# Day 1
# Monday 31/8/09

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

## Today's Lectures

- Today we will start by examining 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

## 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

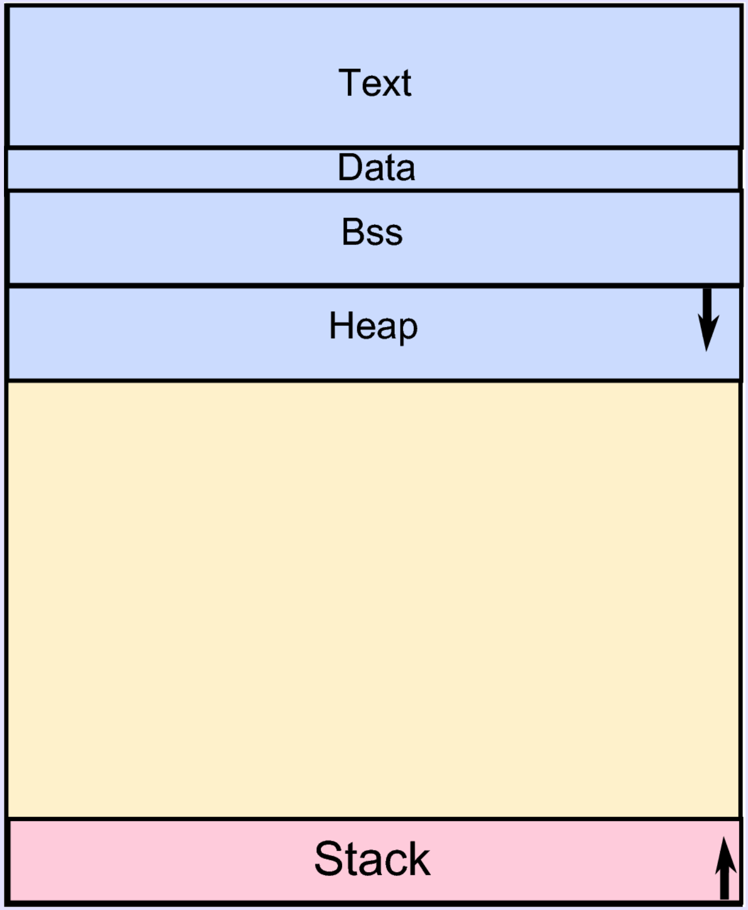
## IPC
## Inter-Process Communication

- 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)

