real-Time scheduling, as such, the `runqueue` might contain both **periodic** and **aperiodic** tasks;
- This set of slides are focused on **aperiodic** tasks and scheduling algorithms (e.g., RR, Priority, CFS);

Scheduler selection (2/3)

`scheduler_pick_next_task` is a **centralized** function used by the scheduler to get the next process to execute, and **internally** this function calls the currently selected scheduling algorithm.
Based on the selected scheduling algorithm, the next process can be chosen differently. MentOS supports the following three **aperiodic** algorithms:

- **RR** - Round-Robin (`__scheduler_rr`);
- **Priority** - Highest Priority First (`__scheduler_priority`);
- **CFS** - Completely Fair Scheduler (`__scheduler_cfs`).

Pay attention!

In the following algorithms, we use the doubly-linked list defined in Linux Kernel, to collect all processes in running state.

Scheduler selection (3/3)

As shown in the following code, the `scheduler_pick_next_task` function, executes the scheduling algorithm based on the selected **cmake** option (e.g., `SCHEDULER_RR`, `SCHEDULER_CFS`, etc):

```c
task_struct *scheduler_pick_next_task(runqueue_t *runqueue) {
    ...

    // Create a pointer to the next task to schedule, and call the algorithm.
    task_struct *next = NULL;
#if defined(SCHEDULER_RR)
    next = __scheduler_rr(runqueue, false);
#elif defined(SCHEDULER_PRIORITY)
    next = __scheduler_priority(runqueue, false);
#elif defined(SCHEDULER_CFS)
    next = __scheduler_cfs(runqueue, false);
#elif defined(SCHEDULER_EDF)
    next = __scheduler_edf(runqueue);
#elif defined(SCHEDULER_RM)
    next = __scheduler_rm(runqueue);
#elif defined(SCHEDULER_AEDF)
    next = __scheduler_aedf(runqueue);
#else
#error "You should enable a scheduling algorithm!"
#endif

    ...
    return next; // Return the next process.
}
```

Developer notes

Students or **developers** should implement their version of these algorithms inside the provided functions by filling the **missing pieces**, i.e., those strange comments

```
    // Get its virtual runtime.
    time_t min = /* ... */;
```

If you wants to implement the Highest Priority scheduler, the way to go is to fill the **missing pieces** the appropriate function:

```
static inline task_struct *__scheduler_priority(runqueue_t *runqueue, bool_t skip_periodic) {
#ifdef SCHEDULER_PRIORITY
    // Get the first element of the list.
    task_struct *next = list_entry(runqueue->curr, task_struct, run_list);
    // Get its static priority.
    time_t min = /*...*/;

    ...

    return next;
#else
    return __scheduler_rr(runqueue, skip_periodic);
#endif
}
```

Select next process (Round-Robin) (1/4)

Round Robin is a CPU scheduling algorithm where a fixed time slice is assigned to each system process, in a cyclic way. It is simple, preemptive, easy to implement, and starvation-free.

Pseudocode of Round-Robin algorithm.

```
Require: Current process c, List of processes L
Ensure: Next process n
nextNode = next(c)
if IsTheHead(L, nextNode) then
    nextNode = next(nextNode)
end if
n = list_entry(nextNode)
```

Select next process (Round-Robin) (2/4)

Here is the current implementation of the Round-Robin algorithm:

```c
static inline task_struct *__scheduler_rr(runqueue_t *runqueue, bool_t skip_periodic)
{
    // If there is just one task, return it; no need to do anything.
    if (list_head_size(&runqueue->curr->run_list) <= 1) {
        return runqueue->curr;
    }
    // Search for the next task (we do not start from the head, so INSIDE, skip the head).
    list_for_each_decl(it, &runqueue->curr->run_list)
    {
        // Check if we reached the head of list_head, and skip it.
        if (it == &runqueue->queue)
            continue;
        // Get the current entry.
        task_struct *entry = list_entry(it, task_struct, run_list);
        // We consider only runnable processes
        if (entry->state != TASK_RUNNING)
            continue;
        // If entry is a periodic task, and we were asked to skip periodic tasks, skip it.
        if (__is_periodic_task(entry) && skip_periodic)
            continue;
        // We have our next entry.
        return entry;
    }
    return NULL;
}
```

Select next process (Round-Robin) (3/4)

The actual implementation in MentOS considers the presence of **periodic** processes, which are discussed in the **Real-Time Scheduler** slides. As such, it is slightly more complex than what is shown in the previous slide.

