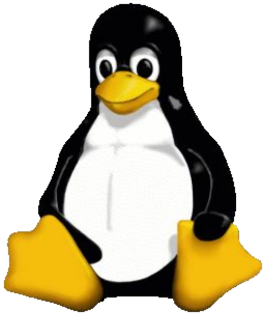


# Linux Systems Programming

Maruthi S. Inukonda

01 April 2019

# License

# Agenda

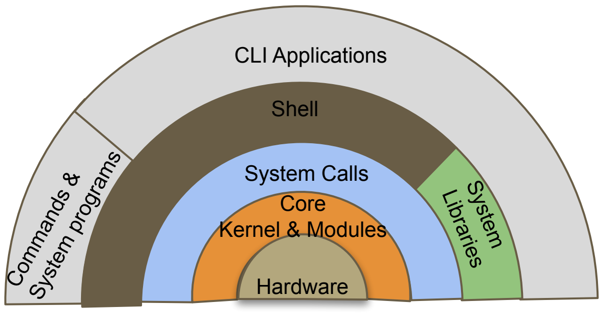
- Linux Architecture (Recap)
- System Libraries - Linking, Loading
- Process Virtual Address Space, Swapping
- Process Hierarchy

# 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 Libraries

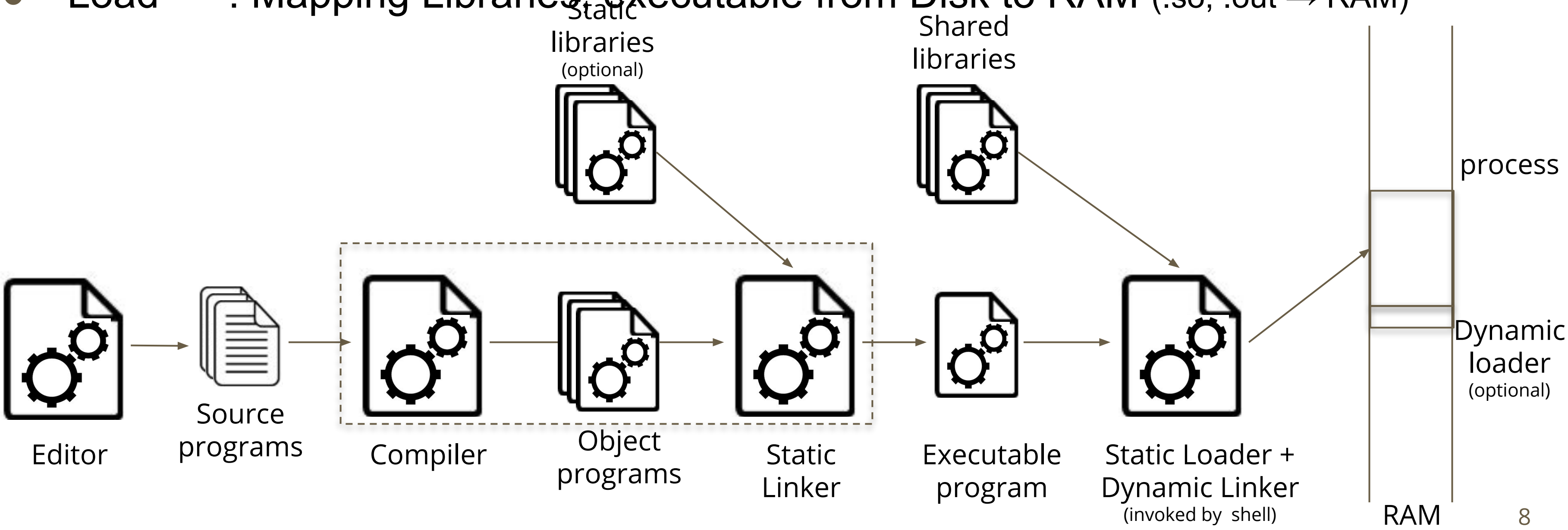
- Reusable routines packaged as `.a` or `.so`
- Every command loads its dependent  libraries at launch time.
- Types of libraries
  - Archive libraries (.a)
  - Shared object (.so)
- Types of linking
  - Compile time (Static) (only with .a) - Deprecated.
  - Load time - (only with .so)
  - Run time (Dynamic) - (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)
```

# 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 happens 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 libraries 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 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 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 happens at run time.
- To dynamically load a library, use `dlopen()`, `dlsym()`, `dlclose()` in the code.

```
...
#include <dlfcn.h>

int main()
{
     ...
      void *handle;
      int (*fact)(int);
      char *error;

      handle = dlopen("libfact.so", RTLD_LAZY);
      if (!handle) { ... }

      dlerror();  /* Clear any existing error */
      *(void **) (&fact) = dlsym(handle, "fact");

      if ((error = dlerror()) != NULL)  { ... }

      f = (*fact)(n);

      dlclose(handle);
      ...
}
```

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`, `-lrdynamic`.

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 (...)
```

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

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

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.

