# Operating Systems
# Structure and Application Services

# Slide Plan

- Operating System Services - Slides 3
- User Operating System Interface – Slide 4-7
- System Calls Internals - Slides 8-14
- Types and sample usage of System Calls – Slides 14-21
- Operating System Design, Structure and Implementation – Slides 22-32
- Systems Utility Programs  for Operating System debugging – Slides 33-38

# Operating Systems Services - Review

- User Interface
- Program loading and execution
- Process start up and scheduling
- Address space separation
- Inter-process communication
- Concurrency control
- Dynamic memory allocation
- File system manipulation
- Input / Output Operations
- Networking
- Fault detection and recovery
- Auditing of OS data structures
- Protection of kernel data

# Operating Systems Services – Loading and Running an Application

- We have looked briefly at the type of OS services an application (let's continue with the basic example of a text editor) typically requires - slide 12 of previous module.

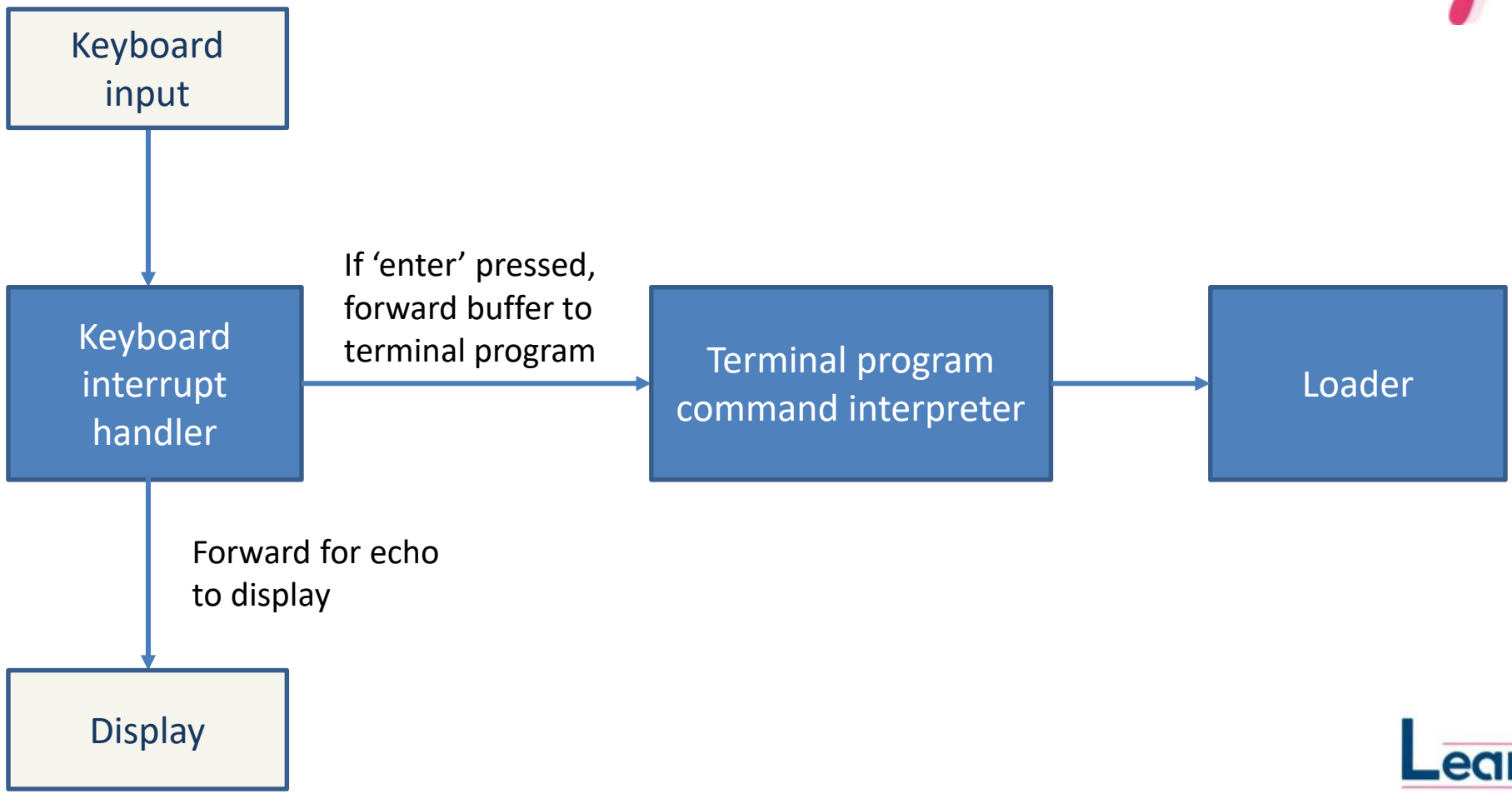
- There's a little bit that the OS needs to do before an application begins to run however:
  - ✓ It needs to have set up a command line/a GUI from where the command to load and run the application can be executed
  - ✓ The application code and data need to be loaded in an area in RAM