However, the idea stays the same, except we need to use a `for` loop to **search** for a viable next process. We need a `for` loop because the `next` process might be a **periodic** process, and we might want to **skip** it.

Pay attention!

The code already helps you by providing most of the code, and you just need to fill the **missing pieces**. So, the code **already contains** the parts required to skip periodic tasks. I'm talking about:

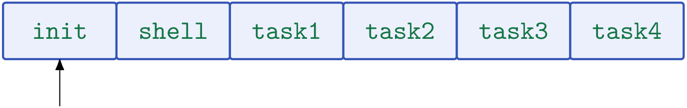
```
if (__is_periodic_task(entry) && skip_periodic)
    continue;
```

Example (Round-Robin) (1/7)

First iteration:

- `current_process = init`
- `__scheduler_rr()` returns `shell`

`runqueue:`



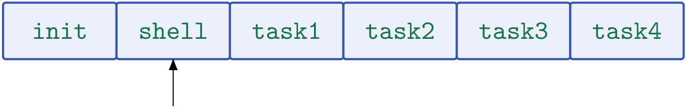
`init` `shell` `task1` `task2` `task3` `task4`

Example (Round-Robin) (2/7)

Second iteration:

- `current_process` = `shell`
- `__scheduler_rr()` returns `task1`

`runqueue:`

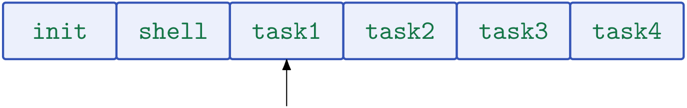


Example (Round-Robin) (3/7)

Third iteration:

- `current_process` = `task1`
- `__scheduler_rr()` returns `task2`

`runqueue:`

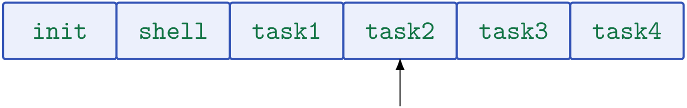


Example (Round-Robin) (4/7)

Fourth iteration:

- `current_process` = `task2`
- `__scheduler_rr()` returns `task3`

`runqueue:`

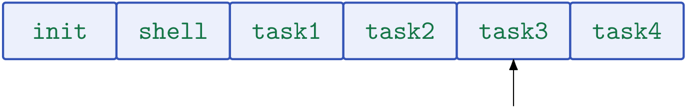


Example (Round-Robin) (5/7)

Fifth iteration:

- `current_process` = `task3`
- `__scheduler_rr()` returns `task4`

`runqueue:`

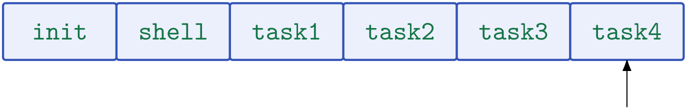


Example (Round-Robin) (6/7)

Sixth iteration:

- `current_process` = `task4`
- `__scheduler_rr()` returns `init`

`runqueue:`

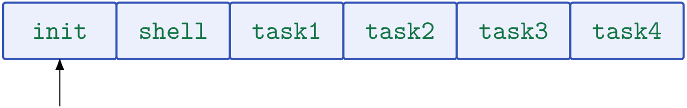


Example (Round-Robin) (7/7)

Seventh iteration:

- `current_process` = `init`
- `__scheduler_rr()` returns `shell`

`runqueue:`



Select next process (Highest Priority First) (1/3)

Round robin scheduling assumes that all processes are equally important. This generally is untrue. We would sometimes like to see long CPU-intensive (non-interactive) processes get a lower priority than interactive processes.

In addition, different users may have different status. A system administrator's processes may rank above those of a student's.

These goals led to the introduction of the *Priority* scheduling algorithm.

Select next process (Highest Priority First) (2/3)

Each process has a static priority. Smaller is the number, higher is the priority of the process.

