# Process Monitoring in Operating System Linux

ZDENEK SLANINA, VILEM SROVNAL
Department of Measurement and Control
VSB Technical University of Ostrava
17. listopadu 15, 708 33 Ostrava-Poruba
CZECH REPUBLIC

*Abstract*: The article deals with a design of system module for the selected processes monitoring in the operating system RT-Linux. The designed module will be able to observe states of selected processes in real-time (start, stop, interruption ...) and visualize changes of states on the remote Linux system. For the better explanation of problems are given basic characteristics of operating systems Linux and RT-Linux. There are described the initiate problems solution, process states monitoring and time sequence of task processing in real time.

*Key-Words:* Linux, RT-Linux, process, scheduling, monitoring, embedded systems, real-time systems

## 1 Introduction

The present technological processes control uses number of technical resources as intelligent sensors, microcontrollers, PLC's, personal computers and workstations. The communication between resources is realized by different types of industrial buses, computer networks and operating systems.

If control systems are realized with personal computers, these computers demand the real-time processing mode. There are required the preemptive multi-processing of concurrent tasks or the pseudo-parallel technique of processing.

The real-time operation systems are ready to process external events any time. The processing of demanded solutions is obtained in prior given intervals. The system can accept data as casual events or data are periodically scanned in intervals in advance with respect of appropriate application.

Real-time systems have to react at signals from external environment, events, according to given time pre-limits. The proper behavior of such system depends not only on evaluation's results executed by processes, but also on the elapsed time for their evaluation. The delayed reaction need not to be up-to-date for the appropriate control action, the delay can cause crash of the corresponding application [4], [5].

The design of control system needs the knowledge of its behavior in many situations as standard or emergency and so on. The creation of monitoring kernel module is very useful for the system debugging and error detection especially in the real-time processing.

## 2 Operating system Linux

### 2.1 Basic description

The operating system Linux is obtainable in the form of free distributed implementation of UNIX kernel [1]. This is the base of lowest operating system level. The operating system core is compiled and installed on the computer with many specific free distributed programs, which make possible to design the complex operating system. Such installations are called Linux systems while kernels are not unique. The complicate installation originates a Linux distribution [2]. Distributions are realized by various mediums (floppy, CD). There are combined kernels and many next support programs, programming languages and utilities. The X-Windows server is involved as graphical user interface of UNIX systems too.

The kernel is the crucial part of each operating system. Linux kernel is compiled by several important subsystems (modules), which are briefly described below. The created interface between the user, operating system and hardware is shown on the figure 1.

Files and devices are controlled by the small number of functions in Linux. These functions are called as system calls. They are Linux components and make interface between the operating system and applications [3].

The problem is the efficiency direct using of these functions for inputs and outputs. The performance of system goes down as a result of switching between user and kernel mode all the time. Function’s libraries are used scores of time. It is possible to use the function, which is dedicated directly to work with the specific device. Linux provides a range of standard libraries as the sophisticated interface for devices and disc files.

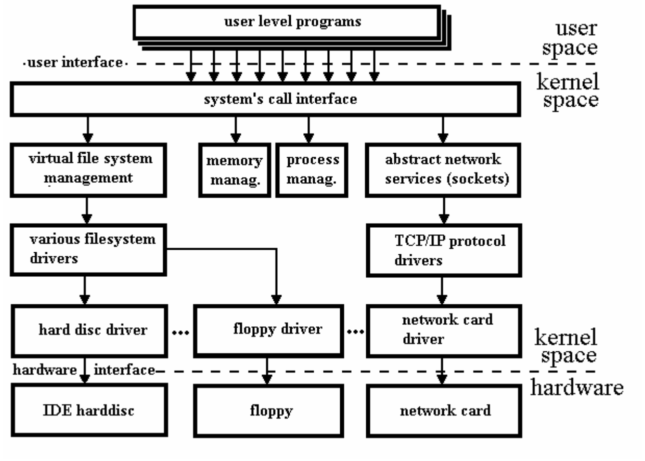

Fig. 1 Interface between user and hardware

## 2.2 Kernel modules

Virtual File System (VFS) creates the universal interface for the using of various file systems. The each type of file systems provides the implementation of specific set of operations, which are common for all file systems.

