Operating Systems - Introduction

Lecture on

Operating Systems

An Introduction

Walter Kriha

Goals for this class

• Understand the structure and workings of an operating system (OS) – in other words: resource management!

• Learn how to write C-language programs which use the features provided by the OS (System Programming)

• Learn how to use and administrate the Linux OS

• Learn how to monitor your application, its environment and the hardware

• Understand the limits of hardware and how to design fast and reliable applications running on top of the OS

This class will NOT turn you into a kernel guru. After the class you should have a much better understanding of the system software that you a using indirectly through different applications.

The Future: How should it be?

Computing will be transformed. It's not just that our problems are big, they are big and *obvious.* It's not just that the solutions are simple, they are simple and right under our noses. It's not just that hardware is more advanced than software; the last big operating-systems breakthrough was the Macintosh, sixteen *years* ago, and today's hottest item is Linux, which is a version of Unix, which was new in 1976. Users react to the hard truth that commerical software applications tend to be badly-designed, badly-made, incomprehensible and obsolete by blaming *themselves* ("Computers for Morons," "Operating Systems for Livestock"), and meanwhile, money surges through our communal imagination like beer from burst barrels. Billions. Naturally the atmosphere is a little strange; change is coming, soon.

from David Gelernter, the second coming – a manifesto. http://www.edge.org/documents/archive/edge70.html . We will come back again and again to compare what we have learned with Gelernters ideas. He may be right after all – look at what is coming with all those PDA‘s, wireless computing etc.

Why learn about Operating Systems?

1. Operating Systems used to be complicated and advanced pieces of software. They had to deal with concurrency, resource allocation and performance and security. Applications were considered to be simpler because they could rely for the critical functions on operating system features. But in the last 10 years we have seen a lot of function move from OS to applications
2. Modern applications no longer run as a single process. They are multi-process engines using lots of internal threading. They use shared memory and large scale storage areas – resources they have to maintain.
3. And if the application programmer moves to new fields like embedded control application then an understanding of those systems and how they are different to comfortable big operating systems is necessary.

The result is that application programmers now need to understand „system thinking“, e.g. how to program concurrent processes using monitors and semaphors or when to use other forms of concurrency.

General System Building Knowledge

- Concurrency: how to avoid data corruption through concurrent processes and how to achieve maximum speed or throughput
- Resource management: how to manage large resources effectively. How to avoid allocation problems. How to keep resources consistens across the lifecycle.
- Architectural know how: layers and abstractions
- Design for scalability across users and machines
- Learn to fear and respect nonfunctional requirements (size, time, independence, energy consumption, quotas)
- Learn to use caching to improve performance while still keeping data consistent
- Understand trade-offs in designs and algorithms
- Get an unterstanding of the „physical side“ of programs and systems

Non-Goals

- Writing device drivers. We will look at the design patterns behind device drivers but writing one is reserved for advanced classes

- This is not a kernel algorithm class. We will look at resource management strategies but we don't implement kernel code yet.

- This is (no longer) a C-language class. We will focus on the runtime system aspects only. We will use C down to the assembly code level but becoming a guru will take more time.

At the end of this class you should know how to how to use OS tools to profile and monitor the programs and in general be able to design an application with a resonable „guestimate“ on where performance problems could be and how they could be avoided.

Schedule

Lecture:

1. OS Introduction
2. Linux Architecture
3. File Management
4. Memory Management and Parallel Programming
5. Processes and Concurrency
6. C runtime system and assembler
7. Unix System Programming
8. Computer Organization
9. Virtual Machines
10. Monitoring

Exercises:

- Linux Certification I, an introduction to self-learning
- Use debugger, tracer, logger
- System Programming Examples
- Monitoring tools and concepts

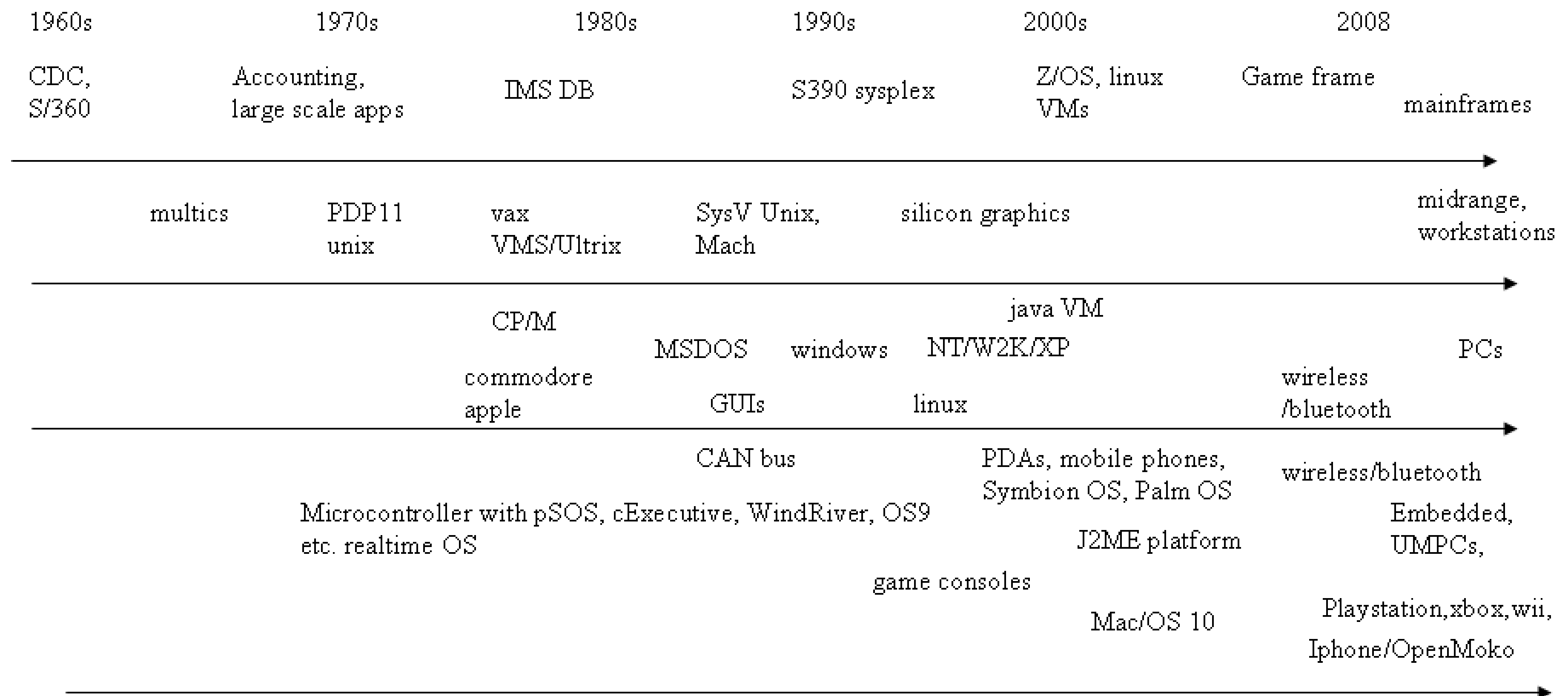
This is a very tight schedule and some reading is constantly expected.

Show Cases

1. How does bootstrapping an OS work?
2. How does the interface between an application and an OS work?
3. How many layers and abstractions are needed for convenient management of resources like files or memory?
4. How to execute a program (through all layers into the kernel)
5. How to trace and track a program with the help of the OS
6. What is happening in the kernel of an OS during a system call?
7. How to extend a system (e.g. with new hardware)?
8. How to manage resources securely and efficiently

After this class you should be able to diagnose problems (OS, environment) by using the proper analytical tools. You should also have a much better understanding of how computing works.

A short history of operating systems



Fact is that most ideas in computing are rather old. A good idea needs the right hardware and users to blossom.

Trends (1)

1. From single computer OS to internet-worked systems: Microsoft vs. Google
2. Mobile computing platforms with desktop capabilities abound: Iphone SDK, OpenMoko, Symbian OS, integrated into enterprise infrastructures.
3. Embedded control applications form the „ambient intelligence“ cloud, creating a huge demand for software.
4. The PC becomes a CC (company computer) losing rapidly its importance in the private area.
5. RAS (reliabiltiy,availability, security) are getting more important: mainframes are high-availability clusters running Linux VMs. Perfect workload management.
6. Workstations are Risc computers or PCs running some Unix or PCs with high end graphics. Are they getting replaced by high-end game consoles running cell chips?
7. From single-owner to single-user to multi-user operating systems, mostly forced by security problems since PCs got networked/Internet.

