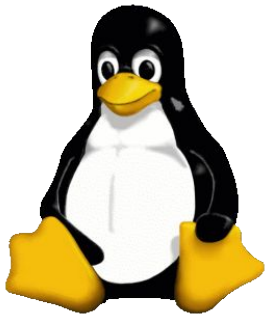


# Linux Systems Programming

Maruthi S. Inukonda

15 April 2021

# License



# Agenda

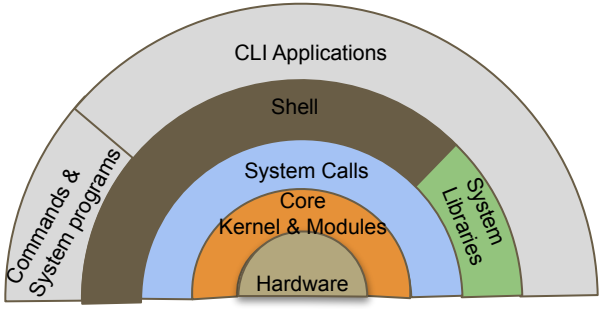
- Linux Architecture (Recap)
- Programs & Libraries - Linking, Loading
- Process Virtual Address Space
- Backing Stores - Paging, Swapping
- File System basics (Recap)
- File I/O
- Memory Mapping

# Linux Architecture (Recap)

Refer “Linux - The Beginning” slides for complete picture.

# Linux Architecture - Command Line Interface (CLI)

- Hardware
  - ➢ CPU, Memory, Disk, Graphics, Network, etc
- Core Kernel & Modules
  - ➢ Process, Memory, File, Network subsystems, Device drivers
- System Calls
  - ➢ read, write, fork, exec, clone, etc
- System Libraries
  - ➢ libc, libpthread, etc
- Commands & System programs
  - ➢ cd, ls, mkdir, top, vi, gcc, etc
- Command Line Interface (CLI) (Shell)
  - ➢ bash, sh, etc
- Command line applications
  - ➢ pine, git, gdb, etc

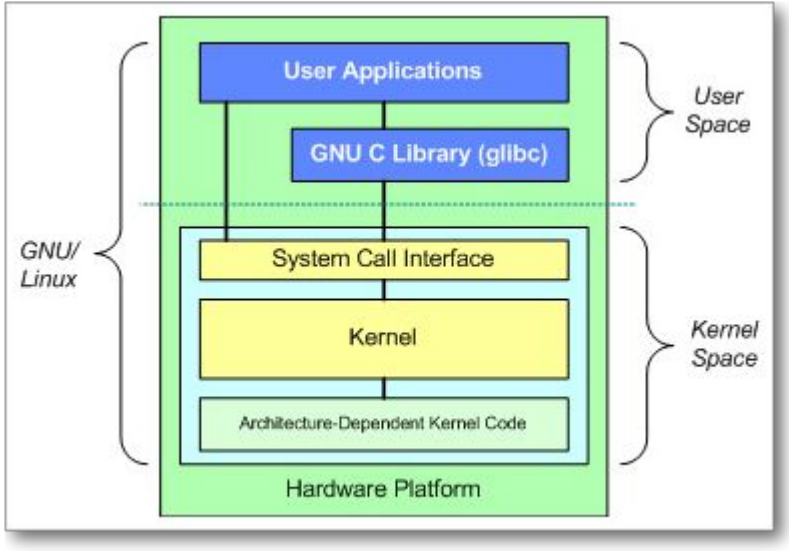


# System calls

- Entry points into the kernel.
- C language APIs.
- About 400 system calls
  - open(), read(), write(), close(), ioctl()
  - fork(), wait(), clone()
  - socket(), connect(), accept(), shutdown()
  - mmap(), munmap(), fadvise()
  - ...
- Using system calls in your program directly makes it
  - portable across Unices.
  - non-portable across Windows/Linux.

```
$ man syscalls

$ uname -o
GNU/Linux
```




Note: Use `strace` command on a live process to count/time system calls

# System Libraries

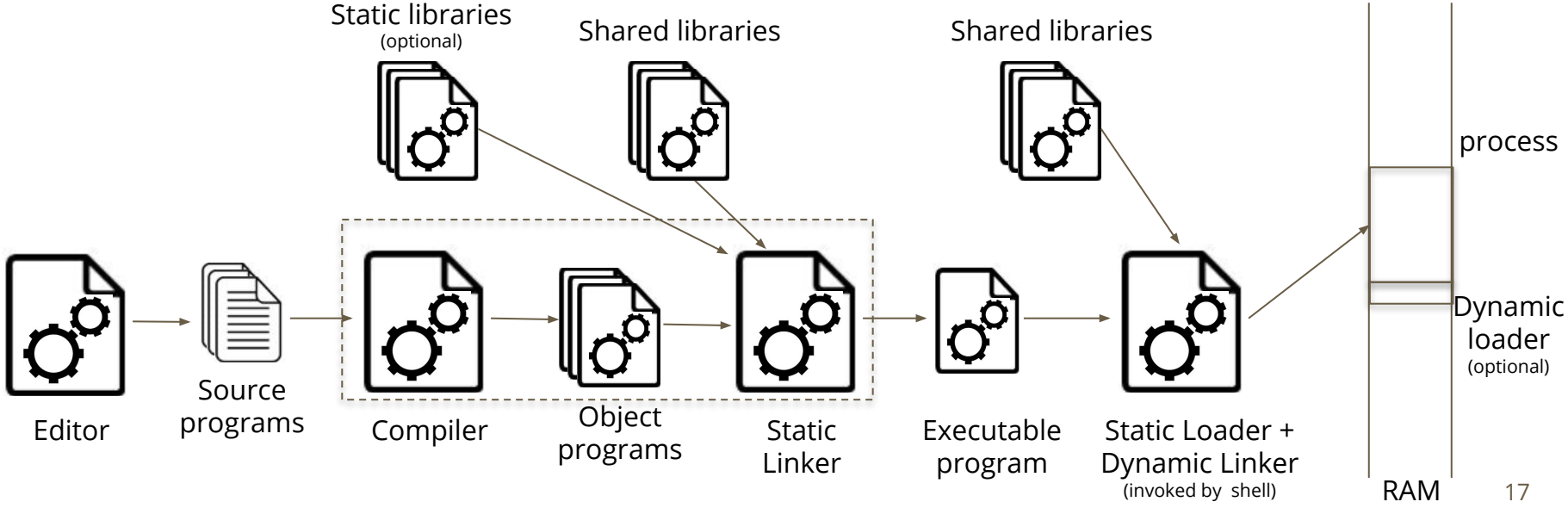
- Reusable routines packaged as `.a` or `.so`
- Every command loads its dependent  libraries at launch time.
- Types of libraries
  - Archive/Static libraries (.a)
  - Shared objects/libraries (.so)
- Types of linking
  - Compile time (Static) (only with .a) - Deprecated.
  - Load time - (only with .so)
  - Run time (Dynamic) - (only with .so)
- To know the dependent libraries use `ldd path_to_program`
- Using standardized library function calls in your program makes it portable.

```
$ ldd /bin/ls
    linux-vdso.so.1 =>  (0x00007ffc1d7eb000)
    ...
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fce2111b000)
    libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fce20a8a000)
```