# Process Virtual Address Space, Swapping

# Virtual Address Space (1/2)

- A process's memory footprint is divided into multiple segments, broadly:
  - Text
  - Initialized data (global variables)
  - Uninitialized data (heap)
  - Stack
- 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)
- Libraries do have their own Text, Initialized data segments. Stack, Heap are shared with program.

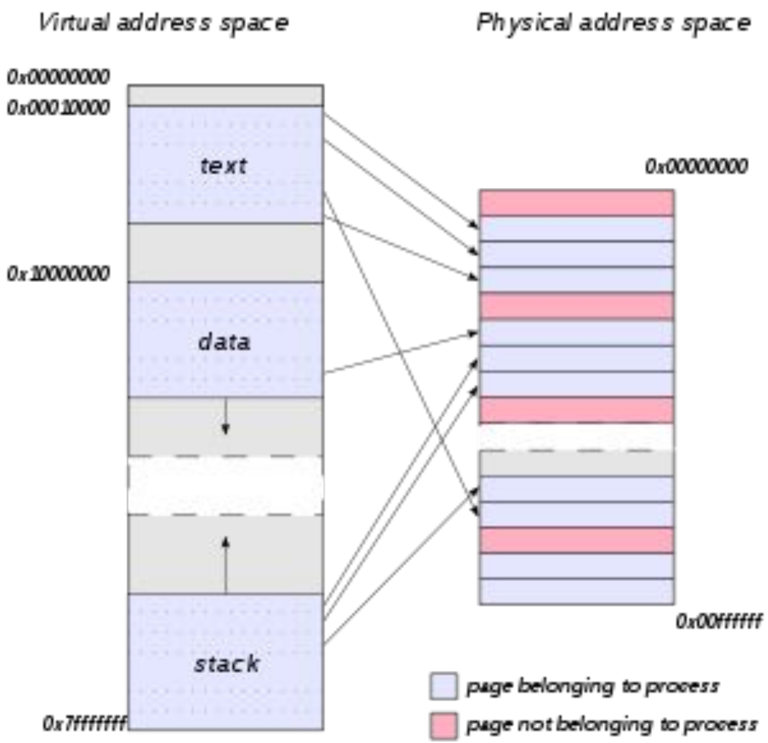



Image courtesy: https://www.wikipedia.org

# Virtual Address Space (2/2)

- Every process in Linux has a separate virtual address space.
- To see VAS of a process,
  use cat /proc/<pid>/maps or `pmap <pid>`

- Each segment in virtual address space is backed by a set of contiguous areas on backing store (FS or Block device)

```
# pmap `pidof sample`
[sudo] password for maruthisi:
25551:   ./sample
0000000000400000      4K r-x-- sample
0000000000600000      4K r---- sample
0000000000601000      4K rw--- sample
0000000002458000    132K rw---   [ anon ]
00007fb993e4b000   1792K r-x-- libc-2.23.so
00007fb99400b000   2048K ----- libc-2.23.so
00007fb99420b000     16K r---- libc-2.23.so
00007fb99420f000      8K rw--- libc-2.23.so
00007fb994211000     16K rw---   [ anon ]
00007fb994215000    152K r-x-- ld-2.23.so
00007fb9943f4000     12K rw---   [ anon ]
00007fb99443a000      4K r---- ld-2.23.so
00007fb99443b000      4K rw--- ld-2.23.so
00007fb99443c000      4K rw---   [ anon ]
00007ffd6cfd1000    132K rw---   [ stack ]
00007ffd6cff7000     12K r----   [ anon ]
00007ffd6cffa000      8K r-x--   [ anon ]
Fffffffff600000      4K r-x--   [ anon ]
 total             4356K
```

Start virt addr, len

# Swapping, Backing Stores (1/2)

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

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

# Swapping, Backing Stores (2/2)

- VAS segments - backing stores
  - Text segment is always paged from file-system program file.
  - Initialized data segments are intialled paged-in from file-system program file. And paged-out to/from swap file/device.
  - Uninitialized data and Stack segments are paged to/from swap file/device.

```
# pmap `pidof sample`
[sudo] password for maruthisi:
25551:   ./sample
0000000000400000      4K r-x-- sample
0000000000600000      4K r---- sample
0000000000601000      4K rw--- sample
0000000002458000    132K rw--- [ anon ]
00007fb993e4b000   1792K r-x-- libc-2.23.so
00007fb99400b000   2048K ----- libc-2.23.so
00007fb99420b000     16K r---- libc-2.23.so
00007fb99420f000      8K rw--- libc-2.23.so
00007fb994211000     16K rw--- [ anon ]
00007fb994215000    152K r-x-- ld-2.23.so
00007fb9943f4000     12K rw--- [ anon ]
00007fb99443a000      4K r---- ld-2.23.so
00007fb99443b000      4K rw--- ld-2.23.so
00007fb99443c000      4K rw--- [ anon ]
00007ffd6cfd1000    132K rw--- [ stack ]
00007ffd6cff7000     12K r---- [ anon ]
00007ffd6cffa000      8K r-x-- [ anon ]
Ffffffffff600000      4K r-x-- [ anon ]
 total             4356K
```

