Day 8
Advanced Linux Programming
Part I

Joel Isaacson
Ascender Technologies Ltd.
http://ascender.com

Today's Lectures

- Today we will concentrate on examining the Linux API.
- This realm is normally called “system programing”.
- Application programming is usually independent of the underlying operating system.

Application Programming

- In the last few years we have witnessed a trend away from system level programming towards high level application programming.
- This trend includes “managed code” such as Java or C#.
- It also includes web based approaches such as JavaScript of PHP.

System Calls

- System programming uses system calls or syscalls.
- Syscalls are invoked from “user space” and they execute in “kernel space”.
- Services like storage, memory, network, process management etc. are managed by the operating system.

System Calls

- For example if a user process wants to read a file, it will have to make 'open' and 'read' system calls.
- Generally system calls are not called by processes directly.
- C library provides an interface to all system calls.

System Calls

- It is not possible to directly link user-space applications with kernel space.
- For reasons of security and reliability user programs can not be allowed to directly execute kernel code.
- The user must somehow “signal” to the kernel that it wishes to invoke some code.

System Calls

- The application tells the kernel which system call to execute and what parameters to use via machine registers.
- Each system call is numbered.
- For example: On the x86 architecture the “open()” syscall is number 5.

What happens in a system call?

- Kernel code is run on request of a user process.
- On the X86 architecture this code runs in ring 0 (with current privilege level, CPL- 0), which is the highest level of privilege in x86 architecture.
- All user processes run in ring 3 (CPL 3).

What happens in a system call?

- To implement system call mechanism, what we need is:
  1) a way to call ring 0 code from ring 3
  2) some kernel code to service the request.

Old Syscall Invocation

- Until some time back, linux used to implement system calls on all x86 platforms using software interrupts.
- To execute a system call, user process will copy desired system call number to %eax and will execute 'int 0x80'.

Old Syscall Invocation

- This will generate interrupt 0x80 and an interrupt service routine will be called.
- For interrupt 0x80, this routine is an "all system calls handling" routine. This routine will execute in ring 0.

New Syscall Invocation

- It was found out that this software interrupt method was much slower on Pentium IV processors.
- To solve this issue, Linus implemented an alternative system call mechanism to take advantage of SYSENTER/SYSEXIT instructions provided by all Pentium II+ processors.

SYSENTER/SYSEXIT

- The SYSENTER instruction is part of the "Fast System Call" facility introduced on the Pentium II processor.
- The SYSENTER instruction is optimized to provide the maximum performance for transitions to protection ring 0 (CPL = 0).
- The vdso segment contains the optimized syscall routines for the CPU architecture.

LD_SHOW_AUXV=1 /bin/true

The C Library

- The C library (libc) is used by virtually all applications.
- Even if you are using another language libc is usually used one way or another.
- The C library usually used is the GNU library, glibc.

The C Library

- This library provides wrappers for system calls.
- It also provided the standard I/O (stdio) library.
- It also contains the POSIX standard libraries.

C Compiler

- The standard C compiler is the GNU C compiler (gcc).
- Over time support for many languages has been added.
- Gcc supports many different processors

API's and ABI's

- Linux's application programming interface (API) is the machine independent definition of the interface to the kernel.
- Linux's application binary interface (ABI) is the machine dependent definition of the interface to the kernel.

API's

- It is common to call an API a contract for services.
- It is a one way contract. The user can decide to use the API or not.
- The user has no influence on these API's.

ABI's

- Whereas an API defines a source interface the ABI defines the low-level binary interface.
- A stable ABI allows us to compile a program and to be sure that it will continue to run as long as the ABI is not compromised.
- Each architecture (X86, ppc) has different ABI's.

Standards - Posix

- Posix (Portable Operating System Interface) was defined in the mid-80's by the IEEE to standardize the system-level interface of Unix.
- Various revisions and extensions to the POSIX standard have been release over the years.

