A Survey on VxWorks

Jia Shi
Delft University of Technology
Delft, The Netherlands
Student ID: 5218845
Email: J.Shi-6@student.tudelft.nl

Kewei Du
Delft University of Technology
Delft, The Netherlands
Student ID: 5209633
Email: K.Du-1@student.tudelft.nl

***Abstract*—VxWorks is one of the most commercially successful real-time operating systems. In this survey, an overview on VxWorks will be provided with respect to the kernel architectures, task creation, drivers, schedulers, tracing and debugging methods and mutual exclusion methods.**

***Index Terms*—Vxworks, RTOS, schedule**

I. Introduction

VxWorks is a real-time operating system (RTOS) developed as proprietary software by Wind River Systems, a wholly owned subsidiary of TPG Capital, US. VxWorks is designed aiming at embedded systems which demands real-time, deterministic performance, safety and security certification with low latency and minimal jitter.

VxWorks supports various architecture, including AMD, Intel, POWER, ARM and RISC-V architecture. This RTOS can be also used in different core configurations: multi-core asymmetric multiprocessing (AMP), symmetric multiprocessing (SMP), and mixed modes designs on 32- and 64-bit processors.[6]

II. High-level introduction

A. Kernel architecture

VxWorks uses a single micro-kernel to handle basic kernel functions [1]. RTOS kernel is responsible for scheduling tasks, while drivers are the interface connecting the tasks with hardware.

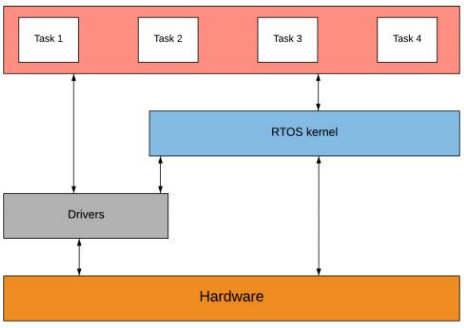

Fig. 1. Architecture Overview for VxWorks

To meet the requirements of determinism and speed required for hard real-time systems, starting from VxWorks 6.0, the operating system adopts the process model, which supports a clear delineation between kernel and applications for real-time processes (RTPs) executed in user mode with features common seen in operating systems with such space segmentation. New applications can be designed and executed in kernel space as well. VxWorks provides a set of core facilities that are included in the kernel of a common multi-tasking operating system.

The operating system can also be extended by adding custom components or modules to the kernel itself like Linux (e.g. new drivers). The system call interface can then be updated by adding custom APIs as well, which would then be available in process-based applications.

B. Task creation

VxWorks provides multiple interface for user to control the tasks. Routine **taskSpawn()** with signature

```
TASK_ID taskSpawn
    (
    char *          name,         /* name of new task
    */
    int             priority,     /* priority of new
    task */
    int             options,      /* task option word
    */
    size_t          stackSize,    /* size (bytes) of
    stack needed plus name */
    FUNCPTR         entryPt,      /* entry point of new
     task */
    _Vx_usr_arg_t arg1,           /* argument 1 to pass
     to entryPt */
    ...,
    _Vx_usr_arg_t arg10           /* argument 10 to
    pass to entryPt */
    )
```

Listing 1. taskSpawn API

, where each parameter respectively stands for the new task's name (an ASCII string), the task's priority, an options word, the stack size, the main routine address, and potential 10 arguments to be passed to the main routine represented by $args$ above to fit the equation into the paper and returns a system-assigned ID, or an error if the task cannot be created successfully.

This routine actually combines the function of two routines **taskCreate()** and **taskActivate()**: creates a new task context, including allocating the stack and setting up the task environment to call the main routine with given arguments. The new task will be assigned to the task queue - which means

activated - and begins execution at the entry to the specified routine. Thus routine **taskCreate()** (providing similar API with **taskSpawn()**) and **taskActivate()** can also be used for the same purpose.

