Linux 2.4

- Non-preemptible kernel
  - A system call might take long time to complete
- Coarse timer resolution
  - Tasks can be released only with 10ms precision
- Virtual memory
  - Introduces unpredictable amount of delays
- Variable priority
  - Each task is assigned a priority which varies over the time; this is to achieve fairness
- Linux will batch several operations for efficient use of H/W delaying all tasks
- Linux also reorders requests from multiple processes for H/W efficiency
- Linux does not preempt low priority task in a system call, instead queues the high priority task
- Linux makes high priority tasks to wait for low priority tasks to release resources

Therefore, Linux 2.4 cannot provide real-time guarantees

Linux

- Linux does not provide a reliable mechanism to wake a task up at a certain time. The sleep timer can only promise to wake a task up after a certain time.
- The kernel can't be preempted as long as system calls executed in the kernel can interfere with timing constraints.
- External interrupts are disabled at several sections in the kernel to protect OS data from corruption (saves the effort of making them reentrant). This adds unpredictability to the amount of time it takes to respond to I/O.
- These has been some progress on some of these issues in the 2.6 Linux kernel:

  1. the kernel is now preemptable although not all of it is.

  2. improvements in I/O handling.
- Despite these changes 2.6 is still not suitable for hard real time performance.

Windows

- Windows (NT based) suffers from most of the same problems Linux does.
- Windows doesn't support the specification of timing constraints for tasks.
- Interrupts may be delayed for an arbitrarily long time in critical sections of the kernel.
- Handling of interrupts is done in two parts: the second part (deferred procedure call) is executed later in a FIFO manner and this may cause delays if a lot of DPCs are awaiting dispatch.
- Requests for synchronization objects such as semaphores are also done in FIFO, leading to similar unpredictable delays.

Windows

- The windows scheduler supports threads on an OS level. Each thread is given a priority (0-31).
- The scheduler has compensation for input bound threads and anti-starvation mechanisms.
- Windows has a “real time” priority for threads but other real-time threads and the kernel can preempt a task with real time priority (15-26).
- A “real time” priority is higher than some parts of the OS, causing most non-real-time apps to become unusable

Fixing These Problems

- Fixing these problems in either OS would require either some form of “hack” into the operating systems design or a massive rewrite of the kernel.
- This has been tried using both of the operating systems as we will see in later case studies (rt-linux, rialto).

Linux

- Real-time extensions of Linux
  - major shortcomings of Linux
    - the disabling of interrupts when a task is in critical sections
    - the disk driver may disable interrupts for a few hundred microseconds at a time
  - scheduling
    - scheduling policies
      - *SCHED_FIFO*, *SCHED_RR*: fixed-priority, applicable to real-time tasks
      - *SCHED_OTHER*: time-sharing basis
    - 100 priority levels
      - can determine the maximum and minimum priorities associated with a scheduling policy
      - *sched_get_priority_min()*, *sched_get_priority_max()*
    - for round-robin policy, the size of the time slices given to a task can be inquired using *sched_rr_get_interval()*

Linux

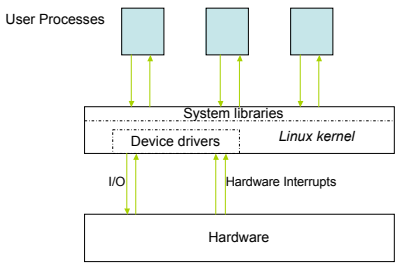
- clock and timer resolution
  - actual resolution of Linux timers: 10 millisecond
  - UTIME high resolution time service: provides microsecond clock and timer granularity
- threads
  - clone(): creates a process that shares the address space of its parent process and specified parts of the parent's context
  - LinuxThreads: provides most of POSIX thread extension API functions
- examples
  - KURT *Kansas University Real-Time System*
  - RT Linux