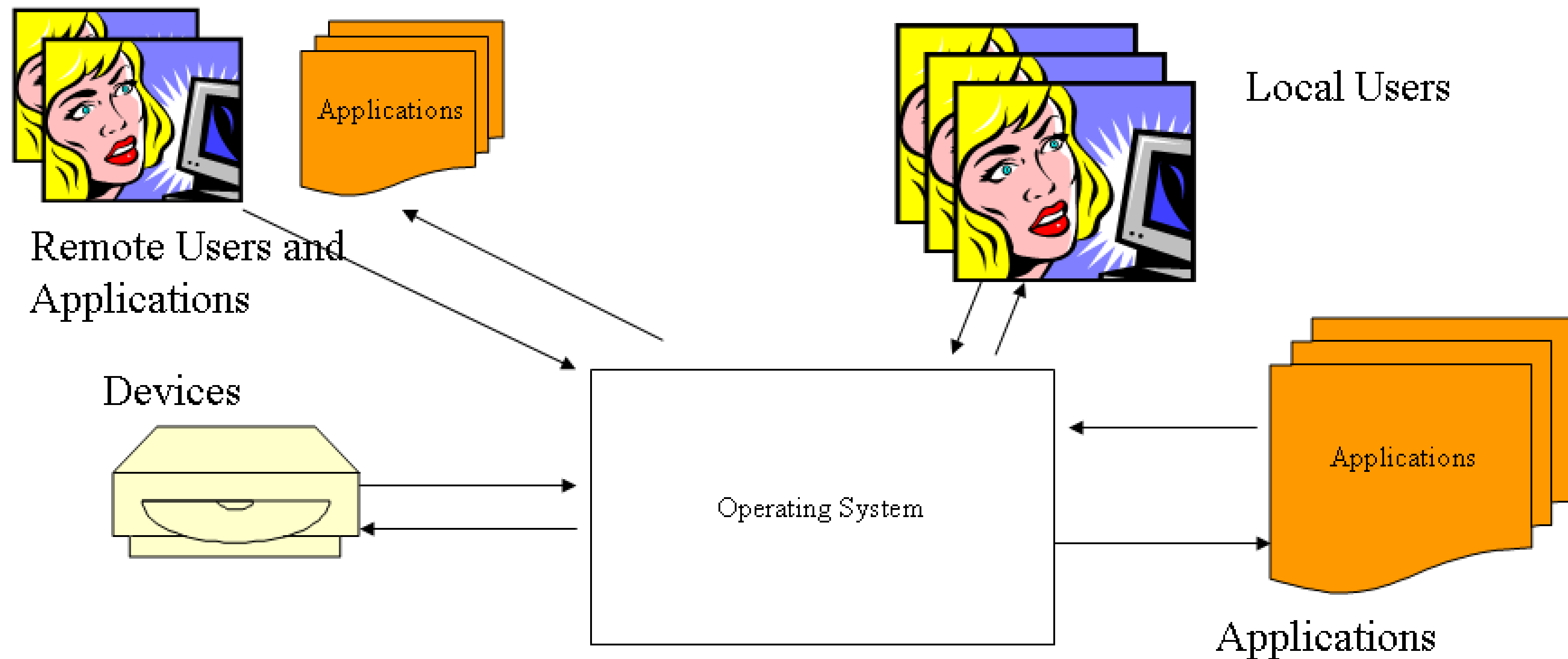
Some of those trends have started long ago and some are quite new like the Linux VMs on mainframes. Some things don‘t change: users want fast and reliable programs and services. Today mobility and interconnection is key.

Trends (2)

1. Kernel threads replace processes. Multi-Core CPUs will have 80+ cores, creating the need for new programming models.
2. Virtual Machines dominate Operating Systems (Java, .NET)
3. Databases for transactional software still hot.
4. Filesystems get atomic updates and become stable. They are implemented using database technology.
5. File names and hierarchies are replaced by attributes and better search engines
6. New security concepts a MUST for embedded control and ambient intelligence.

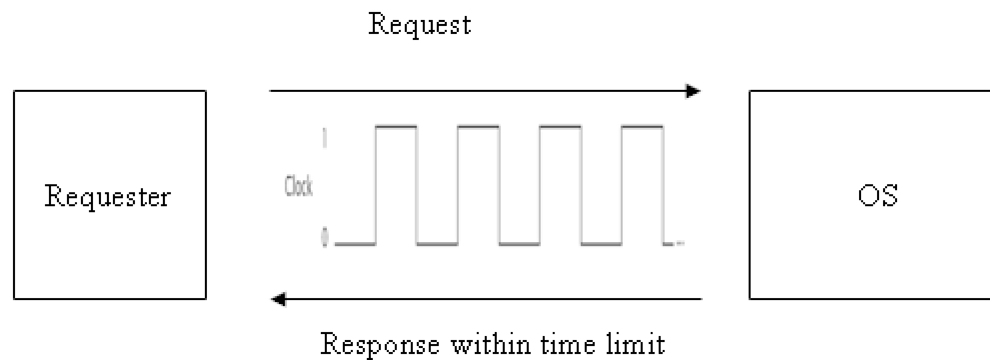
The demand for software will be very high due to „ubiquitous computing“ – the change of our world to a completely computerized one.

What is an operating system?

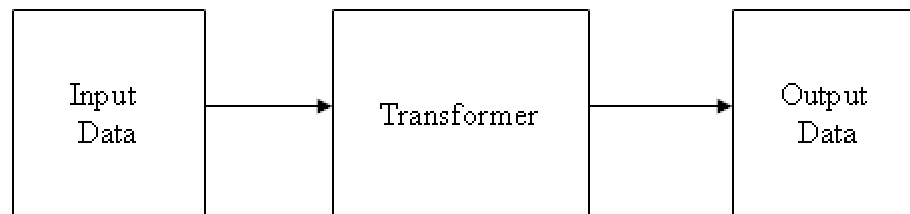


An Operating System is an INTERACTIVE SYSTEM which balances events and requests coming from different sources. It has to keep internal state of applications and of itself consistent while at the same time making sure that users still get responses. An Operating System GENERALIZES over many different use cases, sometimes making necessary compromises e.g. with respect to realtime requirements. Operating Systems are unable to make the same guarantees as reactive or transformative systems.

Other Systems



A reactive systems behavior is completely driven by requests. Responses have to happen within predefined clock cycles. The memory subsystem is an example of this.



A transformative system takes some input and performs transformations on it, thereby producing an output. A compiler is an example of this type.

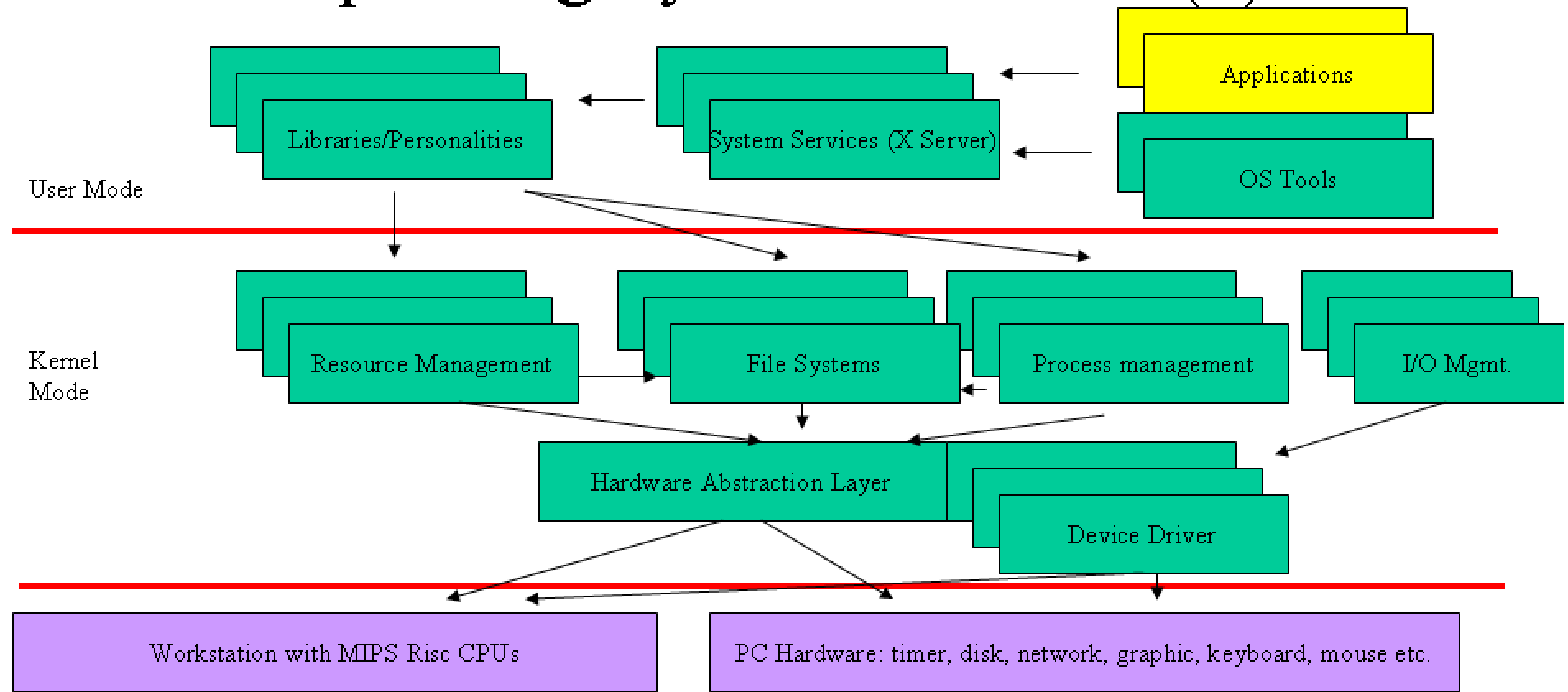
Both types of systems, reactive and transformative, are much less critical with respect to response time and reliable behavior. Unfortunately they are not fit to do an Operating Systems job.