## 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.

# Memory Spaces – Unix(1974)

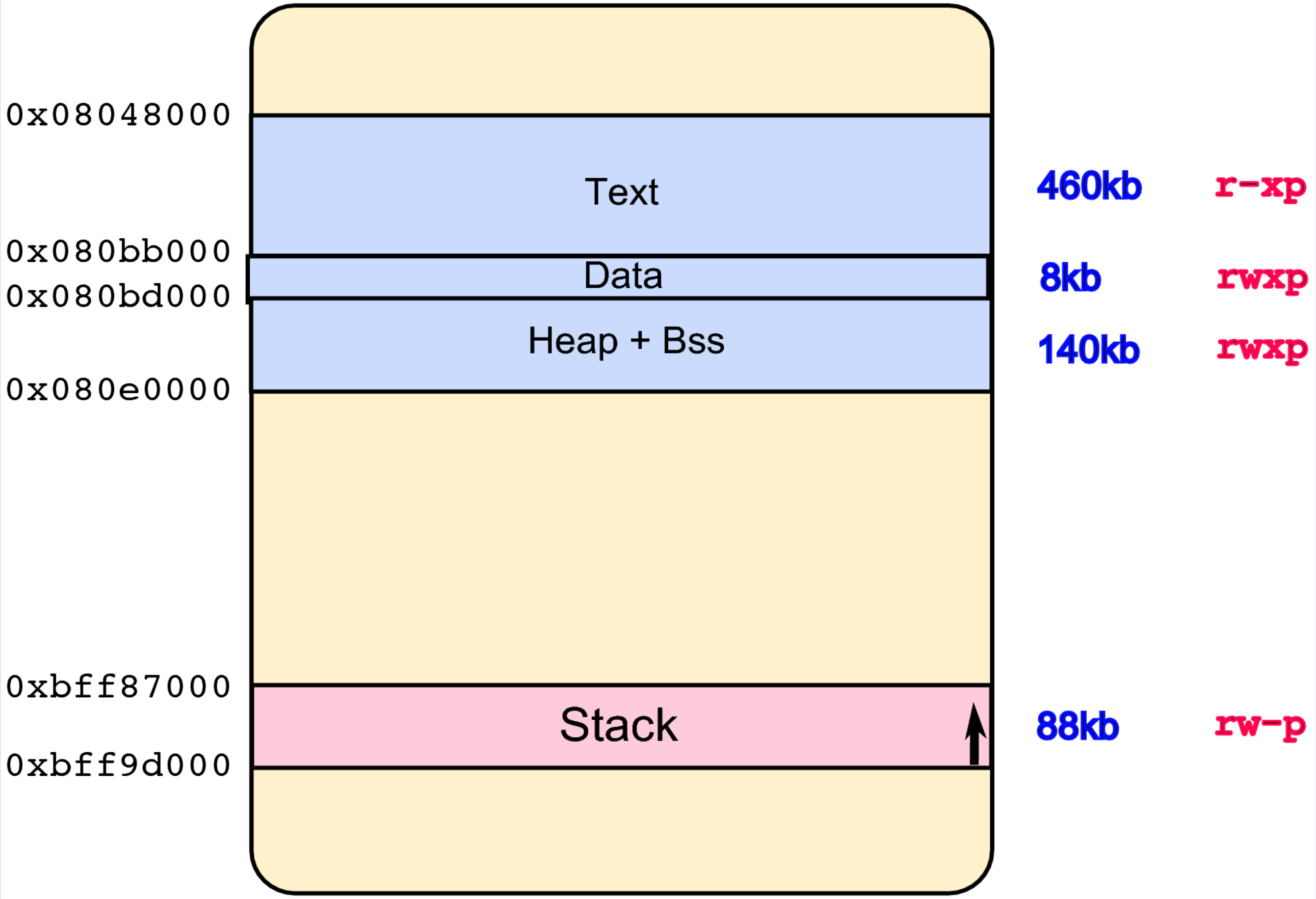


- This is the standard initial configuration of the memory spaces of a process after exec.
- There are five memory segments: text, data, bss, heap and stack

