Advanced System Programming

Advanced System Programming

Agenda

- File operation
- Tracing
- Basic Debugging

File operation - Access

- In Linux, access command is used to check whether the calling program has access to a specified file. It can be used to check whether a file exists or not. The check is done using the calling process's real UID and GID.

int access(const char *pathname, int mode);

Here, the first argument takes the path to the directory/file and the second argument takes flags R_OK, W_OK, X_OK or F_OK.

- F_OK flag : Used to check for existence of file.
- R_OK flag : Used to check for read permission bit.
- W_OK flag : Used to check for write permission bit.
- X_OK flag : Used to check for execute permission bit.

Note: If access() cannot access the file, it will return -1 or else it will be 0.

File operation

dup and dup2 Functions:

- An existing file descriptor is duplicated by either of the following functions.

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

Both return: new file descriptor if OK,-1 on error.

- The new file descriptor returned by dup is guaranteed to be the lowest-numbered available file descriptor.

File operation

- With dup2, we specify the value of the new descriptor with the filedes2 argument. If filedes2 is already open, it is first closed. If filedes equals filedes2, then dup2 returns filedes2 without closing it.
- dup(filedes); is equivalent to

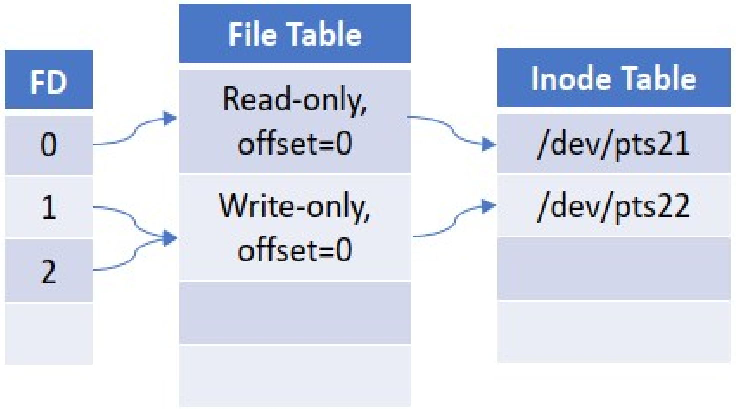
  fcntl(filedes, F_DUPFD, 0);
- dup2(filedes, filedes2); is equivalent to

  close(filedes2);

  fcntl(filedes, F_DUPFD, filedes2);

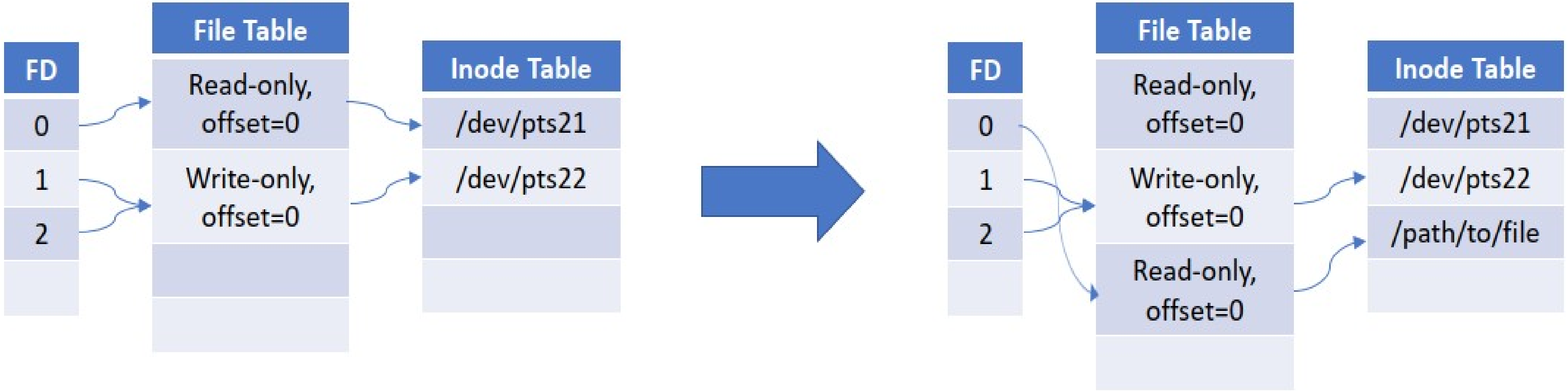
Example: Input Redirection

- Goal: instead of reading from stdin, read from another source



Example: Input Redirection

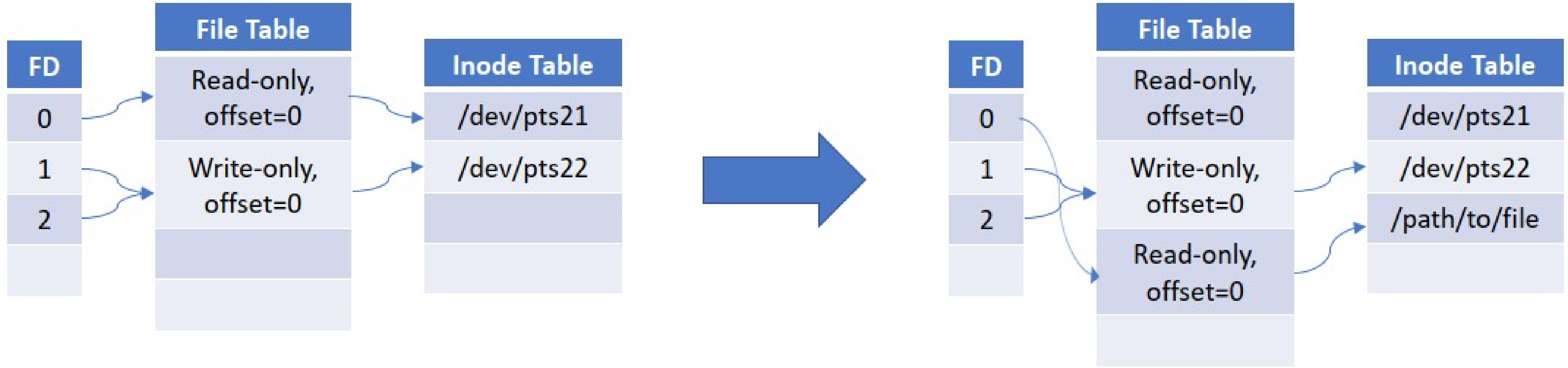
- Goal: instead of reading from stdin, read from another source



