Day 1
Linux Abstractions

Joel Isaacson
Ascender Technologies Ltd

Today's Lectures

- Today we will examine the design of the Linux applicative programmable interface (API).
- You might think that Linux is simply a generic Posix system.
- This view ignores the consistent design principles of Linux that are not present in other Unix implementations

Misconceptions of Unix Users

- “Linux is just Unix spelled differently”
- Users that are used to Unix systems might not optimally use Linux.
- Linux is a more modern version of Unix, supporting the Posix standard.
- Choosing the Posix approach may not be the best choice.

Linux Simple Abstractions

- Linux tries to use a minimal number of concepts and extends their semantics to provide a synergistic effect.
- Many concepts from non-Posix operating systems such as Plan9 have been incorporated into Linux.

Linux Simple Abstractions

- Files
- Processes
- Memory Spaces
- IPC

Files

- The hierarchical file systems is fundamental to Unix-Posix systems.
- The well know open-read-write-close paradigm is used.
- We shall see that Linux has extended file abstractions farther than most Unix systems.

Files – Hierarchical Structure

- The hierarchical file system consists of a singly rooted tree.
- This is different than systems like MS Windows that have multiple roots.
- The `mount` system call grafts a subtree onto the file system.
- The nodes of the tree are directories.
- Directories are unordered lists of files.

Files - Namespace

- Files can be accessed via the file *namespace*.
- Objects within the namespace are simply strings of characters.
- The only character that has any special significance is the '/' character which is the directory delimiter.

Files – API Creation/Deletion

- Files are created/deleted via the standard Posix API's
  - `creat`
  - `open`
  - `mkdir/rmdir`
  - `link/unlink`

Files – File Descriptors

- Most operations that access a file usually use a *handle* to the file called a *file descriptor*.
- Pre-existing files are normally associated with the file's *namespace* via the `open` system call.
- `open` returns a file descriptor.

Files - Read/Write

- The content of files can be accessed via the `read/write` system calls.
- `read/write` copies the contents from/to the file to/from the process's *memory space*.
- Files are frequently used to provide persistence of data.

Files – Regular Files

- Persistent files stored on disk.
- Byte streams
- Each open file has an associated file offset.
- Writing within the file overwrites te previous results
- The file's length can be changed by `truncate`.

Files – Directories

- Provides namespace with which to access files.
- When a fully qualified (name starting with /) filename is opened, the kernel walks the directories in the filename until the regular file is opened.
- Relative filenames start at the current directory

Files – Hard Links

- The files themselves are stored as a linear list of files.
- Each file has a unique index called an i-node.
- A hard link is an entry in a directory that map a filename to an i-node.
- Each file can have more than one hard link.

Files – Symbolic Links

- Symbolic links look like regular files.
- The content of the file is just the name of a file that must be opened to find the actual data of the requested file.
- Symbolic links that don't point to valid files are called *broken*.
- Hard links cannot span different filesystems. Symbolic links can.

Files – Special Files

- Represent kernel objects.
- Linux has four types of special files.
  - Block
  - Character
  - Named Pipes
  - Unix Domain Sockets

Files – Special Files Block

- Devices through which the system moves data in the form of blocks.
- Block special file often represent addressable devices such as hard disks, CD-ROM drives, or memory-regions.
- Access is random read/write.
- Can be mounted as a filesystem.

Files – Special Files Characters

- Linear queue of bytes.
- Not necessarily random read/write.
- Typically keyboard, mouse.
- At end of file returns EOF.

Files – Special Files
Named Pipes

- Often called *fifos*.
- Are used in IPC.
- Regular pipes are used to communicate between related processes.
- Named pipes can be used between unrelated processes.
- I generally prefer Unix Domain Sockets.

Files – Special Files
Unix Domain Sockets

- Uses the standard BSD socket interface.
- Uses a filename in bind-connect rather than an IP number.
- Only works within one computer.

Processes

- Processes contain the basic state of computation.
- Linux has a very efficient process model.
- Linux does not have an independent LWP (Light Weight Process – threading) model.
- Threads are processes.

Processes

- Processes also contain a possibly sparse array of open files.
- The file descriptor is just an index into this array.
- An open file has an associated file pointer that contains the byte count position within that file.

