Chapter 6

Real-time operating systems: Implementation

Overview of the main difficulties

- The implementation of the OS should remain generic, even though some operations (e.g., context switching) are highly architecture-dependent.
- The following operations need to be efficient:
  - Identifying the process with the highest priority in a list, in order to make it active, or to unblock it following an operation on a synchronization or communication object.

    (Ideally, this should have a maximum execution time that is independent from the number of tasks managed by the operating system.)
  - Unblocking processes suspended for a given delay.
  - Performing a context switch.
- There must exist a mechanism for invoking the scheduler immediately upon exiting interrupt routines, if preemption is needed.
- For some applications, the real-time operating system has to share the processor with another operating system.

Keeping the OS implementation generic

Note: For the illustrations in this chapter, we consider FreeRTOS, compiled with GCC on a ARM Cortex M4F architecture (e.g., STM32F4 MCUs).

Principles:

- The main part of the RTOS code is architecture and compiler-agnostic.
- A small subset of platform-dependent functions is provided in specific ports.

Illustration:

```
FreeRTOS-Kernel/list.c
                queue.c
                tasks.c
                timers.c
                portable/GCC/ARM_CM4F/port.c
                                      portmacro.h
                include/
```

Task control blocks

A Task Control Block (TCB) is a data structure that represents a process inside the kernel memory. This structure contains:

- The current priority of the task.

  Note: When the task priorities are fixed and unique, they can also be used as process identifiers.

- The context of the task, i.e., the state of the processor saved the last time that the task became non-running.

  This data consists in either

  - values for all the processor registers, or
  - only the stack pointer(s), with the other registers saved on the process stack.

- Pointers linking the TCB to the global data structures of the kernel (list of ready processes, lists of processes blocked on synchronization or communication objects, list of delayed processes, . . .).

- The current state of the process (ready, active, blocked, or interrupted), if it cannot be determined from other data.
- Data needed for managing priority inversion, if implemented.
- Values of thread-local variables (e.g., `errno`).
- Auxiliary data: statistics, debugging aids (e.g., process name), safety features (e.g., for stack overflow and underflow detection), . . .

Global data structures of the kernel

The global information managed by the kernel essentially contains:

- Sets of task control blocks corresponding to
  - the ready (or active) processes,
  - the processes suspended for a given delay.