If any system component sends the request to use the one of file systems, its request goes through VFS. VFS forwards it to the relevant file system driver. VFS provides the user interface both for file systems (FAT, ext2...) and devices. The kernel provides the unified interface for user applications.

Devices include partly hardware devices (hard disc, tape memory...) partly software devices (/dev/random - device for generating of random data...). Special services require networks. While these services are non-standard (different then for file systems), they belong to the VFS too. Users communicate as with network devices as with standard devices.

The memory manager provides following functions:

- Virtual address space – the operating system provides the virtual memory. The size of virtual memory is much greater than the size of physical memory in the system.
- Memory protection - each process in the system has its own virtual address space. Virtual address spaces are completely mutually separated. The running application process doesn't affect other processes. The hardware mechanism of virtual memory protects relevant memory areas against writing. The code and data are protected in the memory against destructive operations of other applications.
- Memory mapping – the memory mapping serves for mapping program's images and data files to the address space of the process. When the memory mapping is used the content of file is directly linked with the virtual address space of the process.
- Physical memory allocation – the memory manager subsystem allows each running process to allocate appropriate part of the system physical memory.

1. Virtual memory sharing – while the virtual memory allocate to the process the separated address space, within the running of processes are situations when processes need to share virtual memory among themselves. Dynamic libraries are the one example of sharing code by several processes. The shared memory is also a buffer, which is used in the interprocesses communication when information is exchanged among processes. The Linux supports the interprocesses communication by using UNIX system V IPC mechanism.

Linux provides the virtual memory system as the extension of RAM memory. The efficient size of memory is much greater. The kernel swaps contents of just unused memory blocks on the disc and releases memory for other functions. If it is requested the content of blocks is loaded back to the memory. These operations are the fully transparent for users. Running Linux programs allocate only the appropriate size of accessible physical memory and don't take care of the virtual disc space. Of course, disc operations are not as quick as on the physical memory - RAM.

Linux use the plain file or the special disc area for swapping. The advantage of independent disc segment is speed. The advantage of swap file is possibility to change size of swap space simply. If the size of swap space is known, then is better using a disc segment. In case of no direct demands is better using a swap file. Linux provides multiple usages of swap areas or swap files.

The process management module control multitasking. It concerns the creating of processes and switching processor among active processes. Linux threads implementation is called one-to-one executed at kernel level. Each thread means independent process for the kernel. The scheduler of processes doesn't make differences between processes and threads. Disadvantage of this model is too big overhead through threads switching. P-thread library is provided for threads, which are implementing in agreement with POSIX standard.

The data structure *task_struct* enable the process management in Linux. The terms task and process are equivalent in Linux. The *task_struct* describes properties and states of processes in the system. These data structures create the task vector, which is the array of pointers to all structures *task_struct* in the system. It means that maximal number of processes is limited by the size of task vector (512 items implicitly). The new structure *task_struct* is allocated in the memory as the part of vector task during a process creation. There is possible a reference by the *current* pointer to the actual process for the searching facilitation. Individual items of *task_struct* are separated to several areas.

The first is a state - the state of the process changes according to processing conditions. Processes in Linux are found in following states:

- RUNNING - process is running now (actually process) or it is ready to run (process waiting for the processor).
- INTERRUPTIBLE - process is waiting for processing and it is wake up by signal or timer expiration.
- UNINTERRUPTIBLE - process waiting for processing and it can't be wake up.
- ZOMBIE - finished process with structure *task_struct* in vector task by any reason. This process is inactive.
- STOPPED - process was stopped by any signal usually. In that state is a debugged process by example.
- EXCLUSIVE – this state is created as a logical combination of states with state INTERRUPTIBLE and UNINTERRUPTIBLE.

The Linux and Unix use in the file descriptors attributes for the unauthorized access protection. Each file and directory has its owners. Attributes define the access right for user (owner), group and anyone. The basic file protection defines other three protection bits as rights for read, write and execution. Each group of users can have another access rights. For example, the owner can read and write in the file, group can read only and all other users (processes) have the access to the file disabled.