RTLinux

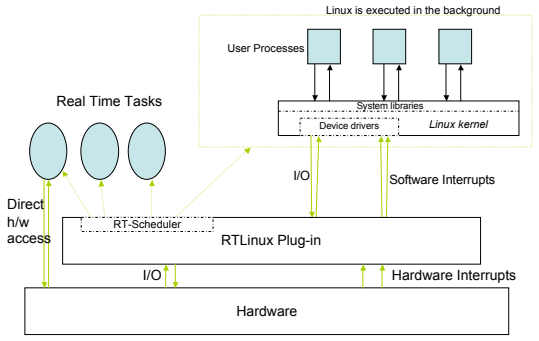
- Open source Linux project
- Supports x86, PowerPC, Alpha
- Available as a patch to the regular Linux kernel
- Provides an RT API for developers
- Runs Linux kernel as lowest priority process

- Configure and compile a fresh Linux kernel (2.4.29)
  - Download the kernel from
    http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.29.tar.gz
- Patch the RTLinux to the Linux kernel
- Recompile the kernel and reboot the system into the new kernel
- Configure the RTLinux kernel and compile it

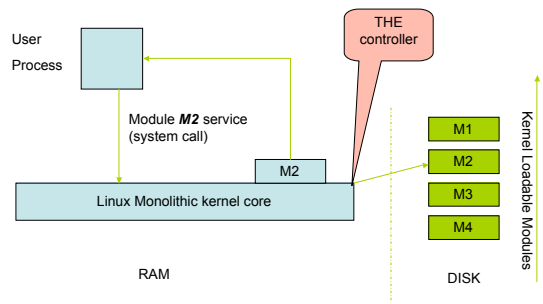
Linux



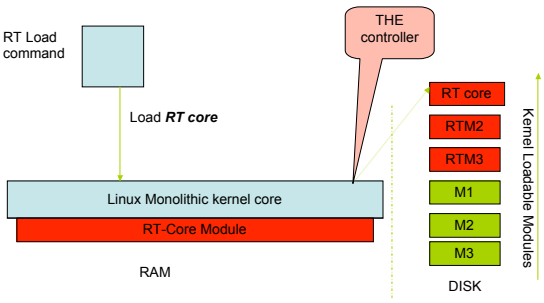
RTLinux



Linux



RTLinux



RTLinux

- Step 1: Write the RT-application program as a kernel module
  - (You are still in Linux)
- Step 2: Compile the module and check for errors
  - (You are still in Linux)
- Step 3: Load the RT-Core (or RT Scheduler, etc..) and the RT-application program module
  - (After this we will be in RTLinux)
- Step 4: Now you are in RTLinux!!!!

RTLinux

- All RT-Tasks are kernel modules.
- A user program runs in user space and the kernel module runs in the kernel space
- A user program starts running at its "**main**" function
- A kernel module starts running at the "**init_module**" function and exits via "**cleanup_module**" function
- Therefore, for a kernel module "init_module" and 'cleanup_module" are as necessary as "main" is necessary for a C program

Typical Kernel Module

Init_module()

{

…….

Perform your module initiation stuff etc..

}

Cleanup_module()

{

…….

Perform cleanup stuff like destroying the threads, freeing memory, etc..

}

RTLinux Module

Init_module() {

For each thread:
//Choose the priority functio
pthread_set_priority(..);

Assign the "task_function" to the thread

//Create the thread

pthread_create(.., task_function,..); }

This function tells the scheduler that I am a periodic task with period "p"

Task_function()

pthread_make_periodic_np(p);

while(1)

{

Pthread_wait_np();

//perform the task function

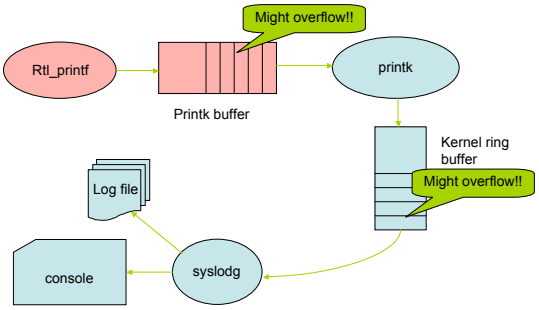
//for this **instance**

}

Cleanup_module()
{
…….
//Delete the thread
pthread_delete_np(..);
}

This function tells the scheduler that I am done for this instance and schedule me after "p" units of time