# Loading and Running an Application (contd.)

- **What exactly is a command line?**
  - ✓ It is an input from the keyboard given to a terminal program, supplied in a line format.
  - ✓ The terminal program, generally a kernel thread, processes the input and calls the corresponding command from a command list that has earlier been set up as part of command line initialization.
  - ✓ The command is a load-and-run-application command in this case.

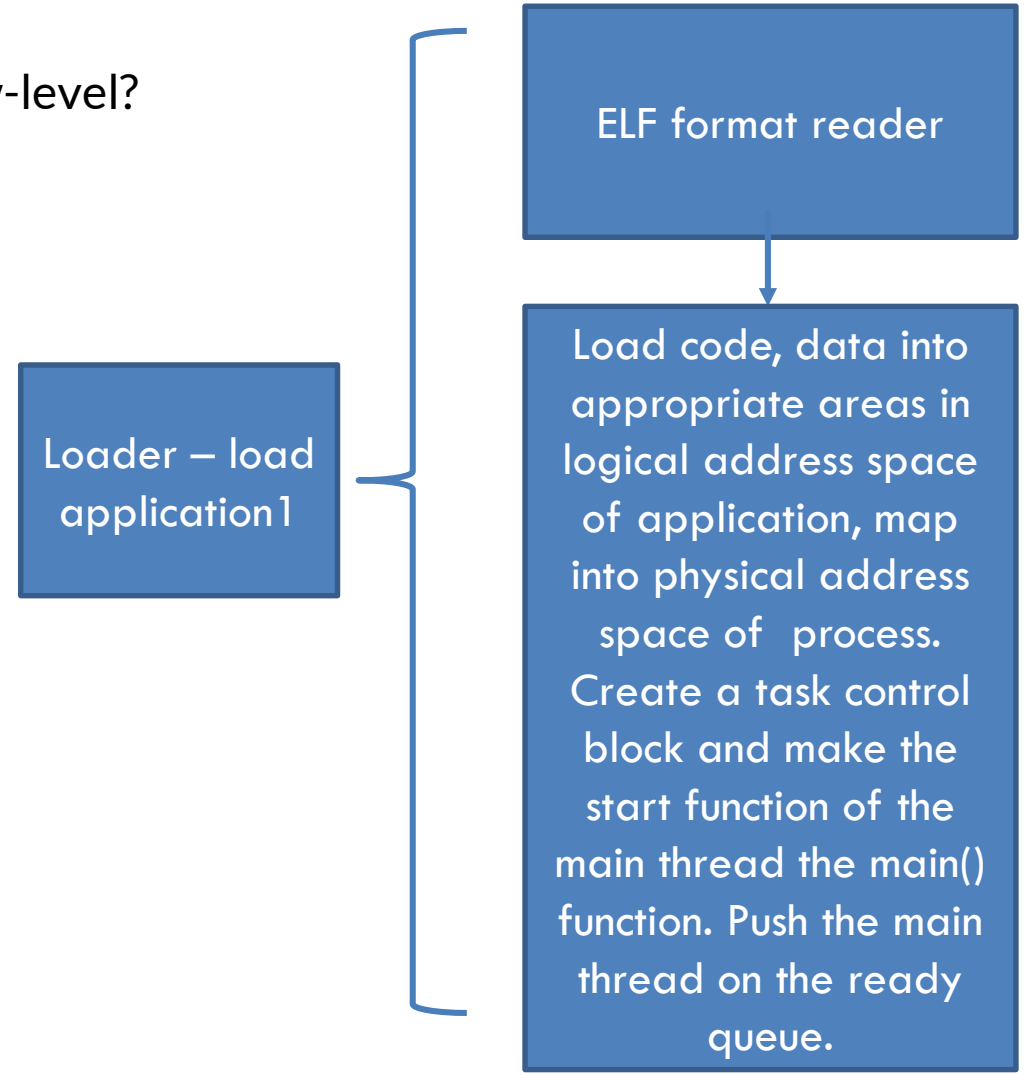
# Loading and Running an Application – Command Interpreter

**How does this work at a low-level?**



# Loading and Running an Application – Loader and Application Execution

How does this work at a low-level?