The group definition enables to assign privileges to the groups of users, not only to one user or all users in the system. The right for process execution is possible assigned to number of groups (maximal number is 32 implicitly). These groups are saved to the group's vector in the structure *task_struct* of each process. If a group has access rights to a file and the process belongs to this group, then the process has group rights to the file.

There are user and group pairs of process attributes in the structure above:

- *uid* and *gid* - identifiers of user and group in the name of user running process
- *effective uid* and *gid* - some processes change their *uid* and *gid* within running process, their own are saved as attributes in *inods* of executing image. These processes are called *setuid* processes and they are very useful because they present way to restrict access to services executed by name of any other as network daemons. *Effective uid* and *gid* are set according to attributes of *setuid* process, values of *uid* and *gid* are unchanged. *Effective uid* and *gid* use the kernel for checking of access rights.
- *filesystem uid* and *gid* - similar to *effective uid* and *gid*. They are used for access rights checking to file system. It is necessary for connected file systems, when NFS server in user mode need access to files as some process. In this case, *filesystem uid* and *gid* are changed instead of *effective uid* and *gid*. This way eliminates the situation when some sends to the server the kill signal. Kill signals are submitted to processes with *effective uid* and *gid*.
- *saved uid* and *gid* - values required by POSIX standard and they are used in processes changing *uid* and *gid* of the process using system calls. When values of *uid* and *gid* are changed, real values of *uid* and *gid* are saved in them.

Each process has its process identifier. Identifier is not an index in the task vector, it is only a number. In the Linux there is no system process to depend on any other processes. All processes have their generic processes excluding the initial process. Each *task_struct* structure of each process contains a pointer to its generic process and siblings (rest processes with the same generic process) and pointers to its descent processes. Moreover all processes in the system are related in the both directions list, its root is *task_struct* structure of init process. The kernel uses this list to the view above all processes in the system.

The kernel keeps information about time of process starting and the total processor time of process. The kernel keeps also values, which processing time is the process in the system and user mode. The Linux supports interval timers of processes. Process can call set timers using system calls to call signal after time period is expired. These timers can be one-off or periodic.

All processes run partly in the user mode and partly in the system mode. These modes are supported by the low-level hardware. There is a specific security mechanism for the switching between user and system mode. In the user mode, a process has obviously minor privileges than in the system mode.

Always when system calls are used, the processing is switched from the user mode to the system mode.

The kernel works in the name of process in the time of system mode.

Linux uses preemptive tasks planning. The one of planning strategies is round-robin. The each process is running a set time (for example 200 ms). When this time expired, other process use processor and previous process has to wait for the next opportunity to run. This time period is called time-slice.

The scheduler decides which process will run. Linux scheduler selects the actual processes on the base of priority algorithm. The scheduler saves the actual process status, values of processor registries and other context information to data structure *task_struct* when the new process is choosing. Then the scheduler restores the state of new planned process. The scheduler keeps following information in structure *task_struct* of each process for a realization of planning strategies:

- *policy* - scheduling strategy is associated to the relevant process. There are two types of processes in Linux - standard and real-time. Real-time processes have higher priorities than all other processes. If real-time process is ready to go, it will be run. Two strategies are applied for real-time processes either round-robin or FIFO (First In First Out). In round-robin scheduling is used the cyclic switching of processes. They are executed cyclic in queue. The strategy FIFO means execution of processes in the order of ready to execution.
- *priority* – the scheduler assigns the priority to the process. It is a quantity of the time (in jiffy units), that process can use, when it is running. The priority of processes is possible to change using system calls and with *renice* command.
- *rt_priority* - Linux supports real-time processes with high priority than other processes in the system. This item allows to scheduler assign to each process its relative priority. The priority of real-time processes is possible to change using system calls.
- *counter* - number of time jiffy when the process is running. At the process planning is this value set as the priority value. The counter is decremented with every time pulse.

The scheduler is activated in several points in the kernel: actual process is transferred to queue of waiting processes; system call is finished; before the switching of process from system to the kernel mode. The next reason is the decrement of counter value to zero.