Apart from **taskSpawn()**, VxWorks also provides a routine **taskOpen()** with a POSIX-like API for creating tasks, which is the most general purpose task-creation routine.

```
TASK_ID taskOpen
    (
    const char *  name,         /* task name -
    default name will be chosen */
    int           priority,     /* task priority */
    int           options,      /* VX_ task option
    bits */
    int           mode,         /* object management
     mode bits */
    char *        pStackBase,   /* location of
    execution stack */
    size_t        stackSize,    /* execution stack
    size */
    void *        context,      /* context value */
    FUNCPTR       entryPt,      /* entry point of
    new task */
    _Vx_usr_arg_t arg1,         /* argument 1 to
    pass to entryPt */
    ...,
    _Vx_usr_arg_t arg10         /* argument 10 to
    pass to entryPt */
    )
```

Listing 2. taskOpen API

VxWorks, like most commercial operating systems, does not support a direct periodic task model for creation and control. To implement a periodic task, when a task finishes its execution, it sleeps until the beginning of its next period. Such periodic behaviour can be implemented in the task by the usage of timers in library *timerLib*, or routine like **taskDelay()** and **taskSuspend()** in library *taskLib*. Listing below shows a possible realization of periodic task creation.

```
#include <vxworks.h>
#include <tasklib.h>
#include "pRTtaskspawn.h"

int pRTtaskSpawn(char *name, int priority, int
    options, int stacksize,
                 int period, int WECT,
                 int a, ..., int j)
{
    // ...
    for (i = 0; i < n; i++) // creating tasks
    uniquely with their names and ids taken into
    consideration
    {
        // ...
        ID[i] = taskSpawn(...);
    }

    // ...

    while (...)
    {
        // suspends a task x ticks
        taskSuspend(ID[i], x);
        // adds 5 ticks to execution time of this
    task
        totalSuspend[i] += x;

        // ...
        // if all tasks are executed successfully in
     a period it starts executing for the next
    period
        if (i++ > n)
        {
            taskDelay(period); // calling the
    taskDelay() to delay the period int j;
            for (j = 0; j < n; j++) // sets the
    incoming task priority to High priority
            {
                // ...
            }
            i = 0; // pointing to the start of the
    array
        }
    }

    for (i = 0; i < n; i++) // if all tasks complete
     execution then the memory is freed
    {
        // ...
    }

    return taskIdSelf(); // returns the taskID for
    the current executing task
}
```

Listing 3. Periodic task creation

Typically, a task does not finish always its execution at the same time point at each period. This is because the execution times and response times vary from one period to another. Hence, timers may not be an easy and precise option for the task who evaluates its time for next period relative to the current time, whenever it finishes its execution. This happens due to the possible occurrence of preemption between the time measurement and calling the sleep function.

C. Drivers

Drivers in VxWorks are different for block and non-block devices. For a type of particular non-block devices, the driver usually implements seven basic I/O functions, which are **creat()**, **remove()**, **open()**, **close()**, **read()**, **write()**, and **ioctl()**. The routine **iosDrvInstall()** installs the implements the device-specific implementation for these general functions with corresponding routines. Not all of the general I/O functions are necessary to be implemented if they are not supported by a particular device. If one routine is not implemented by a driver, a **NULL** function pointer should be used for the corresponding **iosDrvInstall()** parameter when the driver is installed. Any attempt to call a routine that is not implemented will result in returning an **ENOTSUP** error.

I/O system is responsible for mapping user I/O requests to the appropriate routine of the appropriate driver, which done by I/O system maintaining a table containing the address of each routine for each driver. Invoking the I/O system internal routine **iosDrvInstall()** can install drivers dynamically. The arguments to this routine are the addresses of the seven I/O routines for the new driver. The **iosDrvInstall()** routine inserts these addresses in the driver table in a free position, with returning the index of this slot, which index is known as the driver number and is used for associating particular devices with the driver. Figure 2 shows an example for process above

[2].

On the other hand, A driver for a block device commu-

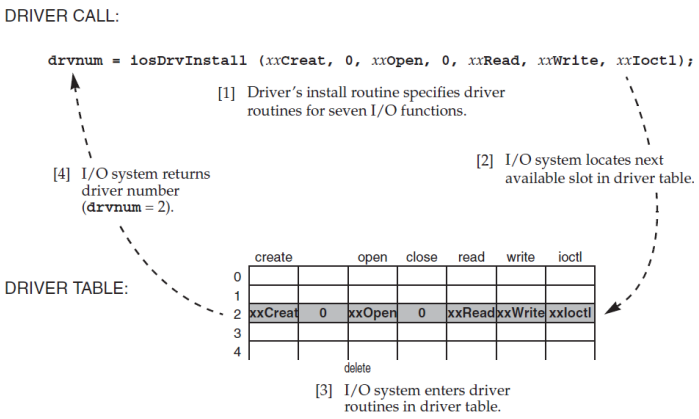

Fig. 2. Installing drivers for VxWorks

nicates with a file system, rather than with the I/O system. Most I/O functions are implemented by the file system, resulting in less routines required to be supported by driver like reading and writing blocks, resetting the device, performing I/O control, and checking device status.