Operating System Functions

- Encapsulate hardware details to keep applications independent from hardware and hardware changes.
- Allow applications easy and abstract access to hardware and services through a uniform interface
- Provide services every application will need like authentication
- Provide resource management functions: allocation, use and control, accounting, garbage collection of resources
- Protect computing resources, applications and users from destruction and each other
- Support inter-process communication and networking

An OS usually provides generic functions. Functions that a lot of applications will need. The OS does this by offering a special interface called the system call interface to applications. An OS designer must judge whether a function is a) really a kernel function which cannot be implemented otherwise and b) whether the function is general enough to be useful for many applications.

Operating System Structure (1)

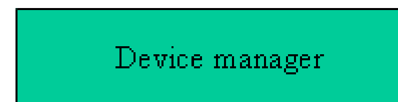


At first glance an Operating System follows the LAYER architectural design pattern. But there is also a lot of inter-component re-use. Please note that an Operating System has kernel mode and user mode parts. E.g. Tools like file manager belong to the OS even though technically they are user mode applications. Most everything within an operating system needs to be extensible or replaceable. New devices need new device drivers, not a new operating system releases.

Operating System Structure (2)

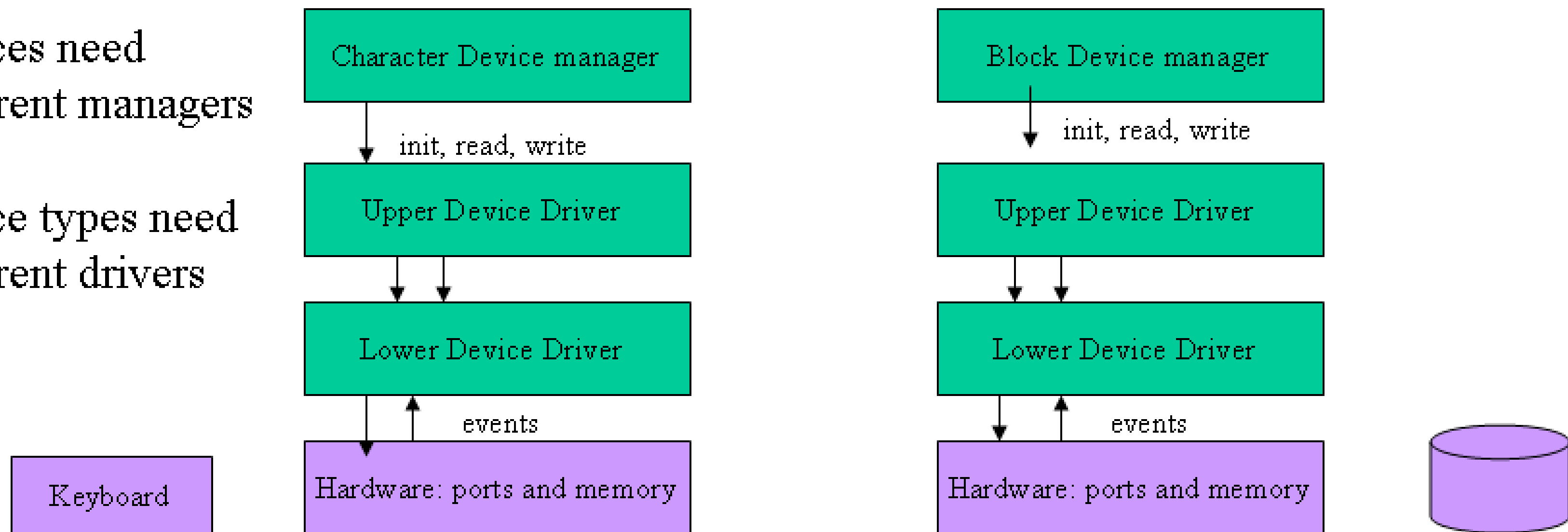
register new device drivers

initialize drivers at boottime or dynamically



devices need different managers

device types need different drivers



Operating Systems are software FRAMEWORKS. They use interfaces to abstract differences in implementation or funtion. A typical example is the device driver interface of an OS which allows new devices to be supported after the OS has been shipped. A new device driver which conforms to the interfaces (template/hook pattern) defined by the OS can be installed (statically or dynamically). The OS will call the driver functions at the proper times. E.g. at boottime the probe() function of each driver is called to see if a certain hardware is present (in case there is no automatic configuration information available)

CPU Protection Levels

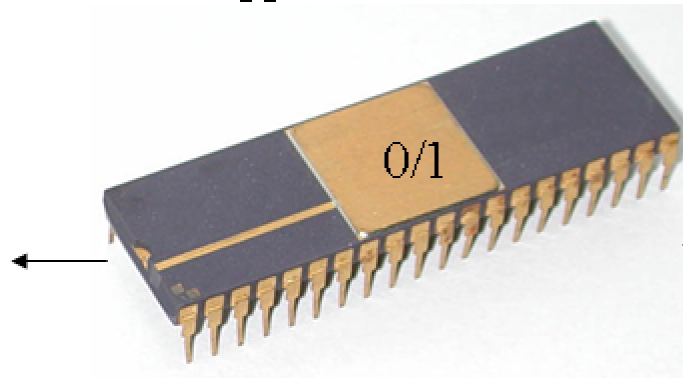
State of protected mode bit:

1 = protected/kernel mode

0 = application/user mode

Sensing operations (I/O)

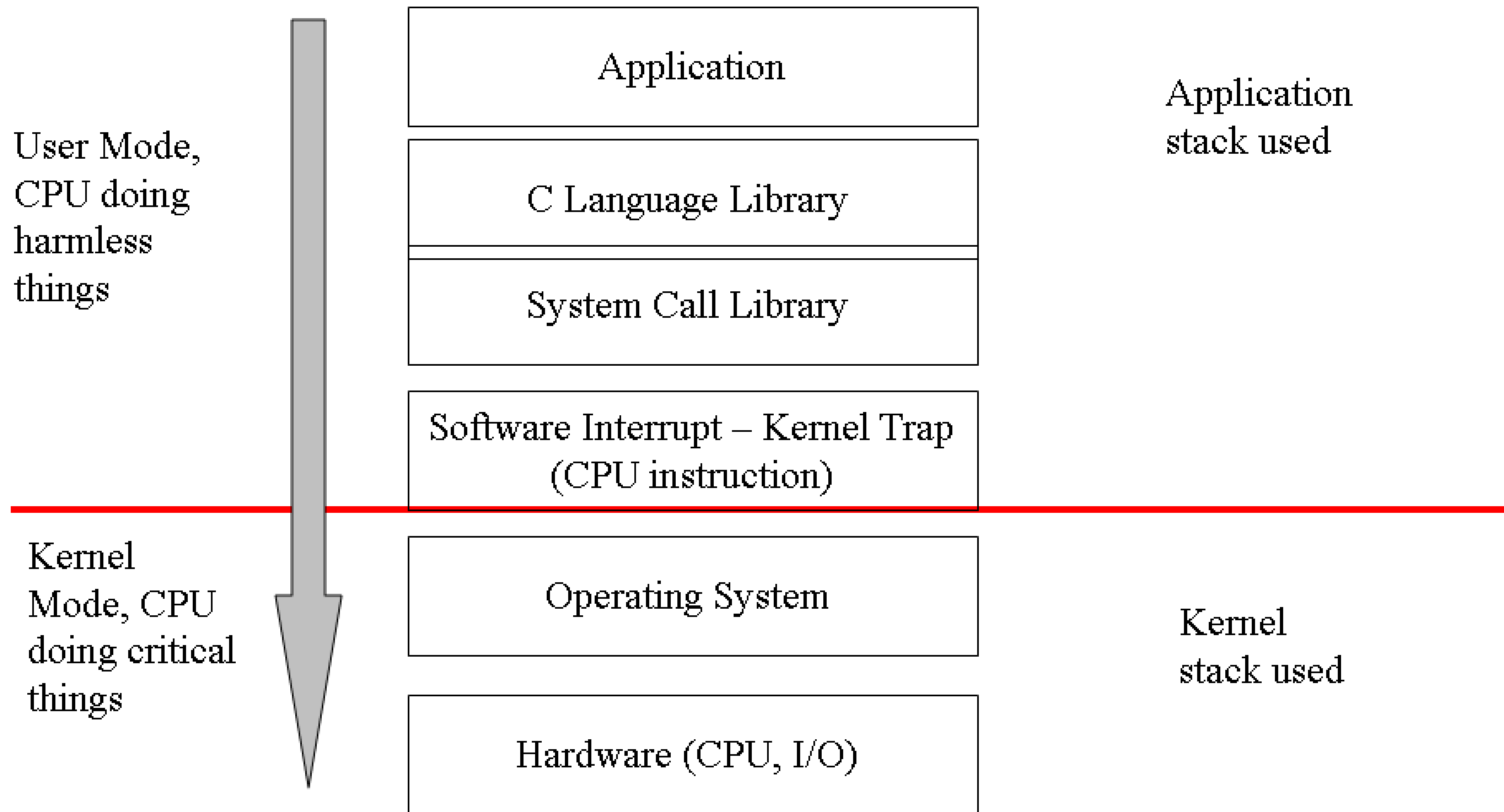
Control operations (halt, memory mgmt.)