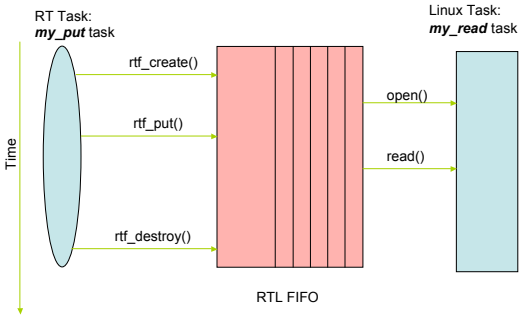
Rtl_printf



RTL-FIFO

- Creating and destroying FIFOS:
  - Using functions: rtf_create and rtf_destroy
  - These functions are called from linux process: open(), read(), write();
- Accessing FIFOs from RTL threads
  - using functions: rtf_put and rtf_get

RTL-FIFO



RTLinux

- Real time threads and interrupt handlers never delayed by non-realtime operations
- Preemptible kernel.
  - Its routines are very small and fast, this does not cause big delays.
  - Interrupts from Linux are disabled.
- RT-Linux has many kinds of Schedulers.
  - FIFO.
    - Used to pass information between real-time process and ordinary Linux process.
    - Designed to never block the real-time task.
  - The “earliest deadline first” scheduler.
  - Rate-monotonic scheduler.

RTAI

- Hard real-time extension to the Linux kernel
- A patch to the Linux kernel which introduces a hardware abstraction layer
- A broad variety of services which make realtime programmers' lifes easier
- RTAI provides deterministic response to interrupts, POSIX compliant and native RTAI realtime tasks.
- Linux application is able to execute without any modification
- RTAI considers Linux as a background task running when no real time activity occurs.

RTAI

- RTAI is very much module oriented
- real time scheduler module
  - Task functions
  - Timing functions
  - Semaphore functions
  - Mailbox functions
  - Intertask communication functions
- Fifo services
- Shared memory
- Posix pthread and pqueue(msg queue)

RTLinux vs RTAI

- RTAI provides better real-time support than RTLinux
  - soft real-time in user space along with hard real-time in kernel space
  - excellent performance in terms of low jitter and low latency
  - better C++ support and more complete feature set

Rialto

- Rialto is a real time system developed at Microsoft Research based on the windows NT architecture.
- The main idea: rewriting the kernel and scheduler so that they support real time deadlines.
- Unlike RTLinux it doesn't require the separation of the hard real time functionality from the rest of the code.
- It uses the beginConstraint and endConstraint function calls that pass the parameters of the real time portion of task to the scheduler.
- The parameters passed to the scheduler:
  - release time
  - deadline
  - running time estimate
  - criticality (critical=hard constraint , non-critical=soft)
- The scheduler returns the feasibility of scheduling the task. This allows the task to perform load shedding if possible, or fail gracefully if not.

Rialto

- The scheduling in Rialto is quite complicated:
  1. Tasks can be granted CPU time reservations. This is the average amount of CPU time they should get.
  2. The scheduler is based on a variation of preemptive LST which uses the maximal amount of time before a task has to start running (deadline-running time estimate).
  3. Critical tasks always run before non-critical tasks.
- The Rialto system is system-resource aware.
- Tasks can negotiate with the **resource manager** in order to reserve the resources they need.
- The resource manager arbitrates the use of resources, and can cause an existing task to relinquish resources allocated to it.

Rialto

- Rialto doesn't handle the problem of the kernel disabling interrupts.
- The schedulability test was not fully implemented originally.
- Summary:
  - Rialto uses the approach of rewriting an existing kernel in order to allow support for real time performance.
  - built as "academic" work, was never widely used.
  - uses a novel scheduling approach, but because it is complicated it is sensitive to many implementation details.

VxWorks

- VxWorks is a commercial hard real time operating system developed by Wind River Systems.
- VxWorks is the most established and most widely deployed device software operating system.
- The main idea: use a monolithic kernel to schedule user tasks according to user defined priorities. Maximize kernel timing predictability.
- Gives the users maximal control.
- A dedicated real time system, not intended as a general purpose OS.
- Lacks many modern OS features that interfere with real time performance (flat memory model, no paging).

VxWorks

- Scheduling is done using a preemptive priority driven approach, priorities are chosen arbitrarily by the user (0-256).
- Priorities can be changed by the user at runtime but this is discouraged.
- A user can lock a task so that it can't be preempted even by higher priority tasks or interrupts.
- This allows the use of the fixed priority response time analysis to check schedulability offline.

