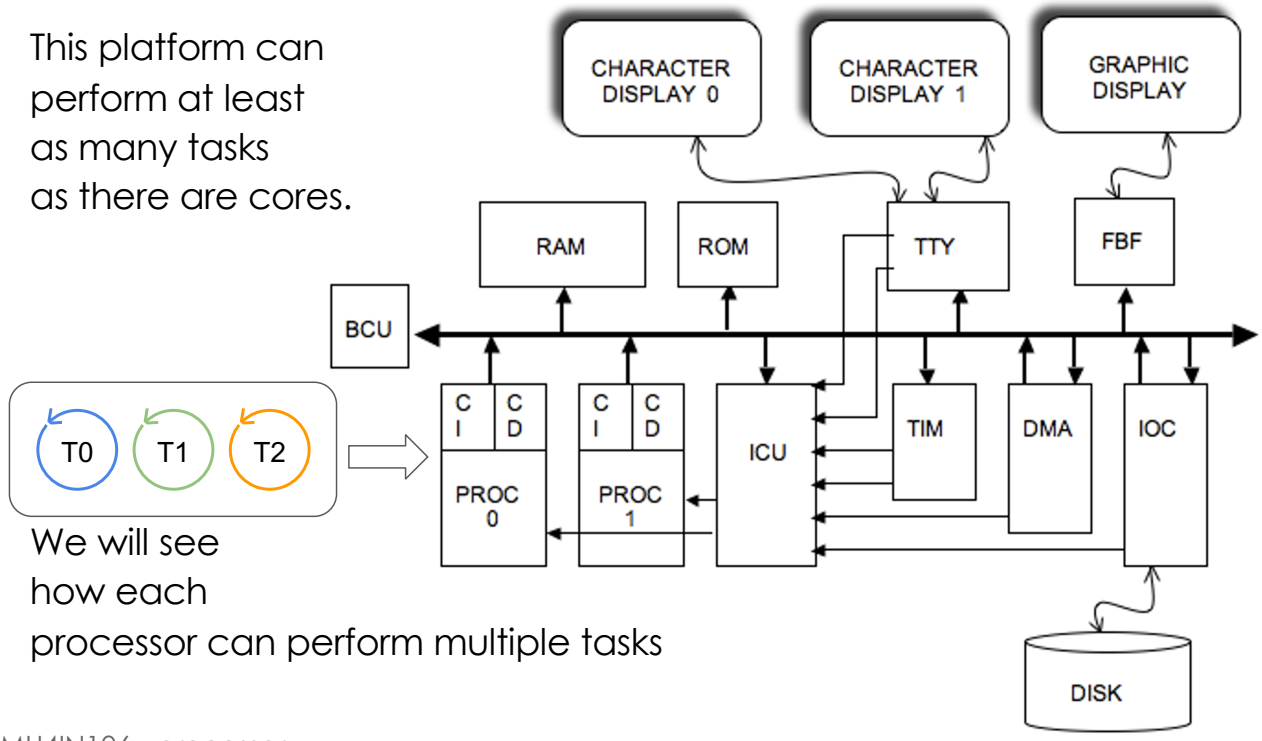
processor sharing

MU4IN106 Multi

Multi-core platform

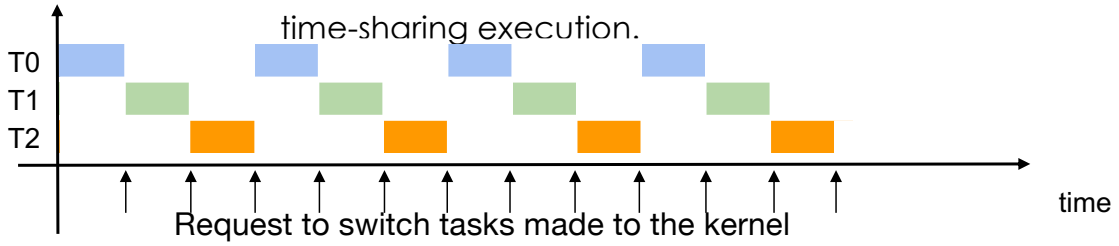
This platform can perform at least as many tasks as there are cores.