# Programs & Libraries - Linking, Loading

# Editing, Compiling, Linking, Loading

- Editing : Writing source code. (.c, .h, .cpp, .hpp)
- Compile: Generation of object code from source code. (.c, .cpp → .o)
- Link : Combining object code to create executable, library (.o, .a → .out, .so)
- Load : Mapping Libraries, executable from Disk to RAM (.so, .out → RAM)



# Editing & Compiling

- To edit a program, use `vi` or `nano` or `emacs`
- To compile use, `gcc -c` with source files. To link use, `gcc` with object/library files.
- To build (compile & link) together, use `gcc` with source and object/library files.
- Use `-o` to specify executable/object name instead of default `a.out`
- To run a program, use `<program>` or ./`<program>` at shell prompt

```
$ vi sample.c

#include <stdio.h>

int main()
{
    char ch;
    printf("Hello\n");
    scanf("%c", &ch);
    return 0;
}
```

Compile, Link in one step

```
$ gcc -o sample sample.c
```

Compile, Link in two steps

```
$ gcc -c -o sample.o sample.c
$ gcc -o sample sample.o
```

To run the program

```
$ ./sample
Hello

$
```

# Dynamic Linking, Loading

- With dynamic linking, binding of all dependent libraries happen just before execution.
- By default `gcc/g++` on Linux uses dynamic linking.
- To see linkage type, use `file`
- To see load-time library dependencies, use `ldd`

```
$ file sample
sample: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/l, for GNU/Linux 2.6.32,
BuildID[sha1]=6990ad4330112d2a473c52f5ec1fbf8fc8f3da98, not stripped

$ ldd sample
    linux-vdso.so.1 =>  (0x00007ffe755fb000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f42933b3000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f429377d000)

$ ./sample
Hello

$
```

Here `ld-linux.so, libc.so, linux-vdso.so` are dynamically linked to `sample` at loading time.

# Static Linking

- With static linking, all dependent library functions are embedded into the program file.
- It makes programs independent of underlying libraries installed on system.
- But make disk space, virtual memory utilization grow abnormally.
- For static linking, use `-static`.

```
$ gcc -o sample_static -static sample.c

$ file sample_static
sample static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
statically linked, for GNU/Linux 2.6.32,
BuildID[sha1]=8bc087d84846f4f45c3651a0b2d8023b3fced667, not stripped

$ ldd sample_static
    not a dynamic executable

$ ls -l sample_static sample
-rwxrwxr-x 1 maruthisi maruthisi   8720 Apr  1 05:54 sample
-rwxrwxr-x 1 maruthisi maruthisi 912808 Apr  1 09:44 sample_static
```

Here `libc.so`, `linux-vdso.so` are statically linked into `sample_static` at build time.

# Multi-file Programs

- Multi-file programs help in modularity & code-reuse.
- Typically split into 3 files (header file(s), implementation file(s), main file)

```
$ vi fact.h
int fact(int n);

$ vi fact.c
#include "fact.h"

int fact(int n)
{
    int i, fact;

    for(fact=1, i=1; i<=n; i++) {
      fact = fact * i;
    }
    return fact;
}
```

```
$ vi mainfact.c
#include <stdio.h>
#include "fact.h"

int main()
{
    int n, f;

    printf("To calculate factorial, enter n: ");
    scanf("%d", &n);

    f = fact(n);

    printf("n!=%d\n", f);

    return 0;
}
```

# Compiling, Static Linking, Loading

- To compile, use `gcc -c`
- To statically link object files to executable, use `gcc`

Compile:

```
$ gcc -c -o fact.o fact.c
$ gcc -c -o mainfact.o mainfact.c
```

Link

```
$ gcc -o mainfact fact.o mainfact.o
```

Here `fact.o` is statically linked into `mainfact`

```
$ ldd mainfact
    linux-vdso.so.1 =>  (0x00007ffd40be5000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f87bb234000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f87bb5fe000)

$ ./mainfact
To calculate factorial, enter n: 5
n!=120
$
```

Here `ld-linux.so, libc.so, linux-vdso.so` are dynamically linked to `mainfact` at loading time.