# What Does an Application Want from the OS?

**Let's look at the text editor slide from the previous module in more detail:**

- A text editor wants to read/write from a file on disk – so what type of information would it need to pass to the OS to get a specific file? (We'll look at just basic stuff here)
  - ✓ Obviously, the filename and directory path
  - ✓ A flag to open a new file if it's a new file that the editor wants to open
  - ✓ Permissions to create the file with

- What type of information would it pass to do a read/write?
  - ✓ Filename and directory path
  - ✓ File offset to start the read/write at

# How does it get what it wants? (System call internals)

**The call to open on Linux (or other Unix-like systems) would look something like:**

**int open(const char *_pathname_, int _flags_, mode_t _mode_);**

- The call returns a file descriptor id that is specific to the process. This id is referenced by subsequent application calls to read/write for the same file.
- Internal to the kernel, inside the open system call handler, this id is associated with a kernel file specific data structure (struct file) that along with the file offset has associations with physical page-related structures that help with reads/writes to/from the page cache. Dirtying the page cache triggers the flush out of the physical page to disk.

# What happens when you execute a System Call?

- **What is a system call?**
  - ✓ A system call is, under the wraps, a system call/trap instruction that is part of processor instruction set architecture
  - ✓ It takes arguments that indicate the filename(in the case of the text editor or a device path for other device operations) whose read/write/control the application is interested in, and the type of call (open/read/write/control in the case of a file operation or various other values for other system calls)
  - ✓ These arguments passed in by way of the system call library will be pushed into appropriate registers within the library function and passed via the registers to the actual system call instruction

# What happens when you execute a System Call?(contd.)

- **What is the purpose of a system call?**
  - ✓ To provide a generic hardware-agnostic interface for the application to call to access an OS service
  - ✓ It switches the processor to kernel mode, allowing privileged data access to kernel memory

