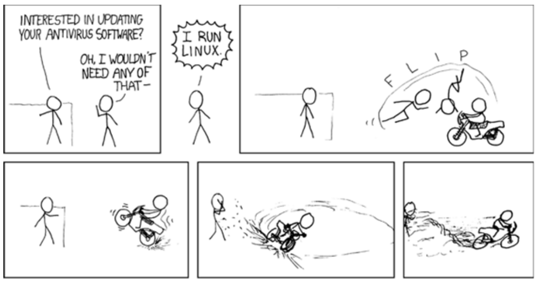
NTNU
Department of Engineering Cybernetics



TTK4147 Real-time Systems

07 Linux and Real-time Variants

25.09.2018

Today

- Linux
  - Brief History
  - Linux Kernel
  - Linux User Space
- Real-time and Linux
  - Real-time Patch
  - Real-time Microkernel
    - Xenomai
  - Slave Microcontroller
- Embedded Linux
  - Embedded development
  - Host - Target
  - Cross-toolchain
  - Build systems

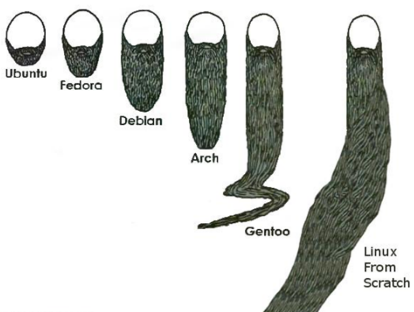
Brief History

- UNIX was first released in 1971 by Bell labs.
- MINIX was first released in 1987 by Andrew S. Tanenbaum, a free, open source UNIX-like operating system created for educational purposes.
- Linus Torvalds posted, in 1991, on an internet MINIX newsgroup that he was working on a "hobby project".
  - Nothing big and professional.
  - Would never run on anything else than 386, or use other hard drives than the one Linus himself had.
  - Later the same year version 0.01 of Linux was released.
- In 2011 the version 3.0 was released, marking the 20 year anniversary.
- The availability of free, open source software from the GNU project have been key to the success of Linux (some insist on the name GNU/Linux).

3 NTNU

Linux Distribution

- A Linux system consists of more than just the Linux kernel.
  - Core utilities (often GNU core utils)
  - Libraries
  - Graphic user interface
  - Programs
- Selecting the software a Linux system contains, and configure it correctly is NOT trivial.
- If you use a Linux distribution, somebody have done all (or most) of these decisions for you.
- There are currently about 780 Linux distributions on www.distrowatch.com

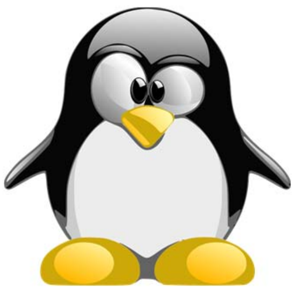


4 NTNU

The Great Linux World Map 2.0
(lots of Linux enthusiasts with spare time)

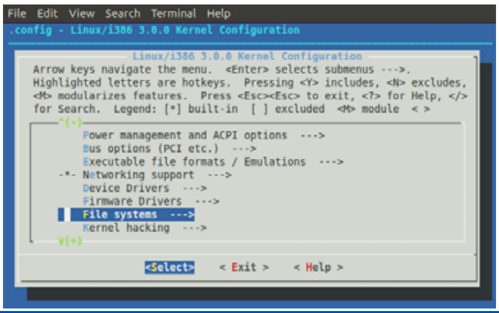
Linux Kernel

- Full equipped UNIX-like operating system.
- Run on a very large number of processor architecture (even if Torvalds only thought it would run on 386).
- Highly configurable.
- Linux uses the GNU General Public License, making it open source and free to use.
- It is a monolithic kernel, with loadable modules.



Building Linux Kernel

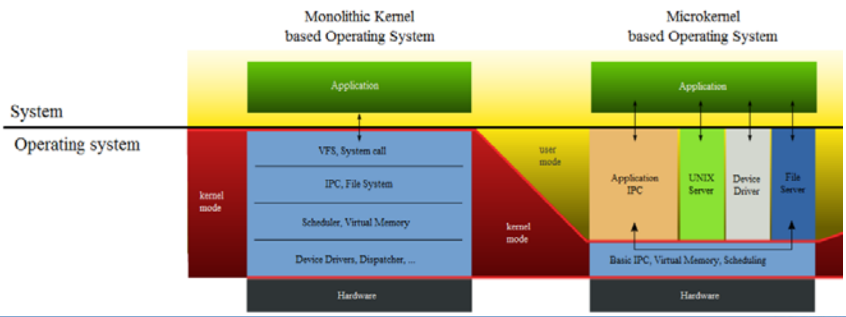
- The Linux kernel is a large piece of software (≈ 6 million lines of code), and has countless configuration options.
- Code available on kernel.org.
- The commands
  - “make menuconfig” allows to configure the kernel.
  - “make” builds the kernel.
  - “make modules” builds the kernel modules.