III. Scheduling Policies

All work carried out in VxWorks is in the form of tasks, each of which has either of four different states: **ready**, **delay**, **pending** and **suspend**. VxWorks provides three kernel scheduler facilities: ***VxWorks native scheduler***, ***POSIX thread scheduler*** and ***kernel scheduler***. Only one scheduler above can be configured active into a VxWorks image at a time. Figure 3 illustrates the task states.

A. VxWorks native scheduler

The routine **taskSpawn()** is responsible for creating tasks in VxWorks, as introduced in previous sections. The ***VxWorks native scheduler*** is the default scheduler, available for two possible scheduling algorithms: *preemptive priority-based* scheduler and a *round-robin* scheduler, enabled by routine **kernelTimeSlice()** [3].

Numbered from 0 through 255, the kernel supports 256 priority levels. Priority 0 is the highest and priority 255 represents the lowest priority. All application tasks should be in the priority ranging from 100 to 255.

1) Round-Robin scheduler: Round-Robin (RR) is a scheduling algorithm similar to *First-Come First-Served (FCFS)* algorithm, where the ready queue is served with *FCFS*, which is shown in Figure 5. However, each task is only executed for up to **Q** slots designated by system or user, after which placed back to the beginning of queue if no preemption occurs. If a task ends before the end of one time slice, it yields the CPU and the next task in the ready queue will run, who is still only able to run for at most the length of a time slice. Figure 4 shows the idea of *Round Robin* scheduling algorithm.

Round-robin has its advantages in terms of that all the task

State Symbol	Description
READY	The task is not waiting for any resource other than the CPU.
PEND	The task is blocked due to the unavailability of some resource.
DELAY	The task is asleep for some duration.
SUSPEND	The task that is unavailable for execution (but not pended or delayed). This state is used primarily for debugging. Suspension does not inhibit state transition, only execution. Thus, pended-suspended tasks can still unblock and delayed-suspended tasks can still awaken.
STOP	The task is stopped by the debugger.
DELAY + S	The task is both delayed and suspended.
PEND + S	The task is both pended and suspended.
PEND + T	The a task is pended with a timeout value.
STOP + P	Task is pended and stopped by the debugger.
STOP + S	Task is stopped by the debugger and suspended.
STOP + T	Task is delayed and stopped by the debugger.
PEND + S + T	The task is pended with a timeout value and suspended.
STOP+P+S	Task is pended, suspended and stopped by the debugger.
STOP+P+T	Task pended with a timeout and stopped by the debugger.
STOP+S+T	Task is suspended with a timeout and stopped by the debugger
ST+P+S+T	Task is pended with a timeout, suspended, and stopped by the debugger.
state **+ I**	The task is specified by *state* (any state or combination of states listed above), plus an inherited priority.

Fig. 3. Task State Symbols

will be finished eventually, and the time slice's size can be easily modified by user. However, as expected, important tasks or tasks with emergent deadline cannot be prioritized thus terminated on time.

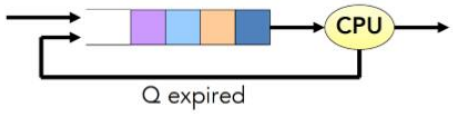

Fig. 4. Round Robin scheduler

2) Preemptive priority-based scheduler: As for this type of scheduler, task are executed based on their fixed priority. Tasks with lower priority can only start or continue its execution after the termination of tasks with higher priority's execution. The most important task always resides in CPU, current task will be interrupted and put back into the ready queue whenever a task with a higher priority arrives.

However, this scheduling method has a potential risk. If the priorities of two or more tasks are equal, whoever comes first will be taken immediately and will lock the CPU. This results in the rest of the tasks not being able to start their execution.

To solve the priority dilemma, whenever several tasks with equal priority are in the ready queue, the scheduler temporarily changes its scheduling method. The first choice is to activate

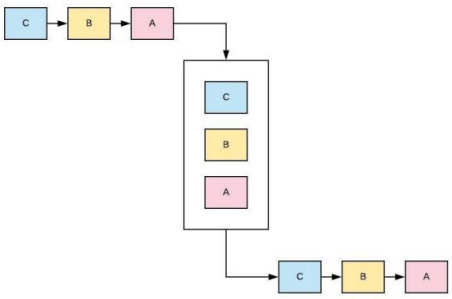

Fig. 5. FIFO Scheduling

the *Round-Robin (RR)* scheduling algorithm to switch between the tasks. The other is using a non-preemptive *First-Come First-Served (FCFS)* algorithm, which indicates a task with higher priority will never be preempted, also being the default solution in VxWorks. Both scheduling algorithms are introduced in previous sections.