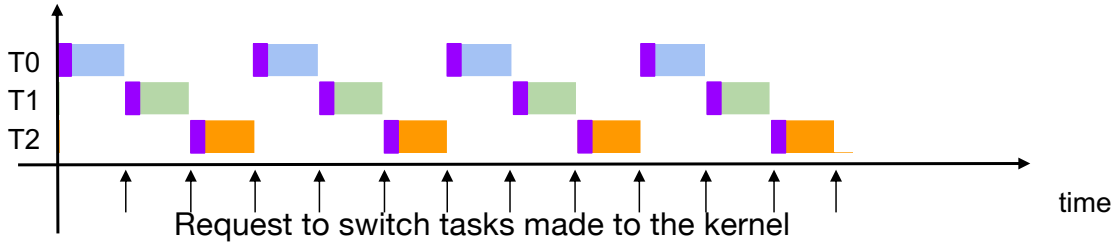
We will see how each processor can perform multiple tasks

Solution principle

Run applications in turn at a high enough frequency that the user feels that all applications are running at the same time.



there is an overhead, the duration of a task switching, it is time lost because during this time the processor does not execute the tasks (here in purple)



When to change the task?

- Periodic switching
  - A periodic IRQ from the TIMER interrupts the task in progress.
    The task switching operation will be performed by the TIMER ISR.
    The duration between two IRQs of the timer is called **tick**
    A switch can be made at each tick or multiple of ticks.
- Task-initiated switching
  - The current task can request itself the task switching, this operation is called **yield**. It loses the processor and a new task chosen by the kernel gains the processor.
  - When a task requests a service from the kernel, but this service cannot be rendered immediately because it depends on a resource (e.g., a device or a software lock), then the kernel may decide to cause a task switch.
    The task loses the processor because it can no longer move forward; when the expected resource arrives, it can resume its work.
- With the GIET, we only see the periodic switch.

How to switch tasks?

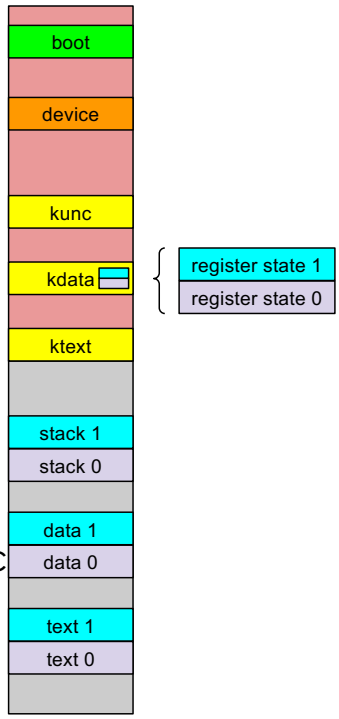
- In a switch, there are
  - the **outgoing task** which is the task that loses the processor,
  - the **incoming task** which is the task that gains the processor.

- To define the task switching, you need to:
  - determine the **context** of a task (its execution state)
  - define **a mechanism** for saving and restoring these contexts,
  - define **a** task scheduling **policy,**
    i.e. define the order in which the tasks must be executed.

- Switching between two tasks is done in 3 parts:
  - **The election of** an incoming task according to the scheduling policy,
  - **saving** the context of the outgoing task,
  - **restoration of** the context of the incoming task.

Context of a task

- Context of a task
  To execute a task, it needs :
  - 3 address segments in memory :
    - stack
    - data
    - text
  - 1 state of the heart registers
- If you have several tasks, they can share the same the same text and data address segments, but each task has its own stack segment
- Here, it is assumed that the address segments remain in memory. On this drawing, the address spac is shared by all the tasks, so there is no security between them.

