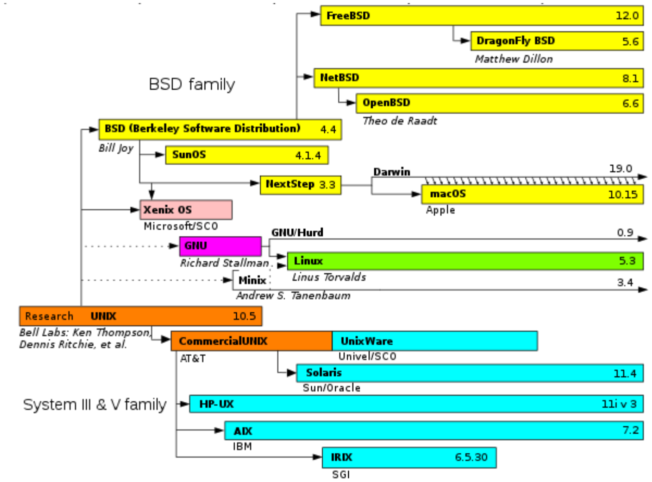
Introduction to System Programming

Concepts of Unix Programming



M1522.000800 System Programming, Fall 2023

Module Outline

- A Brief History of Unix
- Unix Philosophy
- Architecture of *nix Systems
- Module Summary

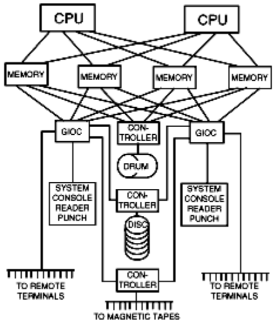
UNIX®

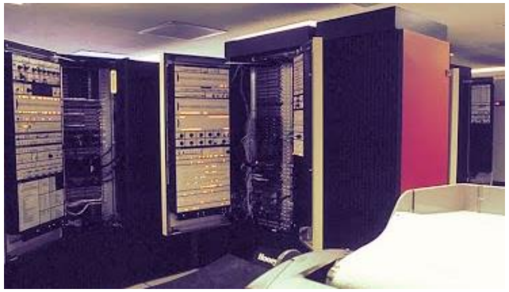
Brief History of Unix

The Inspiration: MULTICS

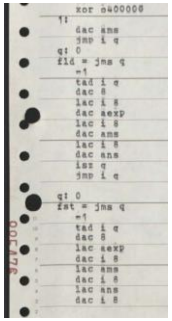
1969, 60 years old

- late 1960: MULTICS (Multiplexed Information and Computing System)
  - ambitious project by Bell Labs, General Electric, and MIT to develop a multi-user, multi-tasking OS for mainframe computers
  - features
    - high-level language implementation
    - multi-processor
    - virtual memory
    - hierarchical filesystem with ACLs, quotas, links, …
    - dynamic linking
    - time-shared scheduler with scheduling classes
    - multi user, security
  - the project "failed", although MULTICS was in use until the year 2000
  - historical reading: https://multicians.org