Preemptive priority-based scheduler has good predictability since the task with the highest priority always run first, hence user can easily control the scheduling with tasks ranging within a large pool of priority levels. However, since the scheduler may activate *RR* method for tasks with identical priority, it is possible that a task with low priority level misses the deadline.

B. POSIX Thread Scheduler

POSIX thread scheduler is also supported by VxWorks, in particular for POSIX threads in user space. This scheduler is similar to ***VxWorks native scheduler*** for VxWorks task scheduling, with extra support for threads with different scheduling policies concurrently. It can apply different scheduling polices like Preempt-RT such as **SCHED_FIFO**, **SCHED_RR**, and **SCHED_OTHER** policies.

C. Kernel Scheduler

The replacement kernel scheduler framework allows users to customize, configure, and initialize a customized kernel scheduler for VxWorks. This mechanism offers a possibility for flexibility. However, the system's behavior cannot be guaranteed since kernel is customized anyway.

IV. Kernel Tracking and Debugging

A. Kernel Debugging

The kernel shell offers the task-level debugging utilities for kernel space if VxWorks has been configured with the **INCLUDE_DEBUG** component identical to host shell. After loading symbols into the debugger, the kernel can be debugged. You can use the host shell in four interactive modes: **C interpreter**, which executes C-language expressions; **Command (Cmd)**, a UNIX-style command interpreter; **Tcl**, to access the WTX Tcl API and for scripting; and **GDB**, for debugging with GNU Debugger commands.

B. Kernel Analysis with kprobes

Dynamic **kprobes** feature is an extension of the Linux kernel **ftrace** function tracer. Currently, *x86* is the only platform supported, where the format is instruction dependent. The **kprobes** tracer can probe instructions inside of kernel functions, which means you are allowed to check which instruction has been executed. The **kprobes** tracer can add new probe points during execution.

One of the design goals of **kprobes** is to allow the insertion and deletion from the command-line without the need for any specialized user tools. Manipulation of **kprobes** is done via the proc and sys virtual file systems. The syntax is complex and probably best implemented in a script for different processes.

V. Mutual exclusion

Mutual exclusion is the key to ensure the correctness of data when concurrent execution mechanism applied. Semaphores are the basis for synchronization and mutual exclusion in VxWorks. They are powerful in their simplicity and form the foundation for numerous VxWorks facilities.

The simplest method to achieve mutual exclusion is to disable interruption, or preemption in the system. However, both methods will harm the real-time property of the system. Say if we disable the interruption, the system prevents itself from responding to external events for the duration of these locks. Also, if disable the preemption, tasks with higher priority are unable to be executed until the locking task leaves the critical region. This happens even though the higher-priority task is not actually in the critical region.

VxWorks semaphores are highly optimized and provide the fastest inter-task communication mechanism in VxWorks. semaphores interlock the access to shared resources to achieve mutual exclusion. Better mutual exclusion with finer grain is provided than either interrupt disabling or preemptive locks.

The interface of VxWorks semaphores is provided in library **semMLib**. **semTake()** is the routine to take a semaphore while **semGive()** is the function to release a semaphore. VxWorks provides three basic types of semaphores: **VXWBSem** for *binary*, **VXWCSem** for *counting* and **VXWMSem** for *mutex*, with two additional semaphore classes **VXWSmBSem** and **VXWSmCSem**. For the purpose of introducing the semaphore related to mutual exclusion, following sections will introduce binary semaphores and mutual exclusion semaphores.

A. *VXWBSem* - binary semaphores

A binary semaphore can be viewed as a flag that is available (full, value 1) or unavailable (empty, value 0). Figure 6 describe the process of taking a semaphore semaphore for binary semaphore. When a task takes a binary semaphore using routine **semTake()**, the outcome depends on whether the semaphore is 1 (full) or 0 (empty) at the time of the call. If the semaphore is available (full), the semaphore goes to unavailable (empty) and the task continues executing immediately. Otherwise, if the semaphore is unavailable (empty), the task will be put back to a queue of blocked tasks and enters a state of pending for the availability of the semaphore.