Regular compute operations (add, mul)

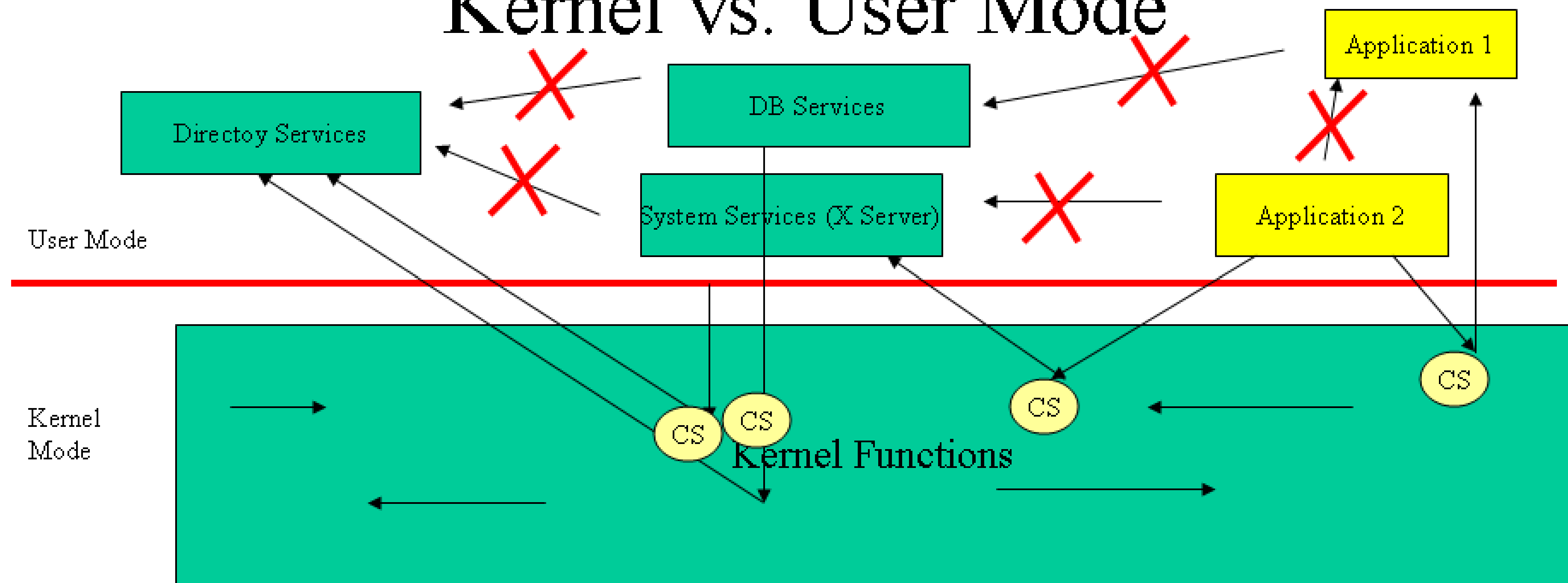
Most CPUs offer a simple protetion scheme. Dangerous operations (sensing, control) are only allowed when the CPU has been put in kernel mode (protection bit is set). Applications can NOT change the state of the CPU arbitrarily. They MUST use certain controlled gates (software interrupts) to change the mode. From then on, operating system code runs!

Switching to Kernel Mode



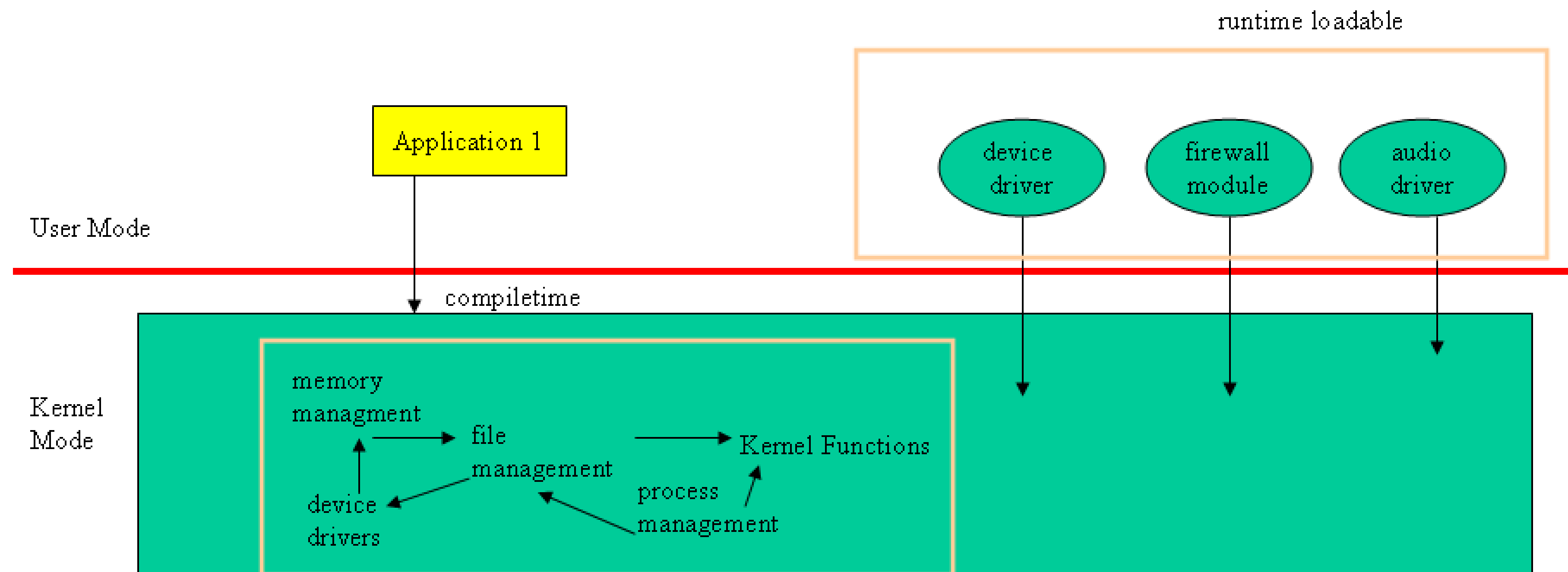
Only in kernel mode will the CPU allow critical instructions. The application will be terminated if it tries to execute critical instructions without changing through kernel traps into protected mode.

Kernel vs. User Mode



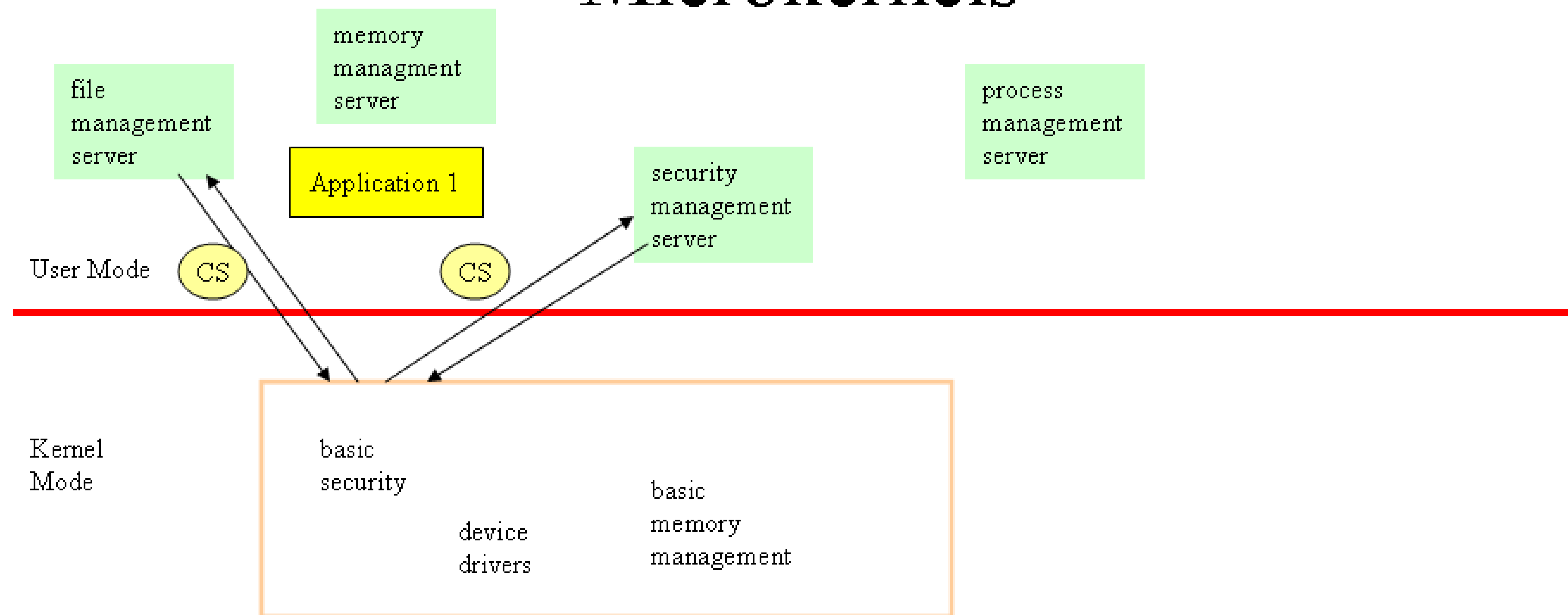
User services are much easier to develop, replace, monitor and debug than kernel services. But this comes at a price: Inter-process communication means going through the kernel which usually means context switches (CS) between processes and copying data back and forth from application to kernel and back to the other application or service. The good news: applications bugs usually do not crash services. And if, then services can be restarted without rebooting the kernel. Internal kernal functions are fast AND DANGEROUS: nothing prevents a kernel mode function from wrecking the system. They run usually with full CPU privileges and can access everything anytime. In times of weak hardware people put every service into the kernel. Nowadays most services are placed in user space.