Standards - SUS

- The Open Group released a standard called the Single Unix Standard (SUS) in 1994.
- The third version of SUS was released in 2002.
- Today the SUS incorporates the POSIX standard.

C Language Standards

- Ritchie's and Kernighan's, *The C Programming Language*, served as the informal standard for the language for many years.
- In 1989 the American National Standards Institute (ANSI) issued a standard for the C language.

Linux and Standards

- Linux is thought to be POSIX (SUS) compatable.
- It has not been formally tested.
- Linux's API has many additional non-POSIX features.
- Linux's kernel API is very fluid.
- Linux's user API is very stable.

Linux Programming Concepts

- Files and Filesystems
- Processes
- Users and Groups
- Permissions
- Signals
- Interprocess Communications

Files and FIlesystems

- Regular files
- Directories and links
  - Hard Links
  - Symbolic Links
- Special files
- Namespaces

Processes

- Object code in execution, data, resources, state and virtual computer.
- Usually starts life as a forked process.
- The object code is switched via “execve”.
- Threads are implemented as Linux processes.

Process Hierarchy

- Each process is identified by a unique integer.
- Processes form a strict hierarchy, know as the process tree.
- The process tree is rooted at the first process.
- New processes are created by clone.

Users and Groups

- The permission model of Linux is based on unique positive integers associated with each process called *user* and *group* id's.
- Linux also supports Access Control Lists (ALC's).

Signals

- Signals are a mechanism for one-way asynchronous notification.
- Signals typically alert a process as to some event, such as segmentation faults, or a user pressing ^C.
- Most signals (accept SIGKILL and SIGSTOP) can be caught.

Error Handling

- Most system calls return a negative value on failure.
- The special variable, errno, can be used to understand exactly what error is indicated by the kernel.
- The variable, errno, is thread-local.

File I/O

- The standard C I/O library (fopen ...) is not supported by the Linux (nor Unix) kernel.
- This library is layered over the more basic POSIX I/O library (open ...).
- The standard C I/O library is buffered. Each I/O operation does not necessarily translate into kernel I/O operations.

File Descriptors

- open returns a small integer, the file descriptor, that is interpreted as an index to the per-process file table in the kernel.
- By default, upon fork, a process inherits the file table entries of its parent.
- It is important to understand that the file offset pointers are shared between the parent and child.

Stdin, Stdout, Stderr

- By convention the three numerically lowest file descriptors are connected to
  - 0 - The standard input
  - 1 - The standard output
  - 2 - The standard error

open(name, flags, mode)

- name: is the file name
- flags: tells how the file is opened
- mode: this gives the file accessibility (RWE) mode when a file is created.

flags

- The flags must include one of the following:
  - O_RDONLY
  - O_WRONLY
  - O_RDWR

flags

- In addition, zero or more file creation flags and file status flags can be bitwise-or'd in flags.
- The file creation flags are
  - O_CREAT
  - O_EXCL
  - O_NOCTTY
  - O_TRUNC

flags

- Other file flags are:
  - O_APPEND
  - O_ASYNC
  - O_CLOEXEC
  - O_DIRECT
  - O_DIRECTORY
  - O_LARGEFILE

flags

- Other file flags are:
  - O_NOATIME
  - O_NOFOLLOW
  - O_NONBLOCK
  - O_SYNC

read(fd, buf, len)

- read() will read from the opened file up to *len* bytes into the location *buf* starting from the current file offset.
- The file offset is increased by the number of byte read.
- Any return value in the range [-1, *len*] can be returned.

Write – Append Mode

- If the file is opened with the O_APPEND flag writes will be done atomically to the end of the file.
- This is the only way you can guarantee that log files are written properly.

pread and pwrite

- Read from or write to a file descriptor at a given offset.
  - pread(fd, buf, count, offset);
  - pwrite(fd, buf, count, offset);
- The file offset is not changed.

readv and writev

- Readv and writev allow scather/gather I/O to be performed with one operation.
- Instead of a character buffer argument a list of iovec's is used:

```
struct iovec {
  void  *iov_base;    /* Starting address */
  size_t iov_len;     /* Number of bytes to transfer */
};
```

Delayed Writes

- It must be understood that Linux aggressively uses memory to buffer delayed reads.
- Normally this is not noticeable
- If the system crashes the written data might not be consistent.
- Also the data written out at any one time might be incomplete and out of order.