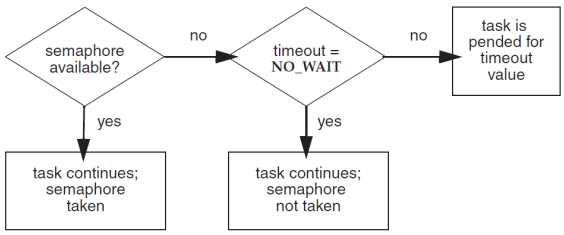

Fig. 6. Taking a Semaphore

Figure 7 describe the process of giving a semaphore semaphore for binary semaphore. When a task gives a binary semaphore, with routine **semGive()**, the result also depends on the state of semaphore - whether the semaphore is available (full) or unavailable (empty) at the time of the call. If the semaphore is available (full), nothing happens when giving back the semaphore. If the semaphore is unavailable (empty) and no task is waiting to take it, then the semaphore becomes available (full). If the semaphore is unavailable (empty) and one or more tasks are pending on its availability, then the first task in the queue of blocked tasks is unblocked, and the semaphore is left unavailable (empty).

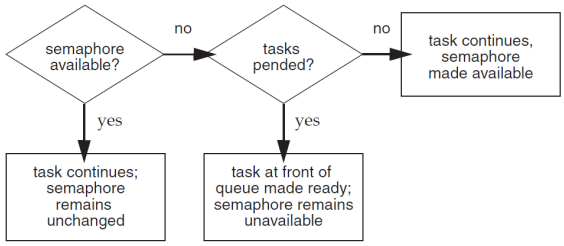

Fig. 7. Giving a Semaphore

Binary semaphores limit the scope of the mutual exclusion to only the associated resource, where the semaphores here are used to guard the source. The semaphore is initially available (full). When a task wants to access the resource, it first takes that semaphore. All other tasks requesting access to the resource are prevented from execution until the semaphore is given back. Thus, all accesses to a resource requiring mutual exclusion are bracketed with **semTake()** and **semGive()** pairs. Here lists an example code.

```
/* includes */
#include <vxWorks.h>
#include <semLib.h>

int main() {
    SEM_ID semMutex;
    /* Create a binary semaphore that is initially
    full. Tasks *
    * blocked on semaphore wait in priority order.
    */
    semMutex = semBCreate(SEM_Q_PRIORITY, SEM_FULL);

    semTake(semMutex, WAIT_FOREVER);
    /* critical region, only accessible by a single
    task at a time */
    semGive(semMutex);
}
```

Listing 4. Binary semaphore example

B. *VXWMSem - mutual exclusion semaphores*

The mutual-exclusion semaphore is a specialized binary semaphore dedicate to issues inherent in mutual exclusion, including priority inversion, deletion safety, and recursive access to resources. Mutual-exclusion semaphore resembles the binary semaphore in fundamental behaviors, except for the following differences:

- It can be used only for mutual exclusion.
- It can be given only by the task that took it.
- Routine **semFlush()** is illegal.

Apart from semaphore mechanism provided by VxWorks, the RTOS also provides thread mutexes (mutual exclusion variables) and condition variables compatible with the POSIX 1003.1c standard. They play actually the same role as VxWorks mutual exclusion and binary semaphores (and are in fact implemented using them!).

these APIs are available within library **pthreadLib**. Mutexes and condition variables have attributes associated with them like POSIX mutex. Mutex attributes are held in a data type called **pthread_mutexattr_t** containing two attributes, **protocol** and **prioceiling**.

POSIX semaphores are also supported by VxWorks. User can configure VxWorks with the **INCLUDE_POSIX_SEM** component to include the POSIX **semPxLib** library hence the POSIX semaphore routines in the system. VxWorks also provides **semPxLibInit()**, a non-POSIX (kernel-only) routine that initializes the kernel's POSIX semaphore library. It is invoked by default at boot time if POSIX semaphores have been included in the system configuration.

VI. Conclusion

In this paper, we briefly introduce one prevailed and powerful RTOS VxWorks. We go through the general architecture of the VxWorks kernel, the kernel scheduling policies, the debugging and tracing methods and mutual exclusion method in the VxWorks. These are essential characteristics making VxWorks a commercially successfully RTOS.

VII. Presentation

Please check the presentation from link https://www.youtube.com/watch?v=wamAiUsVYCc.

References

[1] D. Forsberg and M. Nilsson., "Comparison between scheduling algorithms in rtlinux and vxworks." 2006.
[2] VxWorks, "Vxworks kernel programmer's guide, 6.2," 2011.
[3] G. Palermo, M. Sam, C. Silvan, V. Zaccari, and R. Zafalo, "A comparison between the scheduling algorithms used in rtlinux and in vxworks - both from a theoretical and a contextual view," in *Proceedings of the 13th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 225–228. [Online]. Available: https://doi-org.tudelft.idm.oclc.org/10.1145/764808.764866