Monolithic Kernels



Monolithic kernels run all operating system functions in kernel mode. The kernel itself either contains all necessary code already (compile time extension) or modules can be loaded dynamically (e.g. linux). All kernel code has the same criticality: a bug and the kernel crashes! Performance is good because internal calls are simple procedure calls and not system calls with traps. Maintenance is bad because of millions of lines of code for the kernel with dependencies and no protection between.

Microkernels



Microkernels run most of the traditional kernel functions outside in user mode services. Only the most basic functions like device control and some security and memory handling is performed in kernel mode. This make is easy to change e.g. to a different file management implementation by starting a new service. The price is paid in overhead due to increased numbers of context switches for a single function called.

Single Tasking vs. Multi-Tasking OS

- Only one task runs at any time
- A task is not pre-empted but can give up control
- If a task needs input it usually does a busy wait (polling) for data.

- Several tasks can run concurrently (multi-processor) or quasi-concurrently (single-CPU) by getting a timeslice of CPU time to run. Alternatively priorities decide which task does run (realtime OS)
- If a task needs to wait for data, it blocks (gives up the CPU voluntarily) and the scheduler runs another task
- There are non-interactive background tasks and interactive user oriented tasks

Single tasking operating systems are e.g. MSDOS. This makes e.g. a modern GUI with window technology impossible. The software structure of a single tasking system is much simpler (and safer, that‘s why in some mission critical areas asynchronous, interrupt driven multi-tasking systems where not allowed)

Single User Operating Systems

- The system does not identify the user
- Every task runs with the same built-in authority
- No separate User profiles maintained by the operating system.

Typical examples are MS-DOS, WINDOWS 9.xx and embedded control operating systems. The most critical feature is definitely the lack of a role concept that would allow privilege de-escalation for most tasks. When those types of operating systems are connected to networks or the Internet horrible things happen because the OS has no concept of principals and roles (not to mention capabilities). If user profiles exist then they are created and maintained by applications, creating a data graveyard of personal settings in different applications.

Pseudo Multi-User Operating Systems

- The system does identify the user and keeps different user profiles
- Some tasks run with system and some with user authority
- Resources are protected by Access Control Lists with different user having different rights.
- Usually only one User is the owner and current user of the machine. This user can shutdown the machine. They can change date and time. No quotas are usually set

Typical examples are Windows NT/2000 and some Linux desktop versions. Those systems do much better when connected to networks. Still, knowing that the single user is most often the only user, i.e. for convenience reasons, some system security is reduced. The rationale is: why should the user have to log-out and log-in as admin just to shut the machine down? Or: if the user wants to fill his disk to the brim, why stop her?

True Multi-User Operating Systems

- The system does identify the user and keeps different user profiles
- Tasks run with different user authorities ranging from admin over sub-admin roles to regular users.
- Resources are protected by Access Control Lists with different user having different rights.
- Many users or background tasks are active at any time. Regular users cannot shutdown the machine because this would interrupt other peoples work. Quotas are set for all resources on this system to prevent one user excluding others from service.
- Accounting is performed either for control or charging reasons
- New hardware, device drivers or applications do NOT require a reboot

There is still some technical difference to pseudo systems because of resource protection necessary due to many concurrent users. Mainframes are high end systems with perfect resource management (do you want to reboot a system with 2000+ concurrent users to install a printer driver? One hour downtime costs you 2000 times 150 Dollar). Unix systems are not so perfect with respect to resource management (they are much cheaper as well).

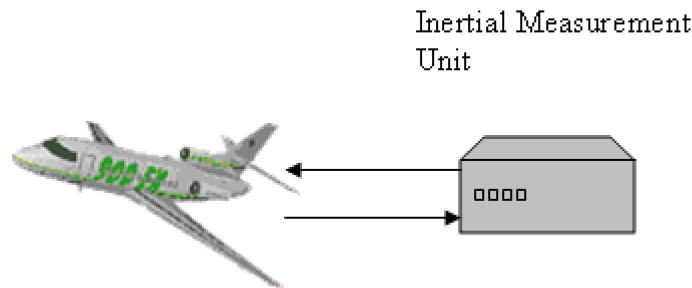
The true difference

What really distinguishes multitasking and multi-user systems today is not so much technology. Instead, it is the set of security policies implemented and enforced. Most systems could run as true multi-user systems but it is a fact that a system used by only one user will be very akward to use if it is run like a full multi-user system with separate identities and roles for administration and regular use. But the truth is: it is security tha t makes the difference. In this there really is a difference between a single-user system or a server/multi-user system.

In the end there is no technical difference – just security (or usability) and licensing.

Realtime systems are technically different.

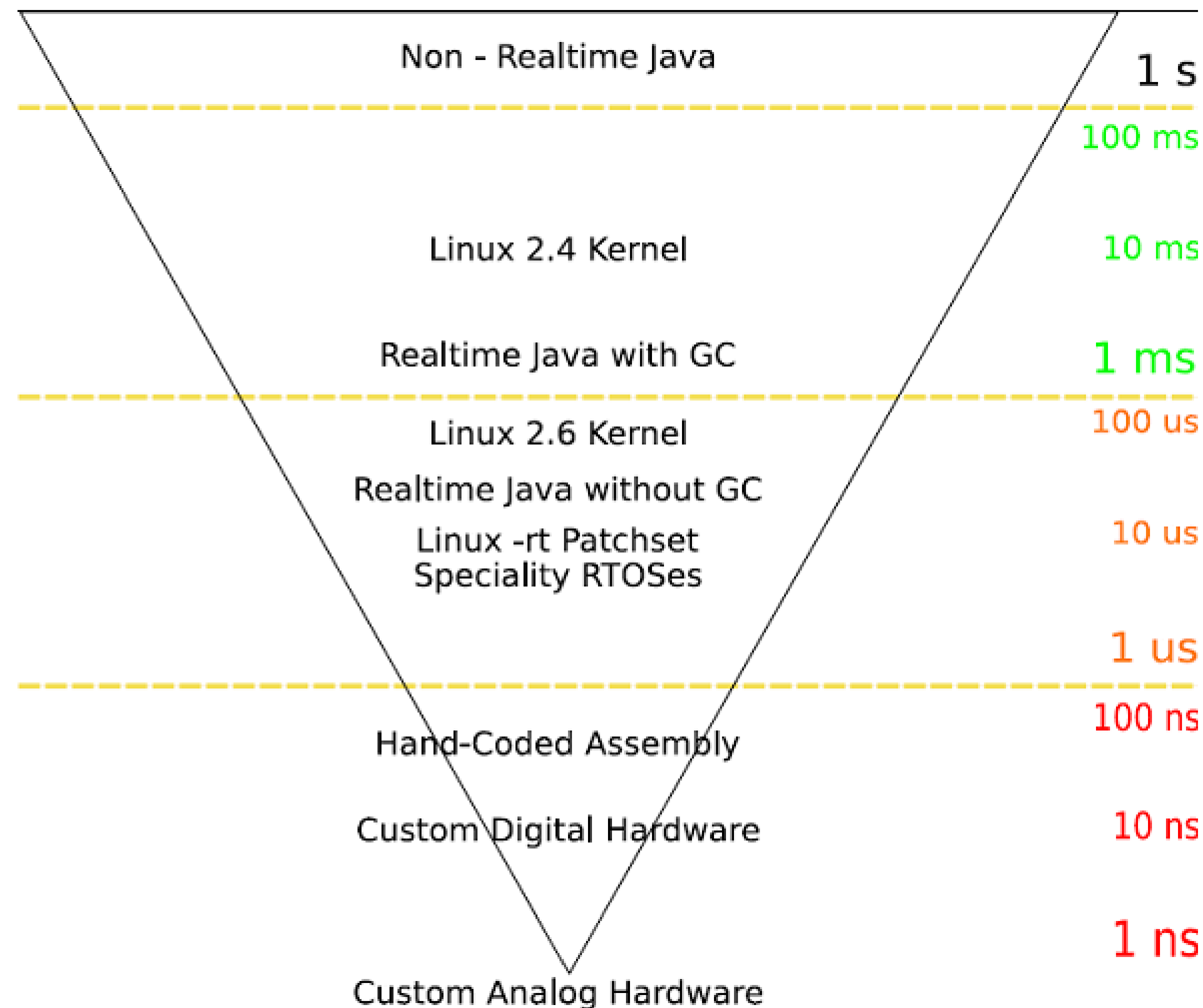
Real-time Operating Systems



- The system needs to react on events within a certain time.
- Tasks need to finish (provide a response) within a certain time
- Hardware and software are redundant (voting, backups etc.)