The Beginning: UNIX

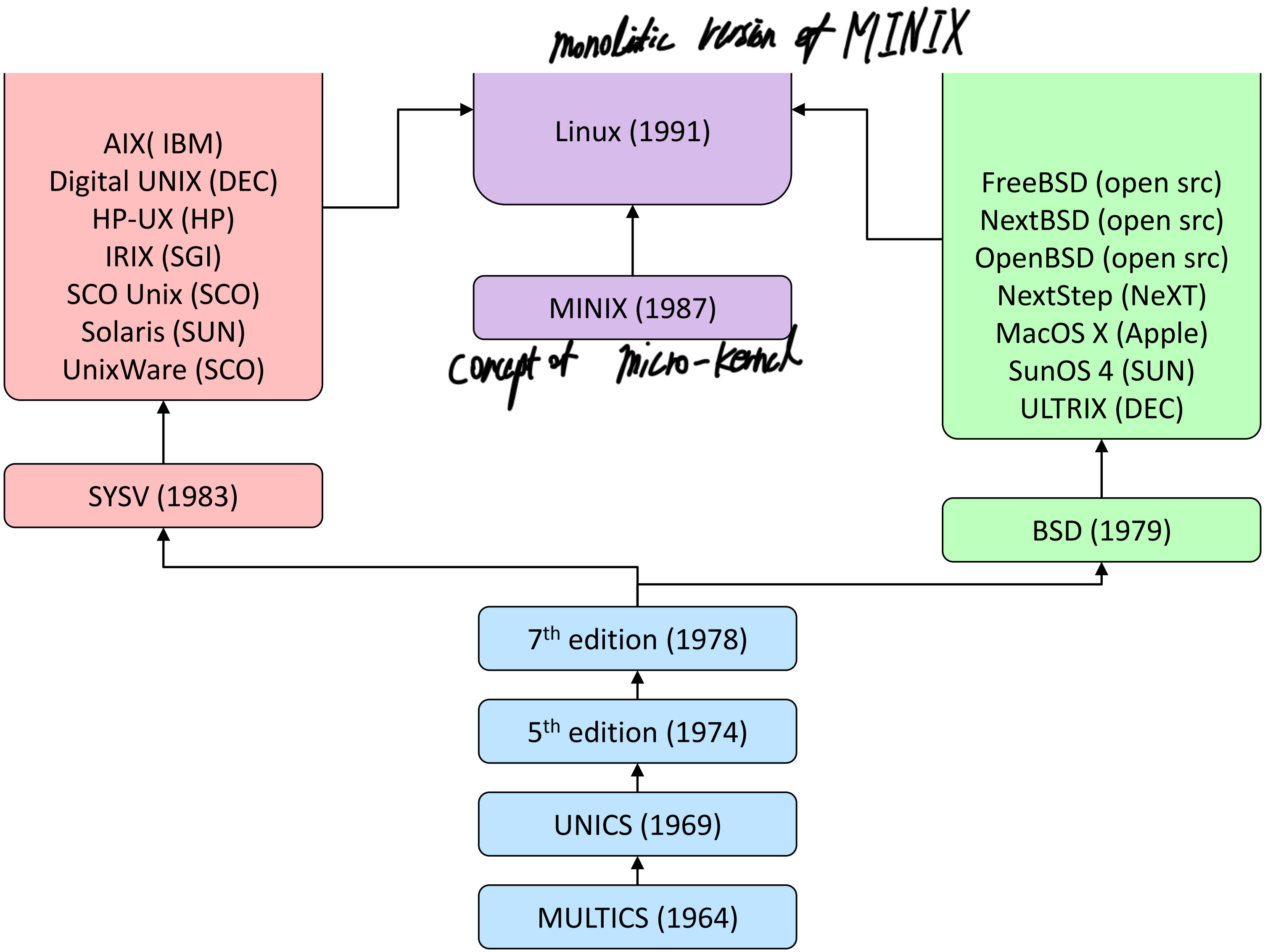


```
        xor o400000
    1:
        dac ams
        jmp i q
    q: 0
    fld = jms q
        -1
        tad i q
        dac 8
        lac i 8
        dac aexp
        lac i 8
        dac ams
        lac i 8
        dac ans
        isz q
        jmp i q

    q: 0
    fst = jms q
        -1
        tad i q
        dac 8
        lac aexp
        dac i 8
        lac ams
        dac i 8
        lac ans
        dac i 8
```

- 1969: Bell Labs drops out of MULTICS project
- Ken Thompson from Bell Labs wrote a simpler version in assembly for a PDP7, called it Unics (Uniplexed Information and Computing System)

- 1973: Thompson teams up with Dennis Ritchie who had extended Thompson's B language into C (with Brian Kernigan) to rewrite UNIX in C
- 1974: 5th edition of UNIX with a C kernel released to universities
- 1978: 7th first portable edition, UNIX splits into two branches:
  - SYSV (System 5) developed at AT&T
  - BSD (Berkeley Software Distribution) developed at UC Berkeley
- further reading: http://www.unix.org/

UNIX Family Tree



monolitic version of MINIX

Linux (1991)

concept of micro-kernel

MINIX (1987)

AIX(IBM)
Digital UNIX (DEC)
HP-UX (HP)
IRIX (SGI)
SCO Unix (SCO)
Solaris (SUN)
UnixWare (SCO)

SYSV (1983)

FreeBSD (open src)
NextBSD (open src)
OpenBSD (open src)
NextStep (NeXT)
MacOS X (Apple)
SunOS 4 (SUN)
ULTRIX (DEC)

BSD (1979)

