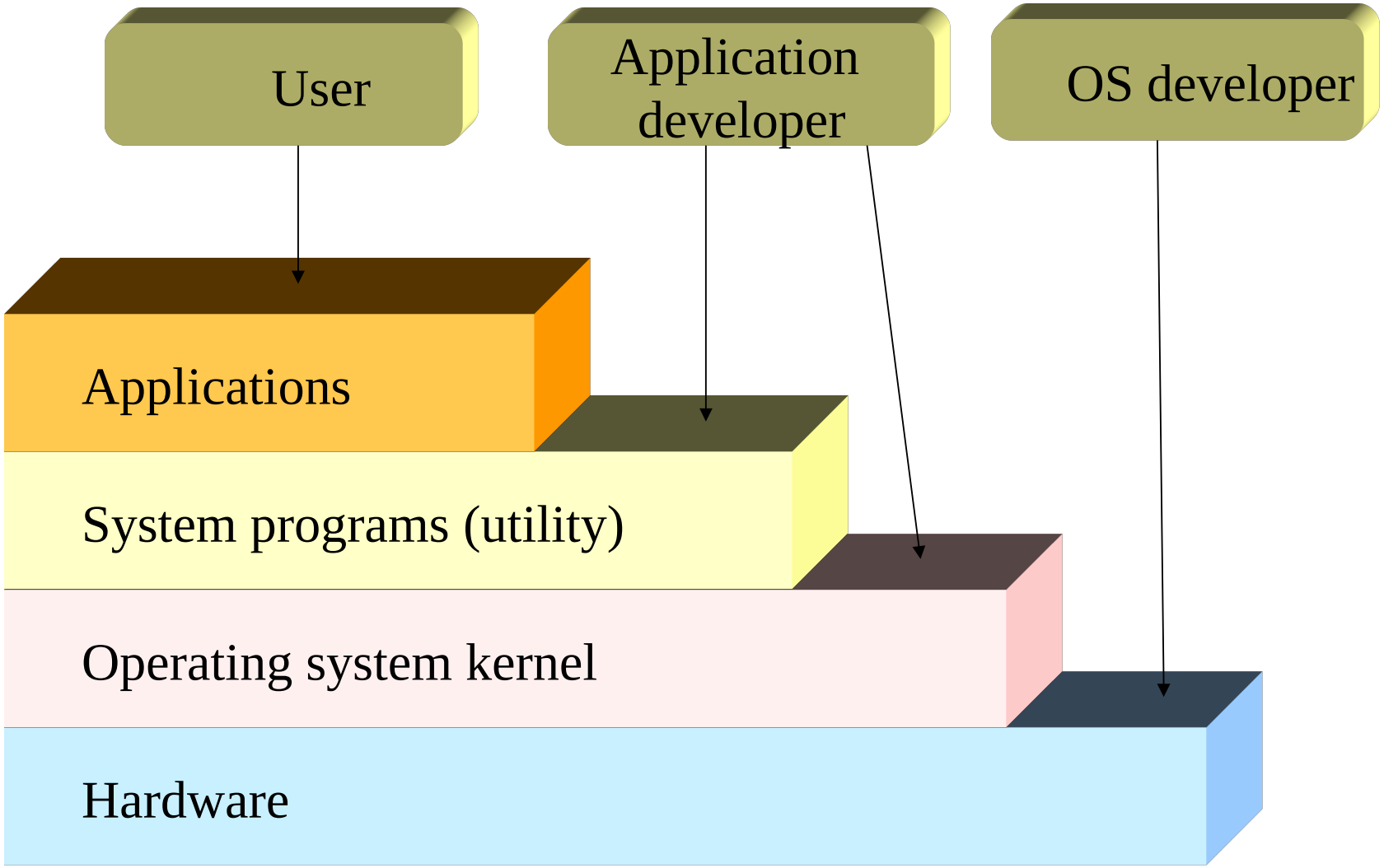
Operating Systems and Networks
AE4B33OSS

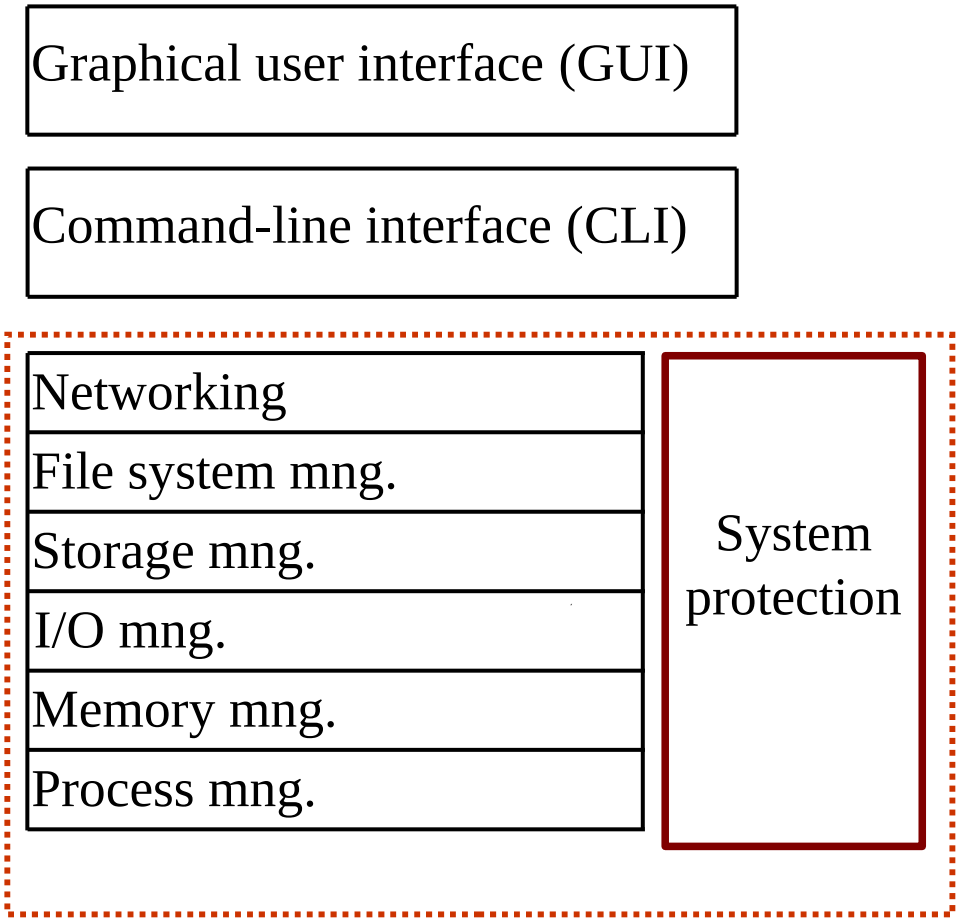
RNDr. Petr Štěpán, Ph.D

System call

Structure of computer



Operating System structure



Process Management

A process is a program in execution. It is a unit of work within the system. Program is a *passive entity*, process is an *active entity*.

- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

Process Management Activities

The operating system is responsible for the following activities in connection with process management:

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

Memory Management

- All data in memory before and after processing
- All instructions in memory in order to execute
- Memory management determines what is in memory when
  - Optimizing CPU utilization and computer response to users
- Memory management activities
  - Keeping track of which parts of memory are currently being used and by whom
  - Deciding which processes (or parts thereof) and data to move into and out of memory
  - Allocating and deallocating memory space as needed

Storage Management

- OS provides uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit – **file**
  - Each medium is controlled by device (i.e., disk drive, tape drive)
    - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what
  - OS activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and dirs
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

I/O Subsystem

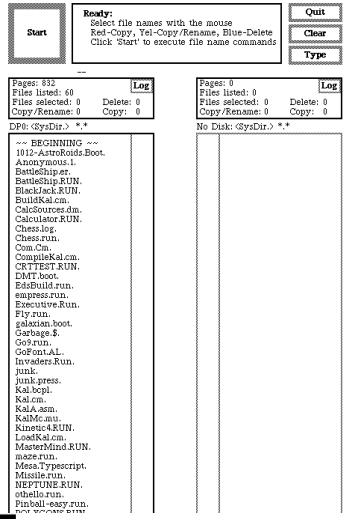
- One purpose of OS is to hide specialities of hardware devices from the user
- I/O subsystem responsible for
  - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
  - General device-driver interface
  - Drivers for specific hardware devices

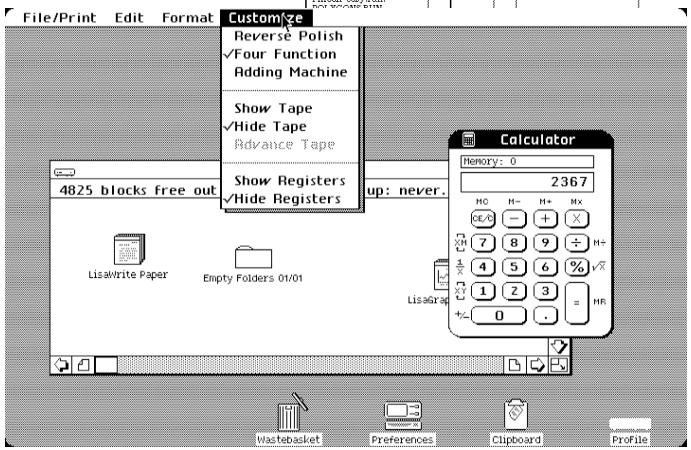
User Operating System Interface

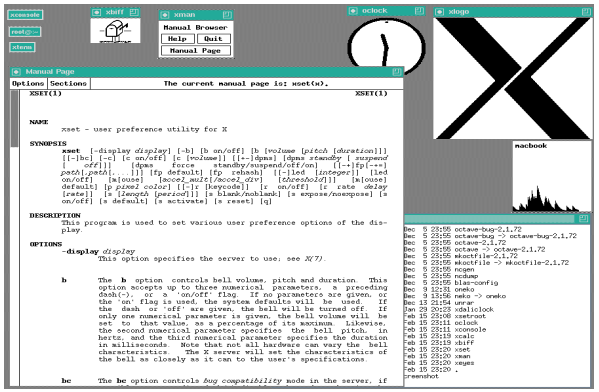
- **CLI** allows direct command entry
  - Sometimes implemented in kernel, mostly by system programs
  - Sometimes multiple flavors implemented – **shells**
  - Primarily fetches a command from user and executes it
    - Some commands are built-in, sometimes just names of programs
    - If the latter, adding new features doesn't require shell modification

User Operating System Interfac

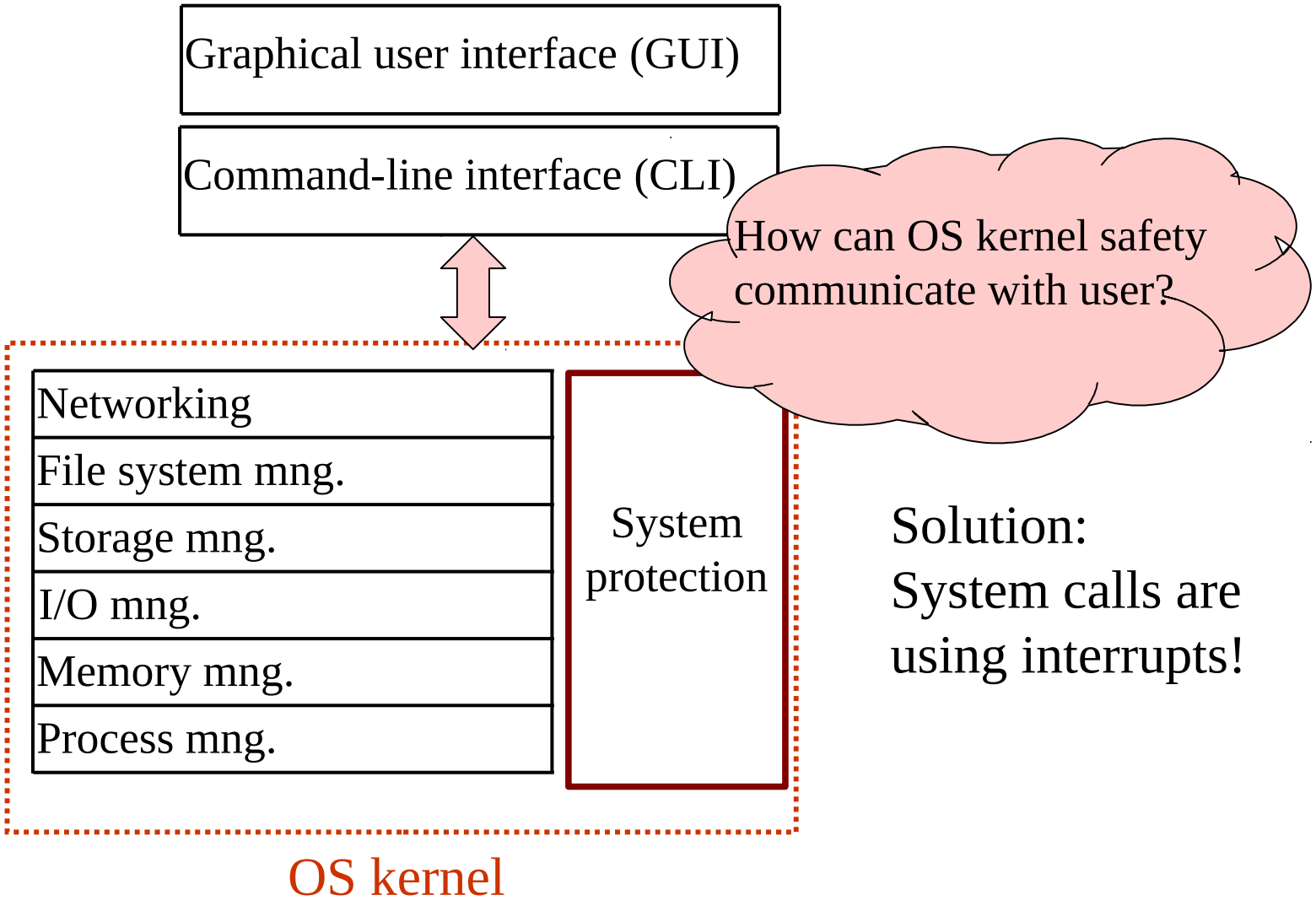
- **GUI** – a user-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc.
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox Alto 1973, followed by Apple Lisa 1983, X windows (client-server)1984 and MS Windows 1.0 -1985
- Many systems include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "cmd" shell
  - Apple Mac OS X as "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)