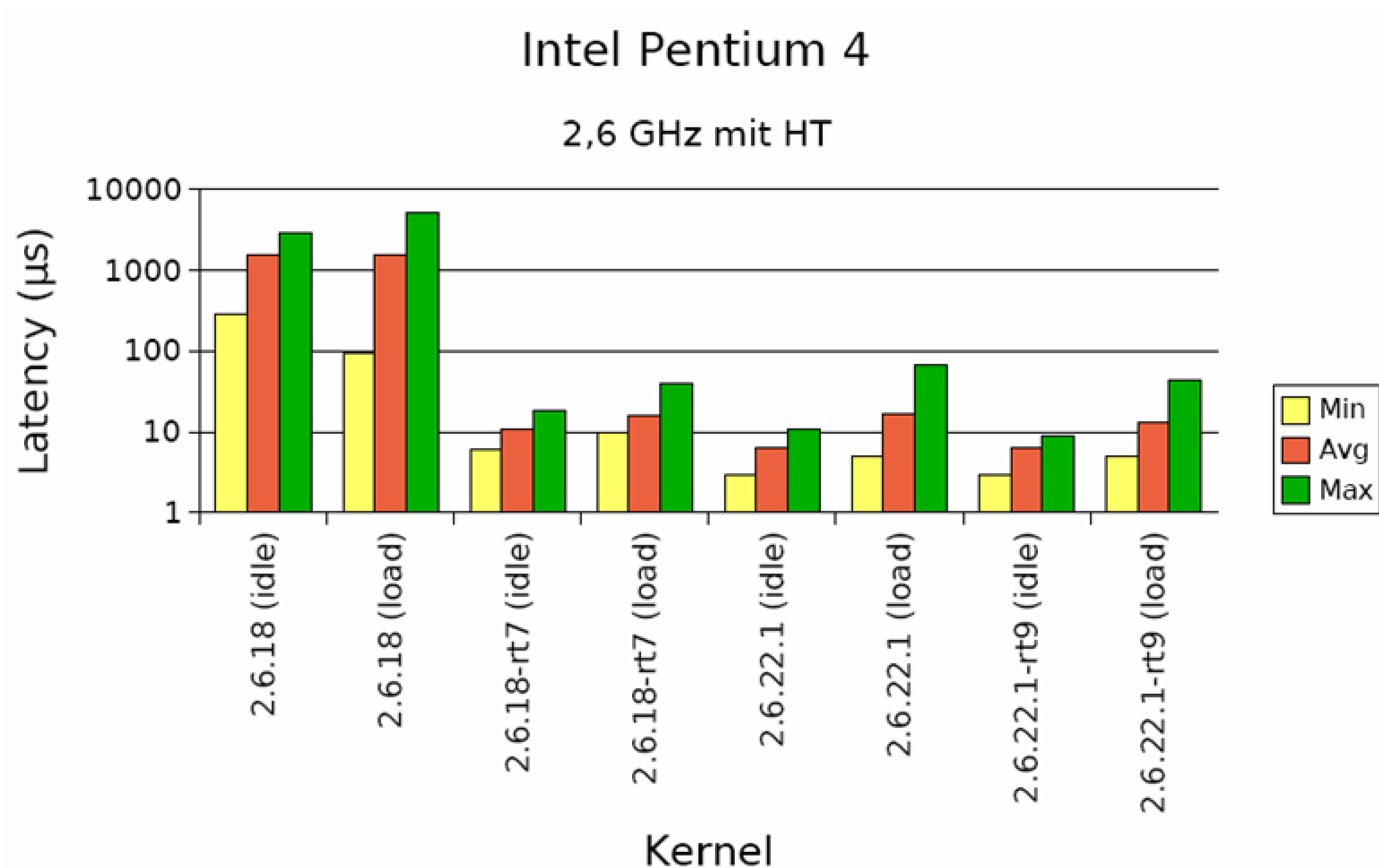
Real-time systems do not use timeslices like interactive systems. They assign priorities to processes. Whenever a high-priority process becomes runnable the scheduler will immediately preempt a low-priority process. A large amount of simulation goes into those systems (see www.ilogics.com for simulation software). Soft real-time systems are regular interactive systems with enough CPU power to generally fulfill timing requirements. The design of the OS kernel decides about whether a system qualifies for real-time requirements.

Latency



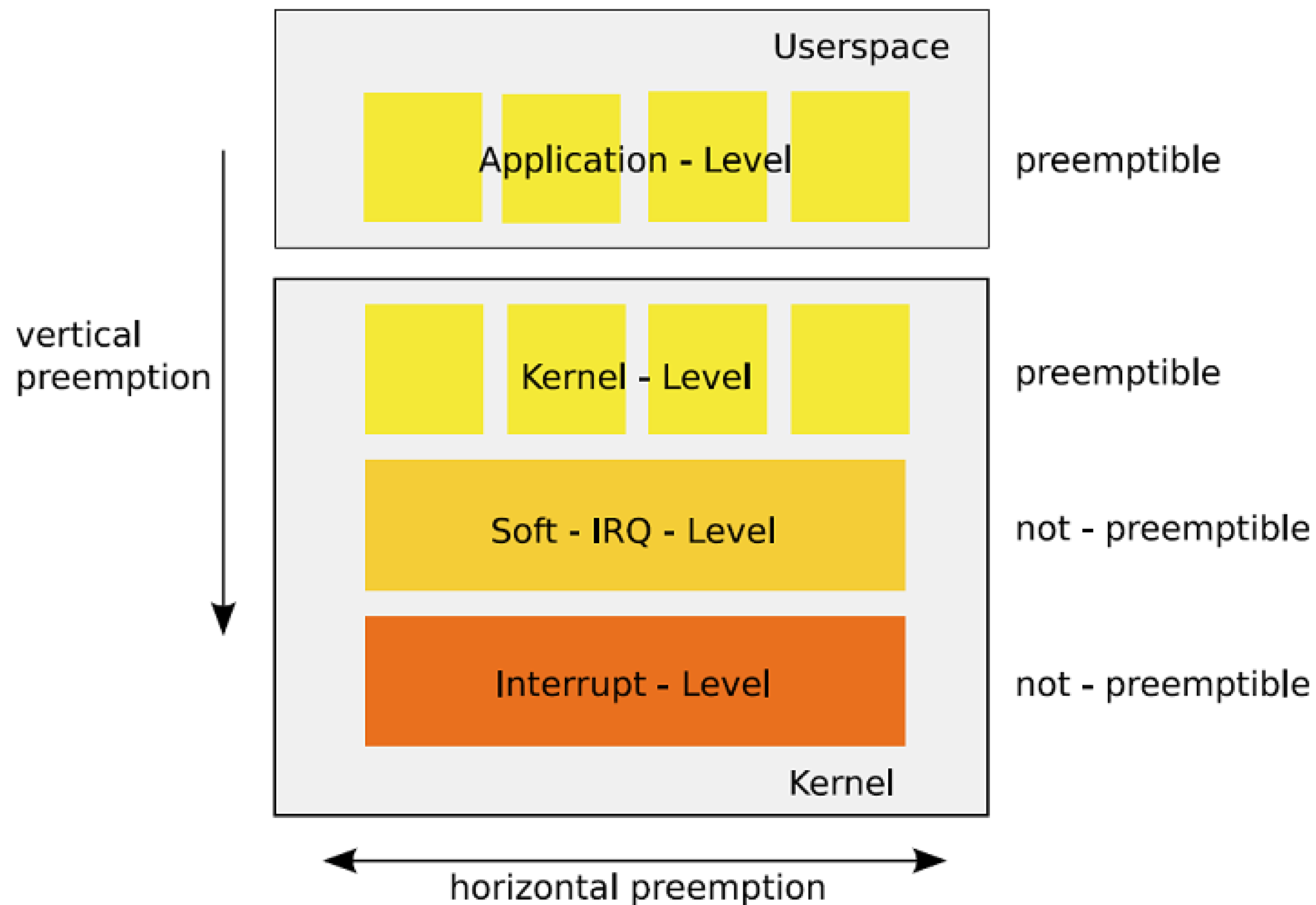
The following diagrams are from P.Klabinus thesis on Realtime-Extensions to Linux, architecture and performance (see resources) based on Paul Kenney, SMP and embedded realtime