## Static Compilation

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

## Statically Linked



```
$ 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.

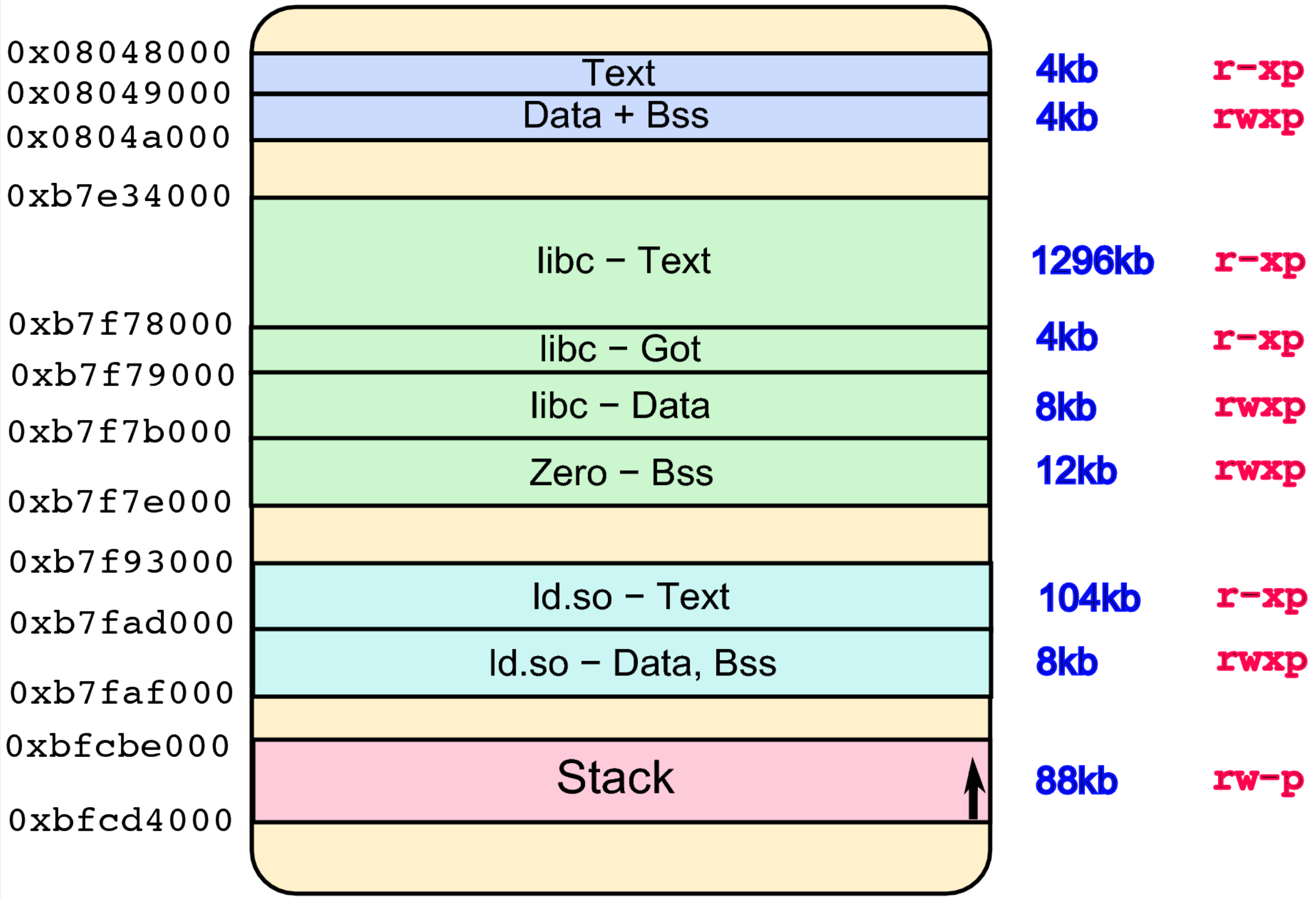
## Shared Library Compilation