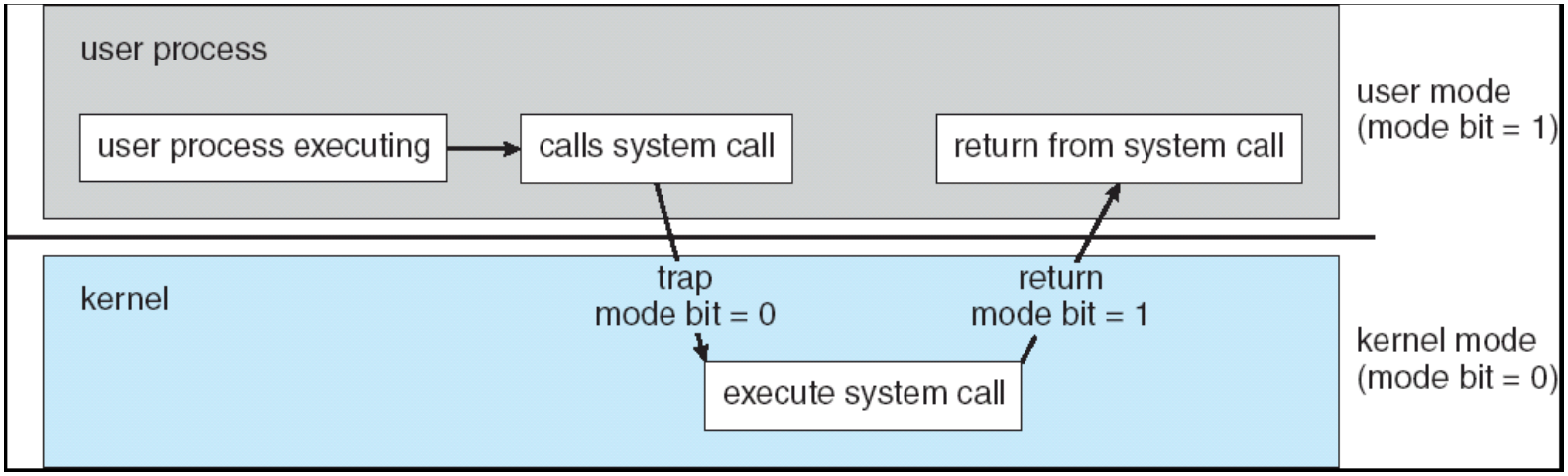
How Operating System works?



Solution:
System calls are using interrupts!

OS protection by System services

Transition from User to Kernel Mode and back



- User processes are running in protected mode
- Kernel is running in supervisor mode

System Call Implementation

- Each system call is using the same interrupt number (0x80)
- System calls are distinguished by number associated with each system call
- The arguments are passed to kernel by registers
- The result values of systems call must be stored in user space, that was prepared by user program
- Result of system call is returned in register
- The user cannot execute it's own program in kernel

How the System Call Interface is Implemented

X86 System Call Example Hello World on Linux

```
  .section .rodata
greeting:
    .string "Hello World\n"
    .text

_start:
   mov $12,%edx                 /* write(1, "Hello World\n", 12) */
   mov $greeting,%ecx
   mov $1,%ebx
   mov $4,%eax                  /* write is syscall no. 4 */
   int $0x80

   xorl %ebx, %ebx             /* Set exit status and exit */
   mov $0xfc,%eax
   int $0x80

   hlt                          /* Just in case... */
```

System API Standards

- Direct use of system calls is very complicated (you need to know parameter assignment, return values, use assembler)
- Three most common API (Application Programming Interface) standards are
  - ***POSIX*** API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - ***Win32*** API for Windows
  - ***Java API*** for the Java virtual machine (JVM)
    - out of this course scope
- POSIX (IEEE 1003.1, ISO/IEC 9945)
  - Very widely used standard based on (and including) C-language
  - Defines both
    - ***system calls*** and
    - compulsory ***system programs*** together with their functionality and command-line format
      - E.g. `ls –w dir` prints the list of files in a directory in a 'wide' format
  - Complete specification is at

    http://www.opengroup.org/onlinepubs/9699919799/nframe.html
- Win32 (Micro$oft Windows based systems)
  - Specifies system calls together with many Windows GUI routines
    - VERY complex, no really complete specification

POSIX

- **P**ortable **O**perating **S**ystem **I**nterface for Uni**x** – IEEE standard for system interface
- Standardization process began circa 1985 – necessary for system interoperability
- 1988 POSIX 1 Core services
- 1992 POSIX 2 Shell and utilities
- 1993 POSIX 1b Real-time extension
- 1995 POSIX 1c Thread extension
- After 1997 connected with ISO leads to POSIX:2001 and POSIX:2008
- http://www.opengroup.org/onlinepubs/9699919799

POSIX example

- Standard defines:
  - Name - system call name(for example read)
  - Synopsis - ssize_t read(int *fildes*, void **buf*, size_t *nbyte*);
  - Description – detailed text description of system call functions
  - Return value – define all possible return values, often describes how to recognize errors
  - Errors – define all possible errors of this function
  - Examples – sometimes are listed examples how to use this call
  - See also – list of systems calls related to described system call

POSIX definition is available in each UNIX like system by command ***man***

POSIX example

NAME

abort - generate an abnormal process abort **SYNOPSIS**

#include <stdlib.h>

void abort(void);

DESCRIPTION

[CX] The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1-2008 defers to the ISO C standard.