Memory Spaces

- Memory spaces provides the glue that bind Files and Processes.
- They are usually associated with a process and contain memory areas that can map:
    - Files
    - Shared Memory
    - Device Memory

Memory Spaces Continued

- A memory area might have:
  - COW (Copy On Write – Private) semantics.
  - Shared memory semantics

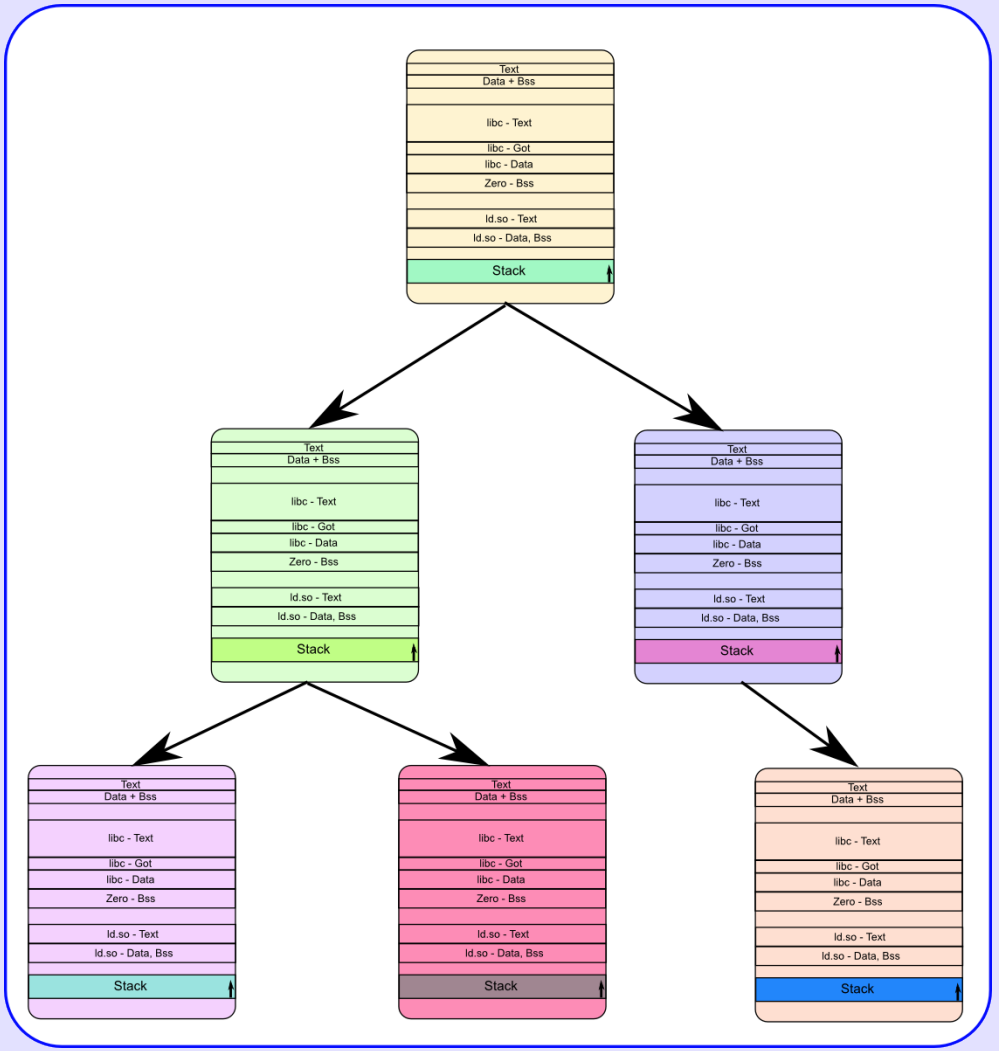
IPC
Inter-Process Communication

- Linux's IPC is based on the BSD socket API.
- It works both within one computer (Unix domain sockets) and between computer (usually via TCP/IP)