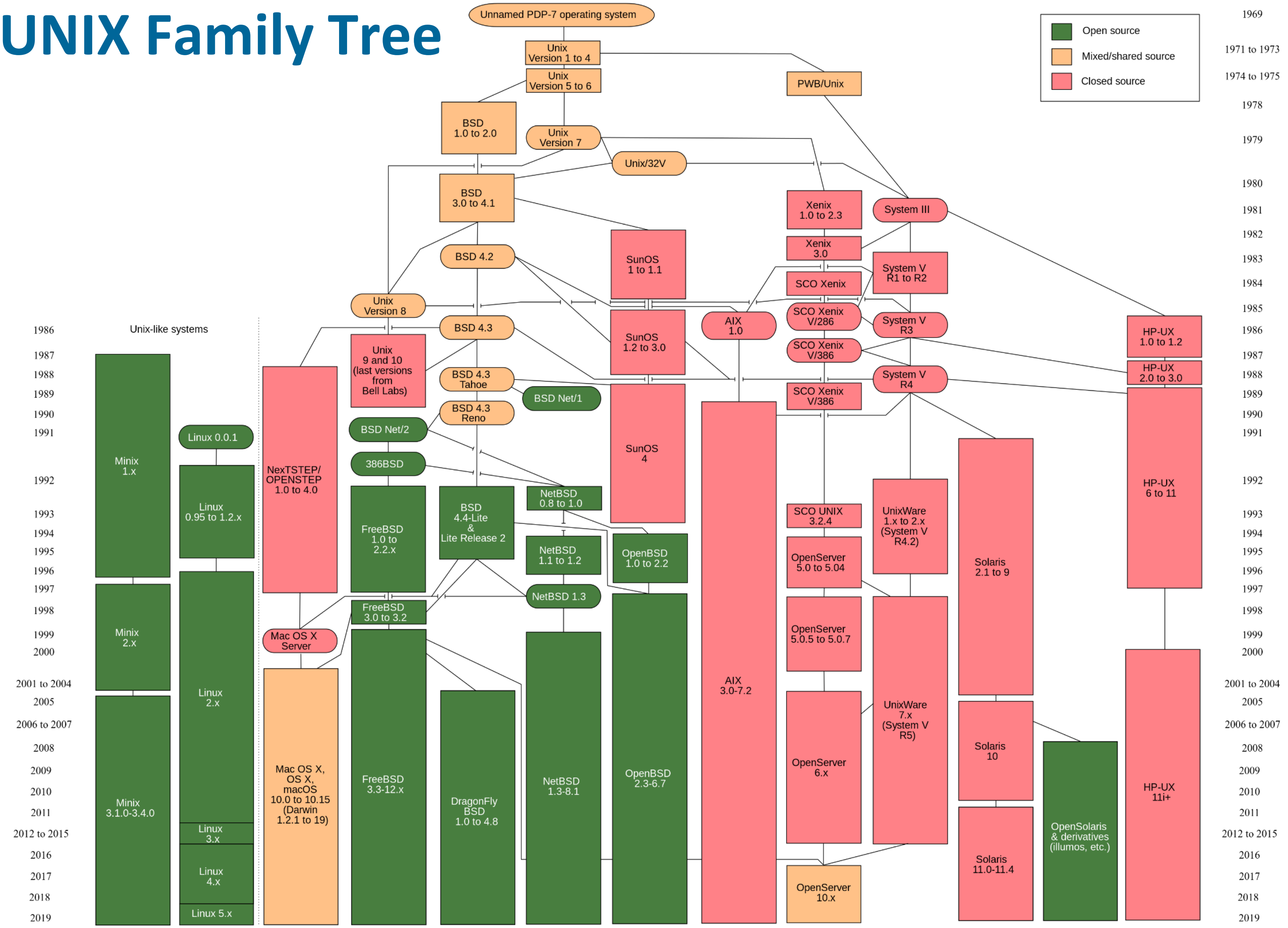
7th edition (1978)

5th edition (1974)

UNICS (1969)

MULTICS (1964)

UNIX Family Tree

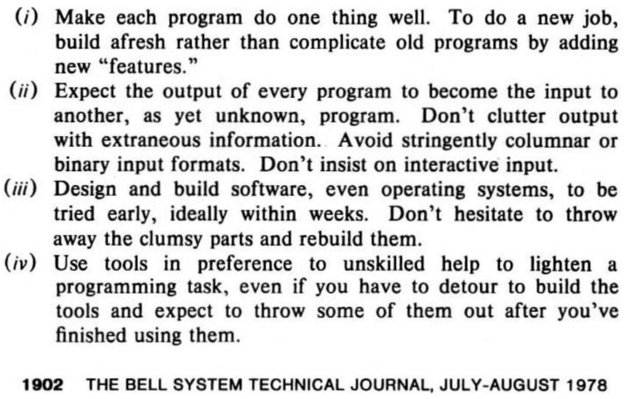


source: Wikipedia

The IEEE POSIX Standard

- Portable Operating System Interface (POSIX) (set of API functions)
  - IEEE Computer Society standard to maintain compatibility between OSes
  - defines the kernel API (application programming interface), command line shells, and utility programs (Interface)
  - based on Unix
  - POSIX-certified
    - AIX, EulerOS, HP-UX, IRIX, macOS >= 10.5, Solaris, UnixWare, …
  - POSIX-compliant
    - *BSD, MINIX, Linux, Darwin, VMware ESXi, VxWorks, Android, …
  - MS Windows
    - POSIX subsystem (1990-2000),
    - more recently: Windows Subsystem for Linux (WSL)
    - compatibility layers: Cygwin, MinGW, …

Unix Philosophy



(*i*) Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features."

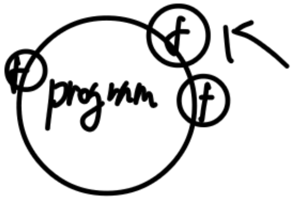
(*ii*) Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.

(*iii*) Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.