The *abort*() function shall cause abnormal process termination to occur, unless the signal SIGABRT is being caught and the signal handler does not return.

[CX] The abnormal termination processing shall include the default actions defined for SIGABRT and may include an attempt to effect *fclose()* on all open streams.

The SIGABRT signal shall be sent to the calling process as if by means of *raise()* with the argument SIGABRT.

[CX] The status made available to *wait()*, *waitid()*, or *waitpid()* by *abort*() shall be that of a process terminated by the SIGABRT signal. The *abort*() function shall override blocking or ignoring the SIGABRT signal.

POSIX example

RETURN VALUE

The *abort*() function shall not return.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

Catching the signal is intended to provide the application developer with a portable means to abort processing, free from possible interference from any implementation-supplied functions.

RATIONALE

The ISO/IEC 9899:1999 standard requires the *abort*() function to be async-signal-safe. Since POSIX.1-2008 defers to the ISO C standard, this required a change to the DESCRIPTION from ``shall include the effect of *fclose*()" to ``may include an attempt to effect *fclose*()."

The revised wording permits some backwards-compatibility and avoids a potential deadlock situation.

The Open Group Base Resolution bwg2002-003 is applied, removing the following XSI shaded paragraph from the DESCRIPTION:

…..

POSIX example

FUTURE DIRECTIONS

None.

SEE ALSO

exit , *kill* , *raise* , *signal* , *wait* , *waitid*

XBD <*stdlib.h*>

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 6

Extensions beyond the ISO C standard are marked.

Changes are made to the DESCRIPTION for alignment with the ISO/IEC 9899:1999 standard.

The Open Group Base Resolution bwg2002-003 is applied.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/10 is applied, changing the DESCRIPTION of abnormal termination processing and adding to the RATIONALE section.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/9 is applied, changing ``implementation-defined functions'' to ``implementation-supplied functions'' in the APPLICATION USAGE section.

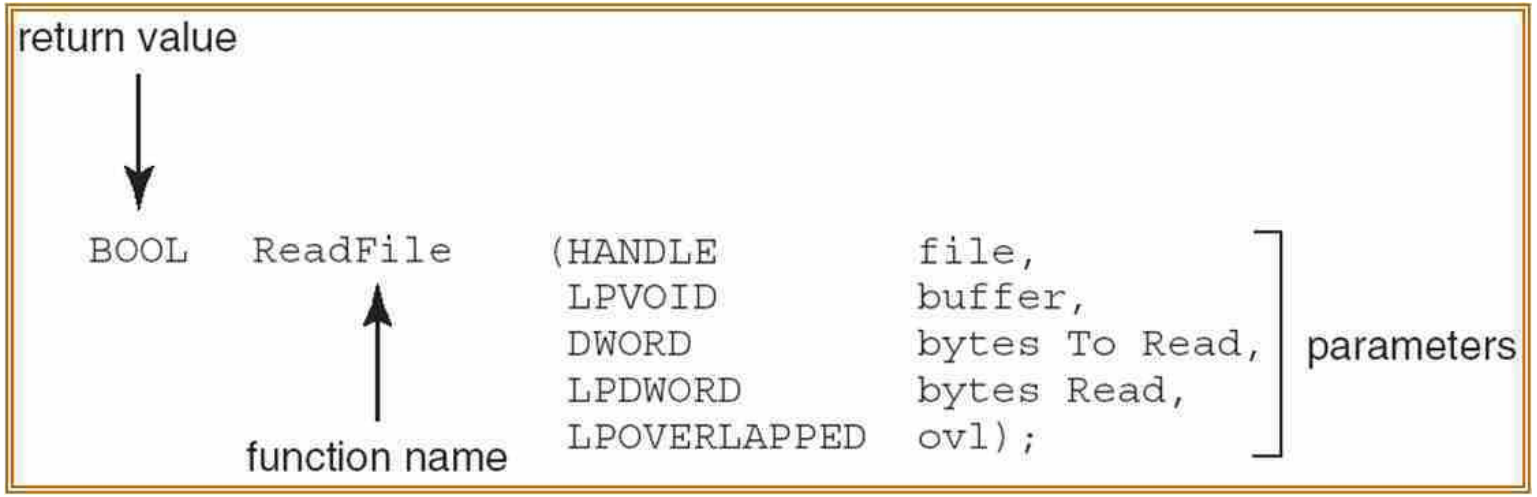
Windows API

- Not fully described – hidden system calls, hidden system functionalities
- MS developers can ask MS for explanation
- Win16 – 16-bit version for Windows 3.1
- Win32 – 32 bit version started with Windows NT
- Win32 for 64-bit Windows – 64 bit version of Win32, main changes only in memory pointer types

- For long time, only MS Visual Studio and Borland's compilers were the only tools to use for Win API

Example of a System Call through a Standard API

- Consider the ReadFile() function in the Win32 API – a function for reading from a file



- The parameters passed to ReadFile() are
  - HANDLE file – the file to be read
  - LPVOID buffer – a buffer where the data will be read into and written from
  - DWORD bytesToRead – the number of bytes to be read into the buffer (buffer size)
  - LPDWORD bytesRead – the number of bytes read during the last read
  - LPOVERLAPPED ovl – indicates if overlapped (non-blocking) I/O is to be used

Types of POSIX System Calls

A set of (seemingly independent) groups of services:

- Process control and IPC (Inter-Process Communication)
- Memory management
  - allocating and freeing memory space on request
- Access to data in files
- File and file-system management
- Device management
- Communications
  - Networking and distributed computing support
- Other services
  - e.g., profiling
  - debugging
  - etc.