7 NTNU

Modular Monolithic Kernel

- Includes virtually all of the OS functionality in one large block of code that runs as a single process with a single address space
- All the functional components of the kernel have access to all of its internal data structures and routines
- Linux is structured as a collection of modules



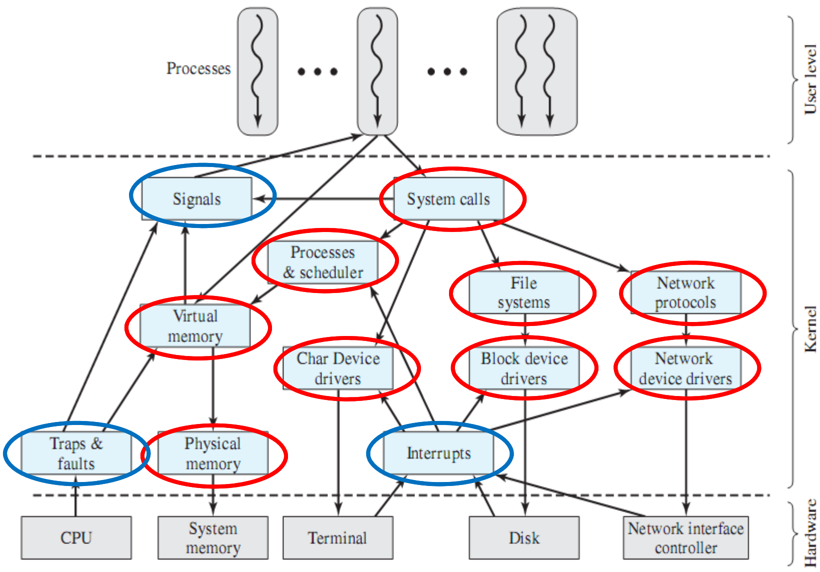
8 NTNU

Loadable Modules

- Relatively independent blocks
- A module is an object file whose code can be linked to and unlinked from the kernel at run-time
- A module is executed in kernel mode on behalf of the current process
- Have two important characteristics:
  - Dynamic linking
  - Stackable modules

    (Hierarchic organized. Individual modules serve as libraries when they are referenced by client modules higher up in the hierarchy, and as clients when they reference modules further down.)

Linux Kernel Components

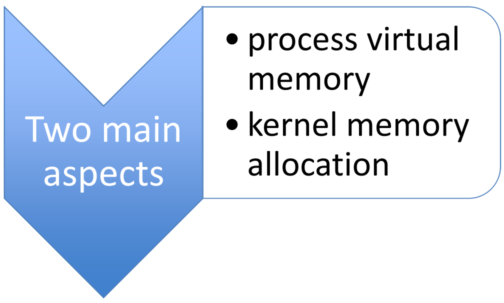


Concurrency in the Linux Kernel

- Four mechanisms are mentioned in Stallings:
  - Atomic operations
    Operations that are guarantied to perform without interruption or interference.
  - Spinlocks
    Works similarly as a mutex, but threads that want to get the mutex will «spin» until it received it, not sleep.
  - (Kernel) Semaphores
    Semaphores for kernel-space code.
  - Barriers
    Enforce the order in which instructions are executed
    (compiler and processor oriented!)

Linux memory management

- Shares many characteristics with UNIX
- Is quite complex



Linux virtual memory

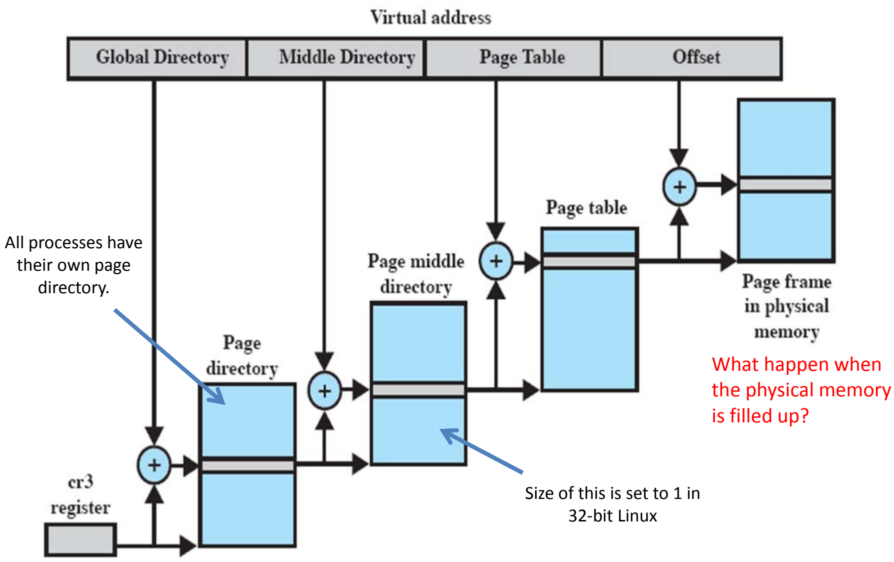
- Three level page-table structure:

Page directory
- process has a single page directory
- each entry points to one page of the page middle directory
- must be in main memory for an active process

Page middle directory
- may span multiple pages
- each entry points to one page in the page table

Page table
- may also span multiple pages
- each entry refers to one virtual page of the process

Linux Page Table



Linux Page Replacement

- The physical memory will sooner or later be filled.
- Page replacement in Linux is a form of «least frequently used policy»

- Each page in main memory has a counter, that is incremented each time it is accessed.
- Linux will periodically decrement these counters.