File operation

Example: Input Redirection

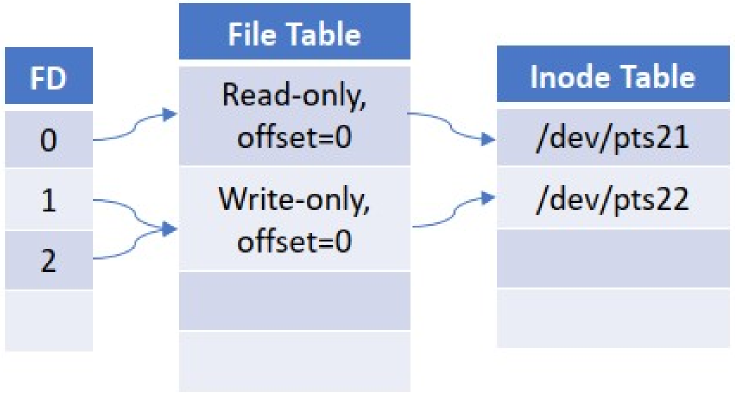
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

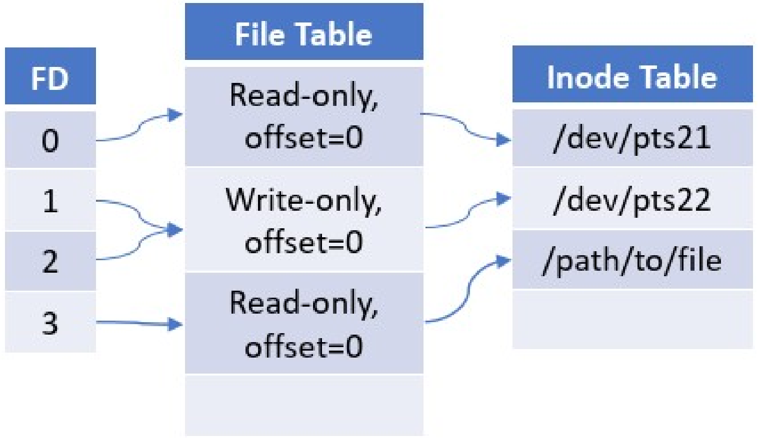
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

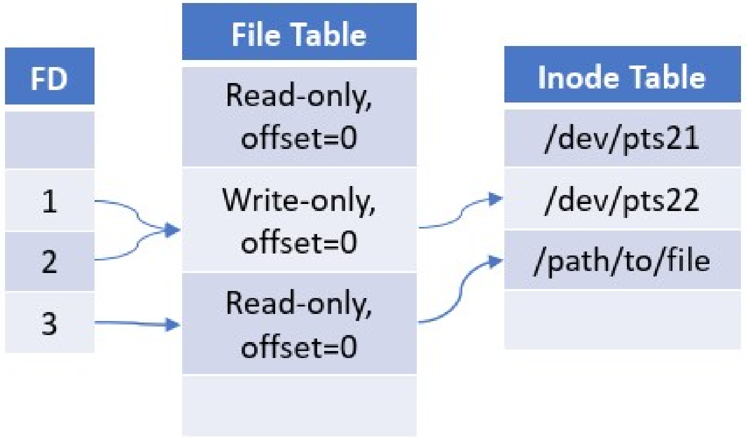
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

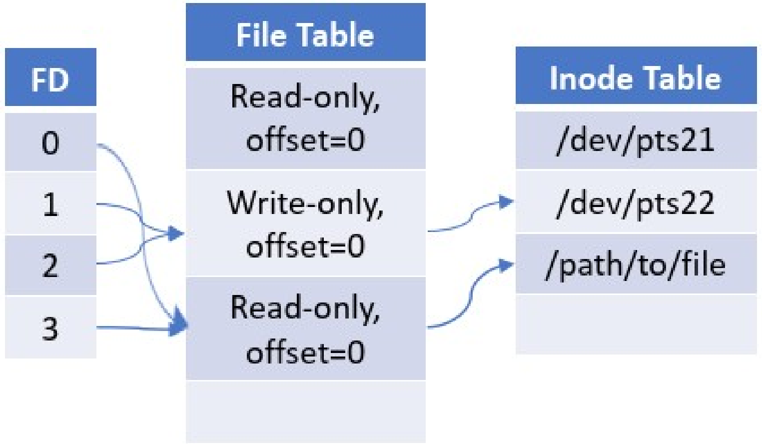
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

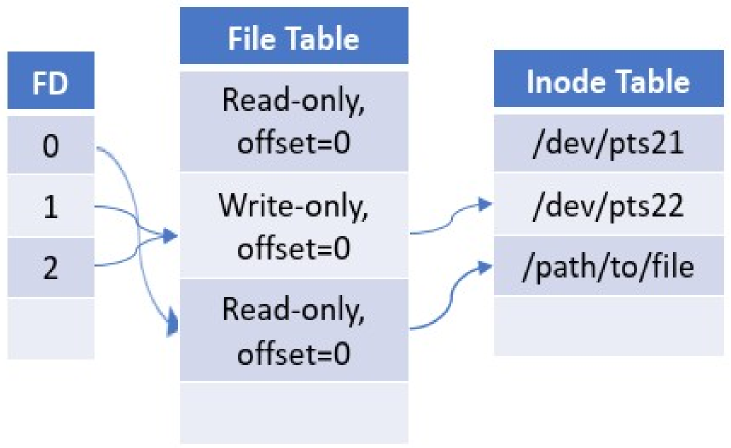
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

File operation – MMAP

- The mmap() function is used for mapping between a process address space and either files or devices. When a file is mapped to a process address space, the file can be accessed like an array in the program. This is one of the most efficient ways to access data in the file and provides a seamless coding interface that is natural for a data structure that can be assessed without he abstraction of reading and writing from files.