# Compiling, Dynamic Linking, Loading

- To compile for creating library, use `gcc -c -fpic`
- To statically link object files and create shared library, use `gcc` with `-shared`
- To dynamically link shared libraries to executable, use `gcc` with `-L , -l`

Compile & Link a Library:

```
$ gcc -c -fpic -o fact.o fact.c
$ gcc -shared -o libfact.so fact.o

$ gcc -c -o mainfact.o mainfact.c
```

Link

```
$ gcc -o mainfact mainfact.o -L . -l fact
```

Here `libfact.so` is deferred for dynamic linking

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
$ ldd mainfact
    linux-vdso.so.1 =>  (0x00007fff217fa000)
    libfact.so => ./libfact.so (0x00007fa55817400)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (...)
    /lib64/ld-linux-x86-64.so.2 (0x00007fa558376000)
```

```
$ ./mainfact
To calculate factorial, enter n: 5
n!=120
$
```

Here `ld-linux.so, libc.so, linux-vdso.so, libfact.so` are dynamically linked to `mainfact` at loading time.

# Dynamic Loading (1/2)

- With dynamic loading, binding of all dependent libraries happen at run time.
- To dynamically load a library, use `dlopen(), dlsym(), dlclose()` in the code.

```
$ vi mainfact_light.c
...
#include <dlfcn.h>

int main() {
      int n, f;

      printf("To calculate factorial, enter n: ");
      if (n < 0) return -1;
      ...
      void *handle;
      int (*fact)(int);
      char *error;

      handle = dlopen("libfact.so", RTLD_LAZY);
      *(void **) (&fact) = dlsym(handle, "fact");
      f = (*fact)(n);
      printf("n!=%d\n", f);
      dlclose(handle);

      return 0;
}
```

Here `libfact.so` is deferred for dynamic loading

Access using pointers

# Dynamic Loading (2/2)

- To create an executable with dynamically loadable libraries, use `-l dl, -rdynamic`.

Compile & Link:

```
$ gcc -o mainfact_light mainfact_light.c -l dl -rdynamic

$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
$ ldd mainfact_light
    linux-vdso.so.1 =>  (0x00007ffcd0ab1000)
    libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (...)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (...)
    /lib64/ld-linux-x86-64.so.2 (...)


$ ./mainfact
To calculate factorial, enter n: 5
n!=120
$
```

Here `libdl.so` is deferred for dynamic linking. No mention about `libfact.so`

Here `ld-linux.so`, `libc.so`, `linux-vdso.so`, `libdl.so`, are dynamically linked to `mainfact` at loading time.

Here `libfact.so` is loaded conditionally at runtime.

# Process Virtual Address Space

# Virtual Address Space (1/3)

- Every process in Linux has a separate virtual address space.
- A process's memory footprint is divided into multiple segments, broadly:
  - Text (code)
  - Initialized data (global variables)
  - Uninitialized data / heap (dynamic variables)
  - Stack (local variables)
- Basically, a segment is a range of contiguous virtual addresses (start, len) of a process.
- Collection of all virtual address segments of a process is Virtual Address Space (VAS)

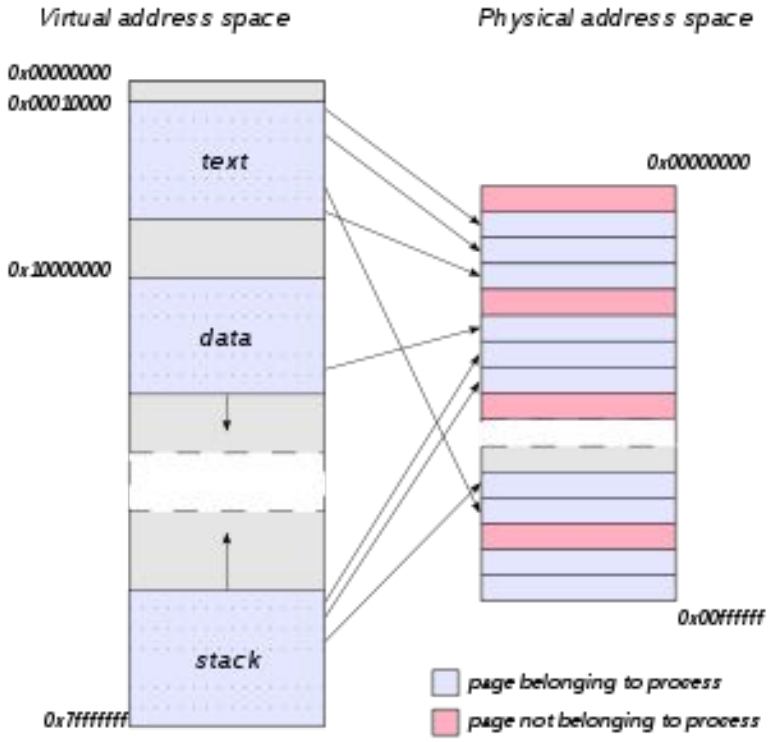



Image courtesy: https://www.wikipedia.org

# Virtual Address Space (2/3)

- Programs have different segment types
  - Code (in text segment)
  - Local variables (in stack segment)
  - Global variables (in initialized data segment)
  - Dynamic variables (in uninitialized data segment)
- Libraries have their own
  - Code in separate text segment.
  - Global variables in separate initialized data segment.
- Entire program has only one heap (may be in multiple segments)
- In multi-threaded programs, multiple stack segments are created.