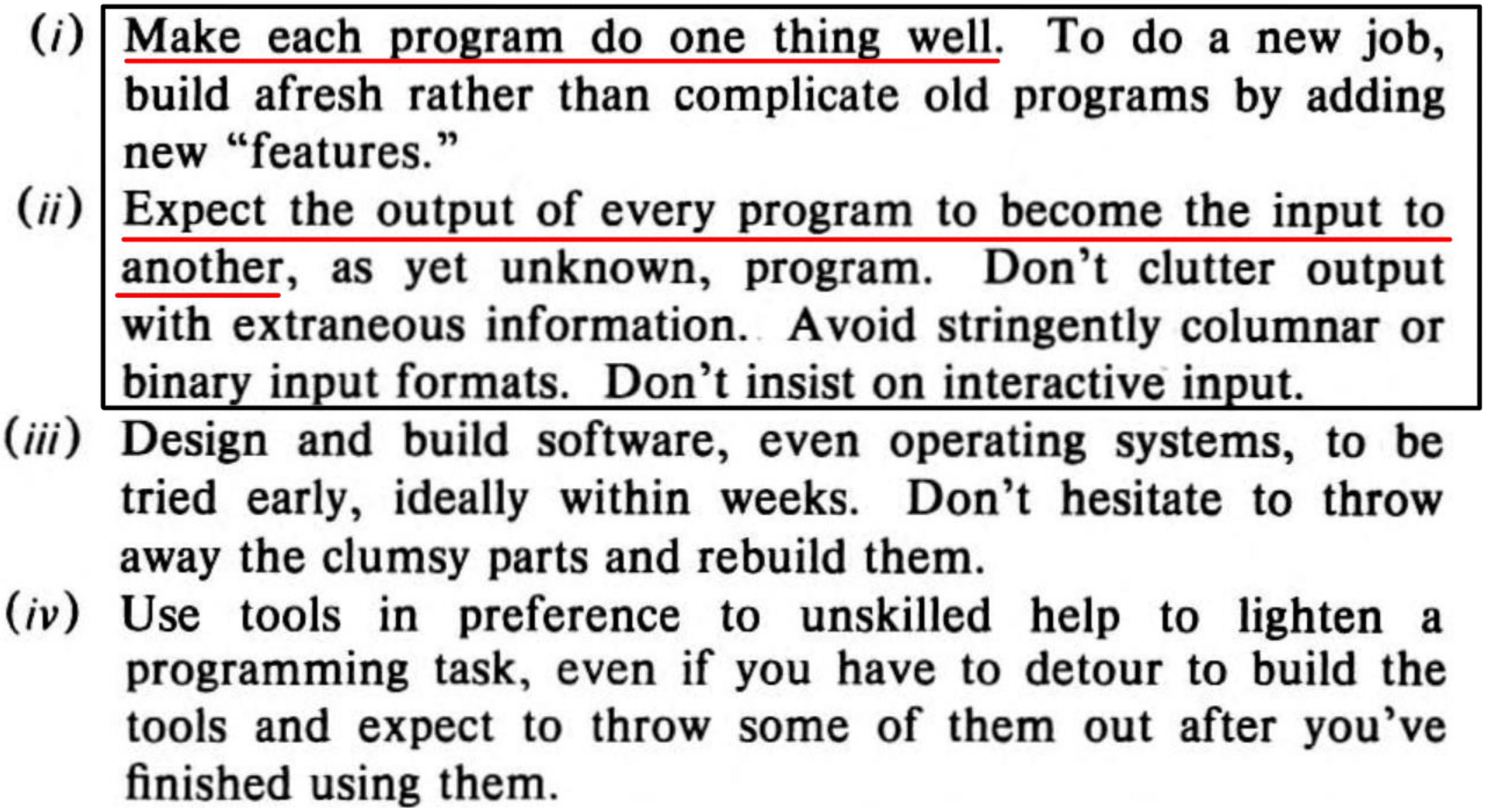
(*iv*) Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

1902 THE BELL SYSTEM TECHNICAL JOURNAL, JULY-AUGUST 1978

Unix Philosophy



make program ugly (separate from origin and build new one)



(*i*) Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features."

(*ii*) Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.

(*iii*) Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.

(*iv*) Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

Bell System Technical Journal, 57: 6. July-August 1978 pp 1899-1904.
UNIX Time-Sharing System: Forward. (McIlroy, M.D.; Pinson, E.N.; Tague, B.A.)
https://archive.org/details/bstj57-6-1899/mode/2up

Unix Philosophy

“Much of the power of the UNIX operating system comes from a **style of program design** that makes programs **easy to use** and, more important, **easy to combine with other programs**. This style has been called the use of software tools, [...] This style was based on the use of tools: **using programs separately or in combination to get a job done, rather than doing it by hand**, by monolithic self-sufficient subsystems, or by special-purpose, one-time programs.”

Program Design in the UNIX Environment, Pike and Kernighan, 1984

“Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.”