VxWorks

- Is resource sharing aware and has a priority inheritance built in.
- Optimizations in implementation of context switches and the return from interrupts.
- The kernel never disables NMI (non-maskable interrupts) so they are always available to the user.
- Lacks many modern OS features.
- Guaranteeing the deadlines is the responsibility of the user at design time.
- Doesn't support most modern applications and APIs (only a small subset of POSIX).
- Despite the flat memory model, dynamic memory allocation still causes memory fragmenting, which increases timing unpredictability.

VxWorks

- Summary:
  - a dedicated and widely used real time system.
  - offers the user maximal control , but also passes him responsibility for the deadlines.
  - lacks many modern OS features.

Lynx OS

- Microkernel design
  - Means the kernel footprint is small
  - Only 28 KB in size
- The small kernel provides essential services in scheduling, interrupt dispatching and synchronization
- The other services are provided by kernel lightweight service modules, called Kernel Plug-Ins (KPIs)
- New KPIs can be added to the microkernel and can be configured to support I/O, file systems, TCP/IP, streams and sockets
- Can function as a multipurpose UNIX OS

Lynx OS

- Here KPIs are multi-threaded, which means each KPI can create as many thread as it want
- There is no context switch when sending a message to a KPI
  - For example, when a RFS (Request for Service) message is sent to a File System KPI, this does not request a context switch
  - Hence run-time overhead is minimum
  - Further, inter KPI communication incurs minimal overhead with it consuming only very few instructions
- Lynx OS is a self hosted system – wherein development can be done in the same system

Lynx OS

- In such a system, there is a need for protecting the OS from such huge memory consuming applications (compilers, debuggers)
- LynxOS offers memory protection through hardware MMUs
- Applications make I/O requests to I/O system through system calls
- Kernel directs I/O request to the device driver
- Each device driver has an interrupt handler and kernel thread

Lynx OS

- The interrupt handler carries the first step of interrupt handling
- If it does not complete the processing, it sets an asynchronous trap to the kernel
- Later, when kernel can respond to the software interrupt, it schedules an instance of the kernel thread to complete the interrupt processing

QNX/Neutrino

- POSIX-compliant Unix-like real-time operating system.
- Microkernel design – kernel provides essential threads and real-time services
- use of a microkernel allows users (developers) to turn off any functionality they do not require without having to change the OS itself.
- The system is quite small, fitting in a minimal fashion on a single floppy, and is considered to be both very fast and fairly "complete."
- The footprint of microkernel is 12kb.

QNX/Neutrino

- Every driver, application, protocol stack, and file system runs outside the kernel, in the safety of memory-protected user space.
- As a result, virtually any component can fail - and be automatically restarted -without affecting other components or the kernel.
- Maximize application portability with extensive support for the POSIX standard, which lets you quickly migrate Linux, Unix, and other open source programs

- QNX is a message passing operating system
  - Messages are basic means of interprocess communication among all threads
  - Follows a message based priority tracking feature

VRTX

- VRTX has two multitasking kernels
  - VRTXsa (scalable architecture)
    - designed for performance
    - Provides priority inheritance, POSIX compliant libraries
    - Supports multiprocessing
    - System calls fully preemptable and deterministic
  - VRTXmc (micro-controller)
    - Designed for low memory consumption
    - Used for cellular phones and hand-held devices
- Rather than providing optional components provides hooks for extensibility – application can add its own system calls

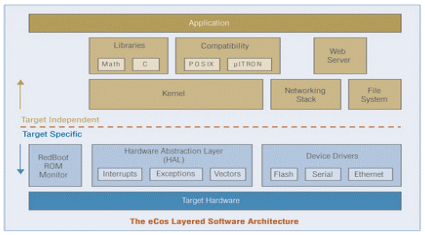
eCos (Embedded Configurable OS)