```
$ vi funcvarptrs.c
int globalvar = 4333;
void func() {
    globalvar++;
}

int main() {
    char ch;
    float localvar;
    double *heapvar;

    heapvar = malloc(sizeof(double));

    printf("pid:%d\n", getpid());
    printf("&globalvar:%p\n", &globalvar);
    printf("&localvar:%p\n", &localvar);
    printf("&heapvar:%p\n", heapvar);
    printf("&main():%p\n", &main);
    printf("&func():%p\n", &func);
    printf("Check virt-phys translations in
separate terminal.\n"
           "Press any key to exit");
    scanf("%c", &ch);

    free(heapvar);

    return 0;
}
```

# Virtual Address Space (3/3)

- To see VAS of a process,
  use **cat /proc/<pid>/maps** or **pmap <pid>**
- Each segment is shown in separate line
- Permission fields
  - r - readable (text, data, stack)
  - w - writable (data, stack)
  - x - executable (text)
  - p - private
  - s - shared

```
$ ./funcvarptrs.out
pid:28542
&globalvar:0x55662de9e010
&localvar:0x7fffdbe6506c
&heapvar:0x55662e427260
&main():0x55662dc9d810
&func():0x55662dc9d7fa
Check virt-phys translations in separate
terminal.
Press any key to exit
```

```
$ pmap `pidof funcvarptrs.out`
28542:   /tmp/funcvarptrs.out
000055662dc9d000      4K r-x-- funcvarptrs.out
000055662de9d000      4K r---- funcvarptrs.out
000055662de9e000      4K rw--- funcvarptrs.out
000055662e427000    132K rw---   [ anon ]
00007f092199e000   1948K r-x-- libc-2.27.so
00007f0921b85000   2048K ----- libc-2.27.so
00007f0921d85000     16K r---- libc-2.27.so
00007f0921d89000      8K rw--- libc-2.27.so
00007f0921d8b000     16K rw---   [ anon ]
00007f0921d8f000    164K r-x-- ld-2.27.so
00007f0921f98000      8K rw---   [ anon ]
00007f0921fb8000      4K r---- ld-2.27.so
00007f0921fb9000      4K rw--- ld-2.27.so
00007f0921fba000      4K rw---   [ anon ]
00007fffdbe45000    132K rw---   [ stack ]
00007fffdbf9c000     12K r----   [ anon ]
00007fffdbf9f000      4K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total             4516K
```

Start virt addr, len

# Virtual to Physical Address Translation

- Each VAS segment can map to one or more physical address segments.
- Virtual-to-Physical address translations can be read from **/proc/<pid>/pagemap** binary file
- Use **sudo ./pagemap <pid> <virt-addr>** to decode the translations

```
$ ./funcvarptrs.out
pid:28542
&globalvar:0x55662de9e010
&localvar:0x7fffdbe6506c
&heapvar:0x55662e427260
&main():0x55662dc9d810
&func():0x55662dc9d7fa
Check virt-phys translations in separate
terminal.
Press any key to exit
```

```
$ sudo ./pagemap.out `pidof funcvarptrs.out`
55662de9e010
Big endian? 0
Vaddr: 0x55662de9e010, Page_size: 4096, Entry_size: 8
Reading /proc/28542/pagemap at 0x2ab316f4f0
[0]0x20 [1]0x28 [2]0x17 [3]0x0 [4]0x0 [5]0x0 [6]0x80
[7]0x81
Result: 0x8180000000172820
PFN: 0x172820
```

```
$ sudo ./pagemap.out `pidof funcvarptrs.out`
55662e427260
Big endian? 0
Vaddr: 0x55662e427260, Page_size: 4096, Entry_size: 8
Reading /proc/28542/pagemap at 0x2ab3172138
[0]0xeb [1]0x73 [2]0x19 [3]0x0 [4]0x0 [5]0x0 [6]0x80
[7]0x81
Result: 0x81800000001973eb
PFN: 0x1973eb
```

# Process meta-data

- Sizes of VAS segments and other process metadata can be seen in
  **cat /proc/<pid>/status**

```
$ cat /proc/`pidof ./funcvarptrs.out`/status
Name:   funcvarptrs.out
Umask:  0002
State:  S (sleeping)
Tgid:   28542
Ngid:   0
Pid:    28542
PPid:   21869
TracerPid:      0
Uid:    1001    1001    1001    1001
Gid:    1001    1001    1001    1001
FDSize:     256
Groups:     27 132 134 1001 64055
NStgid:     28542
NSpid:      28542
NSpgid:     28542
NSsid:      21869
VmPeak:      4512 kB
VmSize:      4512 kB
VmLck:          0 kB
VmPin:          0 kB
VmHWM:        704 kB
VmRSS:        704 kB
RssAnon:       60 kB
RssFile:      644 kB
RssShmem:       0 kB
VmData:       176 kB
VmStk:        132 kB
VmExe:          4 kB
VmLib:       2116 kB
VmPTE:         56 kB
VmSwap:         0 kB
HugetlbPages:   0 kB
CoreDumping:    0
THP enabled:    1
Threads:    1
SigQ:   0/47158
SigPnd:     0000000000000000
ShdPnd:     0000000000000000
SigBlk:     0000000000000000
SigIgn:     0000000000000000
SigCgt:     0000000000000000
CapInh:     0000000000000000
CapPrm:     0000000000000000
CapEff:     0000000000000000
CapBnd:     0000003fffffffff
CapAmb:     0000000000000000
NoNewPrivs:     0
Seccomp:    0
Speculation_Store_Bypass:       thread vulnerable
Cpus_allowed:   f
Cpus_allowed_list:      0-3
Mems_allowed:
00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,0
0000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00
000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,00000000,000
00000,00000001
Mems_allowed_list:      0
voluntary_ctxt_switches:        1
nonvoluntary_ctxt_switches:     0
```

# Backing Stores - Paging, Swapping

# Backing Stores - Paging/Swapping (1/3)

- Paging/Swapping use-cases.
  - Initial program loading
  - During memory pressure
  - During hibernation
  - During kernel dump
  - Inactive processes
- Three types of backing stores for paging/swapping of VAS segments
  - A regular file on a file-system (FS)
  - A swap file on file-system
  - A dedicated swap device (disk or disk partition)
- To see Block/FS devices, use `lsblk`
- To see swap devices/files, use `swapon --show`

```
$ lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda      8:0    0 931.5G  0 disk
├─sda1   8:1    0   512M  0 part /boot/efi
├─sda2   8:2    0  14.9G  0 part [SWAP]
├─sda3   8:3    0   954M  0 part /boot
├─sda4   8:4    0  46.6G  0 part /
├─sda5   8:5    0   954M  0 part
├─sda6   8:6    0  46.6G  0 part
├─sda7   8:7    0 186.3G  0 part /home
├─sda8   8:8    0 605.4G  0 part /misc
└─sda9   8:9    0  29.5G  0 part /share
```

```
# swapon --show
NAME      TYPE      SIZE  USED PRIO
/dev/sda2 partition 14.9G 3.2G   -2
/swapfile file         2G   0B   -3
```

# Backing Stores - Paging/Swapping (2/3)

- VAS segments - backing stores
  - Text segment is always paged from program file on FS.
  - Initialized data segments are initially paged-in from program file on FS. When modifications are done, the segments are COW'd (Copy-On-Write) to swap file/device. After COW, the segments are paged to/from swap file/device.
  - Uninitialized data and Stack segments are paged to/from swap file/device.