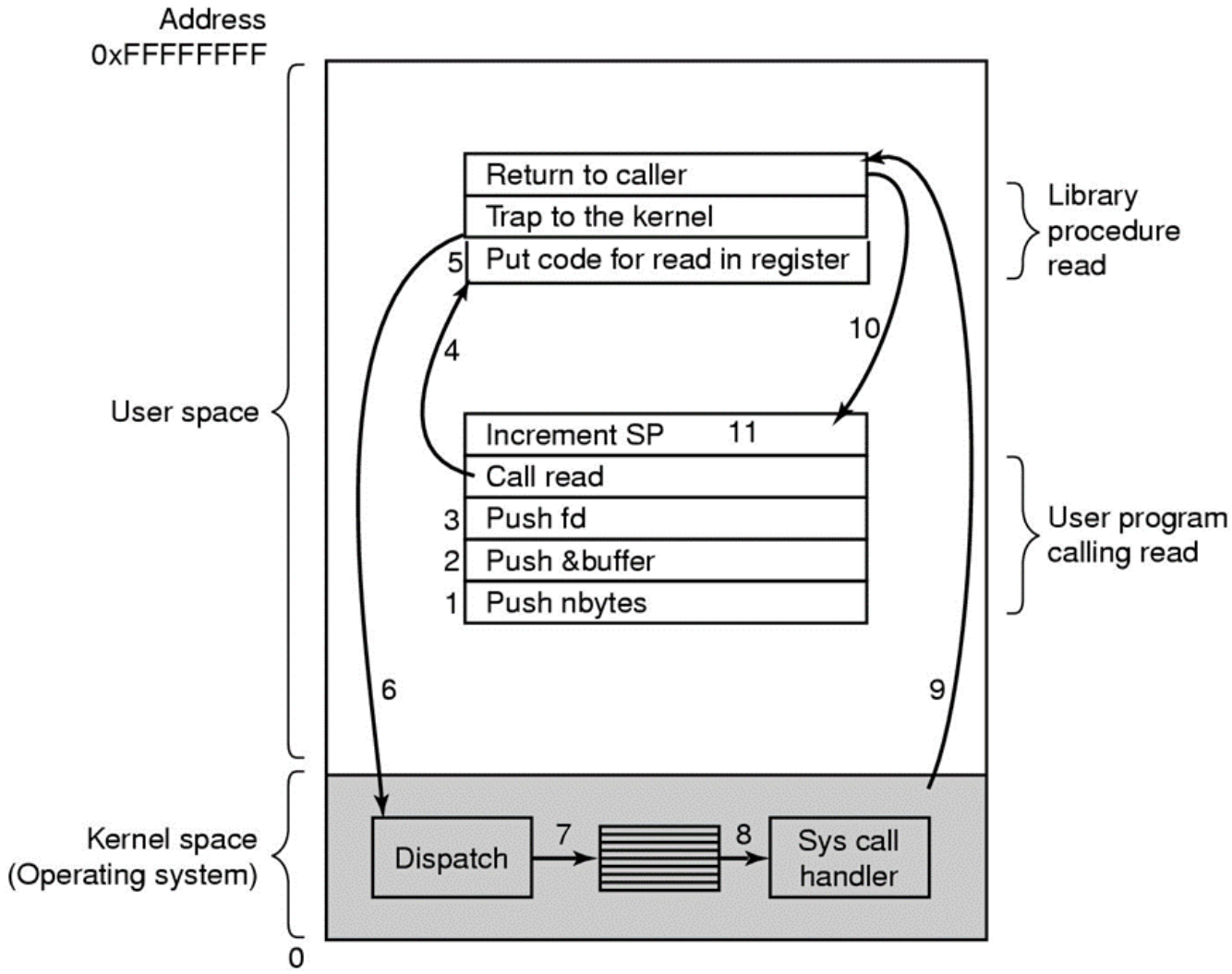
# Why do we need to switch from user to Kernel Mode?

- All logical to physical address mappings (page table entries) contain a protection bit.
- This is constantly (during the execution of each instruction) compared with the current execution mode (in an MSR in PPC, CPL register in x86).
- Only if these match (either both are kernel/protected mode or the execution mode is kernel and the PTE indicates user-space memory) will instructions execute without error. If these don't match (execution mode is user/unprotected and PTE is protected), a protection violation exception is reported.

# Why do we need to switch from user to Kernel Mode? (contd.)

- The reason the kernel data space is kept protected is obvious – you don't want errant application code to be writing to kernel space in user mode
- When you execute a system call/trap instruction, the current execution mode switches to kernel mode so that kernel mode data can be accessed.
- When you return from the exception by way of a return from interrupt instruction, the execution mode switches back to user/unprotected mode

# System Call – A Control Flow Sequence



# A Sample List of System Calls

| %rax | Name | Entry point | Implementation |
|---|---|---|---|
| 0 | read | sys_read | fs/read_write.c |
| 1 | write | sys_write | fs/read_write.c |
| 2 | open | sys_open | fs/open.c |
| 3 | close | sys_close | fs/open.c |
| 4 | stat | sys_newstat | fs/stat.c |
| 5 | fstat | sys_newfstat | fs/stat.c |
| 6 | lstat | sys_newlstat | fs/stat.c |
| 7 | Poll | sys_poll | fs/select.c |
| 8 | Lseek | sys_lseek | fs/read_write.c |
| 9 | Mmap | sys_mmap | arch/x86/kernel/sys_x86_64.c |
| 10 | Mprotect | sys_mprotect | mm/mprotect.c |
| 11 | Munmap | sys_munmap | mm/mmap.c |
| 12 | Brk | sys_brk | mm/mmap.c |

# A Sample List of System Calls (contd.)

| %rax | Name | Entry point | Implementation |
|---|---|---|---|
| 23 | select | sys_select | fs/select.c |
| 24 | sched_yield | sys_sched_yield | kernel/sched/core.c |
| 25 | mremap | sys_mremap | mm/mmap.c |
| 26 | msync | sys_msync | mm/msync.c |
| 27 | mincore | sys_mincore | mm/mincore.c |
| 28 | madvise | sys_madvise | mm/madvise.c |
| 29 | shmget | sys_shmget | ipc/shm.c |
| 30 | shmat | sys_shmat | ipc/shm.c |
| 31 | shmctl | sys_shmctl | ipc/shm.c |
| 44 | sendto | sys_sendto | net/socket.c |
| 45 | recvfrom | sys_recvfrom | net/socket.c |
| 46 | sendmsg | sys_sendmsg | net/socket.c |
| 47 | recvmsg | sys_recvmsg | net/socket.c |