- The lower this counter is, the less frequently has it been used lately.
- A page with an counter of 0 is the best candidate for replacement.

- A form of least frequently used policy

Linux Memory Allocation

- *Kernel memory capability* manages physical main memory page frames
  - Primary function is to allocate and deallocate frames for particular uses

Possible owners of a frame include:
• user-space processes • dynamically allocated kernel data • static kernel code • page cache

- A buddy algorithm is used so that memory for the kernel can be allocated and deallocated in units of one or more pages

- Page allocator alone would be inefficient because the kernel requires small short-term memory chunks in odd sizes

- In addition, Slab allocation
  - Used by Linux to accommodate small chunks

Linux Scheduling

- An important role of an operating system kernel is to schedule the different processes running on the system.
- The three classes of scheduling are used by Linux:
  - SCHED_FIFO: First-in-first-out real-time threads.
  - SCHED_RR: Round-robin real-time threads.
  - SCHED_OTHER: Other, non-real-time threads.
- Within each class multiple priorities may be used.

Linux R-T Scheduling, FIFO vs RR

A	minimum
B	middle
C	middle
D	maximum

(a) Relative thread priorities

D → B → C → A →

(b) Flow with FIFO scheduling

D → B → C → B → C → A →

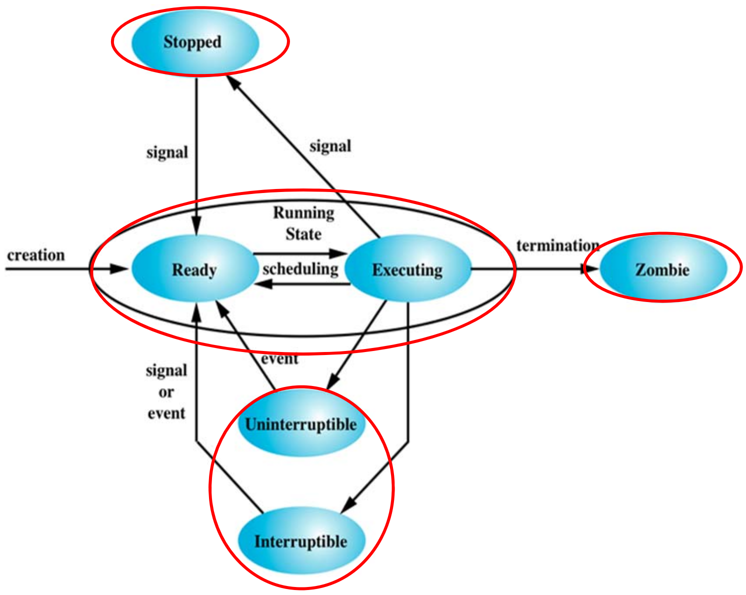
(c) Flow with RR scheduling

Linux Scheduling

- On paper the three classes of scheduling, each with individual priorities, sounds like what we are looking for in a real-time system, but:
  - The Linux kernel was not intended to be preemptive, and can't be fully preemptive regardless of kernel configuration.
  - Linux desktop distributions are typically running a large number of services and applications, and it can be difficult to know what can cause a delay in your program.

19 NTNU

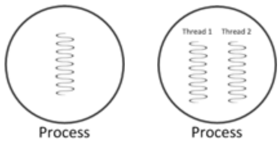
Linux Process / Thread Model



20 NTNU

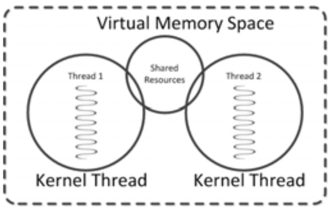
Linux Thread Model

BEFORE:
- Originally there were no threads in Linux, only processes.
- Threads could be implemented in user space inside the processes
- One blocked thread would block the whole process



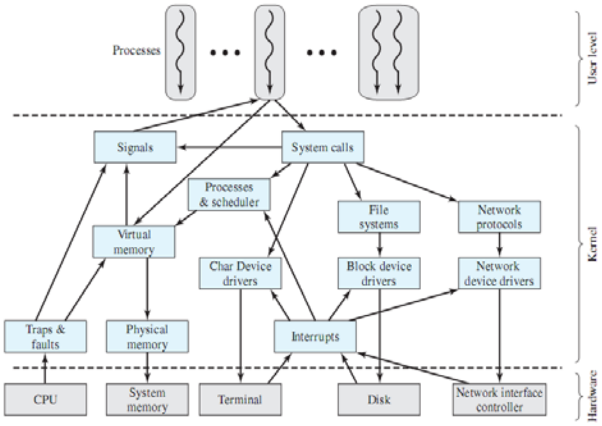
Linux Thread Model

NOW:
- Linux does not recognize a distinction between threads and processes
- Processes and threads are *almost* the same, both are running as separate "kernel threads".
- Threads share the same virtual memory.
- Context switch between threads are faster than between processes.
- The kernel schedule each kernel thread (both processes and threads) individually.