Header File:

#include <sys/mman.h>

Syntax:

void * mmap (void *address, size_t length, int protect, int flags, int filedes, off_t offset)

File operation – MMAP

Arguments:

The function takes 6 arguments:

- **address:**

This argument gives a preferred starting address for the mapping. If another mapping does not exist there, then the kernel will pick a nearby page boundary and create the mapping; otherwise, the kernel picks a new address. If this argument is NULL, then the kernel can place the mapping anywhere it sees fit.

- **length:**

This is the number of bytes which to be mapped.

File operation – MMAP

- **protect:**

This argument is used to control what kind of access is permitted. This argument may be logical 'OR' of the following flags PROT_READ | PROT_WRITE | PROT_EXEC | PROT_NONE. The access types of read, write and execute are the permissions on the content.

- **flags:**

This argument is used to control the nature of the map. Following are some common values of the flags:

- **MAP_SHARED:** This flag is used to share the mapping with all other processes, which are mapped to this object. Changes made to the mapping region will be written back to the file.
- **MAP_PRIVATE:** When this flag is used, the mapping will not be seen by any other processes, and the changes made will not be written to the file.

File operation – MMAP

- **MAP_ANONYMOUS / MAP_ANON**: This flag is used to create an anonymous mapping. Anonymous mapping means the mapping is not connected to any files. This mapping is used as the basic primitive to extend the heap.
- **MAP_FIXED:** When this flag is used, the system has to be forced to use the exact mapping address specified in the address If this is not possible, then the mapping will be failed.

- **filedes:**

This is the file descriptor which has to be mapped.

- **offset:**

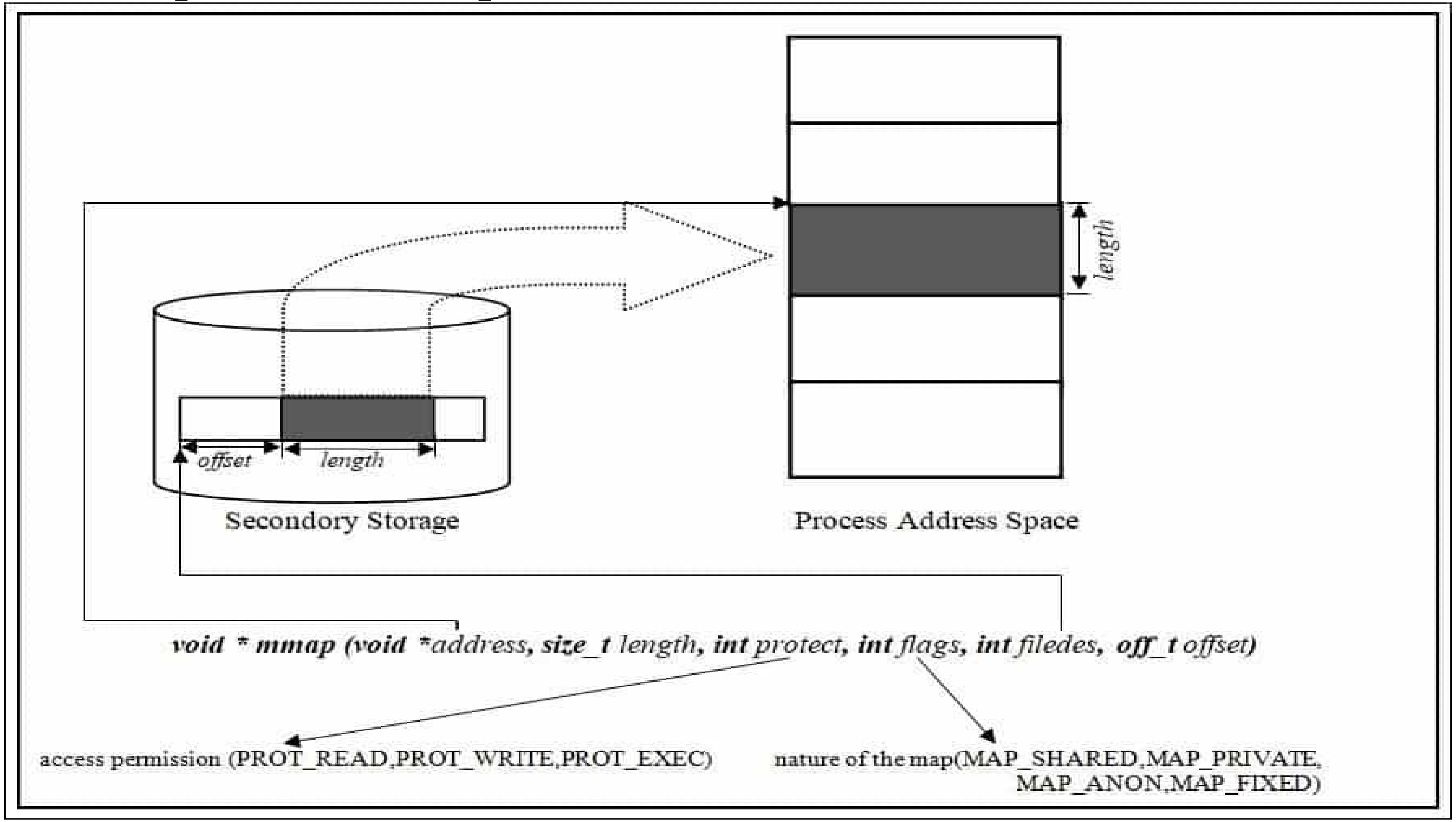
This is offset from where the file mapping started. In simple terms, the mapping connects to (offset) to (offset+length-1) bytes for the file open on filedes descriptor.