# Sample Applications using System Calls : Open, read, write, close

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h> //needed for open
#include <sys/stat.h>  //needed for open
#include <fcntl.h>     //needed for open

int main (int argc, char *argv[])
{
      int inFile;
      int n_char=0;
      char buffer[10];

      inFile=open(argv[1],O_RDONLY);
      if (inFile==-1)
      {
            exit(1);
      }
      //Use the read system call to obtain 10 characters from inFile
      while( (n_char=read(inFile, buffer, 10))!=0)
      {
            //Display the characters read
            n_char=write(1,buffer,n_char);
      }

      close (inFile);
      return 0;
}
```

# Sample Applications using System Calls: fstat

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h> //needed for open
#include <sys/stat.h>  //needed for open
#include <fcntl.h>     //needed for open

int main (int argc, char *argv[])
{
      int inFile, err;
      struct stat statBuf;

      inFile=open(argv[1],O_RDONLY);
      if (inFile==-1)
      {
            exit(1);
      }
      //Use the read system call to obtain 10 characters from inFile
     err = fstat(FD, &statBuf);
     if(err == -1) /* fstat failed? */
         exit(-1);

      printf("The number of blocks = %d\n", statBuf.st_blocks);
      close (inFile);
      return 0;
}
```

# Sample Applications Using System Calls: lseek

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>

int main()
{
    int file=0;
    if((file=open("testfile.txt",O_RDONLY)) < -1)
        return 1;

    char buffer[19];
    if(read(file,buffer,19) != 19)  return 1;
    printf("%s\n",buffer);

    if(lseek(file,10,SEEK_SET) < 0) return 1;

    if(read(file,buffer,19) != 19)  return 1;
    printf("%s\n",buffer);

    return 0;
}
```

# Sample Applications Using System Calls: mmap, munmap

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
int fail(char *filename, int linenumber) {
 fprintf(stderr, "%s:%d %s\n", filename, linenumber, strerror(errno));
 exit(1);
 return 0; /*Make compiler happy */
}
#define QUIT fail(__FILE__, __LINE__ )
int main() {
 // We want a file big enough to hold 10 integers,
 // though the mapping is actually rounded up to a multiple of the page size (4kB on x86 Linux)
 int size = sizeof(int) * 10;
 int fd = open("data", O_RDWR | O_CREAT | O_TRUNC, 0600); //6 = read+write for me!
 lseek(fd, size, SEEK_SET);
 write(fd, "A", 1);
```

# Sample Applications Using System Calls: mmap, munmap (contd.)

```
void *addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  printf("Mapped at %p\n", addr);
  if (addr == (void*) -1 ) QUIT;
  int *array = addr;
  array[0] = 0x12345678;
  array[1] = 0xdeadc0de;
  munmap(addr, size);
  return 0;

}
```

# How do you go about designing an Operating System?

Let’s consider the absolutely mandatory parts of an Operating System first:

- The scheduler
- The Memory Manager
- The Interrupt/Exception handlers

Some basic IO and debug mechanisms are mandatory as well, but let’s focus on the above parts for now.

# How do you go about designing an Operating System(contd.)?

1. **First question you have got to ask is about the type of applications that are expected to run on this OS.**

You have to look at:

- ✓ The nature of the applications from a latency point of view. How big a delay is acceptable from an application data processing point of view? What is the expected throughput of application tasks? Are all applications equally time critical? Is starvation of lower priority tasks acceptable? The answers to these questions will decide the scheduling scheme of the OS.

# How do you go about designing an Operating System (contd.)?

- If starvation of lower priority tasks is absolutely not acceptable, and if delays of the order of milliseconds are acceptable, you might  opt for a fair-share scheduler.
- If you can compromise on relative starvation of lower priority tasks and if delays of the order of milliseconds are acceptable, you might opt for a priority-based scheduler.
- If even delays of the order of milliseconds are not acceptable, i.e. the tasks are extremely time critical, you might opt for a deadline-based scheduler, like the earliest deadline first (EDF) scheduler.