Linux User Space

- Where «normal» programs run.
- Use user-space library (normally GNU C library, but others can also be used).
- Must use *System Calls* to request kernel services.
- Kernel can notify user-space program with *Signals.*



23 NTNU

Some Linux Signals

SIGHUP	Terminal hangup	SIGCONT	Continue
SIGQUIT	Keyboard quit	SIGTSTP	Keyboard stop
SIGTRAP	Trace trap	SIGTTOU	Terminal write
SIGBUS	Bus error	SIGXCPU	CPU limit exceeded
SIGKILL	Kill signal	SIGVTALRM	Virtual alarm clock
SIGSEGV	Segmentation violation	SIGWINCH	Window size unchanged
SIGPIPT	Broken pipe	SIGPWR	Power failure
SIGTERM	Termination	SIGRTMIN	First real-time signal
SIGCHLD	Child status unchanged	SIGRTMAX	Last real-time signal

24 NTNU

Some Linux System Calls

Filesystem related	
close	Close a file descriptor.
link	Make a new name for a file.
open	Open and possibly create a file or device.
read	Read from file descriptor.
write	Write to file descriptor

Process related	
execve	Execute program.
exit	Terminate the calling process.
getpid	Get process identification.
setuid	Set user identity of the current process.
prtrace	Provides a means by which a parent process my observe and control the execution of another process, and examine and change its core image and registers.

Scheduling related	
sched_getparam	Sets the scheduling parameters associated with the scheduling policy for the process identified by pid.
sched_get_priority_max	Returns the maximum priority value that can be used with the scheduling algorithm identified by policy.
sched_setscheduler	Sets both the scheduling policy (e.g., FIFO) and the associated parameters for the process pid.
sched_rr_get_interval	Writes into the timespec structure pointed to by the parameter tp the round robin time quantum for the process pid.
sched_yield	A process can relinquish the processor voluntarily without blocking via this system call. The process will then be moved to the end of the queue for its static priority and a new process gets to run.

Interprocess Communication (IPC) related	
msgrcv	A message buffer structure is allocated to receive a message. The system call then reads a message from the message queue specified by msqid into the newly created message buffer.
semctl	Performs the control operation specified by cmd on the semaphore set semid.
semop	Performs operations on selected members of the semaphore set semid.
shmat	Attaches the shared memory segment identified by shmid to the data segment of the calling process.
shmctl	Allows the user to receive information on a shared memory segment, set the owner, group, and permissions of a shared memory segment, or destroy a segment.

Socket (networking) related	
bind	Assigns the local IP address and port for a socket. Returns 0 for success and –1 for error.
connect	Establishes a connection between the given socket and the remote socket associated with sockaddr.
gethostname	Returns local host name.
send	Send the bytes contained in buffer pointed to by *msg over the given socket.
setsockopt	Sets the options on a socket

Miscellaneous	
create_module	Attempts to create a loadable module entry and reserve the kernel memory that will be needed to hold the module.
fsync	Copies all in-core parts of a file to disk, and waits until the device reports that all parts are on stable storage.
query_module	Requests information related to loadable modules from the kernel.
time	Returns the time in seconds since January 1, 1970.
vhangup	Simulates a hangup on the current terminal. This call arranges for other users to have a "clean" tty at login time.

When User Space is not Enough

Reasons for develop in kernel space:

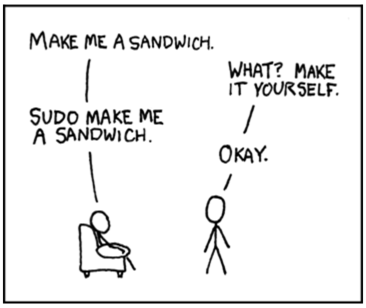
- User space programs can only access the services already provided by the kernel.
  - You need to access an unsupported device.
  - You need to access a special feature of the CPU of the kernel.
- Functionality can be implemented in kernel space to reduce the number of "context switches" which reduce performance.
  - Example: rotary encoder handling.

Disadvantages with kernel space:

- This probably depends on experience, but most developers will find it more difficult to program in kernel space than in user space.
- You can not use the normal user-space libraries when you program in kernel-space, but instead use Linux kernel header files.

Root vs Kernel Space

- Root is NOT the same as kernel space.
- Root is the highest level of privilege you can have in user-space.
- Many operations related to kernel-space requires root access.
  - Installing a module.
  - Mounting a hard-drive (although can be configured to auto-mount).
- Root can be a separate user that you must log in as.
- Or you can get root access with the *sudo* command.



Real-time and Linux

Linux is NOT a real-time system.

- Deterministic “hard real-time” would not be optimal for normal use.
- Kernel is not preemptive.
- Generally very good performance, but tend to get “outliers” with very high latencies.
- Latency will depend on system load.