*Process selection* to execution – the scheduler looks in the priority queue of processes. If the real-time process is in the queue, its rate is higher than standard processes. The weight of standard process is equal to the counter value. The real-time process weight is 1000 higher. It means that real-time processes will execute before standard processes. The actual process, which is running (value counter is decremented) has handicap before other processes with the same priority. When priorities of processes are equal, the scheduler chooses the first process in the queue. The actual process is scheduled for the end of queue at the switching. Processes are executed one by one in the balanced system with same priorities of processes. It is round-robin planning – the cyclic planning of processes. The sequence of waiting processes is possible to change.

*Process switch* - if switching conditions occurs, the actual process is stopped and the new process is ready to run. The running process uses registers and processor and system memories. The every call of routines sets parameters in registers and use values in the stack, for example, to save a return address of calling routine. If the process is suspending, it is necessary to save its state including the program counter and all registers of processor to its *task_struct* structure. Then the state of new planned process is necessary to restore. This operation is a machine dependent, each processor use an own way with the hardware support.

The process switch is the last scheduler's operation. The saved context of previous process is image of hardware context in time of end of process scheduling. So when is loaded a new process, there are know information about the situation before, including the content of counter of instructions and registers.

## 3 Operating system RT-Linux

There are two different approaches to obtain RT tasks executing in Linux:

1. Improving the Linux kernel preemption.
2. Adding a new software layer beneath Linux kernel with full control of interrupts and processor key features.

These two approaches are known as "*preemption improvement*" and "*interrupt abstraction*" respectively. This second approach is the one used by RTLinux.

RT-Linux scheduler uses Linux kernel as its inactive task. Linux is running in the case that no real-time process in the real-time mode is active. The process in Linux unblocks interruption or prevent switch in any time. This mechanism is possible thanks to the software emulation of hardware interruption.

There are important features, which are achieved in real-time processing in the kernel mode:

- Thread processing is in the operating memory of kernel.

- Threads processing is in the kernel mode and threads have complete access to basic layer.
- Application is compiled and installed in the same memory space like real-time operating system. System calls are implemented using simple system call that doesn't use software interruption by the reason of decrement time of operating system overhead.

RT-Linux is following the POSIX 1003.13 minimal realtime operating system standard. The design of RT-Linux is subordinated to POSIX requirements. The system can run on i386, PPC and ARM architectures.

Following tools are provided for applications debugging:

- Debugging at source code level with SMP support at the target machine, cross-debugging is not possible.
- Tracing - kernel tracing and application events.
- POSIX tracing.

Memory management:

- static
- dynamic - dynamic memory allocation is not available (functions *malloc* and *free*); RT-Linux doesn't allows it nor use internally
- protected address space - application threads and RT-Linux threads run at same address space; by some point of view is Linux host system for RT-Linux; Linux has complete control above system memory

Interprocess communication:

- FIFO - communication mechanism is determined to communication between real-time processes and Linux user processes (not compatible with POSIX norm)

Synchronization:

- mutexes - POSIX mutexes; system allows PRIORITY_PROTECT protocol for handling with priority inversion problem
- condition variables - POSIX condition variables
- semaphores - POSIX semaphores

Nowadays are developed various new components for RT-Linux for more effective work with, for measurement and control is Linux interface Comedi, etc.

# 4 Monitoring module

The basic goal of monitoring module is the maximal usage of data structure *task_struct* contains all information about processes in system Linux. Then the process monitoring is possible separate to two basic parts.

The first part is a process status, which is read from the task vector. This procedure has to be very fast. In the case when this procedure is integrated with scheduler, it will have following consequences. Each reading of status evokes a delay of process switching; it can be a problem in real-time systems. The effective processor time is decreased as a result of following actions: switch for status reading; own reading; compare with table of desired monitored processes; writing of data in case of positive result; switching back to the actual process. On the other side, when the monitoring is finished, the processor will have more system time for other process services.