Process Control Calls (1)

- fork() – create a new process

```
pid = fork();
```

  - The *fork*() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except some process' system properties
  - It returns 'twice'
    - return value == 0 ... child
    - return value > 0 ... parent (returned value is the child's *pid*)
    - return value <0 … error in child creation

- exit() – terminate a process

```
void exit(int status);
```

  - The *exit*() function shall then flush all open files with unwritten buffered data and close all open files. Finally, the process shall be terminated and system resources owned by the process shall be freed
  - The value of 'status' shall be available to a waiting parent process
  - The *exit*() function should never return

Process Control Calls (2)

- wait, waitpid – wait for a child process to stop or terminate

```
pid = wait(int *stat_loc);
pid = waitpid(pid_t pid, int *stat_loc, int
options);
```

  - The *wait*() and *waitpid*() functions shall suspend the calling process and obtain status information pertaining to one of the caller's child processes. Various options permit status information to be obtained for child processes that have terminated or stopped.

- execl, execle, execlp, execv, execve, execvp – execute a file

```
int execl(const char *path, const char
*arg0, ...);
```

  - The members of the *exec* family of functions differ in the form and meaning of the arguments
  - The *exec* family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the *new process image file*.
  - There shall be no return from a successful *exec*, because the calling process image is overlaid by the new process image; any return indicates a failure

Memory Management Calls

- System calls of this type are rather obsolete
  - Modern virtual memory mechanisms can allocate memory automatically as needed by applications
  - Important system API calls are:
- malloc() – a memory allocator

```
void *malloc(size_t size);
```

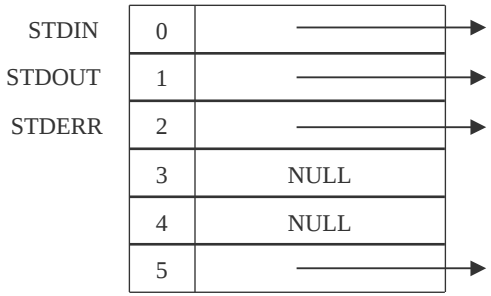
  - The *malloc()* function shall allocate unused space for an object whose size in bytes is specified by `size` and whose value is unspecified.
  - It returns a pointer to the allocated memory space
- free() – free a previously allocated memory

```
void free(void *ptr);
```

  - The *free()* function shall cause the space pointed to by `ptr` to be deallocated; that is, made available for further allocation.
  - If the argument does not match a pointer earlier returned by a *malloc*() call, or if the space has been deallocated by a call to *free()*, the behavior is undefined.

File Access Calls (1)

- POSIX-based operating systems treat a *file* in a very general sense
  - ***File*** is an object that can be written to, or read from, or both. A file has certain attributes, including access permissions and type.
  - File types include
    - regular file,
    - character special file ... a ‘byte oriented device’,
    - block special file ... a ‘block oriented device’,
    - FIFO special file,
    - symbolic link,
    - socket, and
    - directory.
  - To access any file, it must be first *open*ed using an *open()* call that returns a *file descriptor* (*fd*).
    - *fd* is a non-negative integer used for further reference to that particular file
    - In fact, *fd* is an index into a process-owned table of file descriptors
    - Any *open()* (or other calls returning *fd*) will always assign the LOWEST unused entry in the table of file descriptors



File Access Calls (2)

- open – open file

```
fd = open(const char *path, int
oflag, ...);
```

  - The *open()* function shall establish the connection between a file and a file descriptor. The file descriptor is used by other I/O functions to refer to that file. The `path` argument points to a pathname naming the file.
  - The parameter `oflag` specifies the open mode:
    - ReadOnly, WriteOnly, ReadWrite
    - Create, Append, Exclusive, ...

- close – close a file descriptor

```
err = close(int fd);
```

  - The *close*() function shall deallocate the file descriptor indicated by *fd*. To deallocate means to make the file descriptor available for return by subsequent calls to *open*() or other functions that allocate file descriptors.
  - When all file descriptors associated with an open file description have been closed, the open file description shall be freed.

File Access Calls (3)

- read – read from a file

```
b_read = read(int fd, void *buf, int nbyte);
```

  - The *read*() function shall attempt to read `nbyte` bytes from the file associated with the open file descriptor, `fd`, into the buffer pointed to by `buf`.
  - The return value shall be a non-negative integer indicating the number of bytes actually read.

- write – write to a file

```
b_written = write(int fd, void *buf, int nbyte);
```

  - The *write*() function shall attempt to write `nbyte` bytes from the buffer pointed to by `buf` to the file associated with the open file descriptor `fd`.
  - The return value shall be a non-negative integer indicating the number of bytes actually written.

File Access Calls (4)

- lseek – move the read/write file offset

```
where = lseek(int fd, off_t offset, int whence);
```

  - The *lseek*() function shall set the file offset for the open associated with the file descriptor `fd`, as follows:
    - If `whence` is `SEEK_SET`, the file offset shall be set to `offset` bytes.
    - If `whence` is `SEEK_CUR`, the file offset shall be set to its current location plus `offset`.
    - If `whence` is `SEEK_END`, the file offset shall be set to the size of the file plus `offset`.
  - The *lseek*() function shall allow the file offset to be set beyond the end of the existing data in the file creating a gap. Subsequent reads of data in the gap shall return bytes with the value 0 until some data is actually written into the gap (implements *sparse file*).
  - Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, shall be returned.
  - An interesting use is:

```
where = lseek(int fd, 0, SEEK_CUR);
```

    returns the "current position" in the file.

File Access Calls (5)

- dup – duplicate an open file descriptor

```
fd_new = dup(int fd);
```

  - The *dup*() function shall duplicate the descriptor to the open fileassociated with the file descriptor `fd`.
  - As for *open()*, the LOWEST unused file descriptor should be returned.
  - Upon successful completion a non-negative integer, namely the file descriptor, shall be returned; otherwise, -1 shall be returned to indicate the error.

- stat – get file status

```
err = stat(const char path, struct stat *buf);
```

  - The *stat*() function shall obtain information about the named file and write it to the area pointed to by the `buf` argument. The `path` argument points to a pathname naming a file. The file need not be open.
  - The `stat` structure contains a number of important items like:
    - device where the file is, file size, ownership, access rights, file time stapms, etc.

File Access Calls (6)

- chmod – change mode of a file

```
err = chmod(const char *path, mode_t mode);
```

  - The *chmod*() function shall the file permission of the file named by the `path` argument to the in the `mode` argument. The application shall ensure that the effective privileges in order to do this.

- pipe – create an interprocess communication channel

```
err = pipe(int fd[2]);
```

  - The *pipe*() function shall create a pipe and place two file descriptors, one each into the arguments `fd[0]` and `fd[1]`, that refer to the open file descriptors for the read and write ends of the pipe. Their integer values shall be the two lowest available at the time of the *pipe*() call.
  - A read on the file descriptor `fd[0]` shall access data written to the file descriptor `fd[1]` on a first-in-first-out basis.
  - The details and utilization of this call will be explained later.