“Soft” Real-time capabilities can be improved.

- Configure the kernel to be “as preemptive” as possible.
- Make sure the system has the smallest possible footprint.

How to get Hard(er) Real-time in Linux

Three different strategies:

1. Make the Linux kernel fully preemptive, with PREEMPT_RT, thus more predictable.
   - Effect is uncertain.

2. Add a small real-time co-kernel that runs «below» Linux.
   - RTAI (Real-time application interface)
   - Xenomai (which you will test in the exercises)

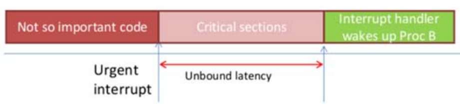
3. Outsource the hard real-time tasks to a slave microcontroller
   - A simple microcontroller without OS like AVR provide better control of latencies.
   - Interface between master CPU and slave microcontroller with UART, SPI, I2C etc.

1. PREEMPT-RT

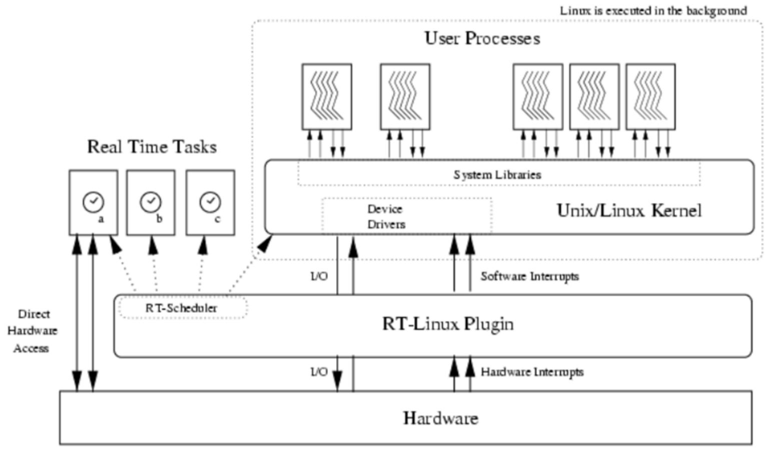
- The default Linux kernel comes in three levels of preemption:
  - No kernel preemption (server)
  - Voluntary kernel preemption (desktop)
  - Preemptive kernel (low-latency desktop)

- Regardless of the level, there are *critical sections* on kernel code that cannot be preempted.

- PREEMPT-RT patch is an effort to reduce the size of such *critical sections*, so that a high priority task can preempt a running task, even kernel task.



2. Generic Linux Co-kernel Concept



Xenomai

- Previous to version 3, Xenomai used a co-kernel or dual kernel approach.
- The relatively new xenomai version 3 have two different modes:
  - A dual kernel approach, where the Cobalt core runs beside the Linux kernel.
  - A single kernel approach, where the native Linux kernel, normally patched with PREEMPT-RT, forming the Mercury core.
- These two modes have the same programming interface (API), thus you can move applications between them.

Dual Kernel Approach

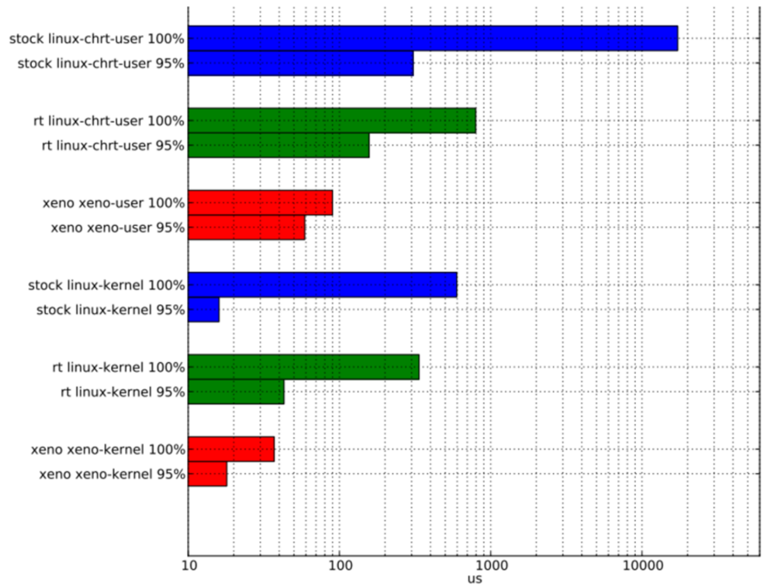
- The best performance, lowest latency.
- Challenges with operating in two contexts.
  - One Linux non real-time context.
  - One Xenomai/Cobalt real-time context.
- Switches between them will take time.
  - Also, if you switch to Linux context, you will loose real-time "privileges".
- Additional development will be necessary.
  - Not all libraries, drivers and programs in Linux can run in Xenomai/Cobalt context.
  - Dual kernel patches only work on some specific Linux kernel versions.