```
# pmap `pidof sample.out`
28142:   /tmp/sample.out
0000562724522000      4K r-x-- sample.out
0000562724722000      4K r---- sample.out
0000562724723000      4K rw--- sample.out
0000562725b28000    132K rw---   [ anon ]
00007fd1d3dec000   1948K r-x-- libc-2.27.so
00007fd1d3fd3000   2048K ----- libc-2.27.so
00007fd1d41d3000     16K r---- libc-2.27.so
00007fd1d41d7000      8K rw--- libc-2.27.so
00007fd1d41d9000     16K rw---   [ anon ]
00007fd1d41dd000    164K r-x-- ld-2.27.so
00007fd1d43e6000      8K rw---   [ anon ]
00007fd1d4406000      4K r---- ld-2.27.so
00007fd1d4407000      4K rw--- ld-2.27.so
00007fd1d4408000      4K rw---   [ anon ]
00007ffdb120d000    132K rw---   [ stack ]
00007ffdb1273000     12K r----   [ anon ]
00007ffdb1276000      4K r-x--   [ anon ]
ffffffffff600000      4K --x--   [ anon ]
 total             4516K
```

FS  Swap  FS initially, COW to Swap

# Backing Stores - Paging/Swapping (3/3)

- Each segment in virtual address space is backed by a set of contiguous areas on backing store (FS or Block device)
- The backing store offset, device and inode can be seen in **cat /proc/<pid>/maps**

```
$ cat /proc/`pidof sample.out`/maps
address                  perms   offset dev   inode       pathname
562724522000-562724523000 r-xp 00000000 08:04 68281729    /tmp/sample.out
562724722000-562724723000 r--p 00000000 08:04 68281729    /tmp/sample.out
562724723000-562724724000 rw-p 00001000 08:04 68281729    /tmp/sample.out
562725b28000-562725b49000 rw-p 00000000 00:00 0           [heap]
7fd1d3dec000-7fd1d3fd3000 r-xp 00000000 08:04 101206351   /lib/x86_64-linux-gnu/libc-2.27.so
7fd1d3fd3000-7fd1d41d3000 ---p 001e7000 08:04 101206351   /lib/x86_64-linux-gnu/libc-2.27.so
7fd1d41d3000-7fd1d41d7000 r--p 001e7000 08:04 101206351   /lib/x86_64-linux-gnu/libc-2.27.so
7fd1d41d7000-7fd1d41d9000 rw-p 001eb000 08:04 101206351   /lib/x86_64-linux-gnu/libc-2.27.so
7fd1d41d9000-7fd1d41dd000 rw-p 00000000 00:00 0
7fd1d41dd000-7fd1d4206000 r-xp 00000000 08:04 101038510   /lib/x86_64-linux-gnu/ld-2.27.so
7fd1d43e6000-7fd1d43e8000 rw-p 00000000 00:00 0
7fd1d4406000-7fd1d4407000 r--p 00029000 08:04 101038510   /lib/x86_64-linux-gnu/ld-2.27.so
7fd1d4407000-7fd1d4408000 rw-p 0002a000 08:04 101038510   /lib/x86_64-linux-gnu/ld-2.27.so
7fd1d4408000-7fd1d4409000 rw-p 00000000 00:00 0
7ffdb120d000-7ffdb122e000 rw-p 00000000 00:00 0           [stack]
7ffdb1273000-7ffdb1276000 r--p 00000000 00:00 0           [vvar]
7ffdb1276000-7ffdb1277000 r-xp 00000000 00:00 0           [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0   [vsyscall]
```

# Backing Stores - Performance

- Performance
  - File system has little overhead, but flexible to extend, shrink.
  - Partition has no overhead, but difficult to extend, shrink.
- On Linux
  - Swapping to swap file has little overhead compared to swap device due to extra FS layer of abstraction.
- FS and Swap file/device are options for memory pressure, inactivation use-cases.
- For hibernation and kernel dump use-cases, swap file/device is the only option.

```
Eg. 100 parallel direct(unbuffered) I/Os of 1GiB to swap file and swap device

Swap file  : 13m12.500s
Swap device: 10m44.449s
```

# (Recap)
# File System Basics, Operations, Dirent, I-node, Links

Excerpt from “Linux Commands” slides

# File Types

- In Unix/Linux, everything in file-system is a file.

- There are many types of files:

  Regular File

  ```
  -rw-r--r-- 1 root root 35913142 Feb  3 04:34 initrd.img-4.4.0-31-generic
  ```

  Directory

  ```
  drwxr-xr-x 5 root root     4096 Nov 14  2016 grub
  ```

  Block (buffered) device special file

  ```
  brw-rw---- 1 root disk     8, 0 Feb  2 10:40 /dev/sda
  ```

  Character (unbuffered) device special file

  ```
  crw--w---- 1 owner tty   136, 0 Feb  3 04:36 /dev/pts/0
  ```

  Symbolic Link (aka soft link)

  ```
  lrwxrwxrwx 1 root root 19 Nov 14  2016 /etc/mtab -> ../proc/self/mounts
  ```

  Socket special file