```
$ 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

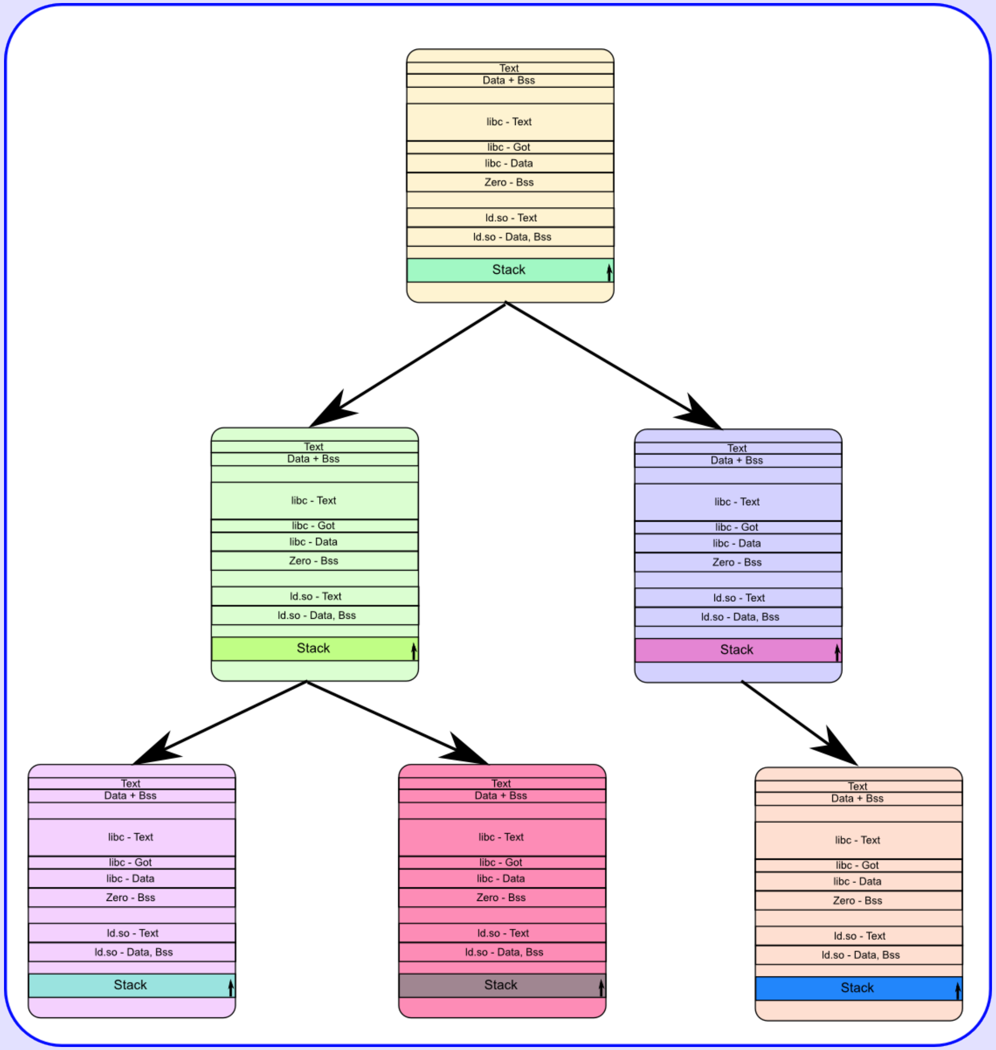


```
$ size a.out
   text    data     bss     dec     hex filename
    952     272       4    1228     4cc a.out
```