Real-time Device Model (RTDM)

- If a Xenomai program use a Linux driver, e.g. some hardware device, the driver will execute in Linux context, not Xenomai.
  - Not hard real-time.
  - Inefficient due to many switches between Linux and Xenomai.
- RTDM can be used to develop full Xenomai drivers.
  - Additional work.
  - Specialized knowledge required.
  - Can be very difficult.
  - Documentation is lacking.

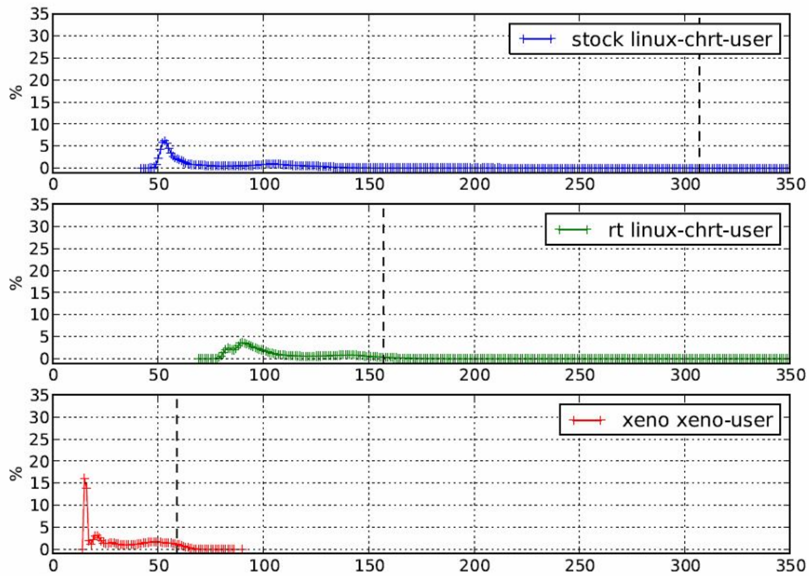
  → Not mature

Comparison between solutions



35 NTNU

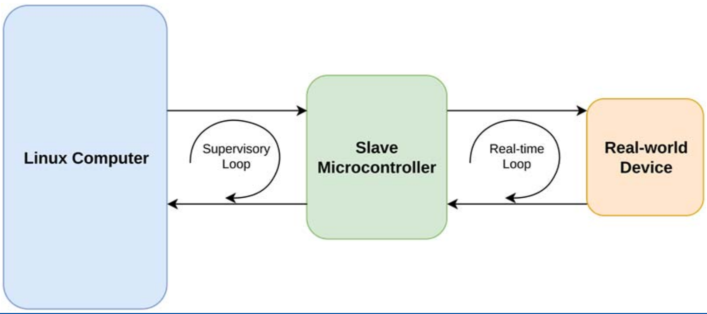
Comparison between solutions



36 NTNU

3. Slave Microcontroller

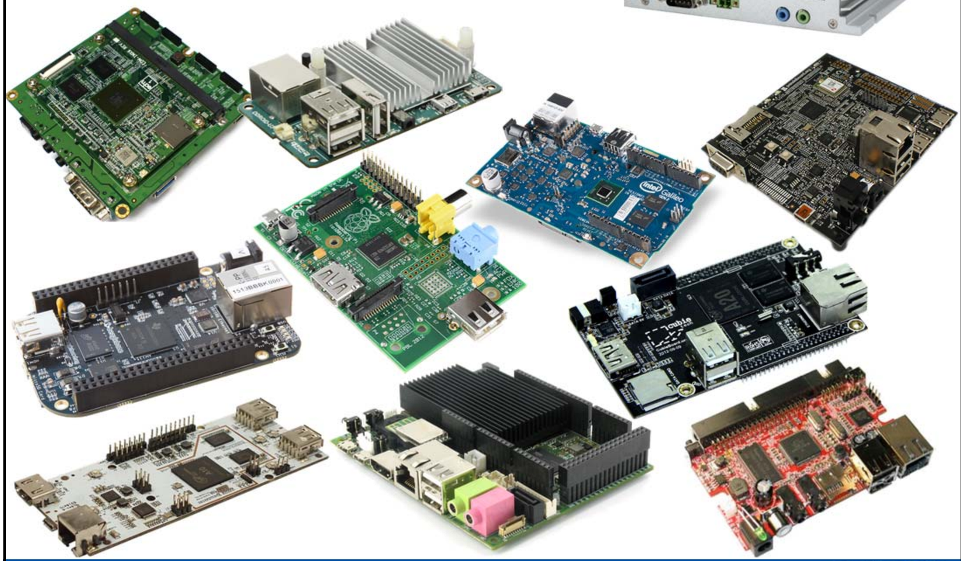
- Outsourcing some of the real-time tasks to a specialized microcontroller.
  - The Linux computer act as a supervisor.
- Suitable (and usual) for systems that have a small, separate real-time component.



Embedded Linux

- Advantages:
  - Customizable and configurable.
  - Runs on a large number of architectures.
  - Huge, active user community.
  - Free.
- Disadvantages:
  - Not hard real-time by default.
  - There are too many alternatives.
  - Difficult
  - Versions
  - Software packages
  - Could be less expensive to buy a working system than to spend a long time configuring a free Linux system.
  - Hardware capability issues (but better than many OSes).