Doug McIlroy, in A Quarter Century of Unix, Peter H. Salus, Addison-Wesley, 1994

Unix Philosophy

- Unfortunately, (typically) ex-Windows programmers do not seem to remember or respect the philosophy anymore

init system program — do same thing / replacable (greatly modularized)

- systemd as the prime example
  - goal: speedup & modernize the boot up process of Linux
  - method: replaced a set of scripts and small binaries with a huge system, even though internally composed of several binaries, with large dependencies
  - author actively advocates ignoring POSIX compatibility

    modularized

    - huge codebase that takes over more and more independent services of Unix
    - only works on the Linux kernel
    - binary logs
    - “Not having to care about portability has two big advantages: …” [source]
    - “Yes, the Open Source community is full of assholes” [source]

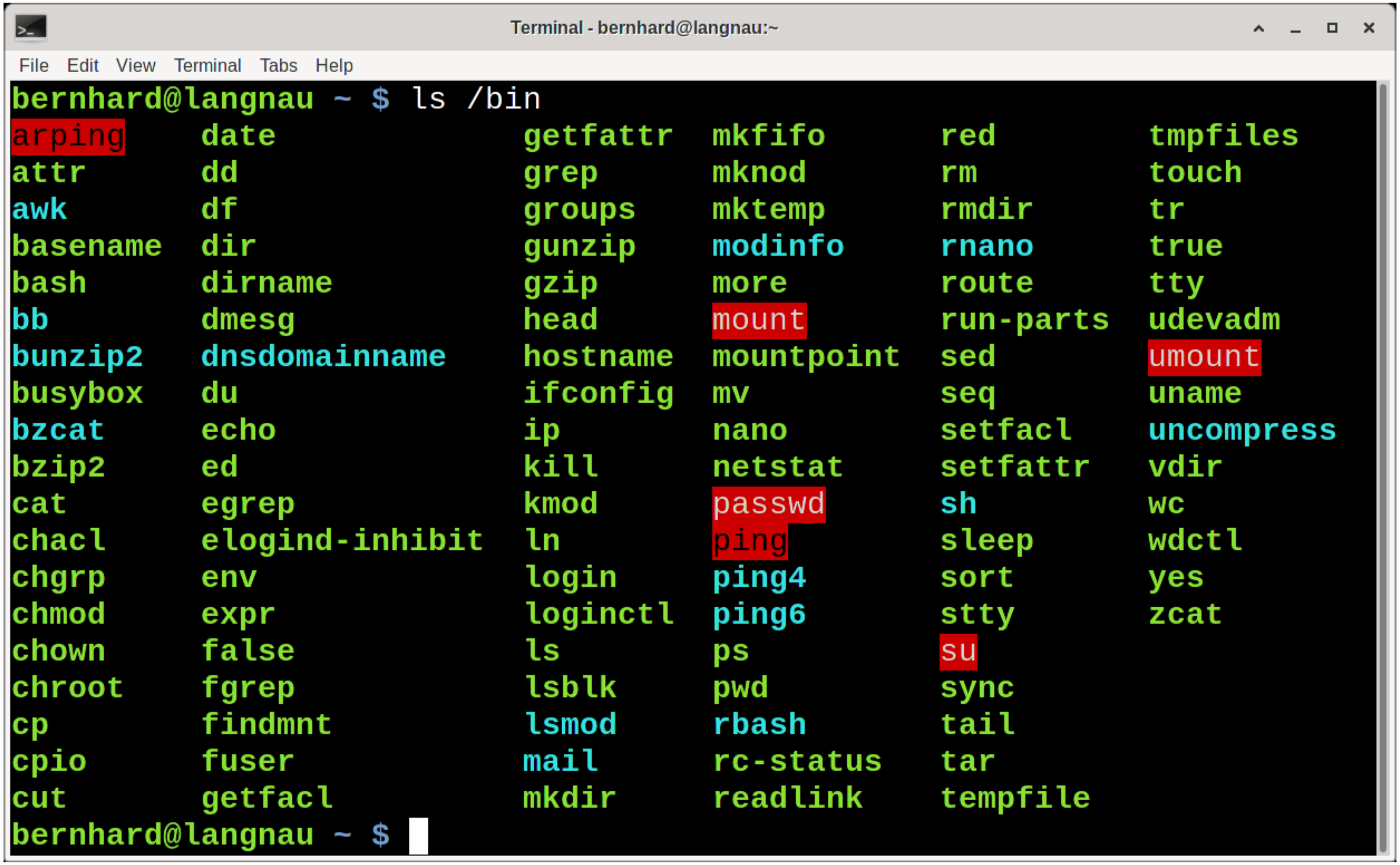
- Gnome (and GTK, unfortunately) is another victim of the devs know it best disease

Unix Philosophy

- A note on backwards compatibility

Mac os:
Can you install this 5 year old program?
Nooooo, I can't! this program is too old!
Windows:
Can you install this 25 year old program?
Yes, I can! Installing... done!
Linux
can you install this 25 year old program
It's already installed

Unix Philosophy


Terminal - bernhard@langnau:~
File Edit View Terminal Tabs Help

```
bernhard@langnau ~ $ ls /bin
arping     date              getfattr  mkfifo      red        tmpfiles
attr       dd                grep      mknod       rm         touch
awk        df                groups    mktemp      rmdir      tr
basename   dir               gunzip    modinfo     rnano      true
bash       dirname           gzip      more        route      tty
bb         dmesg             head      mount       run-parts  udevadm
bunzip2    dnsdomainname     hostname  mountpoint  sed        umount
busybox    du                ifconfig  mv          seq        uname
bzcat      echo              ip        nano        setfacl    uncompress
bzip2      ed                kill      netstat     setfattr   vdir
cat        egrep             kmod      passwd      sh         wc
chacl      elogind-inhibit   ln        ping        sleep      wdctl
chgrp      env               login     ping4       sort       yes
chmod      expr              loginctl  ping6       stty       zcat
chown      false             ls        ps          su
chroot     fgrep             lsblk     pwd         sync
cp         findmnt           lsmod     rbash       tail
cpio       fuser             mail      rc-status   tar
cut        getfacl           mkdir     readlink    tempfile
bernhard@langnau ~ $
```