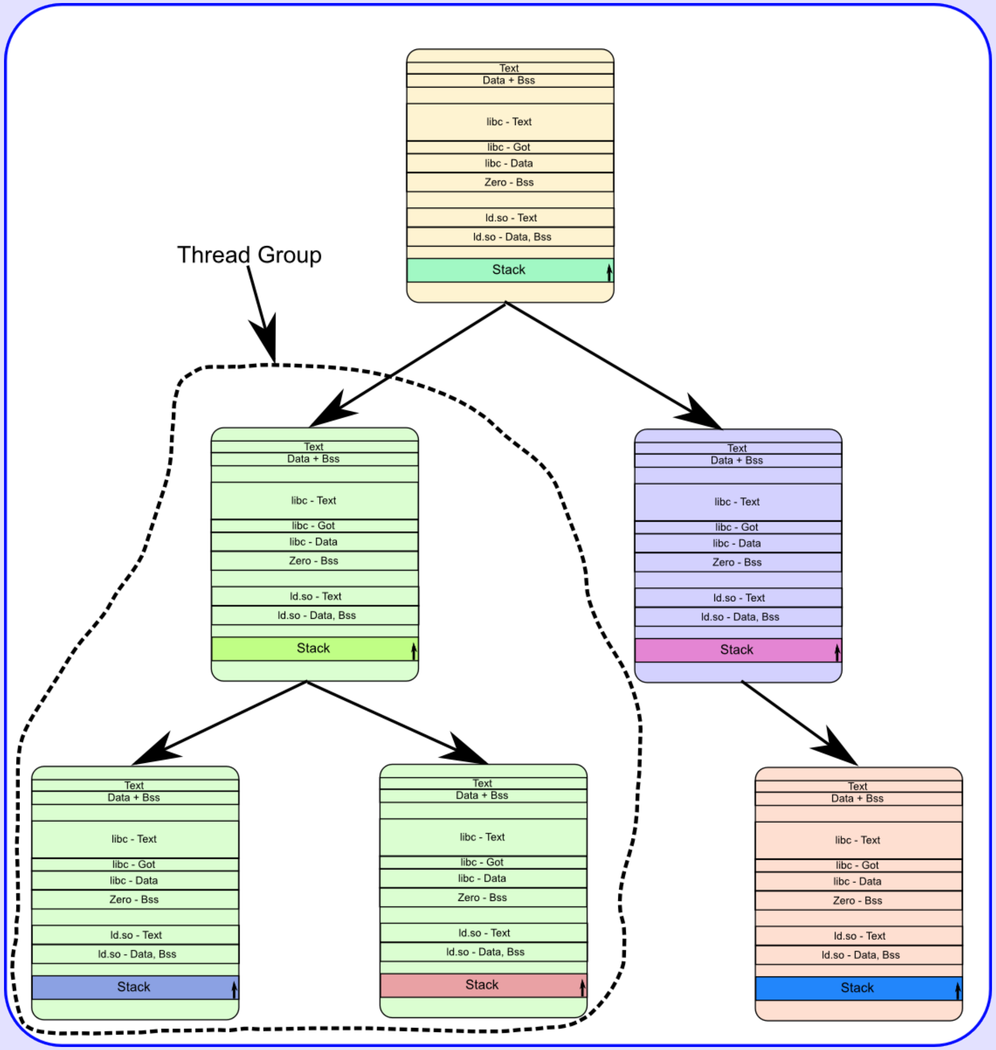
## Fork/Clone

- 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



# Posix Threads



## 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

## Kernel Threads

- What is this strange creature?
- Simply a process that runs without a attached memory space (mm == 0).
- The kernel memory space is mapped.
- Scheduled exactly as any other process.
- Used to provide an independent “user context” to kernel functionality.

## Real Programmers Don't Use

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

## Kernel-User API

- User ABI-API stable.
- Kernel API unstable.

## Processes

Process == Task == Thread

Linux uses these three names interchangeably

## Scheduling

- Linux is a multitasking operating system, having many processes resident in memory at one time.
- Each processes has the illusion that it has a CPU of it own.
- Deciding which processes are to be run is the job of the scheduler.

## Process Scheduling

The fundamental task of the scheduler is to choose, at any one time, from the set of runnable processes, ***m***, only ***n***, where ***n*** is the number of CPU's present, processes to run.
If ***m*** is less than ***n*** this task is trivial.

## Scheduling

- The scheduler is the kernel component that selects among processes that are runnable.
- The scheduler effectively provides a time-multiplexed CPU to many independent processes.

## Scheduling

- Multitasking operating systems come in to varieties:
  - cooperative multiprocessing and
  - preemptive multiprocessing.
- Cooperative multiprocessing (like Windows 3.1) only schedule processes after they yield the CPU.

## Scheduling

- With preemptive multiprocessing processes run for a maximum period of time, called a timeslice (or quantum), and then they are halted to give other runnable processes a chance to run.
- Linux uses a classical preemptive multi-priority queue scheduler with some novel innovations that where present in the first Unix system of the 1970's.

## Scheduling

- The thing that differentiates the Linux scheduler from RTOS schedulers is dynamic-priority setting and the way that processes are suspended from rescheduling until all other processes have completed their time slices.

## Scheduling

- It is important to note that standard Linux (or standard POSIX in fact) actually has three different schedulers.
- There are two real-time (see 'man sched_setscheduler') schedulers, SCHED_FIFO and SCHED_RR, that take precedence over the the standard Linux scheduler, SCHED_OTHER.

## Scheduling

- The scheduler policy is the method by which the dynamic priorities are determined.
- The main rational is to allow good interactive behavior (low latency) and to globally optimise CPU utilization.

## Scheduling

- Conceptually it is useful to divide processes into two classes:
  - I/O bound (or interactive) and
  - CPU bound (or compute intensive).
- Of course in general a process may be somewhere between these two definitions and might change its class many times during its execution.

## Scheduling

- The general idea is to give interactive processes higher priority and longer timeslices so that intercative users of the system experience low latency.
- A interactive processes is defined as a processes that spends most of its time on a wait queue, in a unrunnable state.