The scheduler simply picks the highest priority process to run. A process is **preempted** whenever a higher priority process is available in the run queue.

Advantage: priority scheduling provides a good mechanism where the relative importance of each process may be precisely defined.
Disadvantage: If high priority processes use up a lot of CPU time, lower priority processes may starve and be postponed indefinitely, leading to **starvation**.

Select next process (Highest Priority First) (3/3)

Pseudocode of Highest Priority First.

```
Require: Current process c, List of processes L
Ensure: Next process n
n = c
for all listNode $\in$ L do
    if !IsTheHead(L, listNode) then
        t = list_entry(listNode)
        if priority(t) $<$ priority(n) then
            n = t
        end if
    end if
end for
return n
```

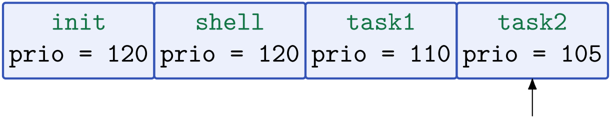
The implementation of this algorithm is given to the student.

Example (Highest Priority First) (1/3)

First block of iteration:

- `current_process` = `task2`
- `__scheduler_priority()` returns `task2` until no process with an higher priority is present in the system.

`runqueue:`



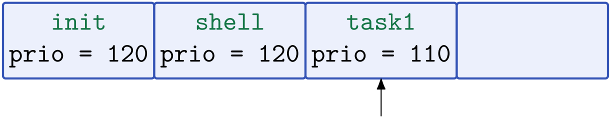
init	shell	task1	task2
prio = 120	prio = 120	prio = 110	prio = 105

Example (Highest Priority First) (2/3)

Second block iteration:

- `current_process` = `task1`
- `__scheduler_priority()` returns `task1` until no process with an higher priority is present in the system.

`runqueue:`

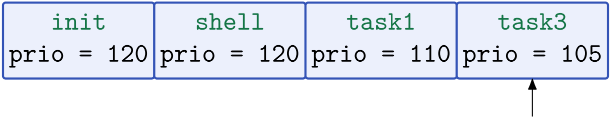


Example (Highest Priority First) (3/3)

Third block of iteration:

- `current_process` = `task3`
- `__scheduler_priority()` returns `task3` until no process with an higher priority is present in the system.

`runqueue:`



How much time do init and shell have to wait to get the CPU?

Select next process (Completely Fair Scheduler) (1/6)

Completely Fair Scheduler (CFS) aims to prevent starvation by assigning the CPU fairly to all system processes.

Let consider an example to illustrate the goal of CFS. If there are two tasks A and B, which have a same “weight", the portion of available CPU time given to each task is 50%.

However, if the “weight" of task A increases on CPU by 10%, then task A's portion of the CPU is 55%, meanwhile task B's portion of the CPU becomes 45%.

Select next process (Completely Fair Scheduler) (2/6)

CFS's idea: let use the priority of each process to “weight" its overall execution time (virtual runtime).

Processes having a low priority have a virtual runtime increasing faster than processes with a higher priority. Scheduler always picks the process with the lowest virtual execution time!

Select next process (Completely Fair Scheduler) (3/6)

Scheduler needs to know the weight of the task to estimate its CPU time's portion. Hence, the priority number has to be mapped to such a weight; this is done in the array `prio_to_weight`:

```
static const int prio_to_weight[] = {
    /* 100 */     88761, 71755, 56483, 46273, 36291,
    /* 105 */     29154, 23254, 18705, 14949, 11916,
    /* 110 */      9548,  7620,  6100,  4904,  3906,
    /* 115 */      3121,  2501,  1991,  1586,  1277,
    /* 120 */      1024,   820,   655,   526,   423,
    /* 125 */       335,   272,   215,   172,   137,
    /* 130 */       110,    87,    70,    56,    45,
    /* 135 */        36,    29,    23,    18,    15
};
```

Select next process (Completely Fair Scheduler) (4/6)

A priority number of 120, which is the priority of a normal task, is mapped to a weight of 1024.

Note that the ratio of two successive entries in the array is almost 1.25. This number is chosen such that:

- if the priority of a task is reduced by one, then it gets 10% higher share of the available CPU time.
- if the priority of a task is increased by one, then it gets 10% lower share of the available CPU time.