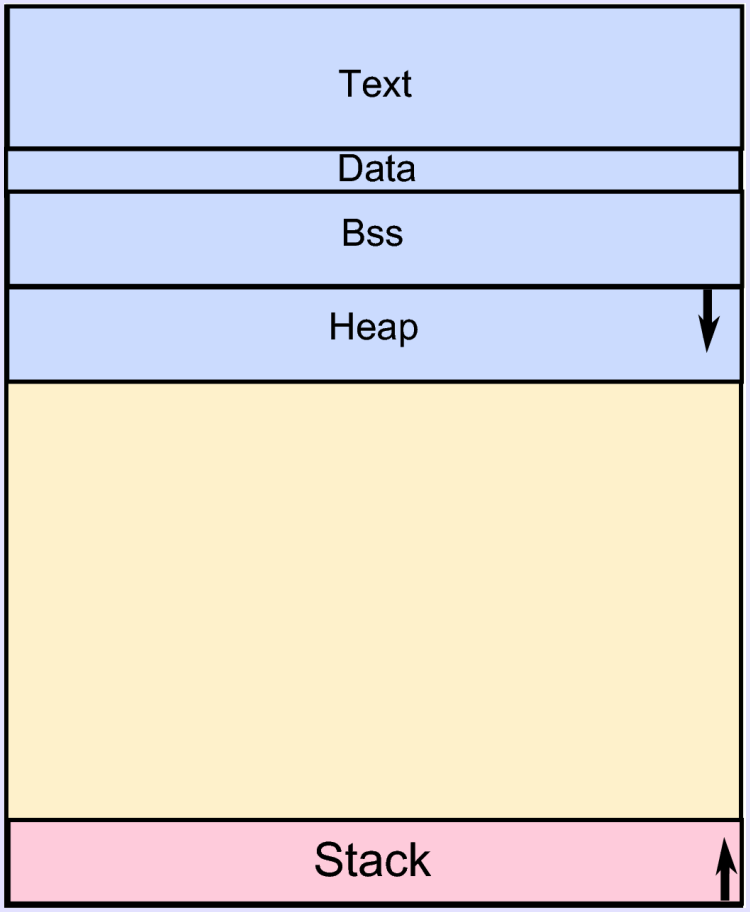
Fork/Clone

- The only way to create a process is through the clone (fork) system call.
- The default is to create a process that inherits the parent's processes memory spaces through COW semantics.
  - A thread is created via the clone system call with the CLONE_VM argument.
  - Threads are no more efficient than processes.

Posix Processes



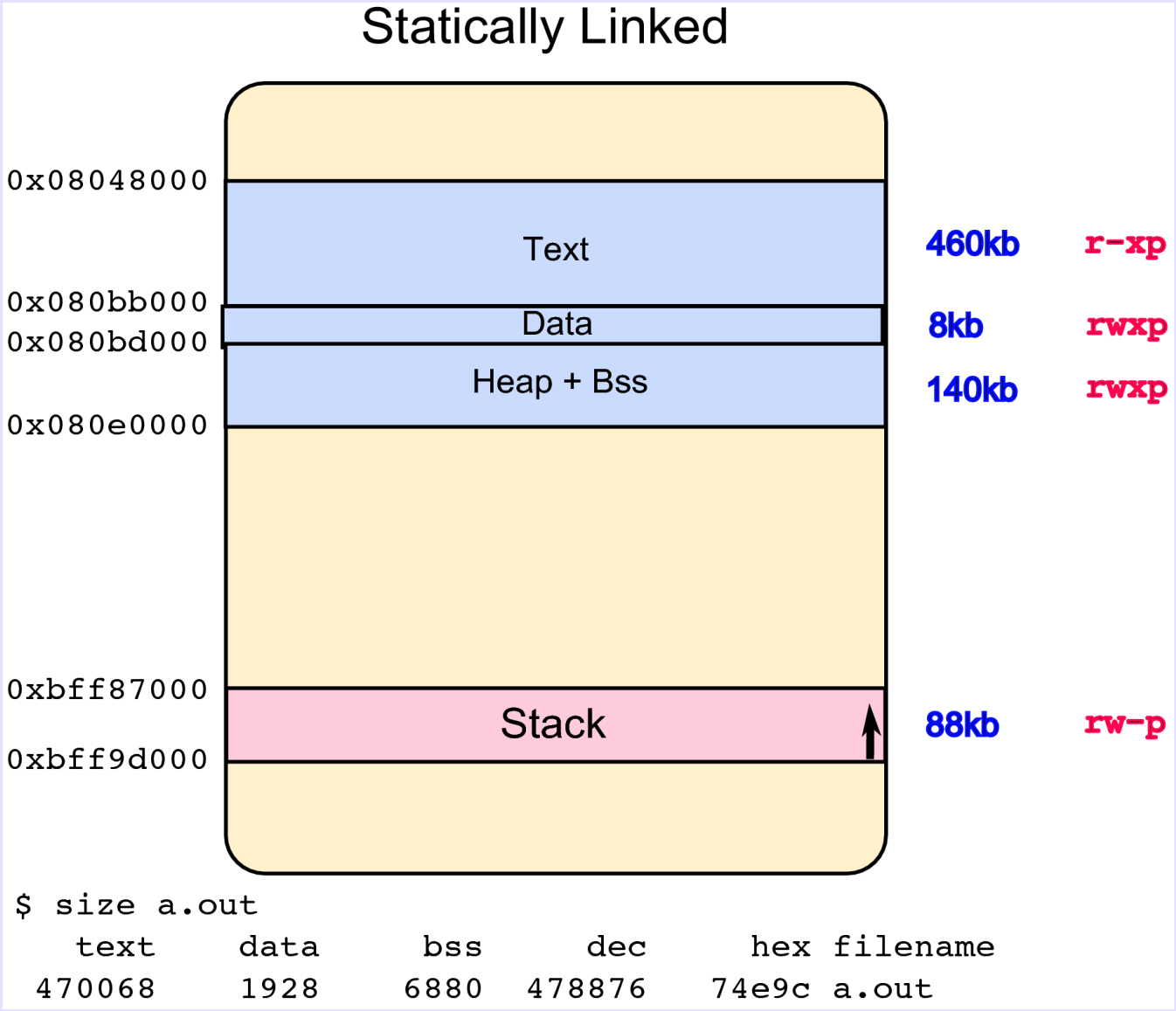
Memory Spaces – Unix(1974)