## Scheduling

- Compute bound processes are mostly in a runnable state.
- This observation has proven extremely successful and is the basis of all Unix schedulers for the last 35 years.

## Scheduling

- For example: if the system has two runnable processes, an editor, and a process to compute the value of $\pi$, to 20000 digits.
- We would want to give the editor a higher priority. Otherwise the interactive use will be quite unhappy.

## Scheduling

- This naturally happens with the standard Linux scheduler since the text editor is normally spending 99% of the time in a wait queue waiting for keyboard input.

## Scheduling

- In addition Unix and Linux allows the user to give a nice value which is added to the dynamic priority in order to give the scheduler information from the user about the relative importance of different processes.

# Scheduling

- The nice value effect the amount of time the process will consume in relationship to other processes, but no process will starve for CPU time.

## Scheduling – O(1) Algorithm

- During the Linux 2.5.x development period, a new scheduling algorithm was one of the most significant changes to the kernel.
- The Linux 2.4.x scheduler, while widely used, reliable, and in general pretty good, had several very undesirable characteristics.

## Scheduling – O(1) Algorithm

- The undesirable characteristics were quite embedded in its design.
- The fact that the Linux 2.4.x scheduling algorithm contained O(n) algorithms was perhaps its greatest flaw, and subsequently the new scheduler's use of only O(1) algorithms was its most welcome improvement.

## Scheduling – O(1) Algorithm

- The Linux O(1) scheduler does not contain any algorithms that run in worse than O(1) time.
- That is, every part of the scheduler is guaranteed to execute within a certain constant amount of time regardless of how many tasks are on the system.

## Scheduling – O(1) Algorithm

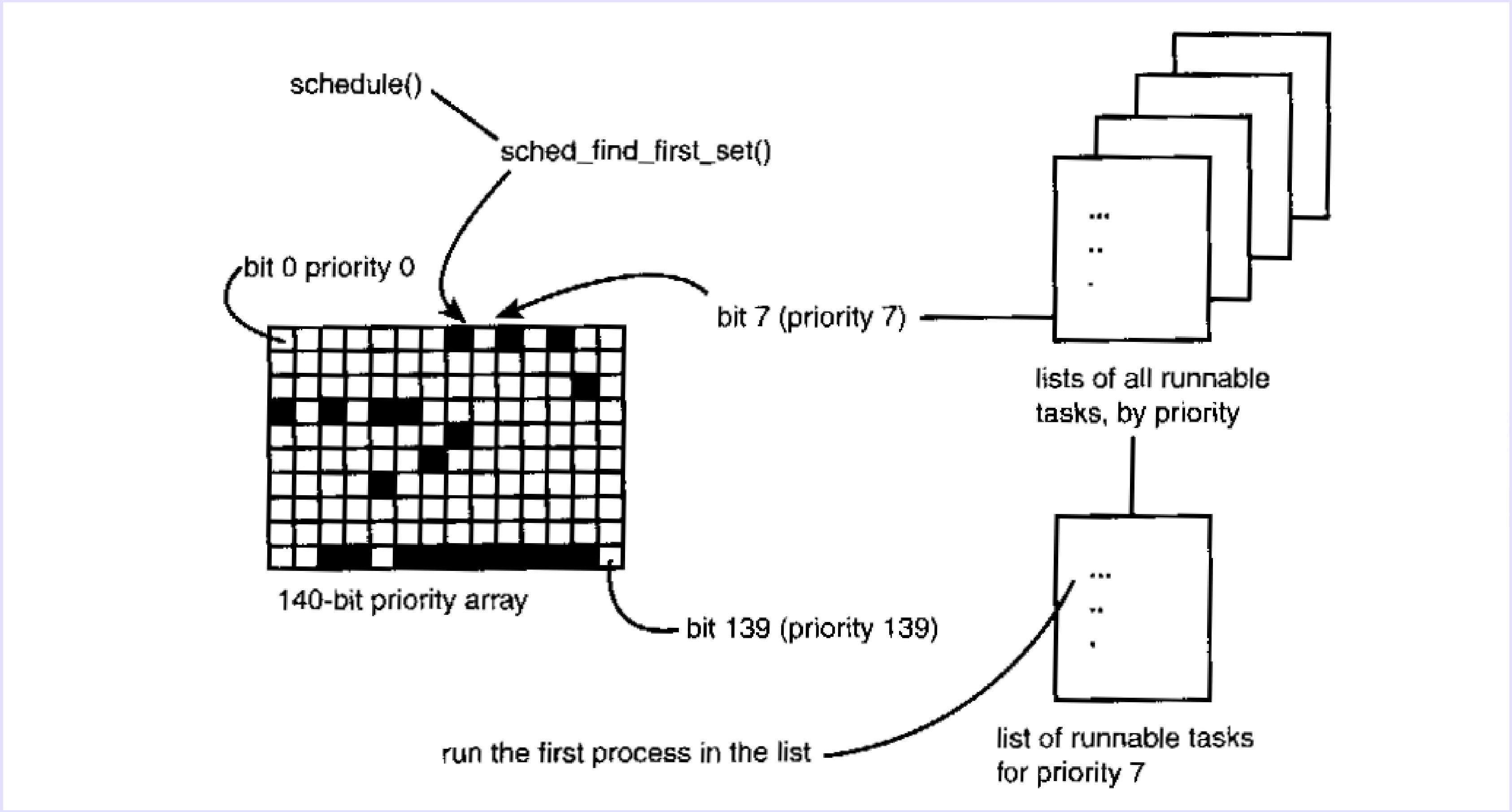
- This allows the Linux kernel to efficiently handle massive numbers of tasks without increasing overhead costs as the number of tasks grows.
- There are two key data structures in the Linux O(1) scheduler and its design revolves around them:
  - runqueues and
  - priority arrays.