Select next process (Completely Fair Scheduler) (5/6)

Given the array `prio_to_weight` we can update the virtual runtime of a process p, namely its weighted overall execution by using the formula:

```
vruntime += delta_exec * (NICE_0_LOAD / weight(p))
```

where:

- `vruntime` is the virtual run time of the process;
- `delta_exec` is the last amount of time spent by p in the CPU;
- `NICE_0_LOAD` is the weight of a task with normal priority (1024);
- `weight(p)` is the weight of p defined by the array prio_to_weight.

Select next process (Completely Fair Scheduler) (6/6)

Pseudocode of Completely Fair Scheduler.

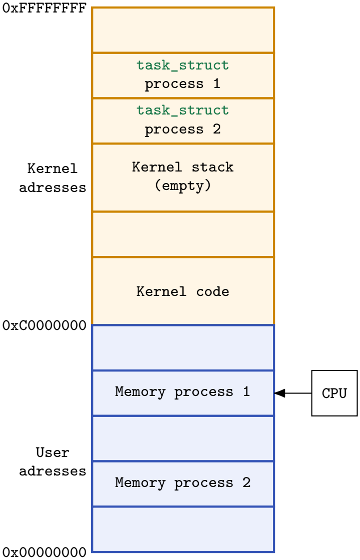
```
Require: Current process c, List of processes L
Ensure: Next process n
updateVirtualRuntime(c)
n = c
for all listNode ∈ L do
    if !IsTheHead(L, listNode) then
        task = list_entry(listNode)
        if virtualRuntime(task) < virtualRuntime(n) then
            n = task
        end if
    end if
end for
return n
```

The implementation of this algorithm is given to the students.

Scheduler

Context switch

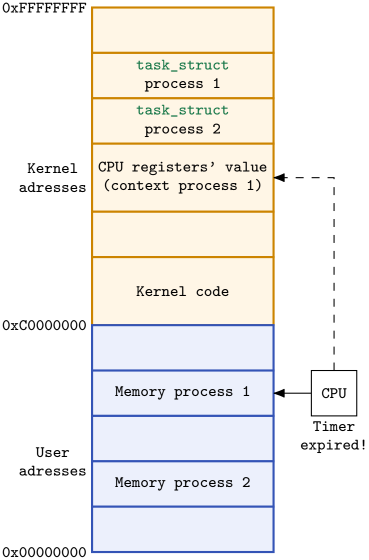
Context switch



The CPU performs a context switch to change the process executed by CPU.

The following example shows the steps performed by the operating system to save the current process's state *(process 1)*, and then resume the execution of a previously stopped process *(process 2)*.

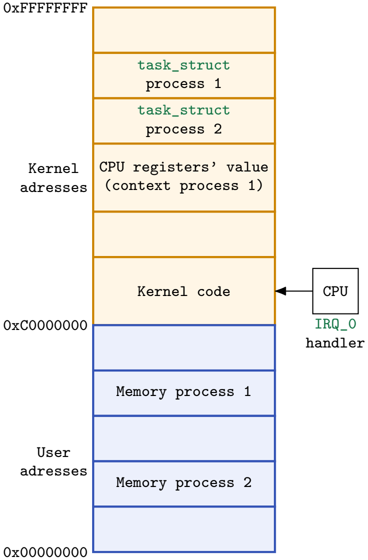
Context switch



(1) Time expired! It is time to give back control of CPU to kernel. Timer device rises the signal *INTR* and present 0 in *irq* line.

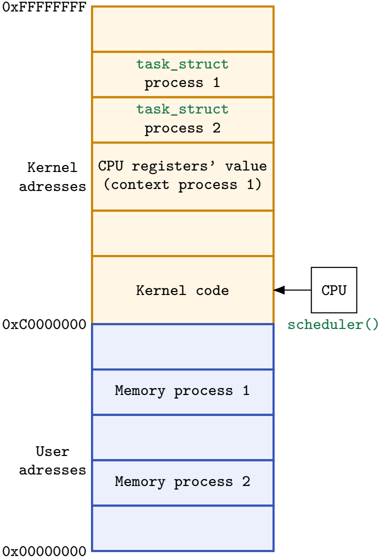
When *INTR* is risen, the CPU moves from Ring 3 (user mode) to Ring 0 (kernel mode). After the CPU privilege level change, the values of CPU registers are pushed in the Kernel's stack.