File & Directory Management Calls (1)

- mkdir – make a directory relative to directory file descriptor

```
err = mkdir(const char *path, mode_t mode);
```

  - The *mkdir*() function shall create a new directory with name `path`. The new directory access rights shall be initialized from `mode`.

- rmdir – remove a directory

```
err = rmdir(const char *path);
```

  - The *rmdir*() function shall remove a directory whose name is given by `path`. The directory shall be removed only if it is an empty directory.

- chdir – change working directory

```
err = chdir(const char *path);
```

  - The *chdir*() function shall cause the directory named by the pathname pointed to by the `path` argument to become the current working directory. Working directory is the starting point for path searches for *relative* pathnames.

File & Directory Management Calls (2)

- link – link one file to another file

```
err = int link(const char *path1, const
char *path2);
```

  - The *link*() function shall create a new link (directory entry) for the existing file identified by `path1`.

- unlink – remove a directory entry

```
err = unlink(const char *path);
```

  - The *unlink*() function shall remove a link to a file.
  - When the file's link count becomes 0 and no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible. If one or more processes have the file open when the last link is removed, the link shall be removed before *unlink*() returns, but the removal of the file contents shall be postponed until all references to the file are closed.

Device Management Calls

- System calls to manage devices are hidden into ‘file calls’
  - POSIX-based operating systems do not make difference between traditional files and ‘devices’. Devices are treated as ‘special files’
  - Access to ‘devices’ is mediated by opening the ‘special file’ and accessing it through the device.
  - Special files are usually ‘referenced’ from the `/dev` directory.
- ioctl – control a device

```
int ioctl(int fd, int request, ... /* arg */);
```

  - The *ioctl*() function shall perform a variety of control functions on devices. The `request` argument and an optional third argument (with varying type) shall be passed to and interpreted by the appropriate part of the associated with `fd`.

Other Calls

- kill – send a signal to a process or a group of processes

```
err = kill(pid_t pid, int sig);
```

  - The *kill*() function shall send a signal to a process specified by `pid`. The signal to be sent is specified by `sig`.
  - *kill*() is an elementary inter-process communication means
  - The caller has to has to have sufficient privileges to send the signal to the target.
- signal – a signal management

```
void (*signal(int sig, void (*func)(int)))(int);
```

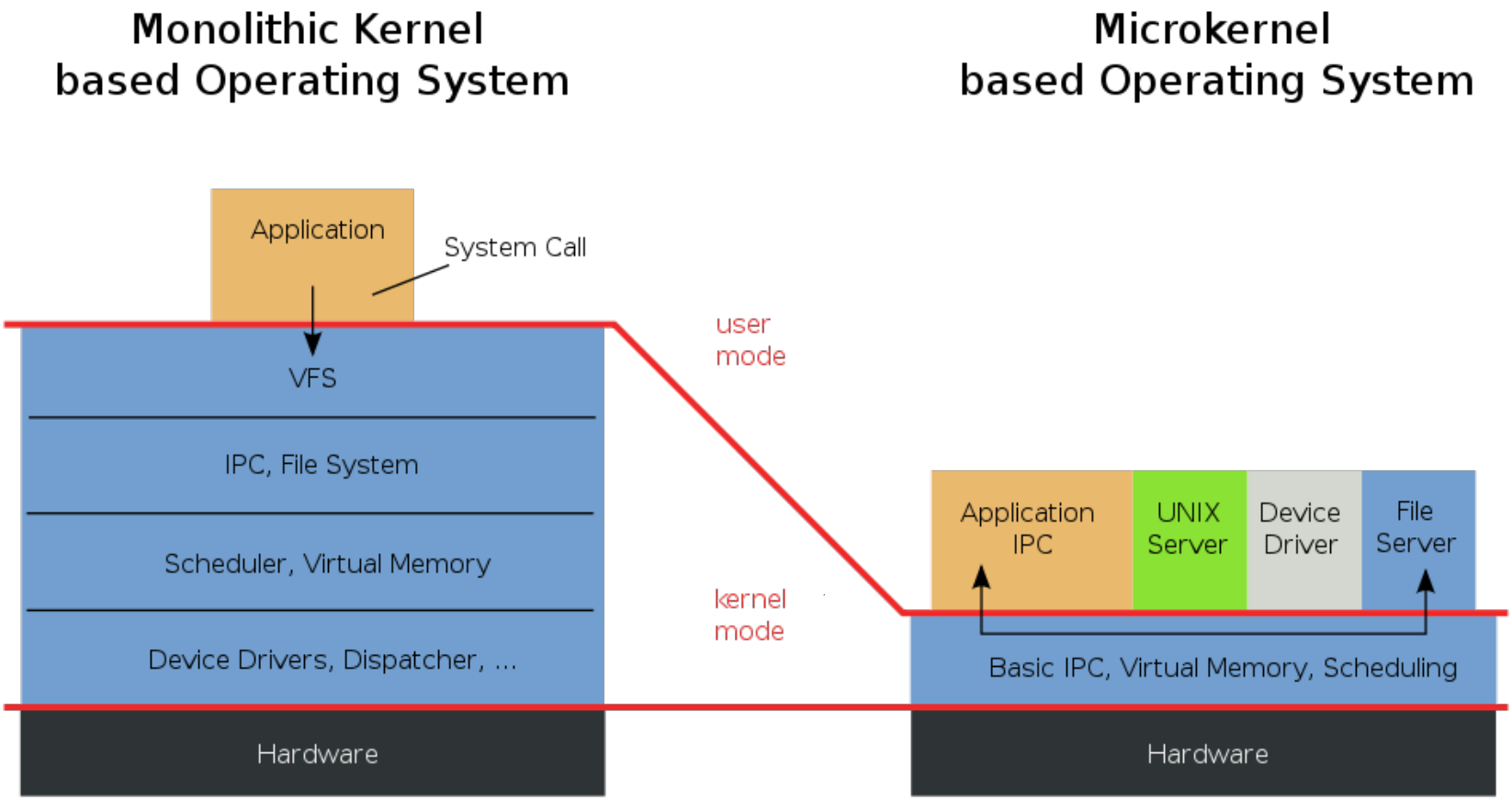
  - The *signal*() function chooses one of three ways in which receipt of the signal `sig` is to be subsequently handled.
    - If the value of `func` is `SIG_DFL`, default handling for that signal shall occur.
    - If the value of `func` is `SIG_IGN`, the signal shall be ignored.
    - Otherwise, the application shall ensure that `func` points to a function to be called when that signal occurs. An invocation of such a function is called a "*signal handler*".