FS      Swap      FS initially, COW to Swap

# 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 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
```

# Processes Hierarchy

# Process Hierarchy

- All processes in Linux form a hierarchy.
- Top most process is `init` or `systemd` (new Linux distros).
- Child processes are created using `fork(2)` system call.
- Parent: Process call the `fork(2)`
- Child: Newly created process by the `fork(2)`.
- Orphan: Child process that terminates before parent. The (orphan) child is reparented to init process.
- If child terminates before parent, the parent gets a `SIGCHLD` signal. Parent cleans up using `wait(2)`.
- Zombie: Terminated child processes that are not yet cleaned up by the parent.

# fork(2)

- `fork(2)` system call creates a duplicate process of calling process.
- Return value: > 0 to the parent, == 0 to the child. If error, < 0 to parent.
- A new process descriptor (`struct task`) is created for the child.
- Parent's page table (and hence VAS) is duplicated.
- All physical pages are also duplicated. (prior to Linux kernel 2.2) [ Time consuming ]
- All page mapping descriptors marked copy-on-write (after 2.2)
- After the control returns, both parent and child continue to run from next line.

```
pid_t pid;

pid = fork();
if (pid > 0)
     printf ("I am the parent of pid=%d!\n", pid);
else if (!pid)
     printf ("I am the child!\n");
else if (pid == -1)
     perror ("fork");
```

# execve(2)

- `execve(2)` system call replaces calling process's VAS with new program.
- The system call never returns.
- No new process descriptor (`struct task`) is created.
- The calling process's page table (and hence VAS) is re-created for new program.
- All required physical pages are also allocated for the new program.

```
pid_t pid;
pid = fork();
if (pid == -1)
	perror("fork");
/* the child ... */
if (!pid) {
	const char *args[] = { "ls", NULL };
	int ret;
	ret = execv("/bin/ls", args);
	if (ret == -1) {
		perror ("execv");
		exit (EXIT_FAILURE);
	}
}
```

# vfork(2)

- `vfork(2)` system call creates a duplicate process of calling process.
- Return value: > 0 to the parent, == 0 to the child. If error, < 0 to parent.
- A new process descriptor (`struct task`) is created for the child.
- Parent's page table (and hence VAS) is NOT duplicated.
- After the control returns from `vfork(2)` child continues to run from next line of code. But the parent is suspended till child calls `execve(2)` or `exit(2)`.
- Child must not make any modifications before calling `execve(2)`.

```
pid_t pid;
pid = vfork();
if (pid == -1)
    perror("vfork");
/* the child ... */
if (!pid) {
    const char *args[] = { "ls", NULL };
    int ret;
    ret = execv("/bin/ls", args);
    if (ret == -1) {
        perror("execv");
        exit(EXIT_FAILURE);
    }
}
```

*vfork() gave big improvement before Linux implemented COW. With COW based VMM, vfork() is not much appealing.*

# clone(2)

- `clone(2)`, Linux specific system call creates a child process or child thread.
- Return value: > 0 to the parent, == 0 to the child. If error, < 0 to parent.
- `clone(2)`, is the underlying re-usable system call for `fork(2)`, `vfork(2)`, `pthread_create(3)`
- The behaviour of clone is adaptable based on `child_stack`, `flags`, `child_tidptr` arguments.

**fork() is implemented as:**

```
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=<addr>)
```

**vfork() is implemented as:**

```
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=<addr>)
```

**pthread_create() is implemented as:**

```
clone(child_stack=<addr>,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS
|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tidptr=<addr>, tls=<addr>,
child_tidptr=<addr>)
```

# Command Execution by Shell

- Every time a command is entered at shell prompt, shell does `vfork(2)` followed by `exec(2)`.
- The parent process (shell) does `wait(2)` to get the exit status of child process. And sets it as `$?` environment variable.
- A command pipeline, involves many calls to `vfork(2)-exec(2)` and a `wait(2)`.
- A shell script involves humongous calls to `vfork(2)-exec(2)-wait(2)`.

```
$ ls
sample   sample.c
```

Here `bash` calls `vfork()` followed by `wait()`. The child calls `exec("/bin/ls")`.

```
$ ./sample
Hello
```

Here `bash` calls `vfork()` followed by `wait()`. The child calls `exec("./sample")`.

```
$ ls | wc -l
2
```

Here `bash` calls one `vfork()` for `wc`. The first child calls `exec("/usr/bin/wc")`. The `bash` continues and calls one more `vfork()` for `ls`. The second child calls `exec("/bin/ls")`. Then the `bash` calls `wait()` for first child (the wc).

# Memory Mapping

# Multiple buffering problem (1/2)

- Three ways to read/write from/to a file.
- ANSI C 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)
  - write(2), pwrite(2)
- mmap(2), munmap(2)
  - Memory mapped read
  - Memory mapped write

# Multiple buffering problem (2/2)

- There could be 3 copies of a file's fragment.
- File fragments are ultimately read into / written from program's buffer/array. Copy # 1
  - One copy per process, per file pointer.
- File fragments are also cached in libc's buffer. Copy #2
  - One copy per process, per file pointer
- File fragments are cached in OS's page cache. Copy #3
  - One copy per system.

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