Examples of Embedded Linux delivered / ported to COTS HW



Android

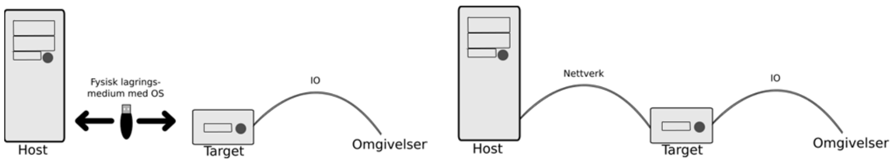
- An embedded environment that runs on Linux.
- Will be covered in a separate lecture.

What about C/C++?

- Real-time programs are *almost* always developed with C/C++.
  - Direct interaction with hardware.
  - No virtual machine run-time (like Python/Java).
- How to develop C/C++ programs for embedded systems?
  - Use the host-target model(?)
  - Where (host-target) should the different tasks be performed:
    - Writing the code
    - Building the code
    - Debugging

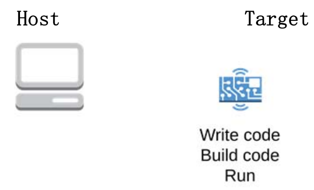
Embedded Development

- In embedded development there is usually one or more targets and one host computer.
  - Host computer is the general purpose computer you use to program and compile the program on.
  - Target computer is the embedded computer that runs the code.
- We compile on the host, because it is normally a more powerful computer.
  - Will often require cross-compiling.
  - Must move compiled software from host to target



Only use Target

- Do everything on the hardware the system will run on.
  - Often less powerful hardware than a desktop or laptop.
  - Often limited development tools.
  - But easy to set up

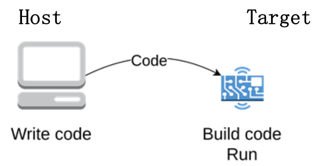


- It takes a Beaglebone Black approximately 12 hours to build its own Linux Kernel

- Not exactly the best development platform...

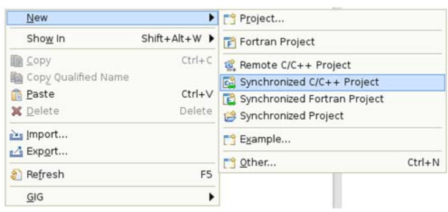
Develop on Host, Build on Target

- A desktop or laptop will typically have better development tools.
  - Develop code on host.
  - Transfer code to target.
  - Build and run on target.



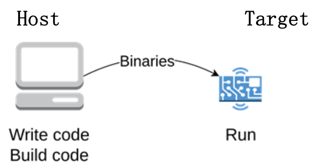
- Still have the problem that target computer might be slow.
  - Might even not have the necessary build tools.

- Eclipse (IDE) functionality for this type of setup, which is called "Synchronized C/C++ project".
  - Available as a downloadable plugin.
  - Syncs code on host and target.
  - Build on target.



Develop and Build on Host, Transfer to Target

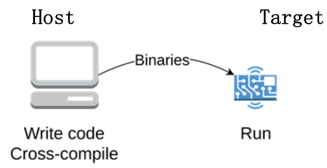
- Develop and build a program as if should run on host.
  - Transfer binaries built on host to target.
  - Program should work the same way on target as it did on host.



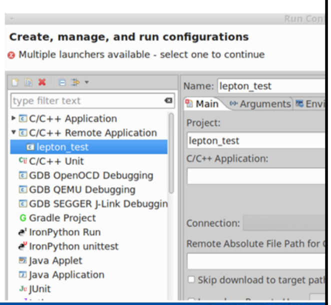
- Will NOT work if host and targets are too different.
  - The same binaries will not work on different CPU platforms.
  - The same binaries will not work is host and targets have different shared libraries.
- Will be difficult to develop and test functionality that only the target has.
  - Connection with other parts of a larger system.
  - I/O devices.

Cross-compile

- A cross-compiler is a compiler that runs on one computer (host) and make binaries for another (target).
- Very useful for embedded development.

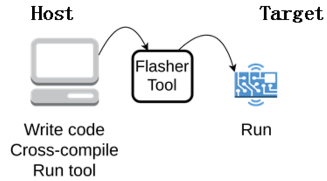


- Challenging to set up, but is the usually the best solution when it works.
  - The host need a specialized tool (the cross compiler).
  - The host need a copy of the targets libraries.
  - Need some method of transferring the binary and run it remotely.
- Eclipse has support for this as well, called "C/C++ Remote Application".
  - Have to set up cross compilation.
  - Special "run configuration" that will transfer binary to target and run it.
  - Can also be used for debugging.



Cross-compile and Flash

- For hardware that does not have removable storage, network etc.
  - Use an external tool or debugger, often to *flash* some content to some internal flash memory.
  - The tool can be external (JTAG used for Atmel micro-controllers)
  - Or on board (Bootloader on Arduino lets you upload programs without external device.

