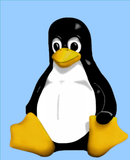
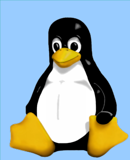


Why use embedded linux?

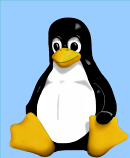
+ Rapid Application Development
+ Debugged Control Structures
+ Code Reuse
+ Hardware Abstraction
+ Application Partitioning
+ Scheduling
+ Priorities
+ Preemption
+ Application and Hardware Interaction

- Memory consumption
- Boot Time
- Interrupt Latency
- Robustness



Motivation of using a Kernel

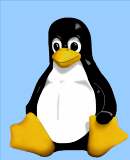
All applications require control code as support for the algorithms that are often thought of as the “real” program. The algorithms require data to be moved to and/or from peripherals, and many algorithms consist of more than one functional block. For some systems, this control code may be as simple as a “superloop” blindly processing data that arrives at a constant rate. However, as processors become more powerful, considerably more sophisticated control may be needed to realize the processor’s potential, to allow the processor to absorb the required functionality of previously supported chips, and to allow a single processor to do the work of many. The following sections provide an overview of some of the benefits of using a kernel on a processor.



Rapid Application Development

The use of uClinux kernel allows rapid development of applications compared to creating all of the control code required by hand. An application or algorithm can be created and debugged on an x86 PC using powerful desktop debugging tools, and using standard programming interface to device drivers. Moving this code base to uClinux kernel running on the Blackfin is trivial because the device driver model is exactly the same. Opening an audio device on the x86 Desktop is done in exactly the same was as on a uClinux/Blackfin system. This allows you to concentrate on the algorithms and the desired control flow rather than on the implementation details. The uClinux kernel and applications supports the use of C, C++, and assembly language. You are encouraged to develop code that is highly readable and maintainable, yet retaining the option of hand optimizing if necessary.

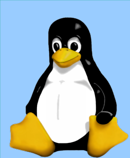
iP THiNKiNG
Embedded platforms



Debugged Control Structures

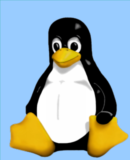
Debugging a traditional hand coded application can be laborious because development tools (compiler, assembler, and linker among others) are not aware of the architecture of the target application and the flow of control that results. Debugging complex applications is much easier when instantaneous snapshots of the system state and statistical run time data are clearly presented by the tools. To help offset the difficulties in debugging software, the uClinux kernel has been tested with the same tests that many Desktop distributions use before releasing a Linux kernel. This ensure that the kernel is as bug free as possible.

iP THINKING
Embedded platforms



Code Reuse

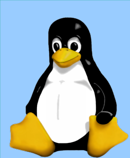
Many programmers begin a new project by writing the infrastructure portions that transfer data to, from, and between algorithms. This necessary control logic usually is created from scratch by each design team and infrequently reused on subsequent projects. The uClinux kernel provides much of this functionality in a standard, portable and reusable manner. Furthermore, the kernel and its tight integration with the GNU development and debug tools are designed to promote good coding practice and organization by partitioning large applications into maintainable and comprehensible blocks. By isolating the functionality of subsystems, the kernel helps to prevent the morass all too commonly found in systems programming. The kernel is designed specifically to take advantage of commonality in user applications and to encourage code reuse. Each thread of execution is created from a user-defined template, either at boot time or dynamically by another thread. Multiple threads can be created from the same template, but the state associated with each created instance of the thread remains unique. Each thread template represents a complete encapsulation of an algorithm that is unaware of other threads in the system unless it has a direct dependency.



Hardware Abstraction

In addition to a structured model for algorithms, the uClinux kernel provides a hardware abstraction layer. Presented programming interfaces allow you to write most of the application in a platform independent, high-level language (C or C++). The uClinux Application Programming Interface (API) is identical for all processors which support Linux or uClinux, allowing code to be easily ported to a different processor core. When porting an application to a new platform, programmers must only address the areas necessarily specific to a particular processor - normally device drivers. The uClinux architecture identifies a crisp boundary around these subsystems and supports the traditionally difficult development with a clear programming framework and code generation. Common devices can use the same driver interface (for example a serial port driver may be specific for a certain hardware, but the application ↔ serial port driver interface should be exactly the same, providing a well defined hardware abstraction, and making application development faster).

iP THINKING
Embedded platforms

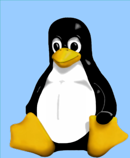


Partitioning an Application

A uClinux application or thread is an encapsulation of an algorithm and its associated data. When beginning a new project, use this notion of a application or thread to leverage the kernel architecture and to reduce the complexity of your system. Since many algorithms may be thought of as being composed of “sub algorithm” building blocks, an application can be partitioned into smaller functional units that can be individually coded and tested. These building blocks then become reusable components in more robust and scalable systems.