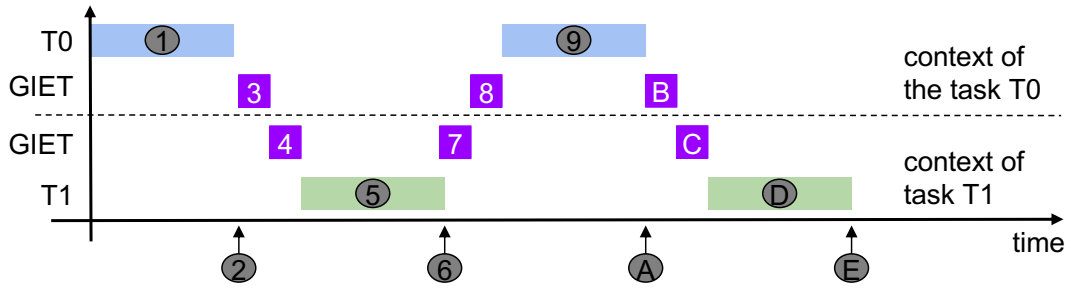


Context to save a task

- It is necessary to save the records of the heart since the heart will have to be given
  and then taken back to the tasks periodically.
- The context registers for MIPS32:
  - 32 GPR registers except $0, $26, $27
    note that $29 contains the pointer to the stack
  - HI / LO outputs of integer multipliers and dividers
  - EPC return address in the task
  - SR state of the core, switching can occur
    while the core is in USER or KERNEL mode

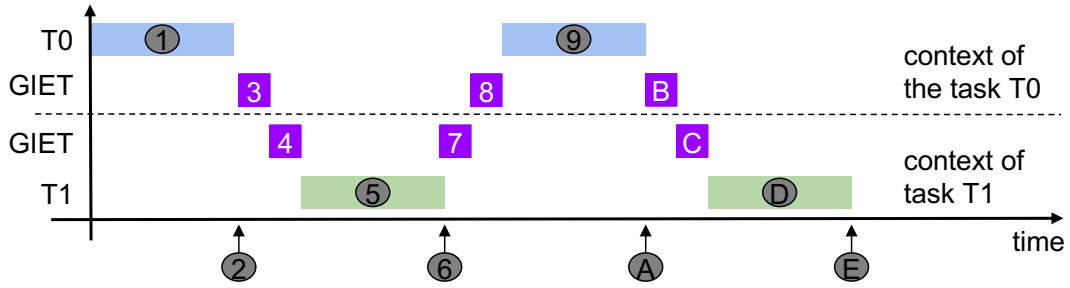
    36 registers, but for saving, we use an array aligned on 64 (2)[6]
- There is no need to save the address segments, since the segments remain in memory.

Principle of task switching



- Steady state
  1. The T0 task is executed
  2. The core receives an IRQ from the timer that interrupts the T0 task
  3. The kernel analyzes the cause and executes the TIMER ISR:
     Entering the task switching function
     a. election of an incoming task
     b. backup of the outgoing task's registers,
        restoration of the incoming job's registers
  4. Output of the task switching function, output of the ISR, output of the GIET
  5. The T1 task is executed...
- We continue like this: 6 = 2, 7 = 3, 8 = 4, 9 = 1, etc.....

Special case of start-up 1/2



In this chronogram, we are in stationary mode.

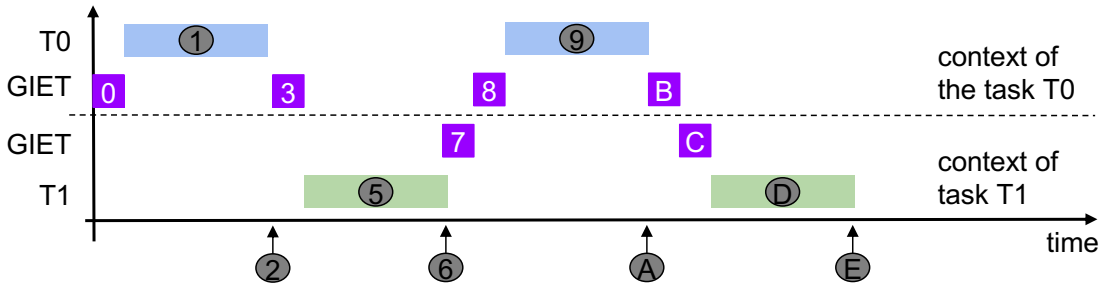
The T0 task that is interrupted by the IRQ ② enters the GIET in 3 which it will exit in 8

This means that the T1 task that is elected and then restored had already been elected 4 and that is the output of a switch that took place in the past 8 as is the output of 3

When we go from (in 3) to (in T1), 4 we stay in the GIET to exit an ISR.