# How do you go about designing an Operating System (contd.)?

2. **Second question you have to ask is how fault tolerant the system should be.**

You have to look at:

- ✓ Whether there should be system recovery for rarest of rare scenarios, i.e. what is known as five 9s, six 9s reliability in telecom carrier grade systems. This requires extensive hardware support and proprietary CPU-memory buses and controllers, with kernel-level software primitives that support isolation of CPUs and memory elements.
- ✓ If the fault scenarios are in the software domain, you need kernel audit tasks auditing ready queues, software watchdogs looking for starvation and similar kernel-level checks for memory management data structure inconsistencies.

# How do you go about designing an Operating System (contd.)?

- If CPU resets are the favoured recovery mechanisms in some scenarios, then migration of the tasks on the offending CPU's ready queue need to be migrated to suitably less busy CPUs etc.

- Software upgrades are a very critical part of highly fault tolerant systems. Active and passive storage partitions need to be supported and boot/kernel file consistency checks need to be used, and fallback mechanisms need to be in place in case of errors. In-field patching of code might also need to be supported.

# How do you go about designing an Operating System (contd.)?

3. **Third question is what type of operating environment you are going to run the OS and application:**

- ✓ Most embedded systems where the software execution environment is more controlled - definite set of applications requiring very predictable execution on proprietary hardware – you will opt for simplicity in OS design and not worry about flexibility as much.
- ✓ For instance, you might opt to always execute at kernel level, ignoring the extra protection of user-kernel space separation.
- ✓ Modules will get loaded into areas defined at build time.
- ✓ There will be no address space related context switch expenses when the scheduler switches between tasks.
- ✓ On more general purpose systems, the reverse design assumptions hold and you might opt for flexibility and protection over simplicity and time saving.

# How do you go about designing an Operating System(contd.)?

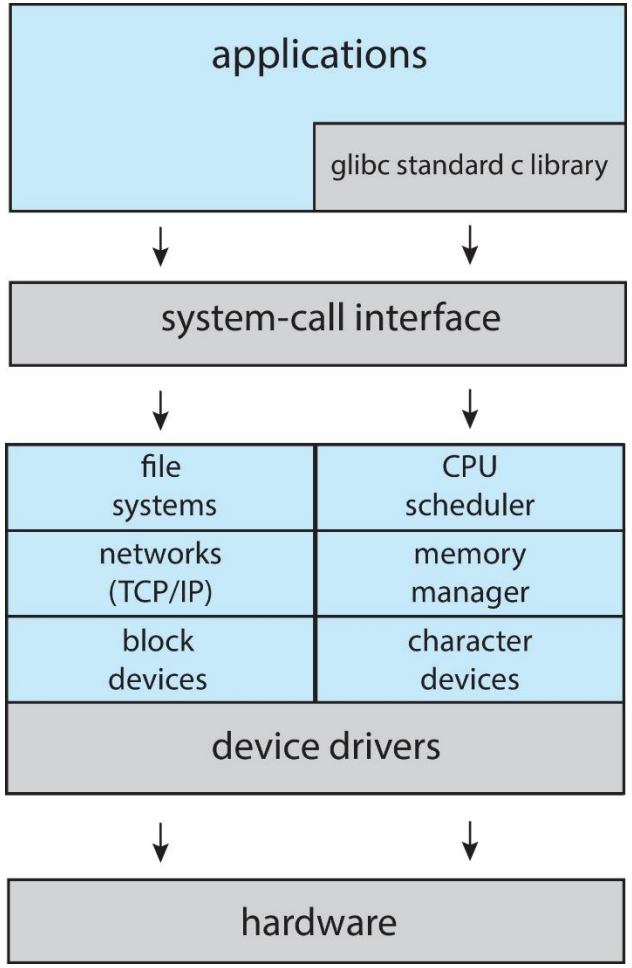
4. **What type of modularity does the OS need to support?**

- ✓ Can we afford some parts of the kernel to be in user space?
- ✓ For instance, can the File System and some Device Drivers be in user space and can these services be accessed by message passing rather than system calls?
- ✓ We will have to compromise a bit on performance or develop specialised fast messages specifically for these OS services.
- ✓ It will gives us flexibility in terms of easier debugging in user space and less time spent in implementation because user space development is an easier option.