  ```
  srw-rw-rw-. 1 root root 0 Feb  3 03:34 /run/cups/cups.sock
  ```

  Named Pipe special file

  ```
  prw------- 1 root root 0 Feb  2 10:41 /run/systemd/inhibit/6.ref
  ```

# File Types

- Know file type use `ls -l` command, notice the first letter in output.
- Regular files can be further differentiated based on content.
- Know file type based on its content using `file` command. Works based on magic number stored in `/usr/share/misc/magic.mgc` and `/etc/magic`

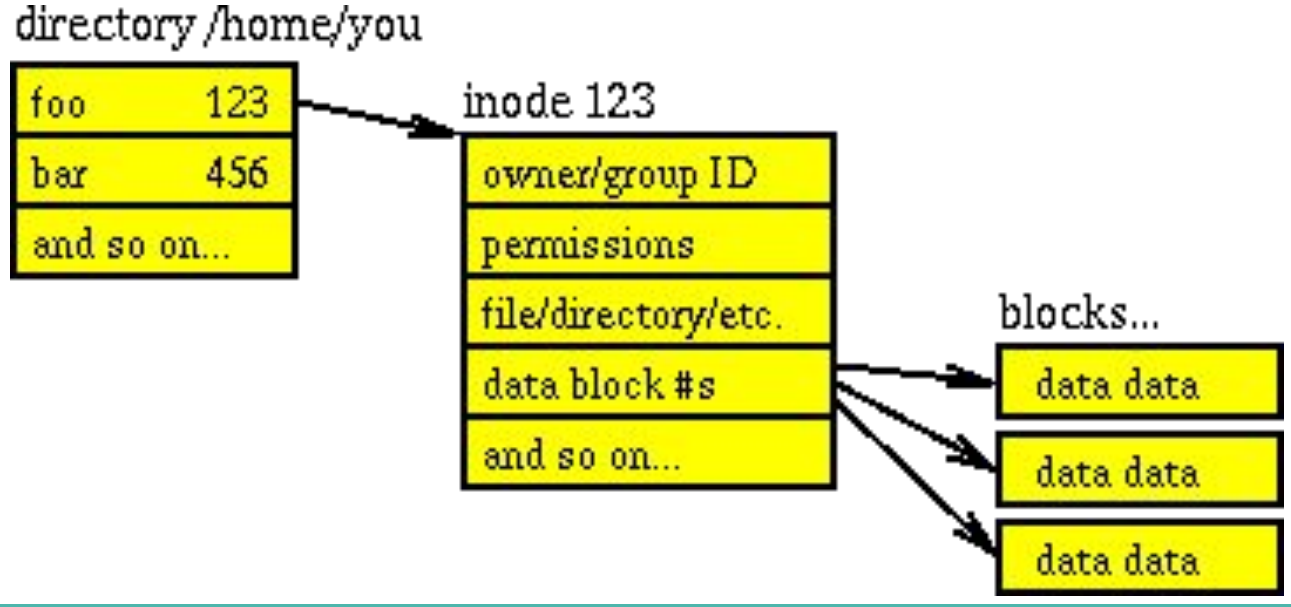
```
$ ls -l
-rw-r--r-- 1 root root 35913142 Feb  3 04:34 initrd.img-4.4.0-31-generic

$ file Documents
Documents: directory

$ file .bashrc
.bashrc: ASCII text
$ file sizesof.c
sizesof.c: C source, ASCII text
$ file Documents/ds-3808.pdf
Documents/ds-3808.pdf:  PDF document, version 1.4
```

# On-disk Index node and Directory Entry

- Every file has few ondisk data structures for metadata.
- Index node (`inode`) with a unique number (`ino`). Filename is not part of inode.
- Directory entries (`dirent`) which stores the file name, and inode number.



# Knowing file metadata (1/2)

- To know file's metadata use `stat` command.

```
$ ls -li
  6894908 drwxrwxrwx 3 maruthisi maruthisi 16 Jan 19 05:20 dir1
421776638 -rw-rw-rw- 1 maruthisi maruthisi 12 Jan 19 05:15 file1.txt

$ stat dir1
  File: 'dir1'
  Size: 16              Blocks: 0          IO Block: 4096   directory
Device: 807h/2055d      Inode: 6894908     Links: 2
Access: (0777/drwxrwxrwx)  Uid: ( 1001/maruthisi)   Gid: ( 1001/maruthisi)
Access: 2019-01-19 05:19:54.122106408 +0530
Modify: 2019-01-19 05:20:02.774092126 +0530
Change: 2019-01-19 05:20:02.774092126 +0530
 Birth: -

$ stat file1.txt
  File: 'file1.txt'
  Size: 12              Blocks: 8          IO Block: 4096   regular file
Device: 807h/2055d      Inode: 421776642   Links: 1
...
```

# Knowing file metadata (2/2)

- Attributes from dirent:
  - File: name
- Attributes in inode:
  - Size: apparent file size
  - Blocks: allocated file size (in 512 byte blocks)
  - IO Block: Unit of I/O size by underlying block device driver.
  - Device: id of device on which this inode exists.
  - Inode: inode number
  - Links: number of dirents pointing to this inode.
  - Access: permissions
  - Uid: user owner's id
  - Gid: group owner's id
  - Access: the last time the file was read
  - Modify: the last time the file was modified (content has been modified)
  - Change: the last time meta data of the file was changed (e.g. permissions)

# Symbolic link

- To create a symbolic link (symlink), use `ln -s`
  -f : force recreation
- Symlinks can be created within and across file-systems.
- Separate inode and dirent is created for symlink.
- A symlink can become dangling link if target is deleted.
- `ls -l` shows “l” in the file type column.
  -L : to traverse symlink. Useful for detecting broken links.