File operation – MMAP

Return values:

On success, the mmap() returns 0; for failure, the function returns MAP_FAILED.

Pictorially, we can represent the map function as follows:



File operation – MMAP

- For unmap the mapped region munmap() function is used :

Syntax:

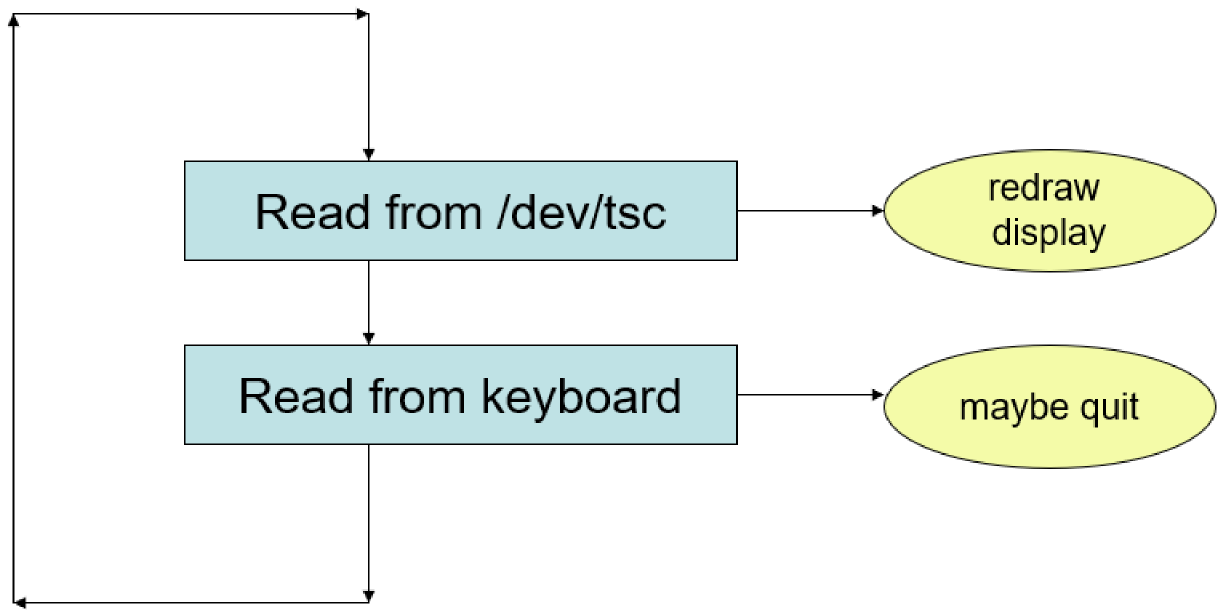
int munmap(void *address, size_t length);

Return values:

On success, the munmap() returns 0; for failure, the function returns -1.

File operation

Illustrates I/O Multiplexing



More than just one source for input

File operation

Remember ‘read()’ semantics

If device-driver’s ‘read()’ function is called before the device has any data available, then the calling task is expected to ‘sleep’ on a wait-queue until the device has at least one byte of data ready to return.

The keyboard’s device-driver behaves in exactly that expected way: it ‘blocks’ if we try to read when there are no keystrokes.

File operation

Blocking versus Multiplexing

If we want to read from multiple devices, we are not able to do it properly with the read() system-call

If one device has no data, our task will be blocked, and so can’t read data that may become available from some other device

File operation

Idea: use ‘nonblocking’ i/o

When we ‘open()’ a device-file, we could specify the ‘O_NONBLOCK’ i/o mode

File operation

```
char buf[BUFSIZ];
ssize_t nr;

start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1) {
        if (errno == EINTR)
                goto start; /* oh shush */
        if (errno == EAGAIN)
                /* resubmit later */
        else
                /* error */
}
```

File operation

Linux provides two system calls for truncating the length of a file, both of which are defined and required (to varying degrees) by various POSIX standards. They are:

```
#include <unistd.h>
#include <sys/types.h>

int ftruncate (int fd, off_t len);
```

and:

```
#include <unistd.h>
#include <sys/types.h>

int truncate (const char *path, off_t len);
```

Both system calls truncate the given file to the length given by `len`. The `ftruncate( )` system call operates on the file descriptor given by `fd`, which must be open for writing. The `truncate( )` system call operates on the filename given by `path`, which must be writable. Both return `0` on success. On error, they return `-1`, and set `errno` as appropriate.

File operation

The `select( )` system call provides a mechanism for implementing synchronous multiplexing I/O:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

A call to `select( )` will block until the given file descriptors are ready to perform I/O, or until an optionally specified timeout has elapsed.