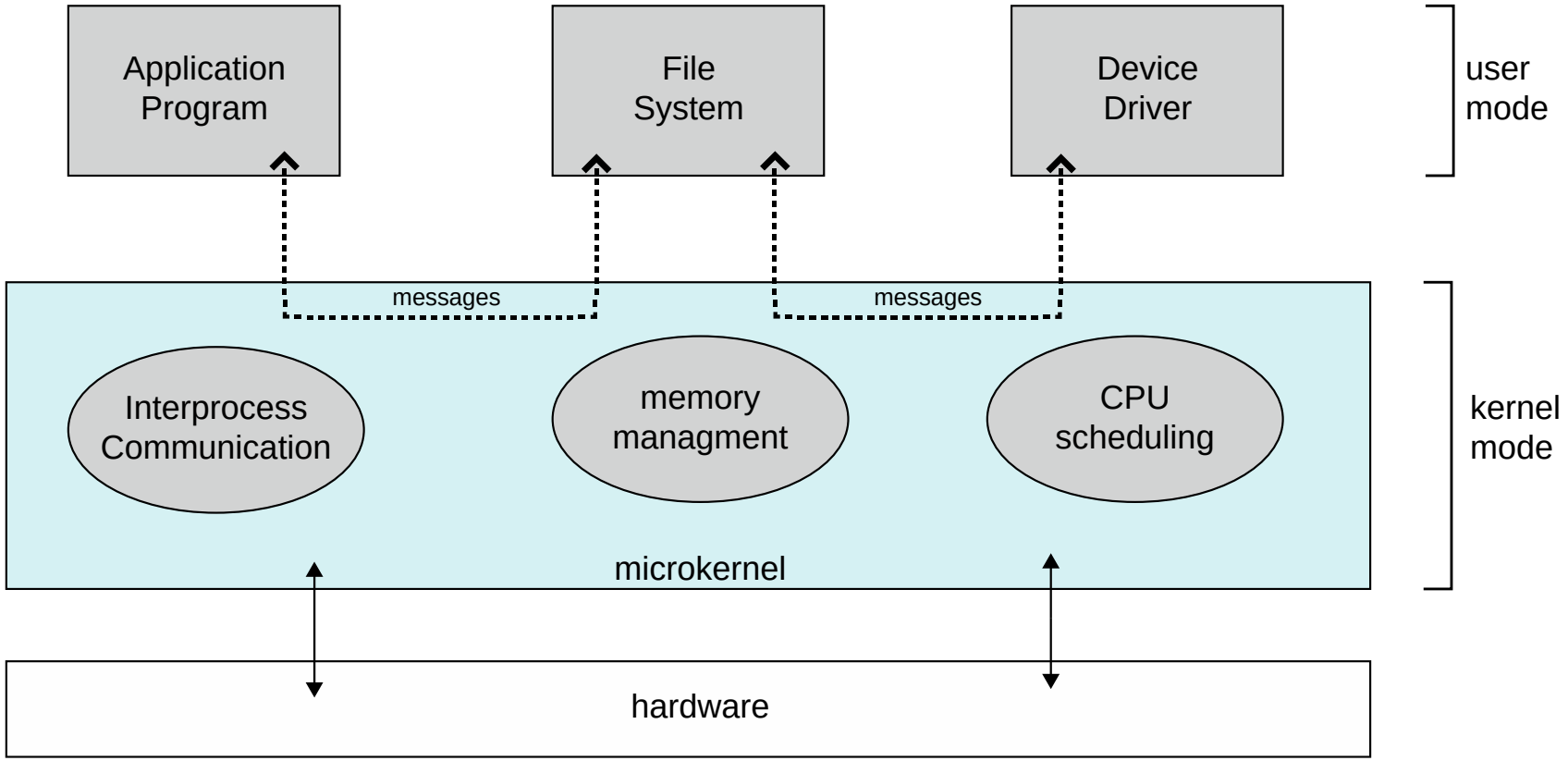
# How do you go about designing an Operating System(contd.)?

- It gives us the option of building kernels with bare functionality and then add on services as required. There might be some systems that do not require sophisticated file system support – there you can go for a more basic raw file system in the kernel and leave out the more sophisticated file system module from the kernel build.