```
$ ln -s abc.txt pqr.txt
$ ls -li
6894904 -rw-rw-rw- 1 maruthisi maruthisi 12 Jan 19 09:48 abc.txt
6894915 lrwxrwxrwx 1 maruthisi maruthisi  7 Jan 19 09:49 pqr.txt -> abc.txt

$ ls -liL
6894918 -rw-rw-rw- 1 maruthisi maruthisi 12 Jan 19 09:56 abc.txt
6894918 -rw-rw-rw- 1 maruthisi maruthisi 12 Jan 19 09:56 pqr.txt
```

# Hard link

- To create a link (hardlink), use `ln`
  -f : force recreation
- Hardinks must be created within file-system.
- A new dirent is created for each hardlink. Inode is shared across all hardlinks of the inode.
- A hardlink cannot become dangling link if target is deleted. Link count reduces.
- `ls -l` shows “-” in the file type column. Link count could be used to differentiate.

```
$ ln abc.txt stu.txt
$ ls -li
6894913 -rw-rw-rw- 2 maruthisi maruthisi 12 Jan 19 09:59 abc.txt
6894913 -rw-rw-rw- 2 maruthisi maruthisi 12 Jan 19 09:59 stu.txt
```

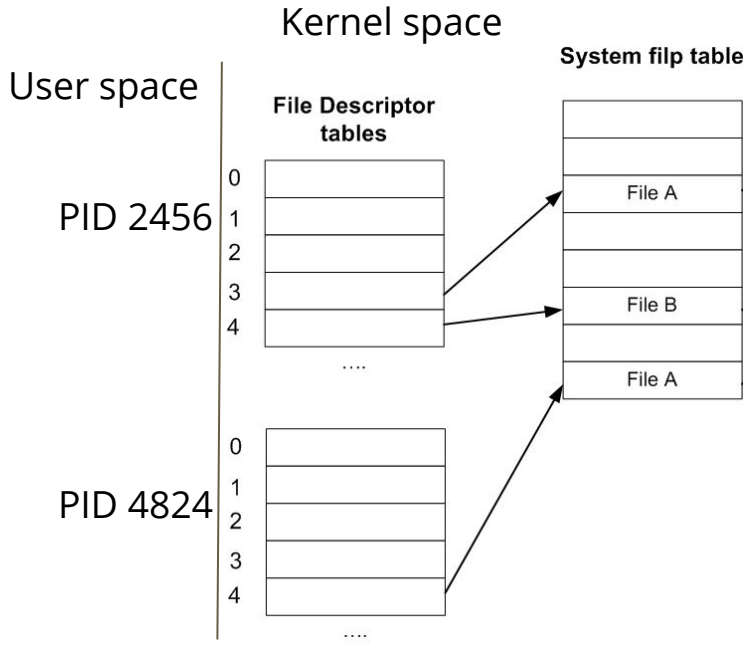
# File I/O

# Types of file I/O

- Three ways to read/write from/to a file.
- ANSI C APIs (or other language specific APIs)
  - `fopen(3), fclose(3)`
  - `fread(3), fget*(3), fscanf(3)`
  - `fwrite(3), fput*(3), fprintf(3)`
- Syscalls
  - `open(2), close(2)`
  - `read(2), pread(2), readv(2)`
  - `write(2), pwrite(2), writev(2)`
- Memory Mapped I/O
  - `mmap(2), munmap(2)`
  - Pointer indirection in *`r-value`*
  - Pointer indirection in *`l-value`*

# File Descriptor Table

- A file-descriptor table is a per-process data-structure with `POSIX_OPEN_MAX` entries.
- It exists in a virtual address space segment called u-area.
- Each file-descriptor entry contains a pointer to a system-wide file entry.
- By default three file-descriptors are opened for every process: `stdin`, `stdout`, `stderr`.




```
$ cat  1> /tmp/out 2> /tmp/err &

$ ls -l /proc/`pidof cat`/fd/
total 0
lrwx------ 1 maruthisi maruthisi 64 Mar 30 13:16 0 -> /dev/pts/2
l-wx------ 1 maruthisi maruthisi 64 Mar 30 13:16 1 -> /tmp/out
l-wx------ 1 maruthisi maruthisi 64 Mar 30 13:16 2 -> /tmp/err
```

# File I/O - System Calls

# open(2)

- `open(2)` system call opens a file.
- On success, the system call returns a non -ve integer (called file-descriptor), and -ve number on failure.
- New file-structure (`struct file`) is created in the Kernel.
- The file structure is a per-open, per-process data structure, which primarily contains read/write position (`f_pos`).
- The position is set to zero.
- An entry is added to the calling process's file descriptor table, which contains pointer to the file-structure.

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>

int main(int argc, char *argv[])
{
      int fd;
      fd = open(argv[1], O_RDWR);
      if (fd<0) { /* error */ }

      // read/write

      close(fd);

      return 0;
}
```

# close(2)

- `close(2)` system call closes an open file
- On success, the system call returns 0, and -ve number on failure.
- The calling process's file-descriptor table's entry is updated to wipe out pointer to the file-structure.
- File structure is deleted.

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>

int main(int argc, char *argv[])
{
      int fd;
      fd = open(argv[1], O_RDWR);
      if (fd<0) { /* error */ }

      // read/write

      close(fd);

      return 0;
}
```

# write(2)

- `write(2)` system call writes given character stream to an open file
- On success, the system call returns number of bytes written, and -ve number on failure.
- In every call, the position (`f_pos`) in file-structure is incremented by number-of-bytes-written successfully.

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
#define BUFLEN 256

int main(int argc, char *argv[])
{
      int fd, ret;
      char buf[BUFLEN];
      fd = open(argv[1], O_RDWR);

      // Process data

      ret = write(fd, buf, BUFLEN);
      if (ret<0) { /* error */ }

      close(fd);

      return 0;
}
```

# read(2)

- `read(2)` system call reads given character stream from an open file
- On success, the system call returns number of bytes read, and -ve number on failure.
- In every call, the position (`f_pos`) in file-structure is incremented by number-of-bytes-read successfully.