Context switch



(2) CPU starts executing *irq_0* interrupt handler to handle the hardware interrupt 0 risen by Timer.

Context switch



0xFFFFFFFF

task_struct
process 1

task_struct
process 2

Kernel
adresses

CPU registers' value
(context process 1)

Kernel code

CPU

scheduler()

0xC0000000

Memory process 1

User
adresses

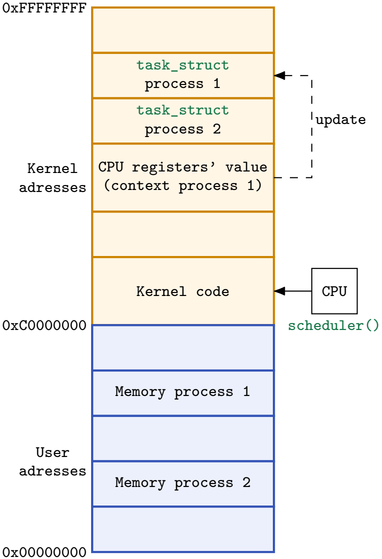
Memory process 2

0x00000000

(3) The scheduler is then called to update the time accounting variables of the interrupted process, and pick the next process to run.

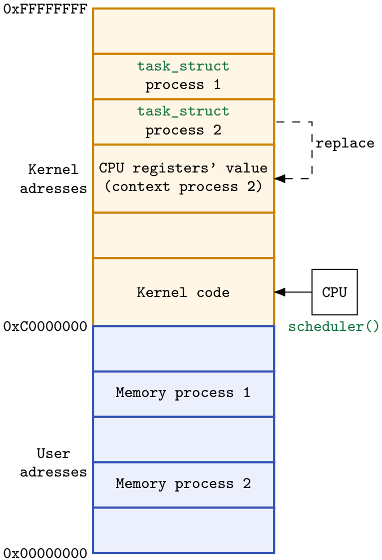
In this example, the scheduler picks the process 2 as the next one.

Context switch



(4) Kernel updates the thread_struct structure of the task_struct of the process 1 in order to save its context.

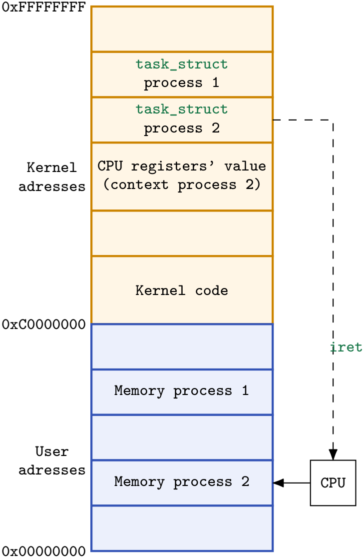
Context switch



(4) Kernel updates the thread_struct structure of the task_struct of the process 1 in order to save its context.

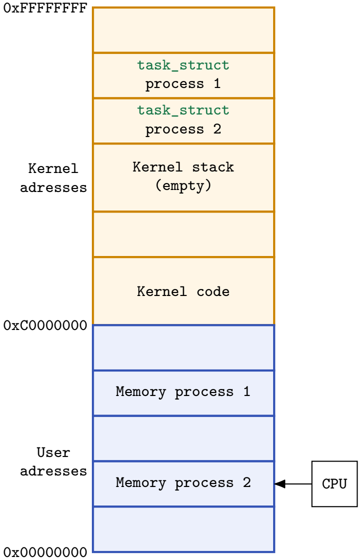
(5) Kernel replaces the context of process 1 with the context of process 2 in its stack memory.

Context switch



(6) Kernel moves the values from its stack to CPU's registers and runs an *iret* assembly instruction, which changes the CPU privilege level from Ring 0 (kernel mode) to Ring 3 (user mode).

Context switch



(7) The context of the process 2 is in the CPU's registers finally. The CPU can keep on executing the code of the process 2 in user mode until the next context switch.