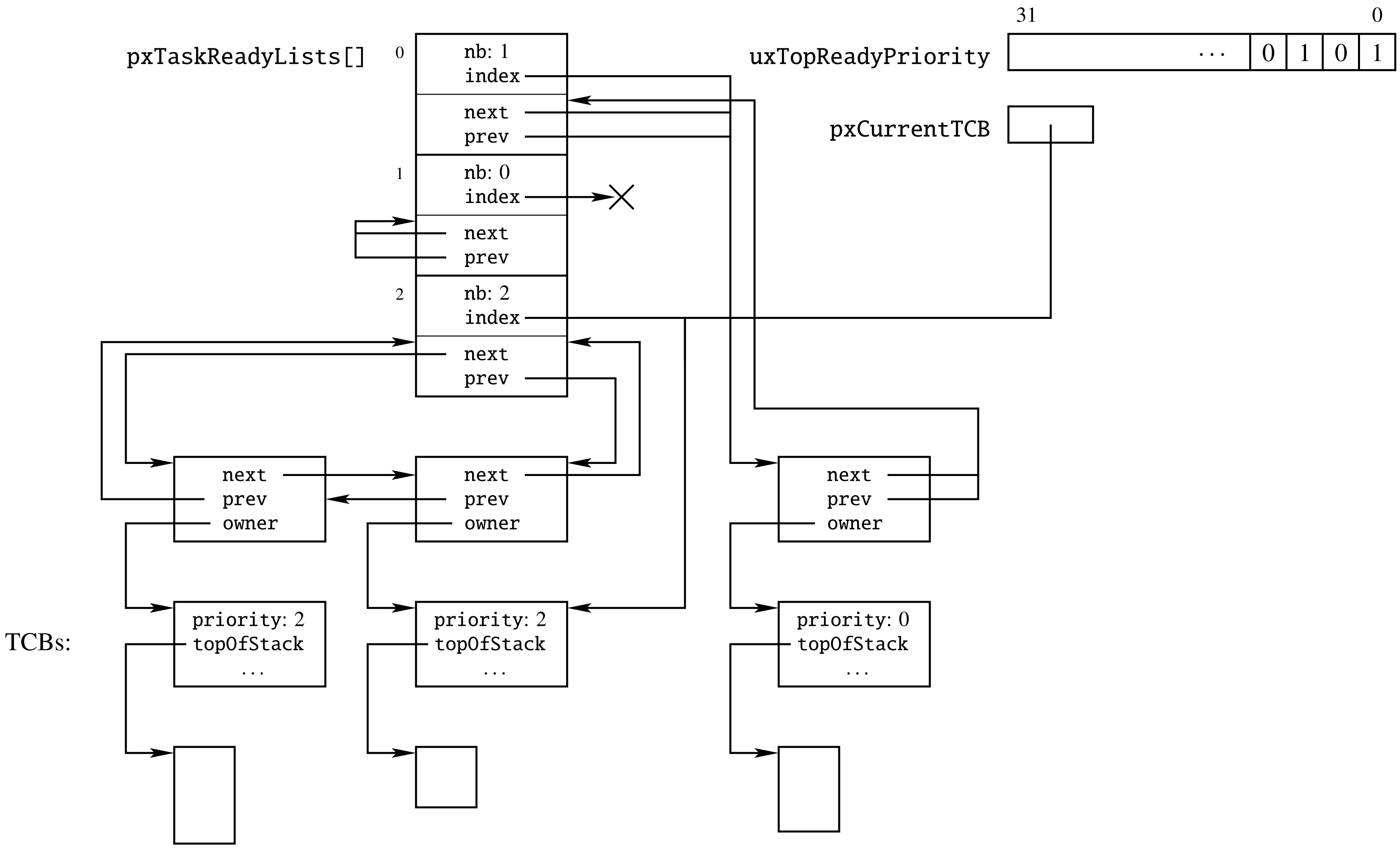
  Those sets are organized as doubly-linked lists, in order to be able to manipulate them in constant time.
- Optionally, a structure for identifying efficiently the ready process with the highest priority.
- An index for accessing directly a task control block from its corresponding process identifier (if those identifiers do not take the form of pointers to TCBs).
- Data structures representing the state of synchronization and communication objects.

Example (FreeRTOS):

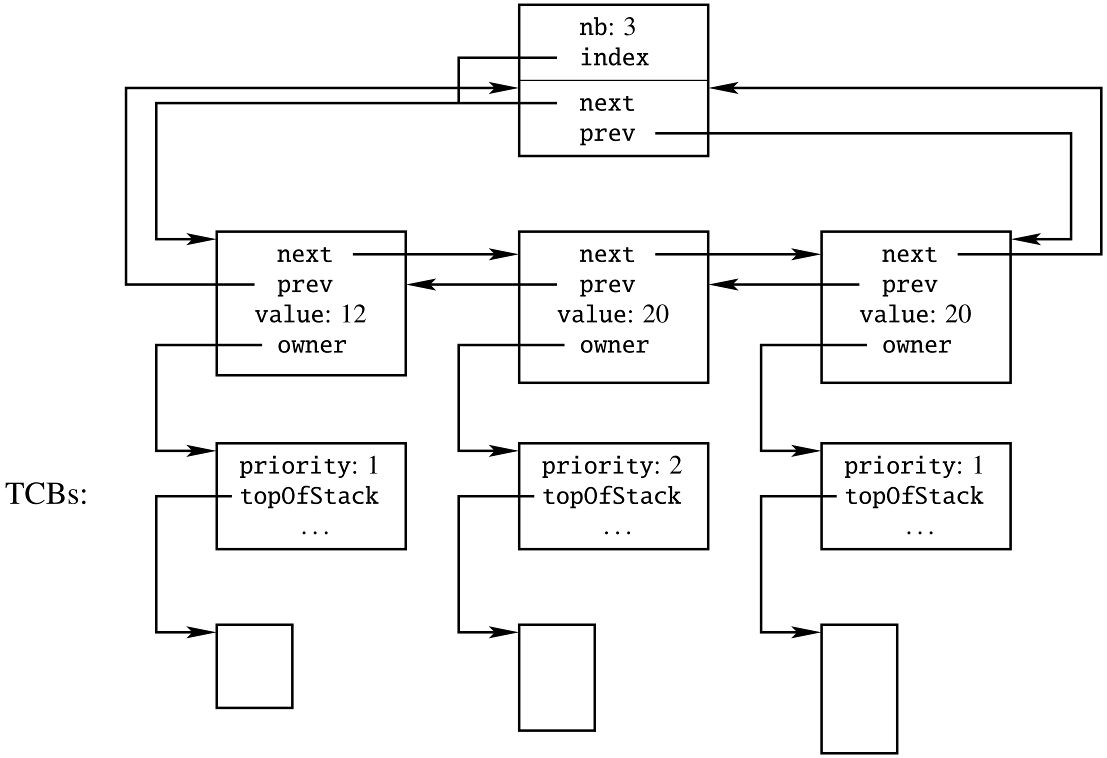
- The maximum number of process priorities is a compile-time configuration parameter of the kernel (`configMAX_PRIORITIES`).
- The set of ready processes is represented by
  - an array `pxReadyTaskLists[]` of circular doubly linked lists of pointers to TCBs, indexed by process priorities.