The `timeout` parameter is a pointer to a `timeval` structure, which is defined as follows:

```
#include <sys/time.h>

struct timeval {
        long tv_sec;          /* seconds */
        long tv_usec;         /* microseconds */
};
```

File operation

The `poll( )` system call is System V's multiplexed I/O solution. It solves several deficiencies in `select( )`, although `select( )` is still often used (again, most likely out of habit, or in the name of portability):

```
#include <sys/poll.h>

int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

Unlike `select( )`, with its inefficient three bitmask-based sets of file descriptors, `poll( )` employs a single array of `nfds` `pollfd` structures, pointed to by `fds`. The structure is defined as follows:

```
#include <sys/poll.h>

struct pollfd {
        int fd;             /* file descriptor */
        short events;       /* requested events to watch */
        short revents;      /* returned events witnessed */
};
```

Each `pollfd` structure specifies a single file descriptor to watch. Multiple structures may be passed, instructing `poll( )` to watch multiple file descriptors. The `events` field of each structure is a bitmask of events to watch for on that file descriptor. The user sets this field. The `revents` field is a bitmask of events that were witnessed on the file descriptor. The kernel sets this field on return. All of the events requested in the `events` field may be returned in the `revents` field. Valid events are as follows:

File operation

POLLIN

There is data to read.

POLLRDNORM

There is normal data to read.

POLLRDBAND

There is priority data to read.

POLLPRI

There is urgent data to read.

POLLOUT

Writing will not block.

File operation

`POLLWRNORM`

Writing normal data will not block.

`POLLWRBAND`

Writing priority data will not block.

`POLLMSG`

A `SIGPOLL` message is available.

In addition, the following events may be returned in the `revents` field:

`POLLER`

Error on the given file descriptor.

`POLLHUP`

Hung up event on the given file descriptor.

`POLLNVAL`

The given file descriptor is invalid.

File operation

File Operations

- File handle : FILE *
- Open a file : fopen
- Input/Output
  - Character IO : getc, putc
  - String IO : fgets, fputs
  - Formatted IO : fscanf, fprintf
  - Raw IO : fread, fwrite
- Close a file : fclose
- Other operations :
  - fflush, fseek, freopen

File operation

Files are opened for reading or writing via `fopen( )`:

```
#include <stdio.h>

FILE * fopen (const char *path, const char *mode);
```

This function opens the file `path` according to the given modes, and associates a new stream with it.

The `fclose( )` function closes a given stream:

```
#include <stdio.h>

int fclose (FILE *stream);
```

Any buffered and not-yet-written data is first flushed. On success, `fclose( )` returns `0`. On failure, it returns `EOF` and sets `errno` appropriately.

File operation

The function `fdopen( )` converts an already open file descriptor (`fd`) to a stream:

```
#include <stdio.h>

FILE * fdopen (int fd, const char *mode);
```

The possible modes are the same as for `fopen( )`, and must be compatible with the modes originally used to open the file descriptor. The modes `w` and `w+` may be specified, but they will not cause truncation. The stream is positioned at the file position associated with the file descriptor.

File operation

The function `fgets( )` reads a string from a given stream:

```
#include <stdio.h>

char * fgets (char *str, int size, FILE *stream);
```

This function reads up to *one less* than `size` bytes from `stream`, and stores the results in `str`. A null character (`\0`) is stored in the buffer after the bytes read in. Reading stops after an `EOF` or a newline character is reached. If a newline is read, the `\n` is stored in `str`.

For some applications, reading individual characters or lines is insufficient. Sometimes, developers want to read and write complex binary data, such as C structures. For this, the standard I/O library provides `fread( )`:

```
#include <stdio.h>

size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

A call to `fread( )` will read up to `nr` elements of data, each of `size` bytes, from `stream` into the buffer pointed at by `buf`. The file pointer is advanced by the number of bytes read.

File operation

Individual characters and lines will not cut it when programs need to write complex data. To directly store binary data such as C variables, standard I/O provides `fwrite( )`:

```
#include <stdio.h>

size_t fwrite (void *buf,
               size_t size,
               size_t nr,
               FILE *stream);
```

A call to `fwrite( )` will write to `stream` up to `nr` elements, each `size` bytes in length, from the data pointed at by `buf`. The file pointer will be advanced by the total number of bytes written.

```
#include <stdio.h>

int fseek (FILE *stream, long offset, int whence);
```

If `whence` is set to `SEEK_SET`, the file position is set to `offset`. If `whence` is set to `SEEK_CUR`, the file position is set to the current position plus `offset`. If `whence` is set to `SEEK_END`, the file position is set to the end of the file plus `offset`.

Upon successful completion, `fseek( )` returns `0`, clears the `EOF` indicator, and undoes the effects (if any) of `ungetc( )`. On error, it returns `-1`, and `errno` is set appropriately. The most common errors are invalid stream (`EBADF`) and invalid `whence` argument (`EINVAL`).

File operation

Writing to a file using fprintf( )

fprintf( ) works just like printf and sprintf except that its first argument is a file pointer.

```
FILE *fptr;
fptr= fopen ("file.dat","w");
/* Check it's open */
fprintf (fptr,"Hello World!\n");
```

File operation

Reading Data Using fscanf( )

- We also read data from a file using fscanf( ).

```
FILE *fptr;
fptr= fopen (“input.dat”,“r”);
/* Check it's open */
if (fptr==NULL)
  {
    printf(“Error in opening file \n”);
   }
fscanf(fptr,“%d%d”,&x,&y);
```

input.dat

20 30

x=20
y=30

File operation

Unlike `lseek( )`, `fseek( )` does not return the updated position. A separate interface is provided for this purpose. The `ftell( )` function returns the current stream position of `stream`:

```
#include <stdio.h>

long ftell (FILE *stream);
```

```
#include <stdio.h>

void rewind (FILE *stream);
```

This invocation:

```
rewind (stream);
```

resets the position back to the start of the stream. It is equivalent to:

```
fseek (stream, 0, SEEK_SET);
```

except that it also clears the error indicator.

File operation

File operation Handson

File operation

```
#include <unistd.h>

ssize_t pread (int fd, void *buf, size_t count, off_t pos);
```

This call reads up to `count` bytes into `buf` from the file descriptor `fd` at file position `pos`.

```
#include <unistd.h>

ssize_t pwrite (int fd, const void *buf, size_t count, off_t pos);
```

This call writes up to `count` bytes from `buf` to the file descriptor `fd` at file position `pos`.