## Runqueue – O(1) Algorithm

- The algorithm uses two structure of type priority_array, in kernel/sched.c,

```
struct prio_array {
   int nr_active;
   unsigned long bitmap[BITMAP_SIZE];
   struct list_head queue[MAX_PRIO];
};
```

# Runqueue – O(1) Algorithm

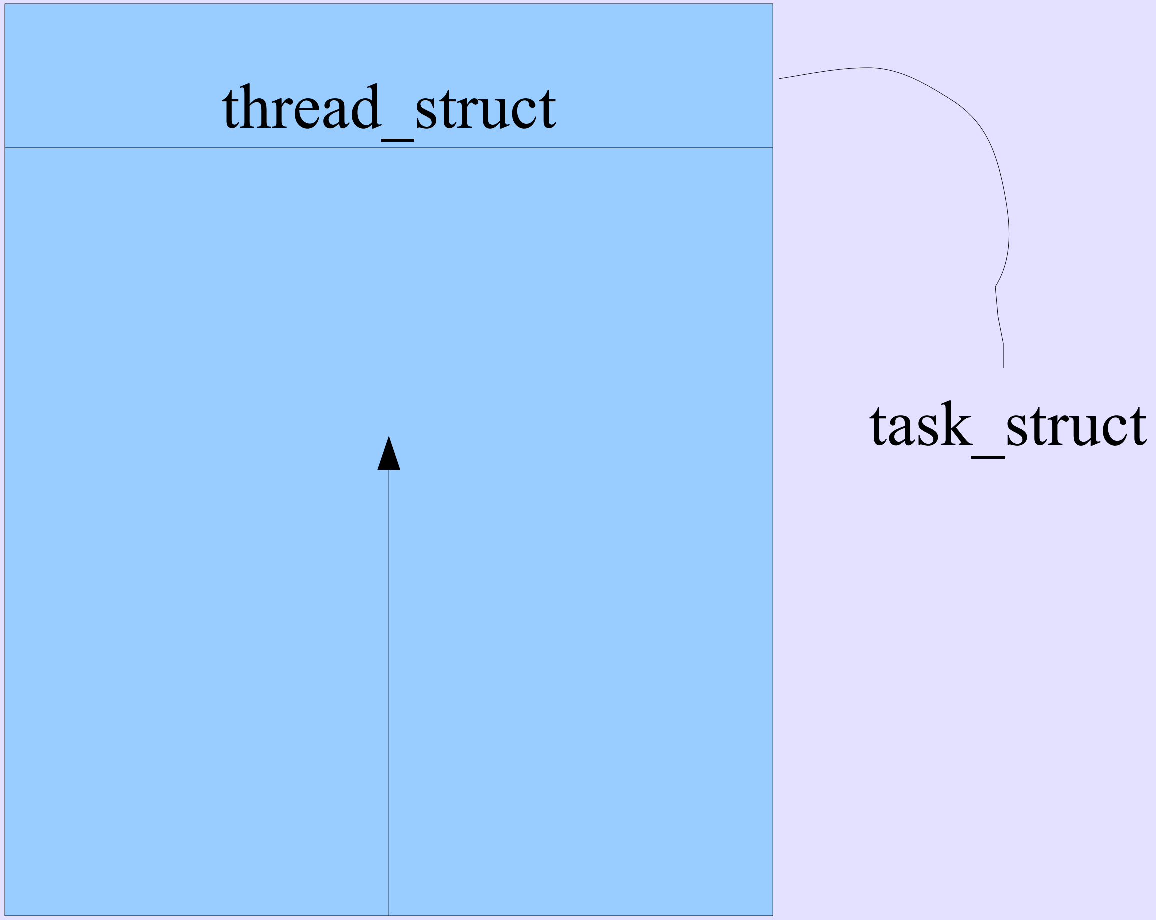

schedule()
sched_find_first_set()
bit 0 priority 0
bit 7 (priority 7)
140-bit priority array
bit 139 (priority 139)
lists of all runnable tasks, by priority
run the first process in the list
list of runnable tasks for priority 7

## The Process Table

- The process table consists of a collection of objects of  types “struct task_struct”.
- Each process has a “struct thread_struct” at the end of the kernel process stack that has as its first element a pointer to “task_struct”.
- The process table can be enumerated by a link in the “task_struct”

## Kernel Mode Stack

4 or 8 Kbytes
page aligned



## Time for a quiz

```
{
   xtype *p;
   char *q;
   p= (xtype *) *(long *)(((long) & q) & ~0x1fff);
}
```

The question is:
What does “p” point to?
What type is “xtype”?
Hint: the kernel stack is 8K bytes.

## The Answer is ...

- This code is very strange.
- I normally wouldn't show this code but it is used heavily in the kernel via inline routines.
- Even experienced kernel programmer might not recognize it since it usually is hidden deep within the processor dependent include files.

## The Answer

- This code is used to return the “task_struct”  of the user process.