- This is the standard initial configuration of the memory spaces of a process after exec.
- There are five memory segments: text, data, bss, heap and stack

Static Compilation

```
$ cat t.c
main()
{
  char b[100];
  gets(b);
}
$ cc -static t.c
$ ./a.out &
[1] 27206
```




```
$ size a.out
   text    data     bss     dec     hex filename
 470068    1928    6880  478876   74e9c a.out
```

Private Segments
Copy on Write (COW)

All memory spaces that we have seen until now have had the 'p' attribute i.e. COW semantics. This means that initially the contents of the space are shared. If there are changes done to the memory space, a page fault is raised which causes a new copy of that page to be allocated. This allows fast execve's since nothing initially has to be copied. This is a form of *lazy* copying.

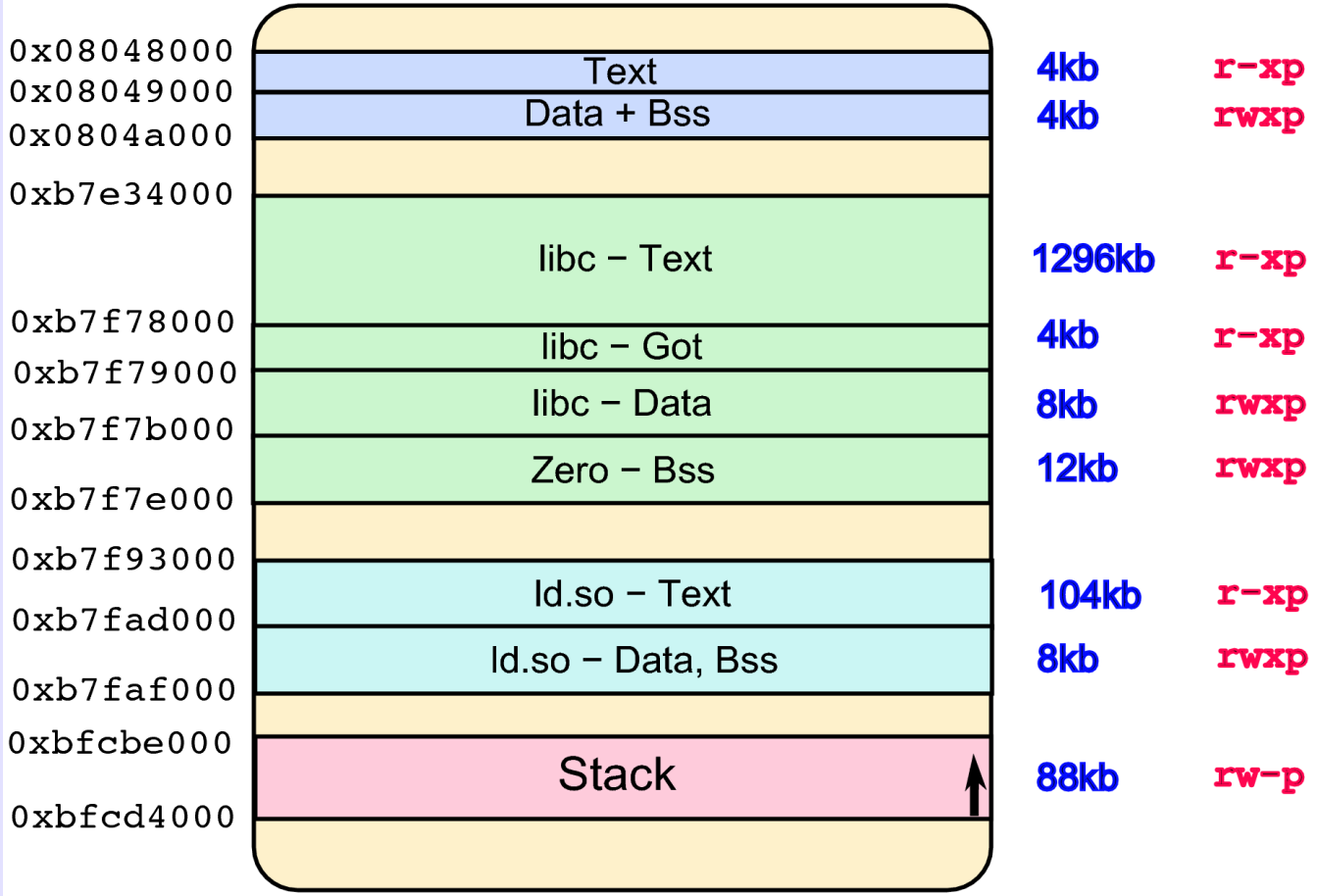
Shared Library Compilation

```
$ cat t.c
main()
{
  char b[100];
  gets(b);
}
$ cc t.c
$ ./a.out &
[1] 27206
```