Unix Philosophy

- Input: a list of strings
- Output: sorted list of strings with no duplicates

- StackOverflow programmer

```
import sys

def dedup(lin):
    lout = []

    if lin:
        for e in lin:
            if e not in lout:
                lout.append(e)

        return lout

    else:
        return lin

if __name__ == "__main__":
    file = open(sys.argv[1])
    strings = dedup(map(lambda it: it.strip(), file.readlines()))
    strings.sort()
    print(*strings, sep='\n')

$ python dedup.py strings.txt
```

Unix Philosophy

- Input: a list of strings
- Output: sorted list of strings with no duplicates

- Me (and you after this class):

```
$ cat strings.txt | sort | uniq
```

Unix Philosophy

- Task: calculate and print the size of all files in a directory (including files in subdirectories)

- Python programmer

```
import os
import sys

def processDir(dn):
    size = 0
    try:
        # get list of all files in directory
        fl = os.listdir(dn)

        # process one by one. Recurse into directories,
        # add size for files
        for e in fl:
            fn = os.path.join(dn, e)
            if os.path.isdir(fn):
                size += processDir(fn)
            else:
                size += os.path.getsize(fn)

    except:
        print("Error enumerating directory '{}'.".format(dn))
        size = 0

    return size

…
```

```
…

if __name__ == "__main__":
    dn = "."    # default directory

    # use path provided on command line
    if (len(sys.argv) > 1):
        dn = sys.argv[1]

        if not os.path.isdir(dn):
            print("'{}' is not a directory.".format(dn))
            sys.exit()


    # recursively enumerate directory
    size = processDir(dn)

    # print size
    print("Total size of '{}': {} bytes".format(dn, size))
```

Unix Philosophy

- Task: calculate and print the size of all files in a directory (including files in subdirectories)

- C programmer

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <dirent.h>

uint64 processDir(const char *dn)
{
  unsigned long long size = 0;
  struct dirent *e;
  struct stat s;
  char *fn;

  DIR *d = opendir(dn);
  if (d == NULL) return 0;

  while ((e = readdir(d)) != NULL) {
    if (strcmp(e->d_name, ".") && strcmp(e->d_name, "..")) {
      if (asprintf(&fn, "%s/%s", dn, e->d_name) == -1) continue;
      if (stat(fn, &s) == 0) {
        if (S_ISREG(s.st_mode)) size += s.st_size;
        if (S_ISDIR(s.st_mode)) size += processDir(fn);
      }
      free(fn);
    }
  }

  closedir(d);
  return size;
}
…
```

```c
…
int main(int argc, char *argv[])
{
  const char CURDIR[] = ".";
  const char *dn = CURDIR;

  // use path provided on command line
  if (argc > 1) {
    dn = argv[1];
    DIR *d;
    if ((d = opendir(dn)) == NULL) {
      printf("'%s' is not a directory.\n", dn);
      return EXIT_FAILURE;
    } else {
      closedir (d);
    }
  }

  // recursively enumerate directory
  uint64 size = processDir(dn);

  // print size
  printf("Total size of '%s': %llu bytes\n", dn, size);

  return EXIT_SUCCESS;
}
```

Unix Philosophy

- Task: calculate and print the size of all files in a directory (including files in subdirectories)

- The Unix way:

find . -printf "%s\n" | paste -s -d+ | bc

calc each size

link with calc +

  - Bash script:

```
#!/bin/bash

# recursively traverse current directory, enumerate all files,
# and sum up the total of the file sizes

DIR="."

if [ -n "$1" ]; then DIR=$1; fi

echo "Total size of '$DIR': `find $DIR -type f -printf "%s\n" | paste -s -d+ | bc` bytes"
```

desig type

  - Bash one-liner:

```
$ DIR="."; printf "Total size of '$DIR': "; sum=0; while read num; do ((sum += num)); done \
    < <(find $DIR -type f -printf "%s\n"); echo "$sum bytes"
```

Unix Philosophy

- Task: calculate and print the size of all files in a directory (including files in subdirectories)

- The Unix way:
  - another bash script:

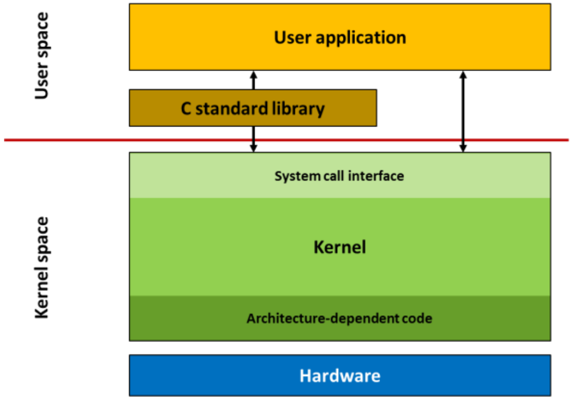
```
#!/bin/bash