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
#define BUFLEN 256

int main(int argc, char *argv[])
{
      int fd, ret;
      char buf[BUFLEN];
      fd = open(argv[1], O_RDWR);

      ret = read(fd, buf, BUFLEN);
      if (ret<0) { /* error */ }

      // Process the data

      close(fd);

      return 0;
}
```

# Memory Mapped I/O

# mmap(2)

- `mmap(2)` system call creates virtutal-to-physical address mapping of a given file's range.
- On success, the system call returns a pointer(virtual address), and `MAP_FAILED` on failure.
- File range must be multiple of page-size, and aligned to page-size.
- This pointer can be used to read the file's content without invoking any system calls.
- There is no system call overhead due to user/kernel mode switching.

```
#include<stdio.h>
#include<fcntl.h>
#include <sys/mman.h>
#define MAPLEN 4096

int main(int argc, char *argv[])
{
      int fd, ret;
      char *mem;
      fd = open(argv[1], O_RDWR);

      // map file into user-space segment.
      mem = (char*) mmap(NULL, MAPLEN,
                PROT_READ, MAP_SHARED, fd,
                0 /*off*/);
      if (mem == MAP_FAILED) { /* error */ }

      // Read/Write

      munmap(mem, MAPLEN);
      close(fd);
      return 0;
}
```

# Memory Mapped Write

- Use pointer indirection on memory address returned by mmap(2).
- There is no need of additional buffer in program.
- For each byte accessed in a new page, page-faults are generated to the file-system/driver.
- If page frame already exist for an address, page fault is not generated.
- The file-sytem or driver flushes dirty pages to disk/device.

```
#include<stdio.h>
#include<fcntl.h>
#include <sys/mman.h>
#define MAPLEN 4096

int main(int argc, char *argv[])
{
      int fd, ret;
      char *mem;
      fd = open(argv[1], O_RDWR);

      // map file into user-space segment.
      mem = (char*) mmap(NULL, MAPLEN,
                  PROT_WRITE, MAP_SHARED, fd,
                  0 /*off*/);

      // Write
      for(int i=0; i<MAPLEN; i++)
            *(mem+i) = "x";

      munmap(mem, MAPLEN);
      close(fd);
      return 0;
}
```

# Memory Mapped Read

- Use pointer indirection on memory address returned by mmap(2).
- There is no need of additional buffer in program.
- For each byte accessed in a new page, page-faults are generated to the file-system/driver.
- If page frame already exist for an address, page fault is not generated.
- The file-system or driver services page-faults and populates the pages with on-disk/device data.

```
#include<stdio.h>
#include<fcntl.h>
#include <sys/mman.h>
#define MAPLEN 4096

int main(int argc, char *argv[])
{
      int fd, ret;
      char *mem;
      fd = open(argv[1], O_RDWR);

      // map file into user-space segment.
      mem = (char*) mmap(NULL, MAPLEN,
                PROT_READ, MAP_SHARED, fd,
                0 /*off*/);

      // Read
      for(int i=0; i<MAPLEN; i++)
           printf("%c", *(mem+i))

      munmap(mem, MAPLEN);
      close(fd);
      return 0;
}
```

# munmap(2)

- `munmap(2)` system call removes virtutal-to-physical address mapping of a given file's range.
- On success, the system call returns 0, and -1 on failure.

```
#include<stdio.h>
#include<fcntl.h>
#include <sys/mman.h>
#define MAPLEN 4096

int main(int argc, char *argv[])
{
      int fd, ret;
      char *mem;
      fd = open(argv[1], O_RDWR);

      // map file into user-space segment.
      mem = (char*) mmap(NULL, MAPLEN,
                PROT_READ, MAP_SHARED, fd,
                0 /*off*/);

      // Read/Write

      munmap(mem, MAPLEN);
      close(fd);

      return 0;
}
```

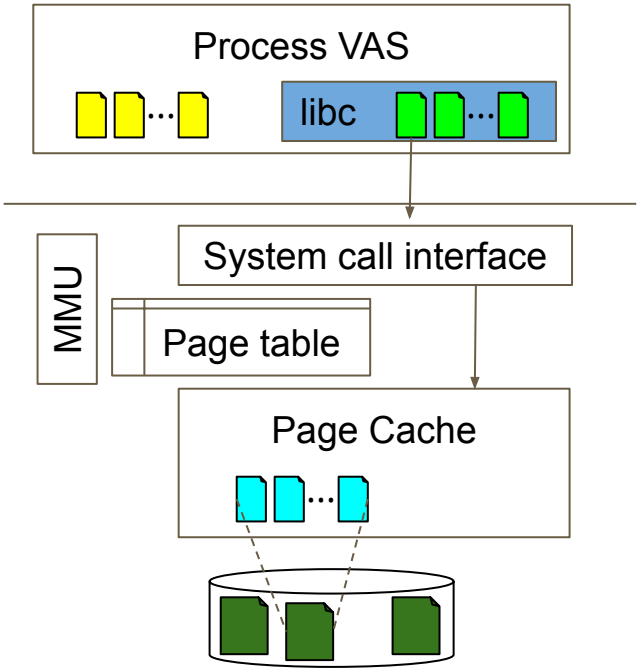
# Multiple buffering problem

There could be 3 copies of a file's fragment when using ANSI libc APIs.

1. File fragments are cached in buffers of OS's page cache.
   - One copy per system.

2. File fragments are also cached in libc's buffer.
   - One copy per process, per file pointer

3. File fragments are ultimately read into / written from program's buffer/array.
   - One copy per process, per file pointer.




Solaris prevents one copy by memory mapping file inside the libc's APIs.

# References

# References


- Linux Systems programming in C++, Terrence Chan, PHI.
- Advanced Programming in Unix Environment, Richard Stevens, PHI.
- Linux Systems Programming, Robert Love, Oreilly.

# Q & A