If the actual process is the one of monitored processes, the status information of actual process is at disposal to the second part of monitoring system. The second part is a visualization system, which is running slowly. Changes in kernel are very quick and in the case of exact visualization, it is not scan able by the operator. The second part writes data in the given format on the appropriate device. It is possible use as the device a monitor, hard disk or Ethernet. In the case of remote visualization and unavailability of devices above is possible to create the special device for direct connection with PC buses (PCI, PC/104 ...).

The format of written data is depended on the used device, for example, in the memory medium (hard disk) is saved names of processes and times. For the visualization on the monitor is used the one of graphic libraries (GtK).

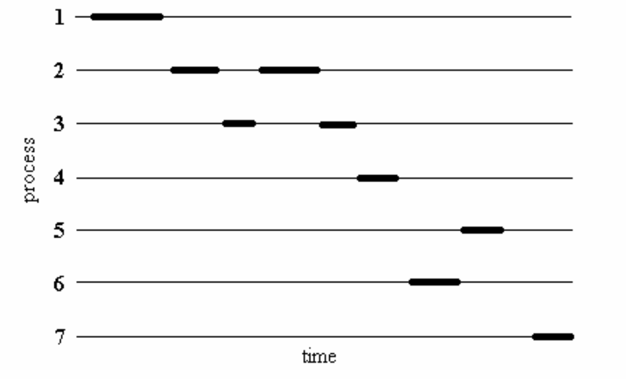

Fig. 2: The monitoring example

The monitoring example is shown on the figure 2. There is an example of processes which are scanning by the monitoring module. There are processes with numbers (1 to 7). Processes 2 to 6 are application processes which are debugging. Processes 1 and 7 are system processes, e.g. drivers for measurement cards, etc. The monitoring module allows saving time data to the file. It is possible analyze a system behavior after the system halt: events in system; exact times of input or output events; times of processing; feedback reactions... It is possible to visualize a behavior on the remote computer.

The block diagram of monitoring system is shown on the figure 3.

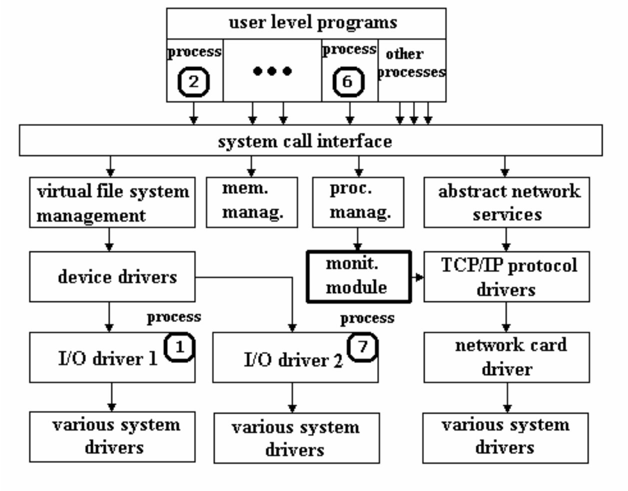

Fig. 3 The block diagram of process monitoring

## 5 Conclusion

The main goal of this project is a support of embedded systems design. The designer obtains information of process behavior in the phase of design, testing and real operation. Because in phase of testing is difficult catch all situations, usage of such system could be expedient. It is possible a testing if the chosen hardware is suitable for the real-time application or it is necessary use a more powerful hardware.

Acknowledgement: The project is solving on VSB Technical University of Ostrava, Czech Republic. The Ministry of Education of Czech Republic supplied the results of the project LN00B029 with subvention. The Grant Agency of Czech Republic supplied the results of the project 102/02/1032 with subvention.

*References:*

[1] Sobell M.G.: *A practical guide to Linux*, Addison-Wesley 1997

[2] Matthew N., Stones R.: *Beginning Linux programming*, Wrox Press 2000

[3] Rubini A., Corbet J.: *Linux device drivers*, Computer Press 2001

[4] Srovnal, V.: *Operating Systems for Real-time Control*, VŠB Technical University of Ostrava 2003 (In Czech)

[5] Kocis T., Srovnal V.: Operating Systems for Embedded Computers. In : *Programmable Devices and Systems 2003-IFAC Workshop*, Pergamon Press-Elsevier 2003, pp. 359-364