POSIX and Win32 Calls Comparison

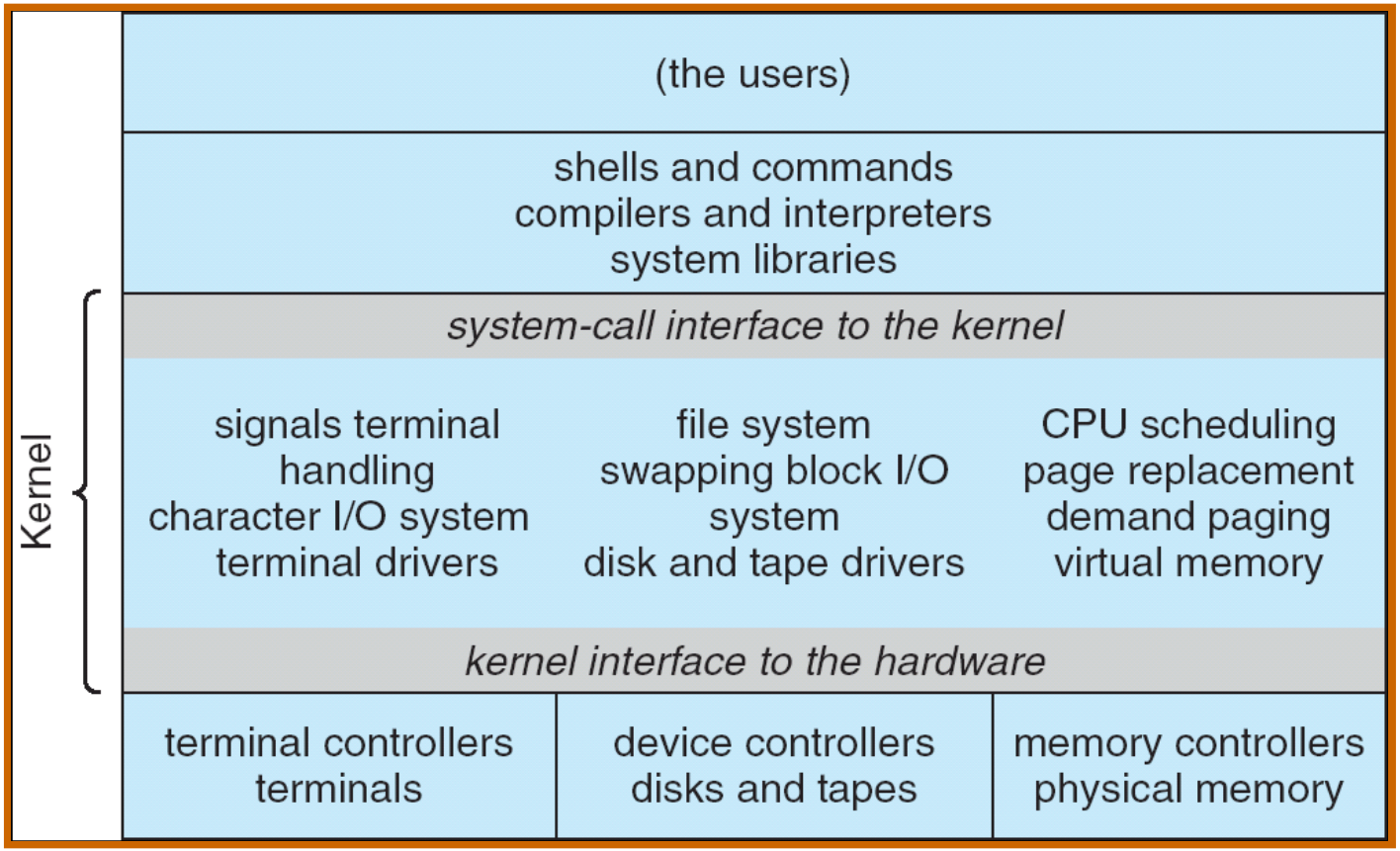
- Only several important calls are shown

POSIX	Win32	Description
fork	CreateProcess	Create a new process
wait	WaitForSingleObject	The parent process may wait for the child to finish
execve	--	CreateProcess = fork + execve
exit	ExitProcess	Terminate process
open	CreateFile	Create a new file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from an open file
write	WriteFile	Write data into an open file
lseek	SetFilePointer	Move read/write offset in a file (file pointer)
stat	GetFileAttributesExt	Get information on a file
mkdir	CreateDirectory	Create a file directory
rmdir	RemoveDirectory	Remove a file directory
link	--	Win32 does not support “links” in the file system
unlink	DeleteFile	Delete an existing file
chdir	SetCurrentDirectory	Change working directory