# recursively traverse current directory, enumerate all files,
# and sum up the total of the file sizes

DIR="."

if [ -n "$1" ]; then DIR=$1; fi

ls -Rlap $DIR | grep -v "/$" | \
  awk '{ size+=$5 } END { printf "Total size of '\''%s'\'': %u bytes\n","'$DIR'",size }'
```

Unix Philosophy

- Task: calculate and print the size of all files in a directory (including files in subdirectories)

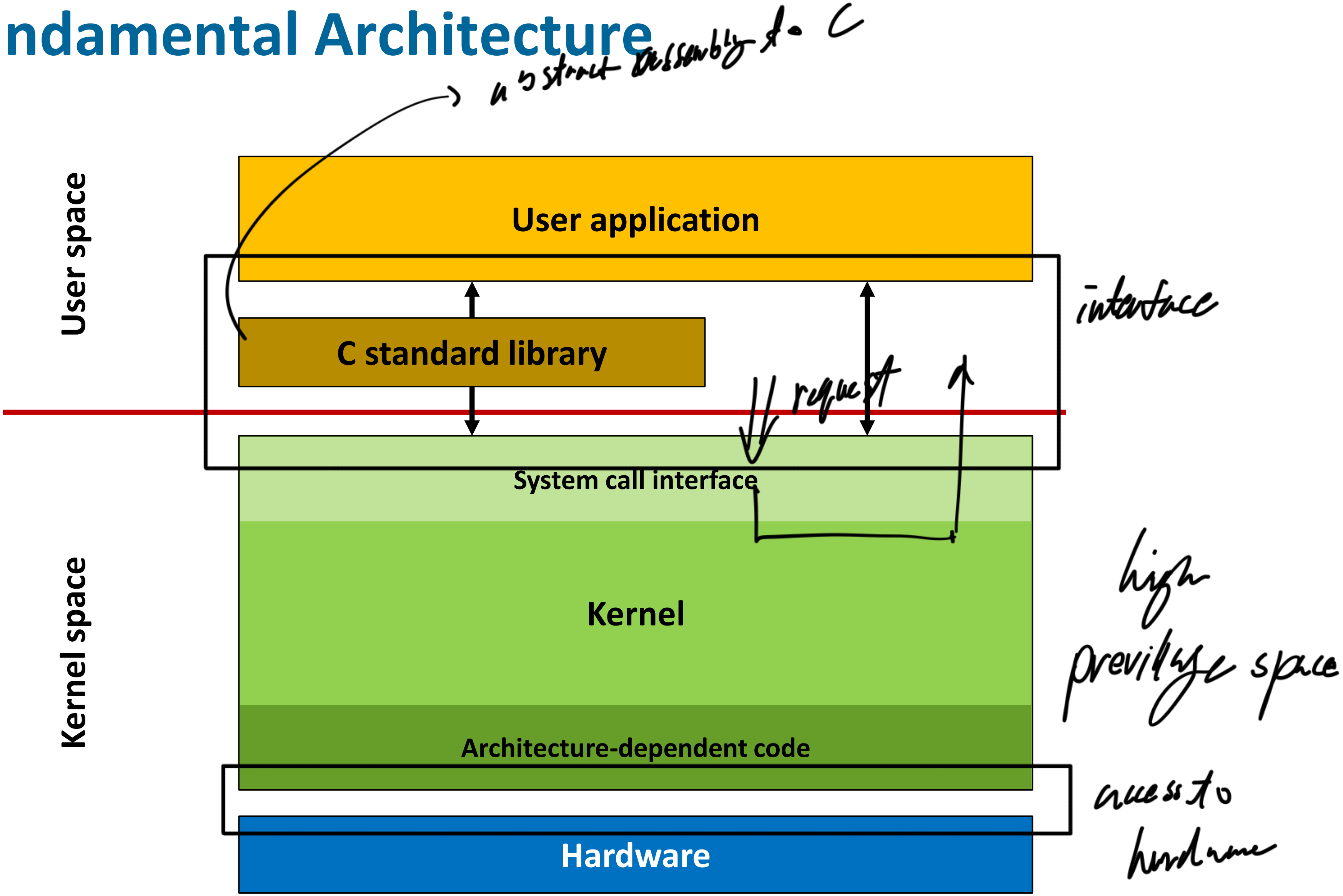
- For real hackers:
  - use xargs, combine several tools:
    - echo: print string
    - inline command execution (`…`)
    - find: find elements in a directory tree
    - xargs: run a command on several inputs – not many understand this properly
    - tail: only print last *n* lines of output
    - awk: pattern scanner & processor

