HÁSKÓLI ÍSLANDS
Iðnaðarverkfræði-, vélaverkfræði- og tölvunarfræðideild

TÖL401G: Stýrikerfi / Operating Systems · Vormisseri 2022

Assignments 5–6 · To be solved until 8.2.2022, 13:00

Assignment 5

Consider a function `scheduler` that realises a scheduler of a single core/single processor time-sharing OS. This function will be called by the `kernel` in one of the following situations:

- The timer interrupt that signals the expiration of a time slice did occur. (In this case, the function `scheduler` is called by the corresponding interrupt handler of the kernel. The interrupt handler did already safe the context of the interrupted process in the PCB of that process.)
- The current process did request an I/O. Therefore, the process needs to be blocked until the I/O is completed. (In this case, the `scheduler` function is called after the corresponding I/O system call implementation has initiated the I/O. The kernel did already safe the context of the process that has requested the I/O in the PCB of that process.)
- An I/O device raised an interrupt to signal that its I/O has been completed. (In this case, the `scheduler` function is called after the kernel has serviced the I/O device. The kernel did already safe the context of the process that has been interrupted in the PCB of that process. The PId of the process that requested the I/O that has just been completed can then be found in the global variable `ioCompleted`[1].)

Fill out the empty branches of the following skeleton (copy it from PDF or use the LaTeX template provided in Canvas/Heimaverkefni/) of the function `scheduler`. Take care to consider also the case that the `ready` queue is empty in case all processes are blocked because all of them are waiting.

```
scheduler() {
  if (called by timer interrupt) { // Time slice of current process expired


  } else
  if (called by I/O system call) { // I/O request by current process


  } else
  if (called by I/O interrupt) { // I/O of  process ioCompleted completed


  }
  // Further code outside if statements (if required)

}
```

[1] Note: this process is not necessarily at the head of the waiting queue: the waiting queue is rather unordered and not a queue with a strict ordering.

Use pseudo code that is close to the *Java* language (full *Java* implementation is also alright). Use only the following data structures and functions that you can consider to be pre-defined:

`running`: Global variable that stores the PCB of the currently running process.

`ready`: Global queue for storing PCBs of ready processes.

`waiting`: Global queue for storing PCBs of blocked/waiting processes. (For simplicity, only one kind of I/O device and thus only one waiting queue is assumed.)

`ioCompleted`: Global variable that stores PId of that process for which the I/O has just been completed.

`addToTail(pcb, queue)`: Append *pcb* to the tail of the queue *queue*.

`removeFromHead(queue)`: Remove the element at the head of the queue *queue* and return this element. If the queue is empty, NULL is returned.

`findAndRemove(pid, queue)`: Find and return PCB entry with PId *pid* in the queue *queue*. Removes that PCB entry from the queue.

`interruptsOn()`: Enables all interrupts again. (Before entering the function `schedule`, the kernel disabled all interrupts to prevent that the scheduler itself gets interrupted.) Do not forget to enable interrupts again, otherwise the scheduler will never get activated again in future by a timer or I/O interrupt.

`sleep()`: Puts the CPU into sleep mode, only an interrupt will wake up the CPU.

`switchTo(pcb)`: Restarts the timer of the time slice expiration interrupt and switches to the context stored in *pcb*. `switchTo` does not return to the caller, but directly switches to the process specified in *pcb* by jumping to the program counter location saved in the *pcb*, i.e. any line following `switchTo` will never get executed.

Example for retrieving a process from the head of the waiting queue and switching to it:

```
running=removeFromHead(waiting)
...
switchTo(running)
```

Assignment 6

1. Find out which system calls the Microsoft Windows API offers for creating and terminating processes:
   - Describe the name of the system call (if the API offers multiple system calls for process creation and/or termination, list *all* of them),
   - Describe *briefly* the parameters of the system call (if several system calls have the same parameters, you are welcome to mention this and describe them only once),
   - Describe *briefly* the functionality provided by the system call.
2. To which pair of POSIX system calls is the Windows API system call for creating a process comparable?