Synchronized I/O

- Kernel buffered I/O provide huge performance gains.
- Sometime we want to know that data has been written out.
- The syscalls fsync() and fdatasync() can be used to do this.
- sync() globally writes out all buffered data.

Direct I/O

- For certain applications you can bypass the kernel caching and buffering algorithms.
- You can use the O_DIRECT flag in the open to do this.
- This is typically used in database applications.

Seeking with lseek

- You can use lseek (or lseek64) to move the file offset.
- You can even seek past the end of a file and the file will be extended with zeros (holes) in the skipped portion.
- Skipped areas of the file are not actually allocated zero'ed blocks but the zero values are understood.

truncate() or ftruncate()

- These functions can be used to either extend or truncate a file.

Multiplexed I/O

- select()
- poll()
- epoll()

Implementation - VFS

- The Virtual FileSystem (VFS) allows the kernel to plug in many different actual filesystems.
- This make it easy to give the different filesystems POSIX semantics.
- A user space read() will eventually translate into a call of a VFS read entry.

The Page Cache

- The page cache uses memory to store parts of the file so that not every access will have to go out to the physical disk.
- The page cache is the first place the kernel look in order to find the data.
- The page cache uses both temporal and spatial locality to make reads more efficient.

Page Writeback

- In order to not block on every write, the Linux kernel maintains write buffers.
- The Linux kernel tries to use memory aggressively to buffer disk writes.
- The dirty pages in the page cache are written back to disk by the *pdflush* kernel threads.

mmap

- You can use mmap to map a file for I/O
- The main difficulty with using mmap is that:
  - The file size granularity is the page size.
  - You can't easily extend the file szie woth mmap.