OS structure



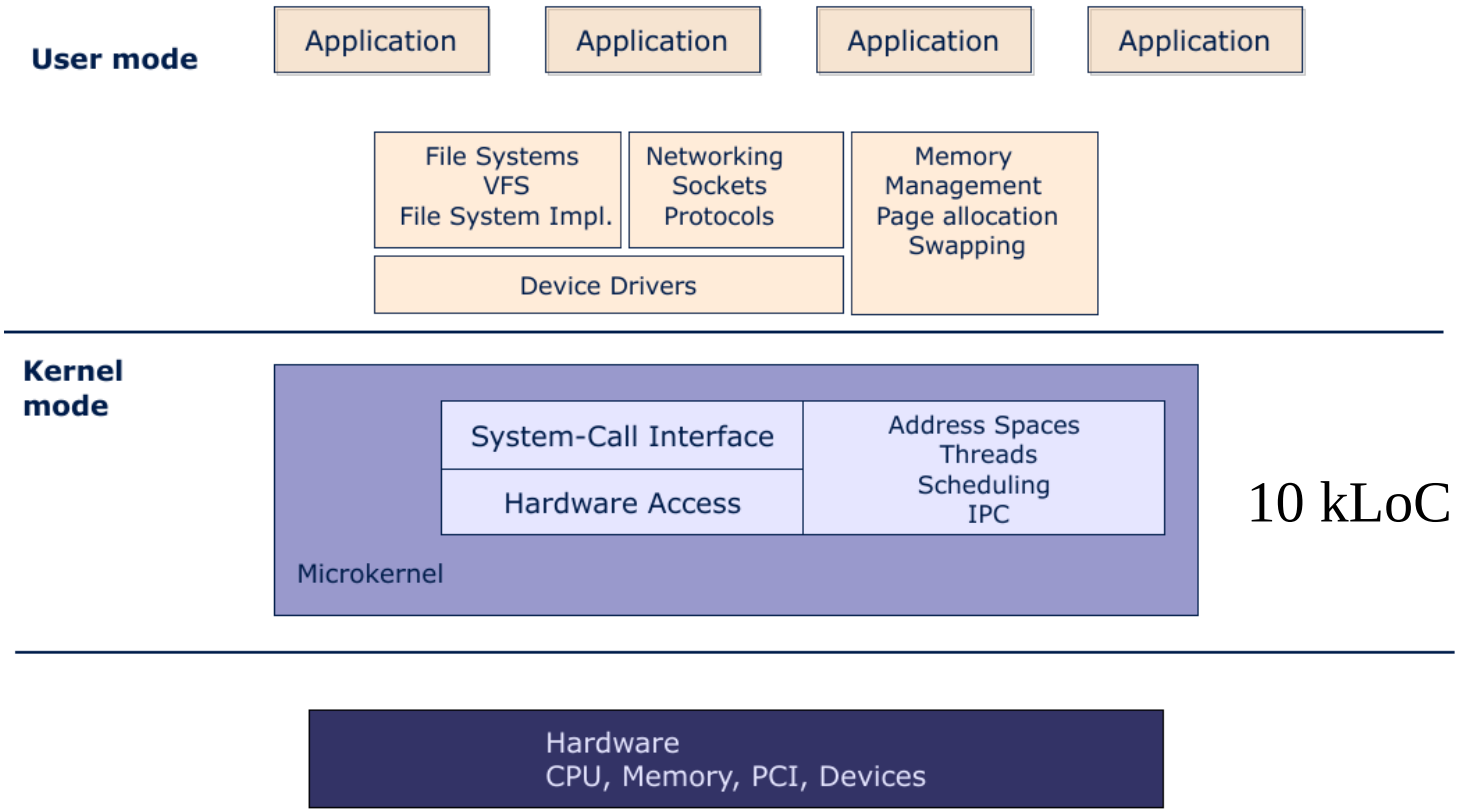
UNIX System Structure



Microkernel System Structure

- Moves as much from the kernel into "*user*" space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a µ-kernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication

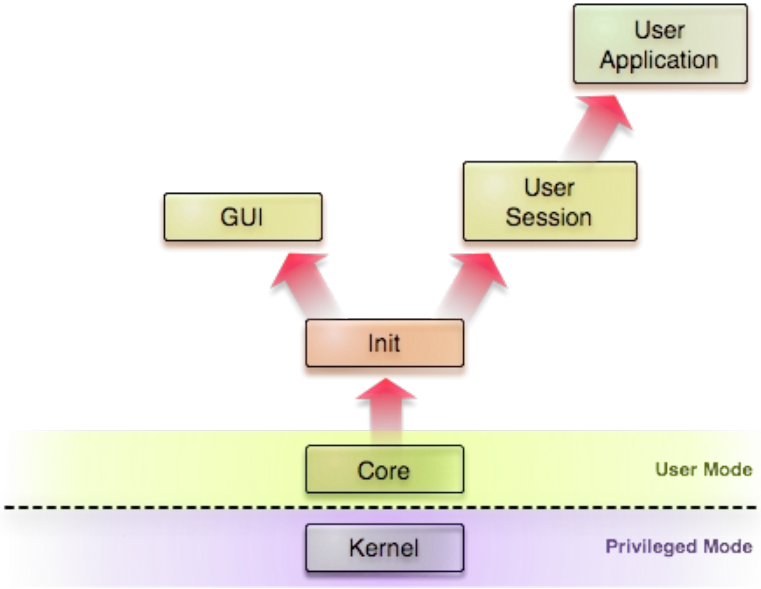
Real systém with µ-kernel L4Re

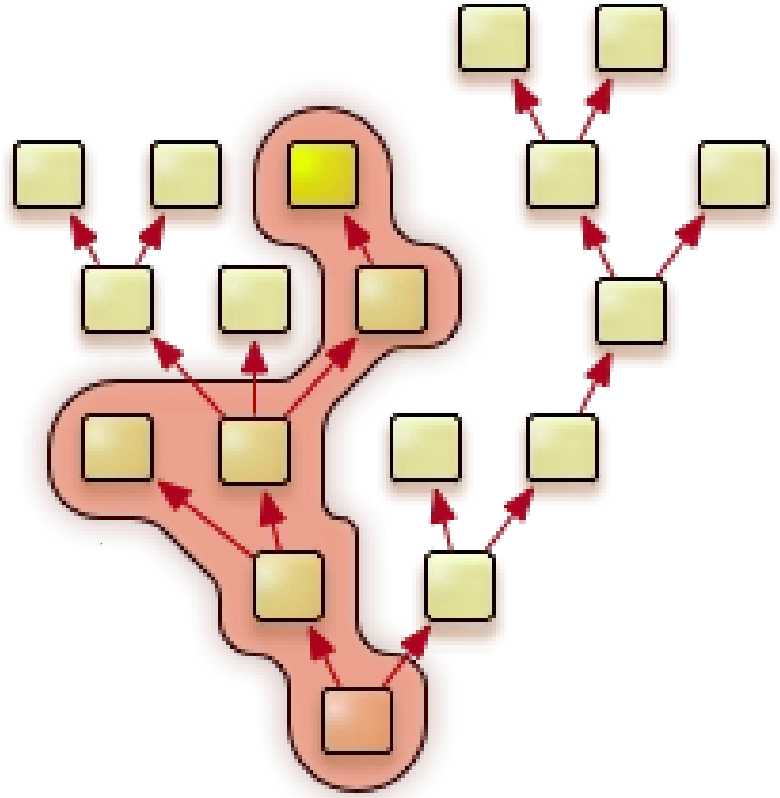


http://os.inf.tu-dresden.de http://www.kernkonzept.com/

Real system with µ-kernel Genode

http://genode.org/





Goal: reduce “Trustued computing base”

System calls µ-kernel NOVA
(http://hypervisor.org/)

- call
- reply
- create_pd
- create_ec
- create_sc
- create_pt
- create_sm
- revoke
- lookup
- ec_ctrl
- sc_ctrl
- pt_ctrl
- sm_ctrl
- assign_pci
- assign_gsi

- No other systém calls
- PD = protection domain = proces
- EC = execution context
- SC = scheduling context
- PT = portal
- SM = semafor

Windows NT - XP

Hybrid operating system