Latency compared across kernel versions



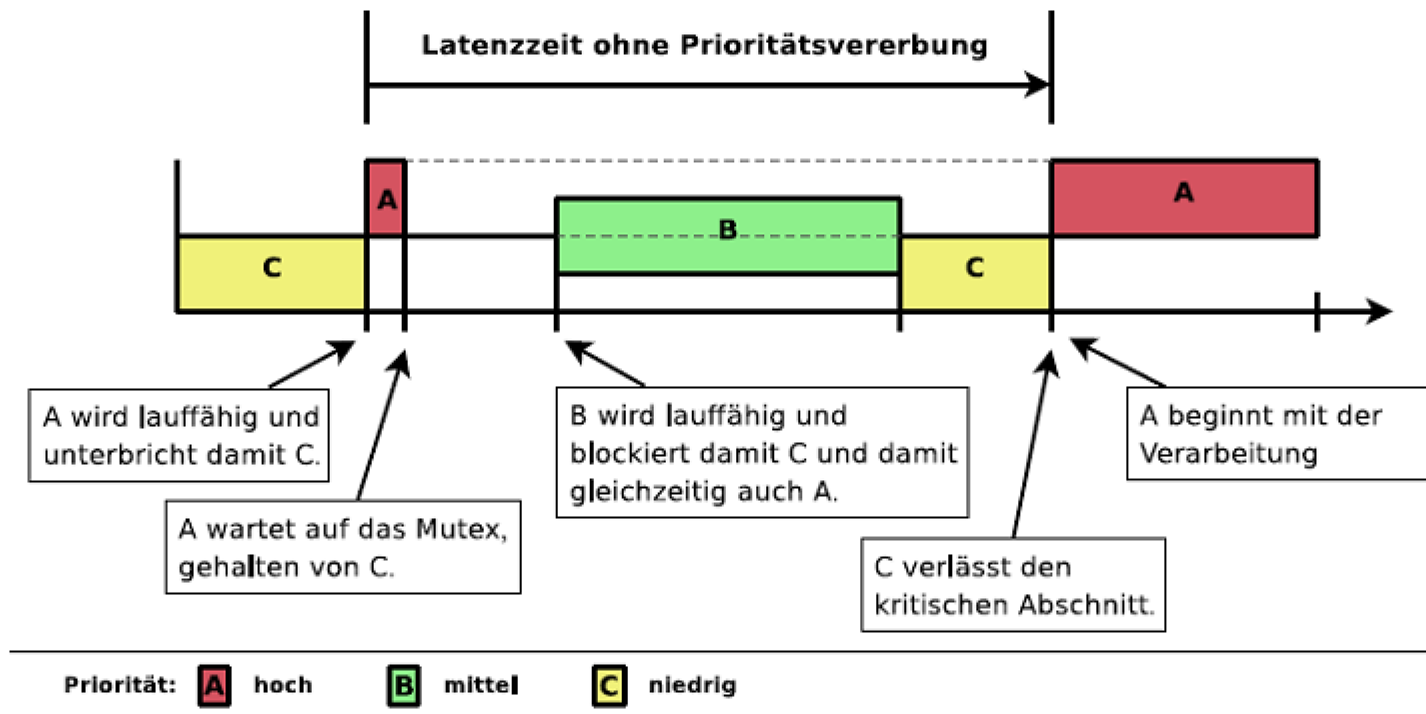
from P.Klabinus thesis on Realtime-Extensions to Linux, architecture and performance (see resources)

Preemption



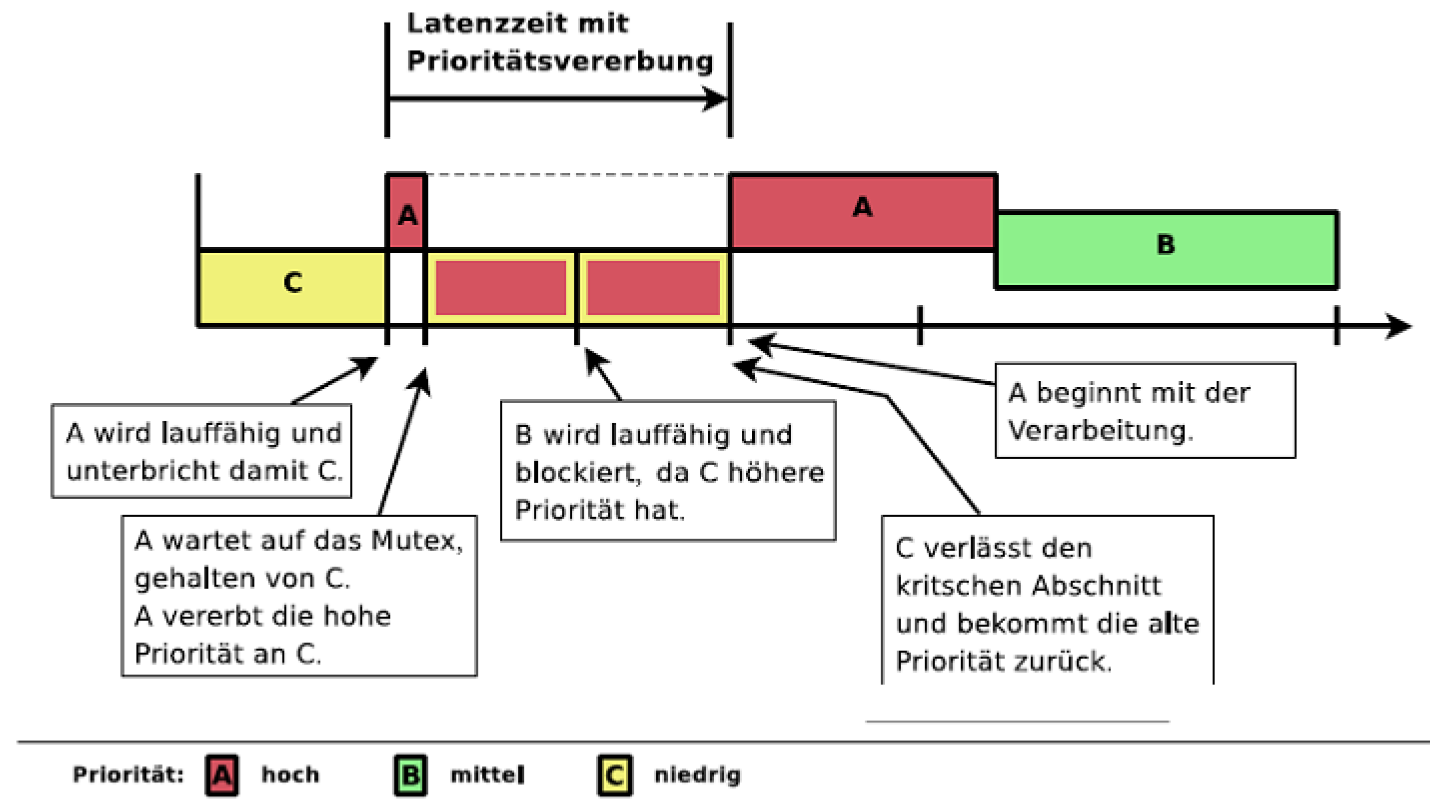
Large blocks of code where the caller cannot be preempted make a quick reaction to external events impossible. One solution is to create kernel threads und allow preemption on that level. This is what the Linux RT extension does. From E.Kunst et.al, see resources)

Priority Inversion Problem



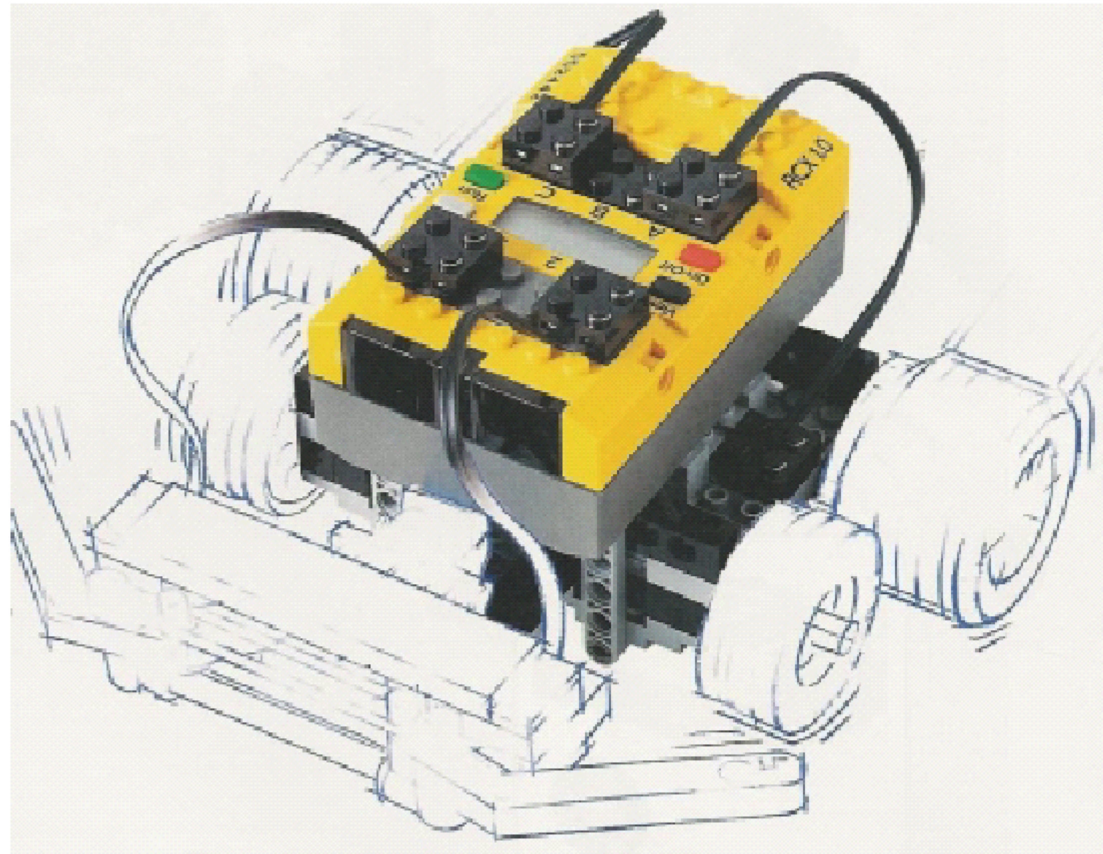
Notice how C is preempted but holds an important resource that is needed by higher priority processes. (Klabinus/Kunst et.al)