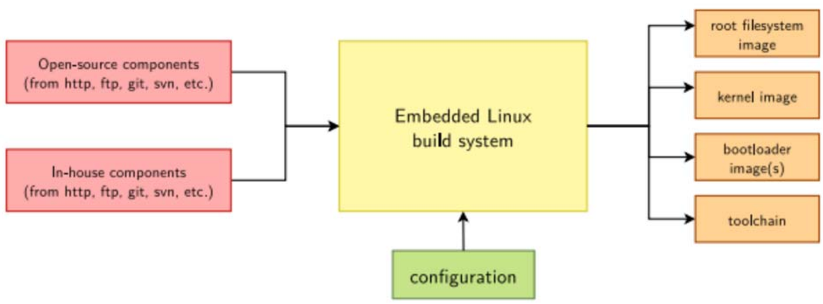


- Examples of this:
  - Different types of Atmel micro-controllers.
  - ARM cortex-m0 and m3 micro-controllers.

47 NTNU

Embedded Linux Build Systems

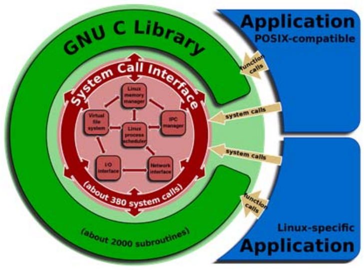
- Compromise between using a *binary distribution* (ubuntu, debian, etc) and having to make everything from scratch.
  - Almost full control over what is included in your system.
  - Built from source.
  - But still relatively easy to make.
- Based on configuration files and recipes.
  - Easy to reproduce a specific solution.



48 NTNU

uClibc

- Most Linux distribution use the GNU library C (glibc).
  - This is a C standard library, which implement basic functions.
- Glibc is often considered too big in embedded situations.
  - Alternative C standard libraries can be used.



- uClibc is an often used alternative for embedded Linux.
  - Lightweight, means less storage space is needed.
  - Configurable, means that you can drop functionality that is not used.
- This sounds great, why don't everybody use uClibc?
  - Not ABI compability between versions.
  - Not everything in glibc is supported in uClibc.

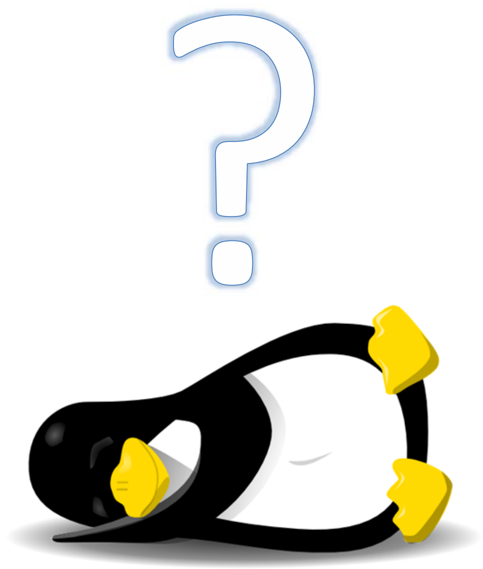
ABI = Application Binary Interface an interface between two binary program modules

Busybox

- All Linux systems (and other UNIX-like systems) needs a set of basic applications, typically found in GNU coreutils.
  - ls - list files
  - cp - copy files
  - ifconfig - manage network devices
  - grep - search for file content
  - etc.
- Busybox - The Swiss Army Knife of Embedded Linux.
  - Implements a collection of the common tools for embedded Linux.
  - Configurable, as uClibc, so you only need to build the tools you need.
- One single binary file
  - All tools are within the same executable, binary file /bin/busybox.
  - E.g. you execute «ls» by running «/bin/busybox ls».

Embedded Linux vs QNX/VxWorks

- This lecture has given a brief overview of the different ways to set up an embedded Linux system and development environment.
- All of this is provided to you if you use a commercial system as QNX or VxWorks.
  - You will get a cross-toolchain that build QNX/VxWorks programs on your desktop computer.
  - You will get tools to configure and make a file system or image file.
- The point is, the *freeness* of Linux comes at a cost.
  - Need to make a compromise between cost of buying a system and cost of setting up a system.



How does the CONFIG_PREEMPT_RT patch work?

The RT-Preempt patch converts Linux into a fully preemptible kernel. The magic is done with:

- Making in-kernel locking-primitives (using spinlocks) preemptible though reimplementation with rtmutexes.
- Critical sections protected by i.e. spinlock_t and rwlock_t are now preemptible. The creation of non-preemptible sections (in kernel) is still possible with raw_spinlock_t (same APIs like spinlock_t).
- Implementing priority inheritance for in-kernel spinlocks and semaphores. For more information on priority inversion and priority inheritance please consult Introduction to Priority Inversion.
- Converting interrupt handlers into preemptible kernel threads: The RT-Preempt patch treats soft interrupt handlers in kernel thread context, which is represented by a task_struct like a common user space process. However it is also possible to register an IRQ in kernel context.
- Converting the old Linux timer API into separate infrastructures for high resolution kernel timers plus one for timeouts, leading to user space POSIX timers with high resolution.

53 NTNU