    Each such list contains a cursor used for managing time slicing, since FreeRTOS allows several processes to share the same priority.
  - an optional bitfield `uxTopReadyPriority` for speeding up the search for the non-empty list of processes with the highest priority.
  - A pointer `pxCurrentTCB` to the TCB of the currently active task.
- The set of processes suspended for some delay is represented by a list `pxDelayedTaskList` of pointers to TCBs, in which the processes are sorted by increasing deadline, then by decreasing priority.

Illustration



`pxDelayedTaskList`



nb: 3
`index`
`next`
`prev`

`next`
`prev`
`value`: 12
`owner`

`next`
`prev`
`value`: 20
`owner`

`next`
`prev`
`value`: 20
`owner`

TCBs:

`priority`: 1
`topOfStack`
…

`priority`: 2
`topOfStack`
…

`priority`: 1
`topOfStack`
…

Note: Identifying quickly the ready process with the highest priority is achieved by exploiting specific processor instructions (e.g., *Count Leading Zeros*, CLZ).

The scheduler

The scheduler is implemented by a kernel function called after each operation that can potentially influence the state of processes:

- Creating or destroying a task, or modifying a task priority.
- Performing an operation on a synchronization or communication object.
- Suspending a task for a given delay.
- Servicing the tick interrupt.

This function must be kept simple and efficient, and only performs the following operations:

1. Checking whether the scheduler is allowed to preempt tasks.
2. Identifying the ready task with the highest priority.
3. Performing a context switch in order to assign the processor to this task.

Note: The possibility of enabling or disabling preemption is offered because

- preempting the current task should be prevented inside interrupt routines (cf. Chapter 5);
- it provides a simple mechanism for manipulating atomically shared variables or communication objects (*preemption locks*). However, this mechanism
  - increases the latency of the tasks (by the duration of the longest interval during which preemption is disabled), and
  - affects all the tasks of the system (not only those that need to be coordinated).

Context switching

The main issue for implementing context switching is to be able to save and restore all processor registers.

Some processors automatically perform (totally or in part) this operation during interrupts:

- When an interrupt routine is called: The current value of the registers (or some of them) is saved on the runtime stack of the interrupted task.
- When an interrupt routine returns: The values extracted from the current stack are loaded into the processor registers.

A simple solution (when allowed by the processor architecture) consists of implementing the scheduler inside an interrupt routine.

If this routine has the lowest possible interrupt priority, then this has the advantage of preventing context switches from occurring in (regular) interrupt routines.

Illustration (FreeRTOS, ARM Cortex M4F)

Principles:

- The scheduler is implemented in the PendSV interrupt routine, which has the lowest priority.
- A call for preemption thus amounts to raising the corresponding interrupt event flag.
- The processor automatically pushes some processor registers (`r0`-`r3`, `r12`, `r14`, . . .) on the stack upon servicing an interrupt.