# Linux – Monolithic Design



# QNX– Microkernel Design



# How do you go about designing an Operating System (contd.)?

5. **What type of debug functionality needs to be factored in?**

- ✓ Kernel debugging is a very different kettle of fish when compared to general application-level debugging. You need to be much more aware of processor and OS architecture.

- ✓ Highly fault tolerant embedded systems will need in-field type debugging tools like tracepoints and not just core dumps and breakpointing in offline type of fault reproduction environments.

# OS Utility Programs for Debugging

- Some sample OS (in this case, Linux) utility programs that help in kernel debugging are given below and some are discussed in more detail in the following slides:

  - ✓ Compile, link, OS kernel build tools (Linux)
  - ✓ Kernel module creation - compile, link makefile
  - ✓ Ftrace kernel tracing
  - ✓ Using gdb, objdump
  - ✓ Memory information (free –m, ….)
  - ✓ Process information (top, ….)
  - ✓ CPU information
  - ✓ /proc files
  - ✓ Dmesg
  - ✓ tcpdump

# Building and Booting an Operating System (Linux)

**Download latest source from www.kernel.org (stable/mainline)**

1. $ sudo apt-get update
2. $ sudo apt-get install libncurses5-dev
3. $ make menuconfig
4. Make changes as follows: Before you go too far, use the “General Setup” menu and the “Local version” and “Automatically append version info” options to add a suffix to the name of your kernel, so that you can distinguish it from the “vanilla” one. You may want to vary the local version string, for different configurations that you try, to distinguish them also.
5. Change EXTRAVERSION in Makefile to changed linux kernel name suffix
6. make (to build the kernel)
7. make modules_install (to build required kernel modules)
8. make install (to install the kernel)
9. When you boot next, Grub will give you the option to load the new kernel (as step 8 includes an update to the Grub list)

# Building a Kernel Module (Linux)

Use a makefile of the following format to build a kernel module:

```
obj-m += new_kernel_module.o
all:
        make -C /lib/modules/Linux-new-version-name/build M="new kernel module
        directory path" modules

clean:
        make -C /lib/modules/build M="new kernel module directory path" clean
```

# Using Ftrace (Linux)

- Ftrace is a crisp tracing tool for use in kernel modules and in the kernel.

- It needs a call to trace_printk with the same arguments as a regular printk from code that is being traced. It works a lot better than printk's especially in code that gets called very frequently at the kernel level. Printk's since they are flushed to the terminal every so often are more time consuming and tend to get overwritten if the calling code is a kernel-level hot spot. Ftrace uses a ring buffer accessed via the debugfs file system and is dumped to display only on user demand.

- At kernel build time, you need to make sure that the .config file has CONFIG_DYNAMIC_FTRACE enabled – by default, it is.

- At trace collection time, tracing needs to be reset, set to function tracing (or other options) and enabled by:

```
> cd /sys/kernel/debug/tracing
> echo 0 > tracing_on
> cat available_tracers
  function_graph function sched_switch nop
 > echo function > available_tracers
 > echo 1 > tracing_on
```

# Using GDB with Core Dumps (Linux)

- First configure core dumps to be of unlimited size: >ulimit -c unlimited
- Here's a simple program that references a NULL pointer:

```
#include <stdio.h>
main(){
  int* null_ptr = NULL;
 *null_ptr = 5;
}
```

> gcc sample.c –g –o sample.o

>./sample.o gives you a core dump

# Using GDB with Core Dumps (contd.)

```
> gdb –c core
 (gdb) bt
#0  0x0000000004004e6 in ?? ()
#1  0x0000000000400500 in ?? ()
#2  0x00007f605ca03830 in ?? ()
#3  0x0000000000000000 in ?? ()
(gdb) display $rax
1: $rax = 0
```

# Using GDB with Core Dumps (contd.)

>objdump –SD sample.o | more gives you:

```
4004e2: 48 8b 45 f8         mov   -0x8(%rbp),%rax
4004e6: c7 00 05 00 00 00   movl  $0x5,(%rax)
4004ec: b8 00 00 00 00      mov   $0x0,%eax
```

The gdb backtrace indicated 0x00000000004004e6 as the offending instruction address and a dump of register eax indicates it has a zero value.

movl $0x5,(%rax) is the offending instruction and if we look at a combination of source and assembly, we see this is the disassembly of the C instruction

*null_ptr = 5;

giving us the complete picture of what the bug is.

# Thank You!