You define the behavior of uClinux applications by creating the application. Many application or threads of the same type can be created, but for each thread type, only one copy of the code is linked into the executable code. Each application or thread has its own private set of variables defined for the thread type, its own stack, and its own C run-time context.

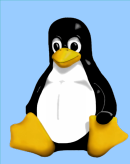
When partitioning an application into threads, identify portions of your design in which a similar algorithm is applied to multiple sets of data. These are, in general, good candidates for thread types. When data is present in the system in sequential blocks, only one instance of the thread type is required. If the same operation is performed on separate sets of data simultaneously, multiple threads of the same type can coexist and be scheduled for prioritized execution (based on when the results are needed).



Scheduling

The uClinux kernel can be a preemptive multitasking kernel. Each application or thread begins execution at its entry point. Then, it either runs to completion or performs its primary function repeatedly in an infinite loop. It is the role of the scheduler to preempt execution of an application or thread and to resume its execution when appropriate. Each application or thread is given a priority to assist the scheduler in determining precedence.

The scheduler gives processor time to the thread with the highest priority that is in the ready state. A thread is in the ready state when it is not waiting for any system resources it has requested.

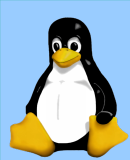


Priorities

Each application or thread is assigned a dynamically modifiable priority. An application is limited to forty priority levels. However, the number of threads at each priority is limited, in practice, only by system memory. Priority level one is the highest priority, and priority thirty is the lowest. The system maintains an idle thread that is set to a priority lower than that of the lowest user thread.

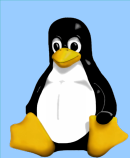
Assigning priorities is one of the most difficult tasks of designing a real time preemptive system. Although there has been research in the area of rigorous algorithms for assigning priorities based on deadlines (for example, rate monotonic scheduling), most systems are designed by considering the interrupts and signals triggering the execution, while balancing the deadlines imposed by the system's input and output streams.

iP THiNKiNG
Embedded platforms



Preemption

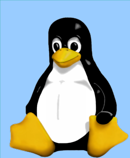
A running thread continues execution unless it requests a system resource using a kernel system call. When a thread requests a signal (semaphore, event, device flag, or message) and the signal is available, the thread resumes execution. If the signal is not available, the thread is removed from the ready queue - the thread is blocked. The kernel does not perform a context switch as long as the running thread maintains the highest priority in the ready queue, even if the thread frees a resource and enables other threads to move to the ready queue at the same or lower priority. A thread can also be interrupted. When an interrupt occurs, the kernel yields to the hardware interrupt controller. When the ISR completes, the highest priority thread resumes execution.



Application and Hardware Interaction

Applications should have minimal knowledge of hardware; rather, they should use device drivers for hardware control. A application can control and interact with a device in a portable and hardware abstracted manner through a standard set of APIs.

The uClinux Interrupt Service Routine framework encourages you to remove specific knowledge of hardware from the algorithms encapsulated in threads. Interrupts relay information to threads through signals to device drivers or directly to threads. Using signals to connect hardware to the algorithms allows the kernel to schedule threads based on asynchronous events. The uClinux run-time environment can be thought of as a bridge between two domains, the thread domain and the interrupt domain. The interrupt domain services the hardware with minimal knowledge of the algorithms, and the thread domain is abstracted from the details of the hardware. Device drivers and signals bridge the two domains.



Downsides of using a kernel

Memory consumption - to have a usable Linux system - your system should have at least 4-8Meg of SDRAM, and at least 2Meg of Flash.

Boot Time - the kernel is fast, but sometimes not fast enough - expect to have a 2-3 second boot time.

Interrupt Latency - On occasions, a uClinux device driver, or even the kernel will disable interrupts. Some critical kernel operations can not be interrupted, and it is unfortunate, but interrupts must be turned off for a bit. Care has been taken to keep critical regions as short as possible as they cause increased and variable interrupt latency.

Robustness - although a kernel has gone through lots of testing, and many people are using it - it is always possible that there are some undiscovered issues. Only you can test it in the configuration that you will ship it.