PendSV interrupt routine:

1. Push remaining registers (`r4`-`r11`, FPU registers, . . .) on the current stack.
2. Store the current stack pointer into `pxCurrentTCB -> pxTopOfStack`.
3. Temporary raise the interrupt enable level in order to prevent reentrancy.
4. Call the function `vxTaskSwitchContext()`, that updates `pxCurrentTCB` with a pointer to the TCB of the task that must become active.

5. Reset the interrupt enable level to its lowest value.
6. Set the current stack pointer to `pxCurrentTCB -> pxTopOfStack`.
7. Pop the register values (`r4-r11`, FPU registers, . . .) from the current stack.
8. Execute a return from interrupt instruction.

Task creation and deletion

Creating or deleting a process essentially amounts to updating the data structures managed by the kernel.

Task creation:

1. Alocate (statically or dynamically) a new TCB, and fill it with user-supplied parameters (priority, process name, . . .).
2. Allocate a stack for the new process.
3. Create an initial context, in which
   - the program counter is equal to the entry point of the new task,
   - the stack pointer points to the appropriate location, and
   - the other registers are either left uninitalized, or set to 0.
4. Insert the new TCB into the list of ready tasks.
5. Call the scheduler.

Note: A parameter can generally be passed to a newly created task, in order to make it possible for several tasks sharing the same code to behave differently.

Task deletion:

1. Remove the TCB of the task from its containing list (ready processes, processes suspended during a given delay, or waiting list of a synchronization or communication object).
2. If needed, unallocate the process stack and the TCB itself.
3. Call the scheduler.

Idle task(s)

Some operating systems systematically create one or many internal tasks, with a lower priority than all other processes.

There are many advantages to this approach:

- The scheduler can be more efficient, since it does not have to check whether there exists at least one ready task.
- Such tasks make it possible to measure the processor utilization, e.g., by repeatedly incrementing a counter that is queried at periodic intervals.
- An idle task can put the processor and some peripherals in a mode that minimizes energy consumption.

Example (FreeRTOS): One idle task that either

- runs an infinite loop, or
- puts the processor in deep sleep mode, and suppresses unneeded tick interrupts (tickless idle mode).

Time management

Quantitative time management is performed by the tick interrupt routine:

1. Increment a counter aimed at measuring elapsed time.
2. Compute the set of blocked tasks that must become ready again.

   Note: This includes tasks blocked on a timeout.
3. Update the state of tasks involved in time slicing.
4. Call the scheduler.

Illustration (FreeRTOS, ARM Cortex M4F):

- The priority of the tick interrupt is immediately above the one of PendSV.
- The tick interrupt routine processes the TCBs at the head of the list of delayed tasks.

  The processing time is linear in the number of tasks to be unblocked.

- Time slicing is implemented by periodically rotating the list of ready tasks at the highest priority.
- This routine also signals a dedicated task (*Timer/Daemon* task), the purpose of which is to manage software timers.

Communication and synchronization objects

In the kernel memory, a communication or synchronization object is represented by a structure containing:

- The type of the object (semaphore, message queue, . . .).
- Data representing the state of the object (e.g., for a semaphore: an integer counter).
- One or several lists of blocked tasks, waiting on specific operations.

  In the case of a priority-based selection policy, such a list can be represented by a doubly-linked list of TCBs, sorted in decreasing priority order.

If necessary, the kernel also maintains a table of allocated objects.

Finally, the TCB of each task waiting for an object contains a pointer to the structure managing this object.

Note: Implementing kernel services that update the state of objects does not require specific instructions such as *test and set*, since it is sufficient to disable interrupts during non-atomic operations.

Combining a real-time and a non-real-time operating systems

It is possible to combine in a single application a real-time operating system (*RTOS*) with another operating system (*host OS*). There are two possibilities:

- The operations of the host operating system are suspended when the real-time OS is started, and get the processor back when the RTOS terminates (e.g., μCOS-III).
- The host operating system is seen as special task that has a lower priority than all the tasks managed by the real-time OS (e.g., RTAI).

  For implementing this approach, it is necessary to ensure that the host OS can never disable interrupts managed by the kernel or the real-time tasks.