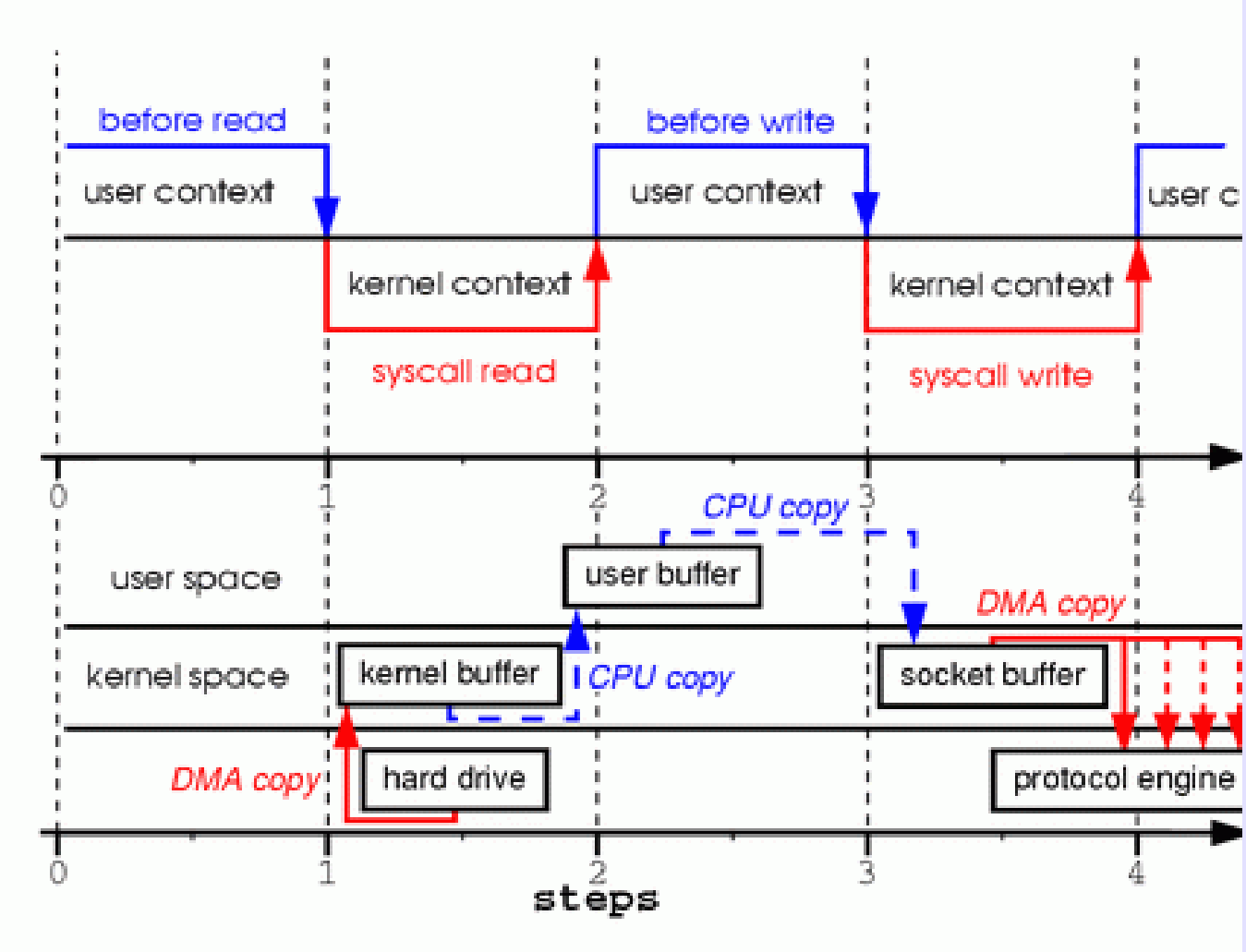
Zero Copy

- Sometimes we can conduct our business without actually transferring data to userspace.
- The classic example of an application that can benefit from “zero copy” is an Ftp server or a Web server.

Two Copy

- Typically in these applications we read data from a file on disk and write it out on a Tcp/Ipp socket.
- The obvious read(file,...)/write(sock,..) code would:
  - Transfer the data to the kernel data cache
  - Read the data to user space
  - Write the data to kernel space
  - Copy the data to the network device