These calls are almost identical in behavior to their non-`p` brethren, except that they completely ignore the current file position; instead of using the current position, they use the value provided by `pos`. Also, when done, they do not update the file position. In other words, any intermixed `read( )` and `write( )` calls could potentially corrupt the work done by the positional calls.

Both positional calls can be used only on seekable file descriptors. They provide semantics similar to preceding a `read( )` or `write( )` call with a call to `lseek( )`, with three differences. First, these calls are easier to use, especially when doing a tricky operation such as moving through a file backward or randomly. Second, they do not update the file pointer upon completion. Finally, and most importantly, they avoid any potential races that might occur when using `lseek( )`. As threads share file descriptors, it would be possible for a different thread in the same program to update the file position after the first thread's call to `lseek( )`, but before its read or write operation executed. Such race conditions can be avoided by using the `pread( )` and `pwrite( )` system calls.

File operation

```
ssize_t readv (int fd,
               const struct iovec *iov,
               int count);
```

The `writev( )` function writes at most `count` segments from the buffers described by `iov` into the file descriptor `fd`:

```
#include <sys/uio.h>

ssize_t writev (int fd,
                const struct iovec *iov,
                int count);
```

The `readv( )` and `writev( )` functions behave the same as `read( )` and `write( )`, respectively, except that multiple buffers are read from or written to.

Each `iovec` structure describes an independent disjoint buffer, which is called a *segment*:

File operation

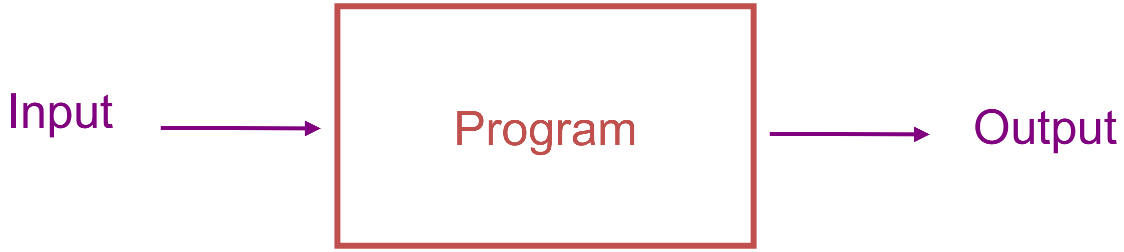
- The main advantages offered by readv and writev are:
  - It allows working with non contiguous blocks of data i.e. buffers need ot be part of an array, but separately allocated.
  - The I/O is 'atomic' i.e if we do writev, all the elements in the vector will be written in one contiguous operation, and writes done by other processes will not occur in between them.

Timing results comparing `writev` and other techniques

	Linux (Intel x86)			Mac OS X (PowerPC)		
Operation	User	System	Clock	User	System	Clock
two `write`s	1.29	3.15	7.39	1.60	17.40	19.84
buffer copy, then one `write`	1.03	1.98	6.47	1.10	11.09	12.54
one `writev`	0.70	2.72	6.41	0.86	13.58	14.72

Profiling – Gprof

- Timing, Instrumenting, Profiling
- How slow is the code?
  - How long does it take for certain types of inputs?
- Where is the code slow?
  - Which code is being executed most?
- Why is the code running out of memory?
  - Where is the memory going?
  - Are there leaks?
- Why is the code slow?
  - How imbalanced is my hash table or binary tree?



Profiling – Gprof

- Gather statistics about your program's execution
  - e.g., how much time did execution of a function take?
  - e.g., how many times was a particular function called?
  - e.g., how many times was a particular line of code executed?
  - e.g., which lines of code used the most time?
- Most compilers come with profilers
  - e.g., **gprof**
- Gprof (GNU Performance Profiler)
  - **gcc –pg –o mymath mymath.c**
- Running the code (e.g., testmath)
  - Produces output file gmon.out containing statistics
- Printing a human-readable report from gmon.out
  - gprof testmath > gprofreport

Profiling – Gcov

Gcov is a source code coverage analysis and statement-by-statement profiling tool. Gcov generates exact counts of the number of times each statement in a program is executed and annotates source code to add instrumentation. **Gcov** comes as a standard utility with GNU CC Suite (GCC)

Gcov provides the following details:

- How often each line of code executes
- What lines of code are actually executed
- How much computing time each section of code uses

Profiling – Gcov

Compilation command for the test code :

 # gcc --coverage lib.c test.c -o test

This will generate the following files:

lib.gcno – library flow graph

test.gcno – test code flow graph

test – test code executable

Now, execute the test code object file. This will generate the following files

lib.gcda – library profile output

test.gcda – test code profile output

Profiling – Gcov

Now we have all the inputs required for gcov to generate the coverage report. To generate the coverage report, run the following command