- Open source, royalty-free
- Highly Configurable nature
  - Small footprint
  - Application specific
  - Multiple implementation of kernel functions including scheduling, allocating memory and interrupt handling
- Easily Portable
  - Hardware Abstraction Language (HAL)
- Native API, POSIX API, µITRON API, C API
- eCos is targeted at high-volume applications in consumer electronics, telecommunications, automotive, and other deeply embedded applications.
- Ecos has kernel mode
  - No user mode
- Implemented using C++
- GNU debugger (GDB) support

eCos

- Features
  - Choice of scheduling algorithms
  - Choice of memory-allocation strategies
  - Timers and counters
  - Support for interrupts and DSRs
  - Exception handling
  - ISO C library , Math library
  - Rich set of synchronization primitives
  - Host debug and communications support

eCos



The eCos Layered Software Architecture

Figure 1

The eCos Layered Software Architecture abstracts target details from the high-level application, providing excellent platform portability, while still maintaining performance at the low-level target drivers and services

MicroC/OSII

- Also known as µC/OS II or uC/OSII
- MicroC/OS has been designed as a small footprint real time pre-emptive OS that was designed for embedded use on 8 bit platforms upwards
- highly portable, ROMable, very scalable, preemptive real-time, multitasking kernel
- has ports for most popular processors and boards in the market
- suitable for use in safety critical embedded systems such as aviation, medical systems and nuclear installations
- Over 100 microprocessors are supported
- approved for use in a DO-178B aerospace system and is (apparently) MISRA-C compliant

MicroC/OSII

- µC/OS II features
  - reentrant functions and is portable to different processors
- kernel is preemptive real time, managing up to 64 tasks, with up to 56 tasks for each application
- Each task has a unique priority and its own stack
- Round robin scheduling is not supported
- operating system uses semaphores to restrict access to resources shared by multiple elements of the system
- Memory management is performed using fixed size partitions.
- µC/OS II is a multitasking operating system
- Each task is an infinite loop and can be in any one of the following 5 states
  - Dormant, Ready, Running, Waiting, ISR
- services such as mailboxes, queues, and semaphores

Other RTOS

- Open source: Nut/OS, TRON Project
- Commercial: BeOS, ChorusOS, OS-9, OSEKtime, pSOS, RMX, RSX-11, RT-11, RTOS-UH

Real-Time POSIX

- Overview
  - POSIX *Portable Operating System Interface*
    - an API standard
    - POSIX 1003.1: defines the basic functions of a Unix os
  - POSIX thread and real-time extensions
    - POSIX 1003.1b: real-time extension
      - prioritized scheduling, enhanced signals, IPC primitives, high-resolution timer, memory locking, (a)synchronized I/O, contiguous files, etc
    - POSIX 1003.1c: thread extension
      - creation of threads and management of their execution
- Threads
  - basic units of concurrency
  - functions
    - create/initialize/destroy threads
    - manage thread resources
    - schedule executions of threads
    - read/set attributes of a thread
      - priority, scheduling policy, stack size and address, etc.

Real-Time POSIX

- Clocks and timers
  - time made visible to the application threads
  - the system may have more than one clock
  - functions
    - get/set time of a specified clock
    - create/set/cancel/destroy timers (up to 32 timers)
    - timer resolution: nanosecond
- Scheduling interface
  - support fixed priority scheduling with at least 32 priority levels
  - a thread may
    - (1) set and get its own priority and priorities of other threads
    - (2) choose among FIFO, round-robin and implementation-specific policies
  - in principle, it is possible to support the EDF or other dynamic priority algorithms, but with high implementation overhead
  - different threads within the same process may be scheduled according to different scheduling policies

Real-Time POSIX

- Synchronization
  - semaphores
    - simple; very low overhead
    - unable to control priority inversion
  - mutexes
    - support both priority inheritance and priority ceiling protocols
  - condition variables
    - allow a thread to lock a mutex depending on one or more conditions being true
    - mutexes are associated with a condition variable which defines the waited-for condition
- Interprocess communication
  - messages: prioritized
  - send/receive: nonblocking
  - receive notification: no check necessary for message arrivals
  - signals
    - primarily for event notification and software interrupt
    - at least eight application-defined signals
    - delivered in priority order
    - can carry data
    - queues blocked signals

Real-Time POSIX

- Shared memory and memory locking
  - a process can create a shared memory object
  - in case of virtual memory, applications can control memory residency of their code and data by locking the entire memory or specified range of address space
- File I/O
  - synchronized I/O
    - two levels of sync: data integrity, file integrity
  - asynchronous I/O
    - I/O concurrently with CPU processing