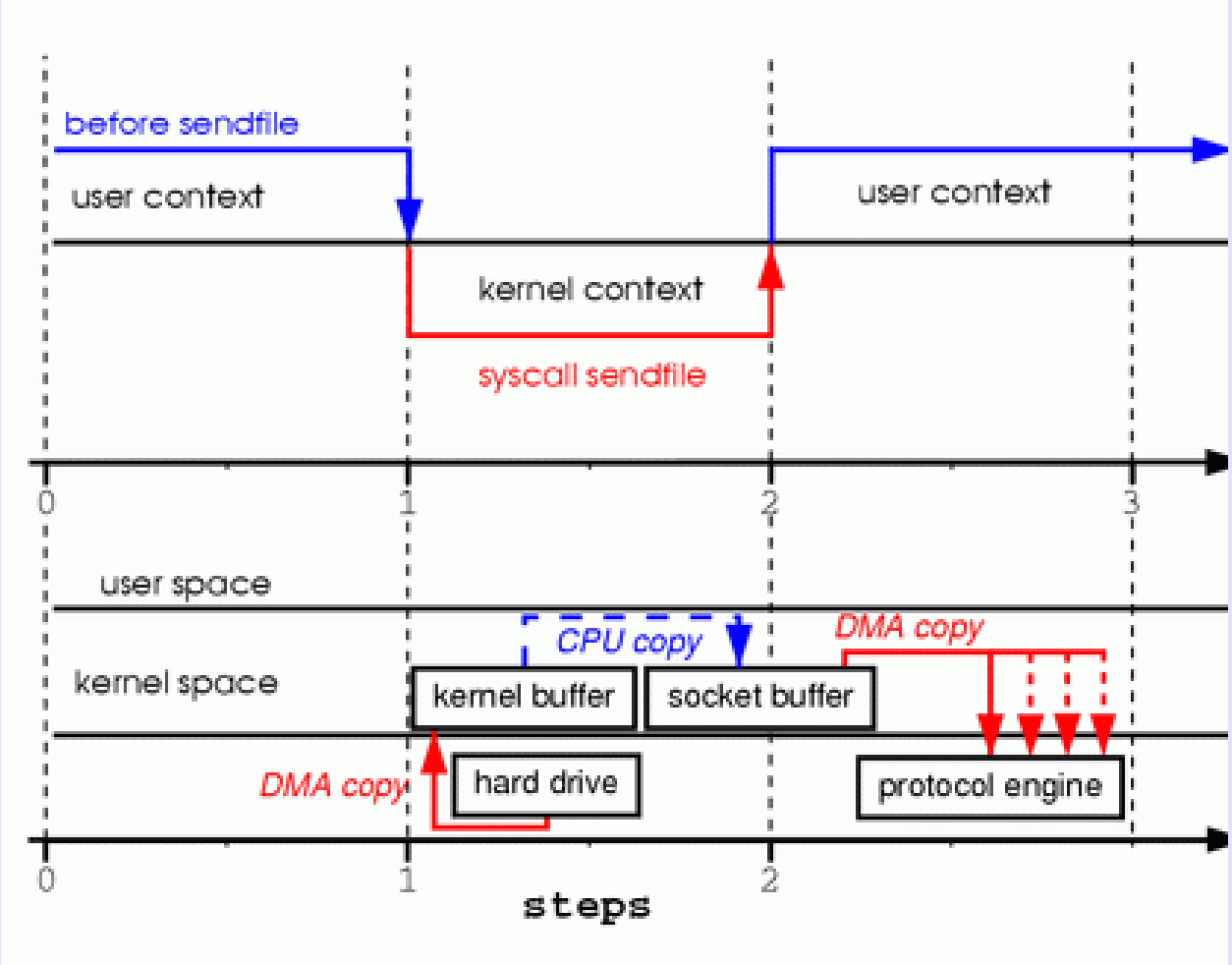
Two-Three Copy



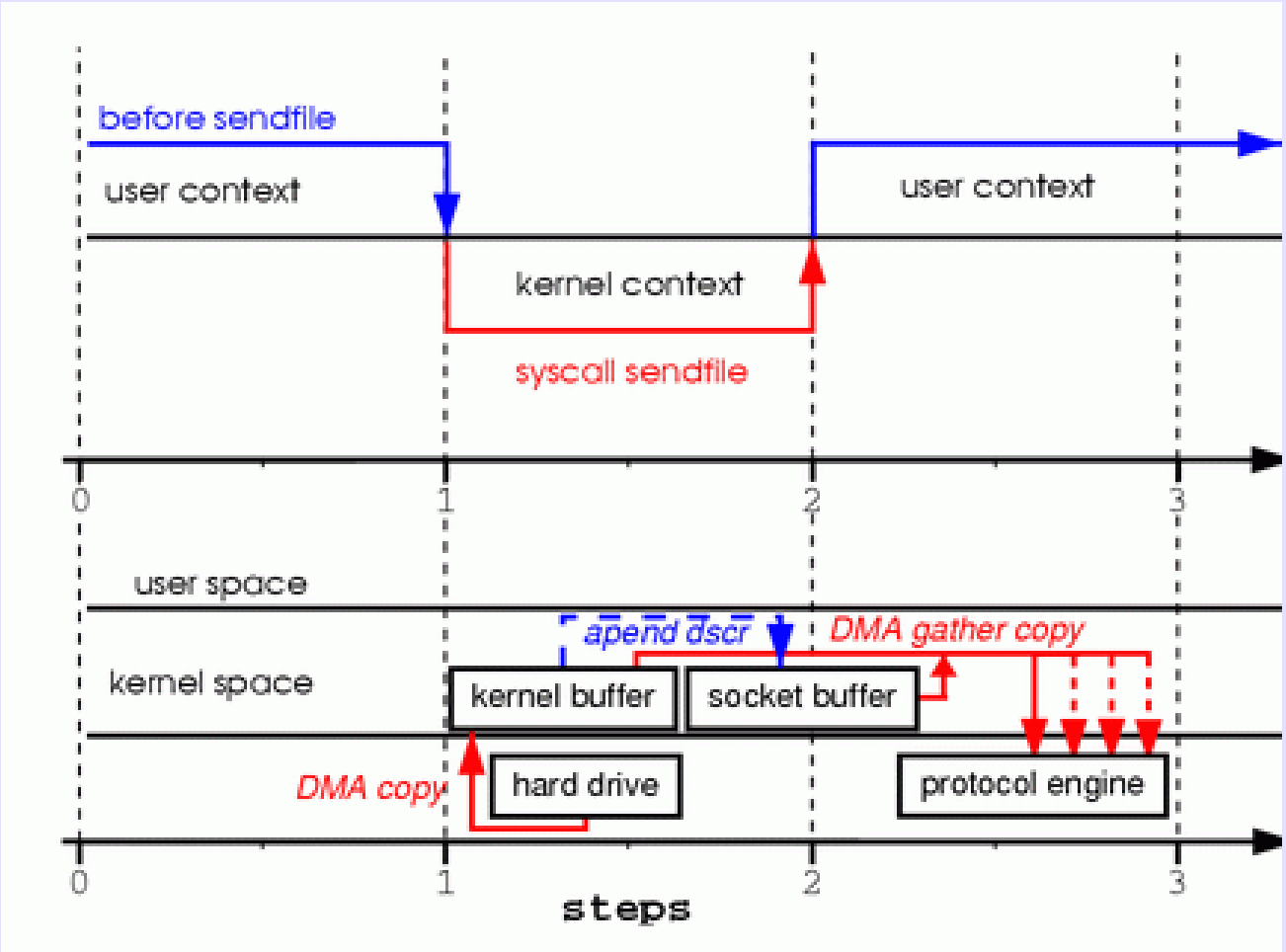
Zero Copy - Sendfile

- In kernel version 2.1, the sendfile system call was introduced to simplify the transmission of data over the network and between two local files. Introduction of sendfile not only reduces data copying, it also reduces context switches.
- Use it like this:
    - sendfile(socket, file, len);

One Copy


before sendfile
user context
user context
kernel context
syscall sendfile
0
1
2
3
user space
kernel space
CPU copy
DMA copy
kernel buffer
socket buffer
DMA copy
hard drive
protocol engine
0
1
2
steps

Zero Copy - Better



Splice/Vmsplice/Tee

- splice() is a system call that copies data between a file handle and a pipe, or between a pipe and user space.
- It does so without actually copying the data, in contrast to other data copying techniques, thereby improving I/O performance.

Splice/Vmsplice/Tee

- The crucial service offered by splice is that one can move data from one file descriptor to another without incurring any copies from user space into kernel space, which is usually required to enforce system security and also to keep a simple and elegant interface for processes to read and write to files.

Splice/Vmsplice/Tee

- The key insight that splice exploits is that a pipe buffer is effectively implemented as an in-kernel memory buffer that is opaque to the user space process.

Splice/Vmsplice/Tee

- This means that the user process can splice the contents of a source file into this pipe buffer, without ever copying the contents of the source file into memory, then into kernel space, then splice the contents of the pipe buffer into the destination file, again without incurring any copies