Priority Inheritance



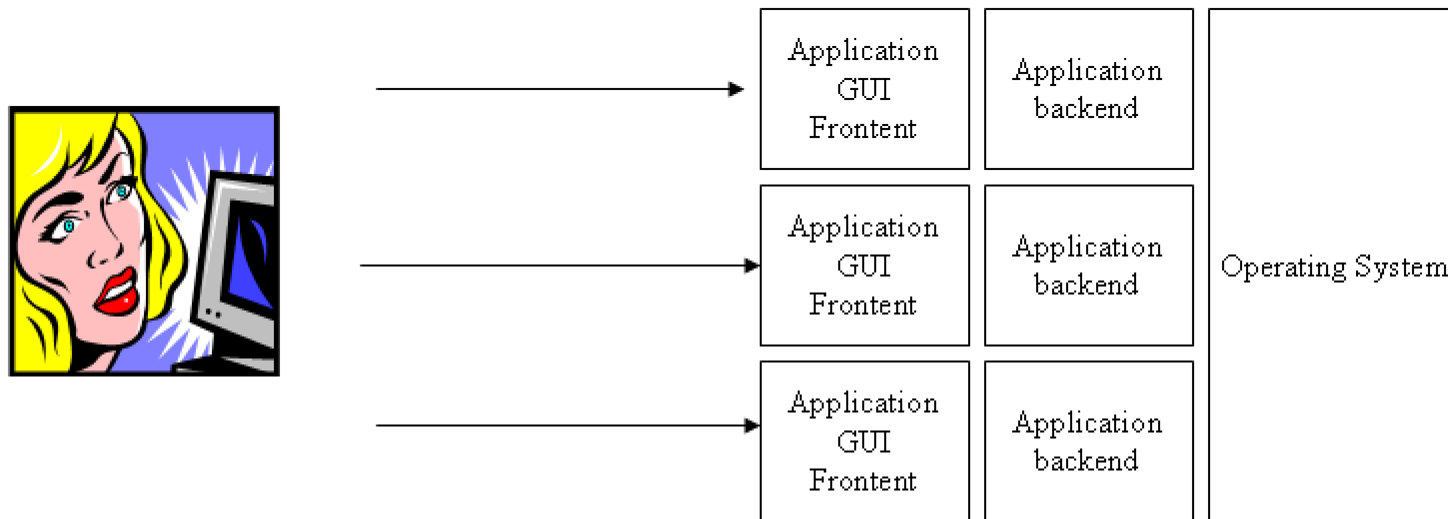
Now C gets temporarily a higher priority to finish processing. This will release the mutex held by C and allow high-priority process A to continue. (Klabinus/Kunst et.al)

Other embedded control platforms



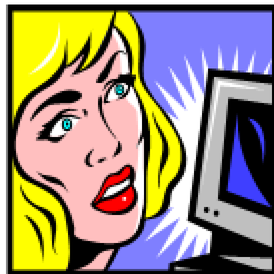
The Lego Mindstorms robot package is an example of a small embedded control platform. Several operating systems exist for this device, e.g. LegOS (c++) and Lejos (Java). A small Hitachi microprocessor and 32k of ram are available. We will use it as an example for embedded control programming (Using cross-compilation, firmware download and subsumption architecture). See http://www.jugs.org/protokolle/02-09-12/leJOS-v1.1.pdf

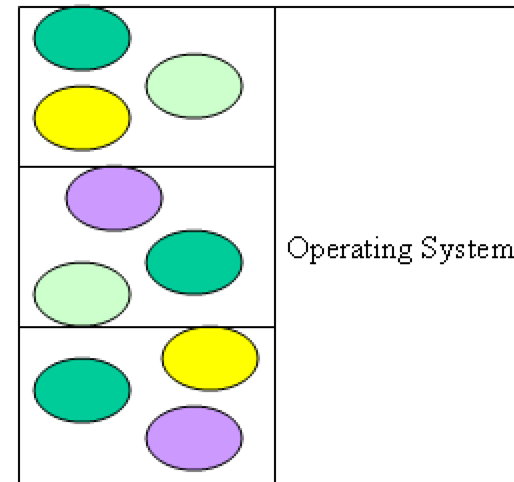
System Philosophies and Business (1)



The most popular systems today are the windows type desktop operating systems. Those systems have a mostly user centric view. They provide a graphical user interface for most tasks and make it very easy to learn the tools. New functions are usually implemented either as extensions of existing applications or a new applications. Users of those systems are usually relatively inexperienced computer users and the knowledge they acquire is mostly in knowing how to use certain applications. No programming is done by users. Software companies serving this customer and technology sector usually depend on updates for applications which bring new features. Automation is very low, everything has to happen manually through the GUI. Most applications get into total feature overload because there is no functional composition used which would require user training.

System Philosophies and Business (2)





Unix systems have traditionally favored a composition pattern. Instead of creating huge applications with hundreds of features small programs were built with very limited capabilities. But those modules could be linked together (via pipes) to generate processing pipelines. Or the shell language could be used to further connect those modules via scripts. This reduced the need to always extend existing modules at the price of the user now having to understand how this composition works – a form of programming. Inexperienced users soon found this challenging. But even worse: the business modell behind favors NOT buying new software with a new feature. Instead, building new solutions by combining existing modules is favored. I believe this is at the core of the decade old Unix vs. Windows discussion. Different user groups and business models. As a side effect Unix modules are NOT supposed to produce output for users. They need to produce output that becomes input for other modules and must therefore be careful not to contain presentation oriented features. That‘s why Unix programs operate „silently“.

Components of Operating Systems

- File management
- Memory management
- Process management (threads, multiprocessing)
- User and Security management
- GUI (window system)
- Command Interpreter (shell)
- Loadable Modules
- Utilities

The next sessions will introduce you to all these subsystems or components.

Compatibility

Layer	Description
Virtual Machine	Programs written for a virtual machine are independent of CPU and OS. Some system characteristics (e.g. RAM size) can still prevent compatibility
Programming Language	A C program does not guarantee compatibility because C does not cover all necessary services. In many cases C programs can be ported to other operating systems with a lot of effort. CPU dependency is small but exists (integer size etc.)
OS	Programs written for different operating systems make use of special OS functions and are hard to port to another OS.
Computer	Computers, especially small ones, differ a lot with respect to RAM size, MMU support etc. This makes it hard to run programs unchanged.
CPU	Programs written for different CPUs cannot be run on all platforms. But sometimes only a re-compile is necessary if the platform is the same otherwise (operating system, computer, language)

Operating Systems are always a hot political and economic topic – resulting from the fact that compatibility of applications is so much tied to the operating system. The only layer that really makes a program largely independent of platform and OS is the virtual machine.

Resources (1)

- Modern Operating Systems, Andrew S.Tanenbaum. The bible of operating systems. If you need to build low-level system code this is your book. Its content stays valid a long time...
- Jean Bacon, Tim Harris, Operating Systems. Concurrent and Distributed Software Design. If you need to understand the concepts behind complex applications, perhaps even distributed, this is a very good book. Not so implementation centric as Tanenbaum. Includes Transactions. I like it better if there is more implementation but ymmv....
- Maurice J. Bach, The design of the Unix Operating System. 1996. At that time I was waiting for this book desperately... Take to it if you need to understand how signals work, how Unix does this and that. Kernel implementation centric.
- Gary Nutt, Operating Systems, A modern perspective, Lab Update. Nice code examples and explanations for Linux and Windows system programming. Covers computer organization as well. Good!

Resources (2)

- Jochen Hiller, Lego Mindstorms – Introduction. An excellent overview of the lego platform and the lejos java operating system which runs on this device. http://www.jugs.org/protokolle/02-09-12/leJOS-v1.1.pdf . This is an ideal chance to see a java virtual machine in source code and a small footprint.
- The lejos homepage, www.lejos.org
- The LegOS homepage, www.legos.org
- Unix Skriptum (Deutsch) – HDM
- http://webcast.berkeley.edu/courses/archive.php?seriesid=1906978284 Berkley lecture "CS 162 Operating Systems and System Programming" either as video stream or mp3 for download. (thanks to Marc Seeger for the link)

Resources (3) Realtime

- McKenney, Paul: Smp and embedded real time. January 2007.
http://www.linuxjournal.com/node/9361 (overview of latencies)

- Love, Robert: Linux-Kernel-Handbuch. Addison-Wesley, 1. Auflage, (process models, syscalls)

- Eva-Katharina Kunst, Jürgen Quade: Kern-Technik - Folge 34 - Das Realtime-Preemption-Patch. Juli 2007
http://linux-magazin.de/heft_abo/ausgaben/2007/07/kern_technik
(Linux preemption model, priority inversion and inheritance)

- Philip Klabinus, Linux Realtime Extension – architecture and performance (Thesis HDM 2008)