If T1 had never been elected in the past, if it was the first time
we would have had to **do something different, because you can't go back to an ISR!**

Special case of start-up 2/2



In this chronogram, we have the case of the start of T0 and the start T1

The task T0 is launched by an application launcher 0
here it is in the reset by a simple : `jal main0 # if main0 is the main() of T0`

The task T0 which is interrupted by the IRQ ② enters the GIET in 3
but exits by starting directly in T1, **there was no** 4

Then, it is a stationary regime, we return in 7 there will be 8 since there was a 3

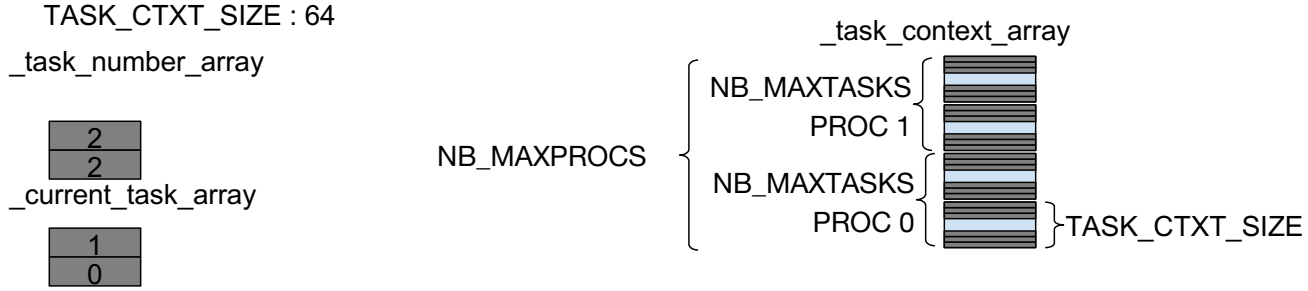
Global variables of the scheduler

The system contains several global variables for context management

- NB_PROCS : maximum number of cores in the machine
- NB_MAXTASKS : maximum number of tasks per core
- TASK_CTXT_SIZE : number of registers of a context (aligned on a then. 2)
- int _task_number_array[NB_PROCS] : array containing the number of tasks for a core
- int _current_task_array[NB_PROCS] : array containing the number of the current task
- int _task_context_array[NB_PROCS * NB_MAXTASKS * TASK_CTXT_SIZE] : context table

For example:

NB_PROCS : 2
NB_MAXTASKS : 2
TASK_CTXT_SIZE : 64



Initial state of the context

When a task has never been executed, you still have to initialize its context with an initial context because this context will be used by the _task_switch() function.

What are the registers to be defined:

- the stack pointer $29 :
  which must point to the top of the stack minus the space for the main() arguments.
- the SR register: with 0xFF13 (UM=1, IE=1 and EXL=1)
  so that at the time of the eret we go in USER mode, interruptions allowed
- the EPC register : with the address of the main() function
- the registers $4 to $7 : with the arguments of the main function
  attention, there are none in the case of GIET
- the register $31 : which contains the return address of the function
  _task_switch() which in steady state returns
  in _ctx_switch() but which initially goes to an
  eret instruction to jump directly into the task

in TME

You are going to run a multi-tasking application on several CPUs.

More precisely, up to 4 tasks on 4 CPUs

- wire the IRQs in the platform
- answer questions about thread management with GIET
- create tasks (in GIET everything is static, create at boot)
- run a multi-tasking application on 1 and then on 2 MIPS