 # gcov -abcfu lib.c

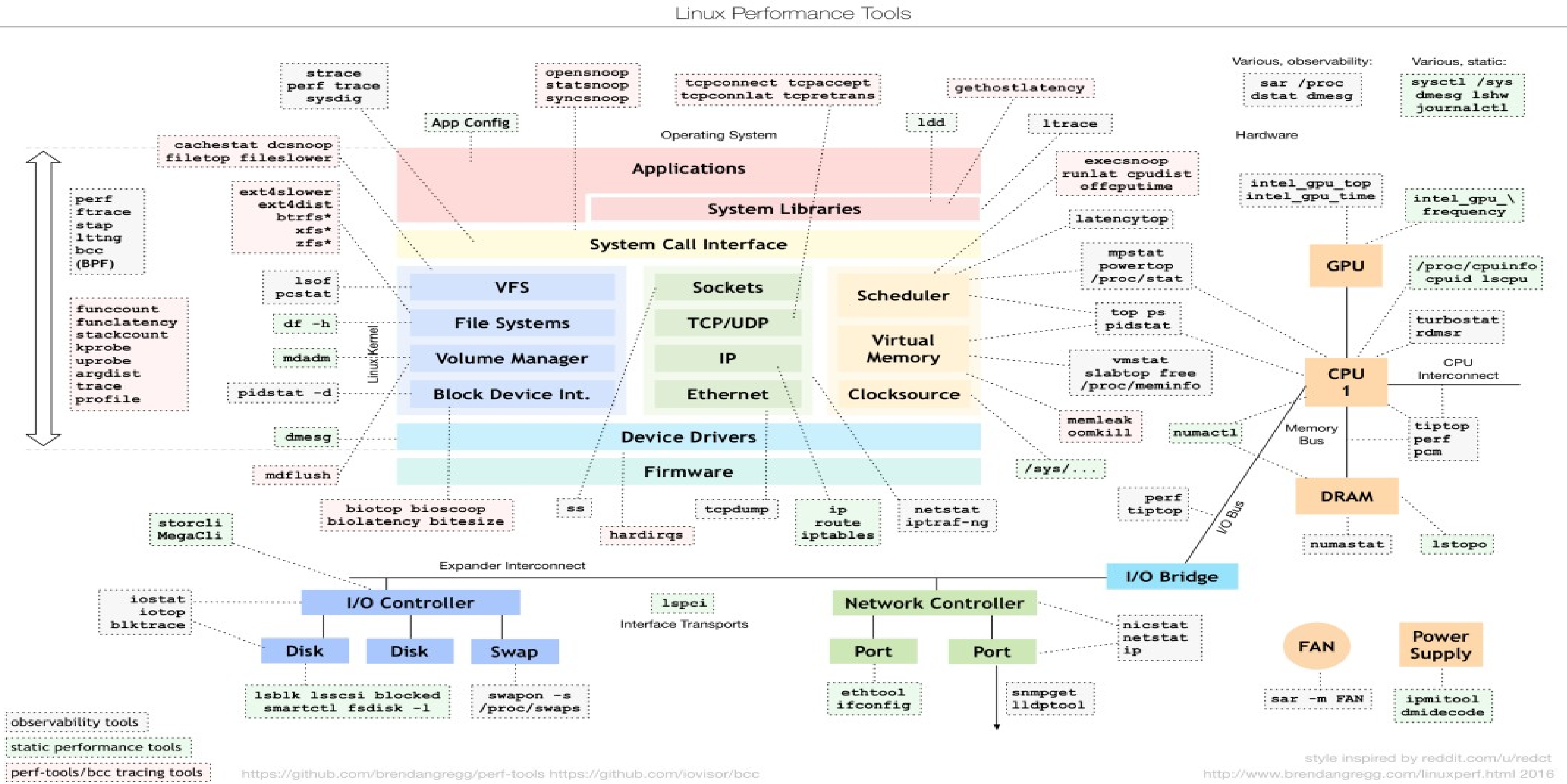
Detailed coverage report will be available in the lib.c.gcov file generated by gcov

LCOV

lcov --directory . --capture --output-file app.info

genhtml app.info

Debugging



Linux Performance Tools
strace
perf trace
sysdig
opensnoop
statsnoop
syncsnoop
tcpconnect tcpaccept
tcpconnlat tcpretrans
gethostlatency
Various, observability:
sar /proc
dstat dmesg
Various, static:
sysctl /sys
dmesg lshw
journalctl
App Config
ldd
ltrace
Operating System
Hardware
cachestat dcsnoop
filetop fileslower
Applications
execsnoop
runlat cpudist
offcputime
intel_gpu_top
intel_gpu_time
intel_gpu_\
frequency
System Libraries
perf
ftrace
stap
lttng
bcc
(BPF)
ext4slower
ext4dist
btrfs*
xfs*
zfs*
System Call Interface
latencytop
mpstat
powertop
/proc/stat
GPU
lsof
pcstat
VFS
Sockets
Scheduler
/proc/cpuinfo
cpuid lscpu
top ps
pidstat
funccount
funclatency
stackcount
kprobe
uprobe
argdist
trace
profile
df -h
File Systems
TCP/UDP
Virtual
Memory
turbostat
rdmsr
mdadm
Linux Kernel
Volume Manager
IP
vmstat
slabtop free
/proc/meminfo
CPU
1
CPU
Interconnect
pidstat -d
Block Device Int.
Ethernet
Clocksource
memleak
oomkill
numactl
Memory
Bus
tiptop
perf
pcm
dmesg
Device Drivers
mdflush
Firmware
/sys/...
perf
tiptop
DRAM
I/O Bus
biotop biosnoop
biolatency bitesize
ss
hardirqs
tcpdump
ip
route
iptables
netstat
iptraf-ng
storcli
MegaCli
numastat
lstopo
Expander Interconnect
I/O Bridge
iostat
iotop
blktrace
I/O Controller
lspci
Interface Transports
Network Controller
nicstat
netstat
ip
Disk
Disk
Swap
Port
Port
FAN
Power
Supply
lsblk lsscsi blocked
smartctl fsdisk -l
swapon -s
/proc/swaps
ethtool
ifconfig
snmpget
lldptool
sar -m FAN
ipmitool
dmidecode
observability tools
static performance tools
perf-tools/bcc tracing tools
https://github.com/brendangregg/perf-tools https://github.com/iovisor/bcc
style inspired by reddit.com/u/redct
http://www.brendangregg.com/linuxperf.html 2016

Debugging - Strace

strace is a powerful command line tool for debugging and trouble shooting programs in Unix-like operating systems such as Linux. It captures and records all system calls made by a process and the signals received by the process.

Trace Linux Command System Calls

You can simply run a command with strace like this, here we are tracing of all system calls made by the free command.