Shared Library

```
$ cat /proc/27206/maps
08048000-08049000 r-xp 00000000 08:05 198109  a.out
08049000-0804a000 rwxp 00000000 08:05 198109  a.out
b7e2e000-b7e2f000 rwxp b7e2e000 00:00 0
b7e2f000-b7f73000 r-xp 00000000 08:03 1934283
libc-2.6.1.so
b7f73000-b7f74000 r-xp 00143000 08:03 1934283
libc-2.6.1.so
b7f74000-b7f76000 rwxp 00144000 08:03 1934283
libc-2.6.1.so
b7f76000-b7f79000 rwxp b7f76000 00:00 0
b7f90000-b7f93000 rwxp b7f90000 00:00 0
b7f93000-b7fad000 r-xp 00000000 08:03 1901413 ld-2.6.1.so
b7fad000-b7faf000 rwxp 00019000 08:03 1901413 ld-2.6.1.so
bfcbe000-bfcd4000 rw-p bfcbe000 00:00 0       [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0       [vdso]
```

Shared Libraries




```
$ size a.out
   text    data     bss     dec     hex filename
    952     272       4    1228     4cc a.out
```

exec

- The exec system call is used to create a new memory space.
- The initial memory space right after the exec system call has four memory areas:
    - Text
    - Data
    - Bss
    - Stack

Files: /proc and /sys file systems

- The /proc file system was pioneered in Bell Labs influential but not very popular Plan9 operating system.
- The /proc and /sys file system has eliminated thousands of application specific API's.

Real Programmers Don't Use

- Threads
- Asynchronous I/O
- Signals
- Semaphores
- Real-time Scheduling
- Unbridled Ioctl's
- Massive Kernel Code Dumping

Threads

- There are very few standard multi-threaded programs under Linux.
- Linux now has very good threaded support.
- It is not impossible to write a multi-threaded nontrivial program. It is just very difficult.
- The most compelling reason to use threads is to utilize multi-core CPU's.

Asynchronous I/O

- New to Linux 2.6.
- Not well supported.
- Only works for O_DIRECT access which is fairly useless.

Signals

- Signals suffer from the same problems that threads have.
- They can't easily be used for reliable inter-process communications.
- They basically create an interrupt programing model with a need for masking interrupts (signals).

Semaphores

- Semaphores are difficult to use properly.
- Forget to use one and your data gets scrambled.
- Use too many and you risk deadlock.

Real-Time Scheduling

- Posix.4 adds real-time, absolute, fixed priority scheduling.
- The *sched_setscheduler* system call can be used to set either *fifo* or *round robin* scheduling policy.

Real-Time Scheduling

- The problem is that:
- The real-time scheduler doesn't work well with the standard (SCHED_OTHER) scheduler.
- Deadlock and starvation can occur.
- The standard scheduler does a very good job.

Unbridled Ioctl's

- Adding functionality via many poorly thought out ioctl calls creates a system that is hard to use and very unportable.
- It is much better to use higher level abstraction such as the /proc or /sys file system.
- Ascii is great. Down with binary.

Massive Kernel Code Dumping

- Just taking a mass of C code from another system and inserting it into the Linux kernel is asking for trouble.
- Some difficulties are:
  - No virtual memory or process memory space
  - Small stack (4k or 8k)
  - No standard C library
  - Locking, Preemption etc. etc

Linux vs uClinux

- No fork (or clone) only vfork.
- No brk or sbrk, Only mmap (and only with MAP_ANONYMOUS) can be used to allocate memory (not contiguous).
- Fixed size stack.
- No memory managment.

Examples

- /proc filesystem
- Static Linking
- Dynamic Linking
- Light weight processes