- The tricky idea is that any address on the kernel mode stack when aligned to the nearest 8 Kb boundary will point to the “thread_struct”.
- The first element of the “thread_struct” points to the “task_struct” which is the process entry.

## current_thread_info()
## asm/thread_info.h

```
/* how to get the current stack pointer from C */
register unsigned long current_stack_pointer asm("esp")
                     __attribute_used__;

/* how to get the thread information struct from C */
static inline struct thread_info *current_thread_info(void)
{
        return (struct thread_info *)(current_stack_pointer
              & ~(THREAD_SIZE - 1));
}
```

## current
## asm/current.h

```
static __always_inline struct task_struct *
get_current(void)
{
        return current_thread_info()->task;
}

#define current get_current()
```

## System Calls

- The system call is (usually) called from user space.
- This causes a switch to kernel mode (ring 3 -> ring 0).
- It also causes the upper 1 giga byte (0xc0000000 – 0xffffffff) to be memory mapped.

## System Calls

- The system call is a synchronous exception.
- It is actually either and “INT 80” or a “sysenter”.
- In Linux a new “vsyscalls” mechanism through the mapping to the kernel page “[vdso]”

## System Calls

- Each Linux system call is given a number.
- The particular link to the called kernel routine is in the file “syscall_table.S”
- The xxxx system call is called sys_xxxx in the kernel.