 # strace free

Debugging - Strace

Trace Linux Process PID

If a process is already running, you can trace it by simply passing its PID as follows; this will fill your screen with continues output that shows system calls being made by the process, to end it, press [Ctrl + C]

 # sudo strace -p 3569

Get Summary of Linux Process

Using the -c flag, you can generate a report of total time, calls, and errors for each system call, as follows.

 # strace –c free

Debugging - Strace

Print Instruction Pointer During System Call

The -i option displays the instruction pointer at the time of each system call made by the program.

 # sudo strace -i free

Show Time of Day For Each Trace Output Line

You can also print the time of day for each line in the trace output, by passing the -t flag.

 # sudo strace –t free

Debugging - Strace

Print Command Time Spent in System Calls

To shows the time difference between the starting and the end of each system call made by a program, use the -T option.

 # sudo strace –T free

Trace Only Specific System Calls

In the command below, trace=write is known as a qualifying expression, where trace is a qualifier (others include signal, abbrev, verbose, raw, read, or write). Here, write is the value of the qualifier.

Debugging - Strace

The following command actually shows the system calls to print df output on standard output.

 # sudo strace –e trace=write free

 # sudo strace -e trace=open,close free

 # sudo strace -e trace=open,close,read,write free

 # sudo strace -e trace=all free

Debugging - Strace

Trace System Calls Based on a Certain Condition

Let's look at how to trace system calls relating to a given class of events. This command can be used to trace all system calls involving process management.

 # sudo strace -e trace=process free

Debugging - Strace

Trace System Calls Based on a Certain Condition

Let's look at how to trace system calls relating to a given class of events. This command can be used to trace all system calls involving process management.

 # sudo strace -e trace=process free

Next, to trace all system calls that take a filename as an argument, run this command.

$ sudo strace -q -e trace=file free

Debugging - Strace

To trace all system calls involving memory mapping, type.

$ sudo strace -q -e trace=memory free

You can trace all network and signals related system calls.

$ sudo strace -e trace=network free

$ sudo strace -e trace=signal free

Debugging - Strace

Redirect Trace Output to File

To write the trace messages sent to standard error to a file, use the -o option. This means that only the command output is printed on the screen as shown below.

$ sudo strace -o df_debug.txt free

strace a program and threads

To trace both program and it threads using option -f

 # strace -f ./program

Debugging - Strace

strace a program and print strings

This will print the first 80 characters of every string.

```
# strace -s 80 -f ./program
```

Debugging - GDB

Debuggers are programs which allow you to execute your program in a controlled manner, so you can look inside your program to find a bug.

gdb is a reasonably sophisticated text based debugger. It can let you:

Start your program, specifying anything that might affect its behavior.

Make your program stop on specified conditions.

Examine what has happened, when your program has stopped.

Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

SYNOPSIS

gdb [prog] [core|procID]

Debugging - GDB

GDB is invoked with the shell command gdb.

Once started,it reads commands from the terminal until you tell it to exit with the GDB command quit.

The most usual way to start GDB is with one argument or two, specifying an executable program as the argument:

 # gdb program

You can also start with both an executable program and a core file specified:

 # gdb program core

You can, instead, specify a process ID as a second argument, if you want to debug a running process:

 # gdb program 1234

would attach GDB to process 1234

(gdb) layout next (to see source also)

Debugging - GDB

To use gdb best, compile your program with:

gcc –g –c my_math.c

gcc –g –c sample.c

gcc –o sample my_math.o sample.o

or:

gcc –o sample -g my_math.c sample.c

That is, you should make sure that –g option is used to generate the .o files.

This option tells the compiler to insert more information about data types, etc., so the debugger gets a better understanding of it.

Debugging - GDB

Here are some of the most frequently needed GDB commands:

b(reak) [file:]function Set a breakpoint at function (in file).

r(un) [arglist] Start program (with arglist, if specified).

bt or where Backtrace: display the program stack; especially useful to find where your program crashed or dumped core.

print expr Display the value of an expression.

c Continue running your program (after stopping, e.g. at a breakpoint).

n(ext) Execute next program line (after stopping); step over any function calls in the line.

Debugging - GDB

s(tep) Execute next program line (after stopping); step into any function calls in the line.

help [name] Show information about GDB command name, or general information about using GDB.

q(uit) Exit from GDB.

l(ist) print the source code

Debugging – GDB commandline

 # gdb --args executablename arg1 arg2 arg3

Or

$ gdb executablename

(gdb) r arg1 arg2 arg3

Debugging – GDB coredump

You can also start with both an executable program and a core file specified:

 # gdb program core

Debugging – GDB Variable

(gdb) print variable

(gdb) print func::variable

print variable in different format

x

Regard the bits of the value as an integer, and print the integer in hexadecimal.

d

Print as integer in signed decimal.

u

Print as integer in unsigned decimal.

Debugging – GDB Variable

o

Print as integer in octal.

t

Print as integer in binary. The letter `t' stands for "two". (1)

a

Print as an address, both absolute in hexadecimal and as an offset from the nearest preceding symbol. You can use this format used to discover where (in what function) an unknown address is located:

(gdb) p/a 0x54320

$3 = 0x54320 <_initialize_vx+396>

Debugging – GDB Variable

c

Regard as an integer and print it as a character constant.

f

Regard the bits of the value as a floating point number and print using typical floating point syntax.

Debugging – GDB Variable

(gdb) set var variable=90

(gdb) print &a

Type **info variables** to list "All global and static variable names".

Type **info locals** to list "Local variables of current stack frame" (names and values), including static variables in that function.

Type **info args** to list "Arguments of the current stack frame" (names and values).

Debugging – GDB Variable

(gdb) condition <breakpoint> <condition>

Example:

(gdb) break 28

(gdb) condition 2 i>5

- **watch**: gdb will break when a *write* occurs
- **rwatch**: gdb will break wnen a *read* occurs
- **awatch**: gdb will break in *both cases*

Thank you