```
$ DIR="."; echo "Total size of '$DIR': `find $DIR -type f | xargs -d '\n' wc -c | tail -n 1
| awk '{ print $1 }'` bytes"
```



Architecture of *nix Systems

Fundamental Architecture



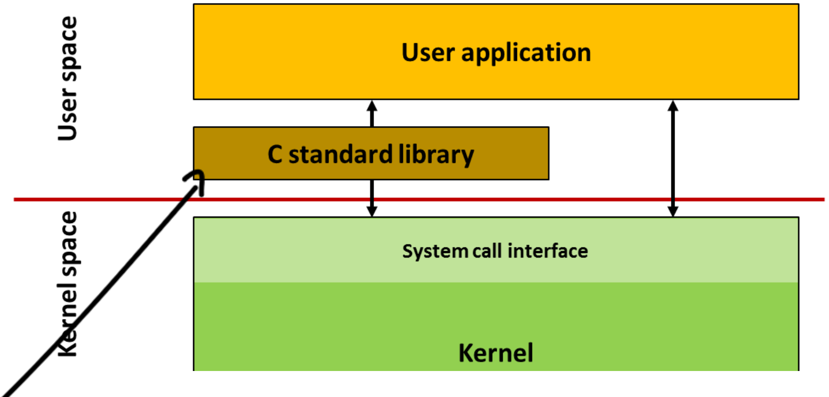
Fundamental Architecture

- User applications is located in **user space**, runs in **user mode**

- Kernel code located in **kernel space**, runs in **kernel mode**

- **User** vs **kernel** mode
  - supported by hardware (the CPU)
  - user mode: unprivileged instructions
    - unprivileged instructions cannot modify important system state
  - kernel mode: unprivileged and privileged instructions
    - privileged instructions can modify system state
      - turn on/off interrupts
      - modify memory translation tables
      - I/O instructions

Fundamental Architecture



- **System call interface**
  - well-defined access points to enter kernel
  - switch from user to kernel mode
  - somewhat difficult to use

abstract system call (assembly)

- **C standard library (libc)**
  - glibc on GNU systems (GNU C library)
  - core set of supported functions by C language (printf, malloc, …)
  - wrappers for system calls

Abstraction 1: Files

- **Example: hexdump**

```c
#define _GNU_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

#define DEFAULT_CHARS 128
#define CHARS_PER_LINE 16

int main(int argc, char *argv[])
{
  int fd, i, n;
  unsigned char c;

  //
  // check arguments
  //
  if (argc < 2) {
    fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
    fprintf(stderr, "(use '-' for stdin)\n");
    return EXIT_FAILURE;
  }

  //
  // open file
  //
  if (strcmp(argv[1], "-") == 0) fd = STDIN_FILENO;
  else if ((fd = open(argv[1], O_RDONLY)) == -1) {
    perror("Cannot open file.");
    return EXIT_FAILURE;
  }

  …
```

```c
  …
  //
  // number of characters to dump
  //
  n = DEFAULT_CHARS;
  if (argc >= 3) n = atoi(argv[2]);

  //
  // read & dump 'n' characters from file
  //
  i = 0;
  printf("%04d: ", i);
  while (i < n) {
    int r = read(fd, &c, sizeof(c));  //read from file
    if (r == sizeof(c)) {
      i++;
      if (c < 32) printf("<%02x>", c);  print hexformat.
      else printf("%c", c);
      if (i % CHARS_PER_LINE == 0) {
        printf("\n");
        if (i < n) printf("%04d: ", i);
      }
    } else {
      fprintf(stderr, "\nError reading file (pos: %d).\n", i);
      break;
    }
  }
  if (i % CHARS_PER_LINE > 0) printf("\n");

  //
  // cleanup & return
  //
  close(fd);

  return EXIT_SUCCESS;
}
```

hexdump.c

Abstraction 2: Virtual Memory

Example: Memory Management (CAREFUL!)

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

// max allocated memory is N * BUF_SIZE
#define N 64
#define BUF_SIZE (256 * 1024 * 1024)

int main(int argc, char *argv[])
{
  struct timespec sleeper = { .tv_sec = 0, .tv_nsec = 250000000L };
  unsigned int i, c = 0, n = 0;
  volatile int v;

  char **memory = (char**)malloc(N*sizeof(char*));

  while (n < N) {
    nanosleep(&sleeper, NULL);
    if ((memory[n] = malloc(BUF_SIZE)) == NULL) break;
    n++;
    printf("[%5d] Allocated %2.4f GB of memory\n", getpid(), (float)BUF_SIZE / (1024*1024*1024) * n);
  }

  while (1) {
    int m, ofs;

    m = random() % n;
    ofs = random() % BUF_SIZE;
    memory[m][ofs] = '!';

    for (i=0; i<5000; i++) v = i;

    if (c++ % (256*1024) == 0) printf("[%5d] Performed %2.3f M random writes\n",
                                     getpid(), (float)c / (1024*1024));
  }
  return EXIT_SUCCESS;
}
```

write to memory

allocator.c

Abstraction 3: Processes

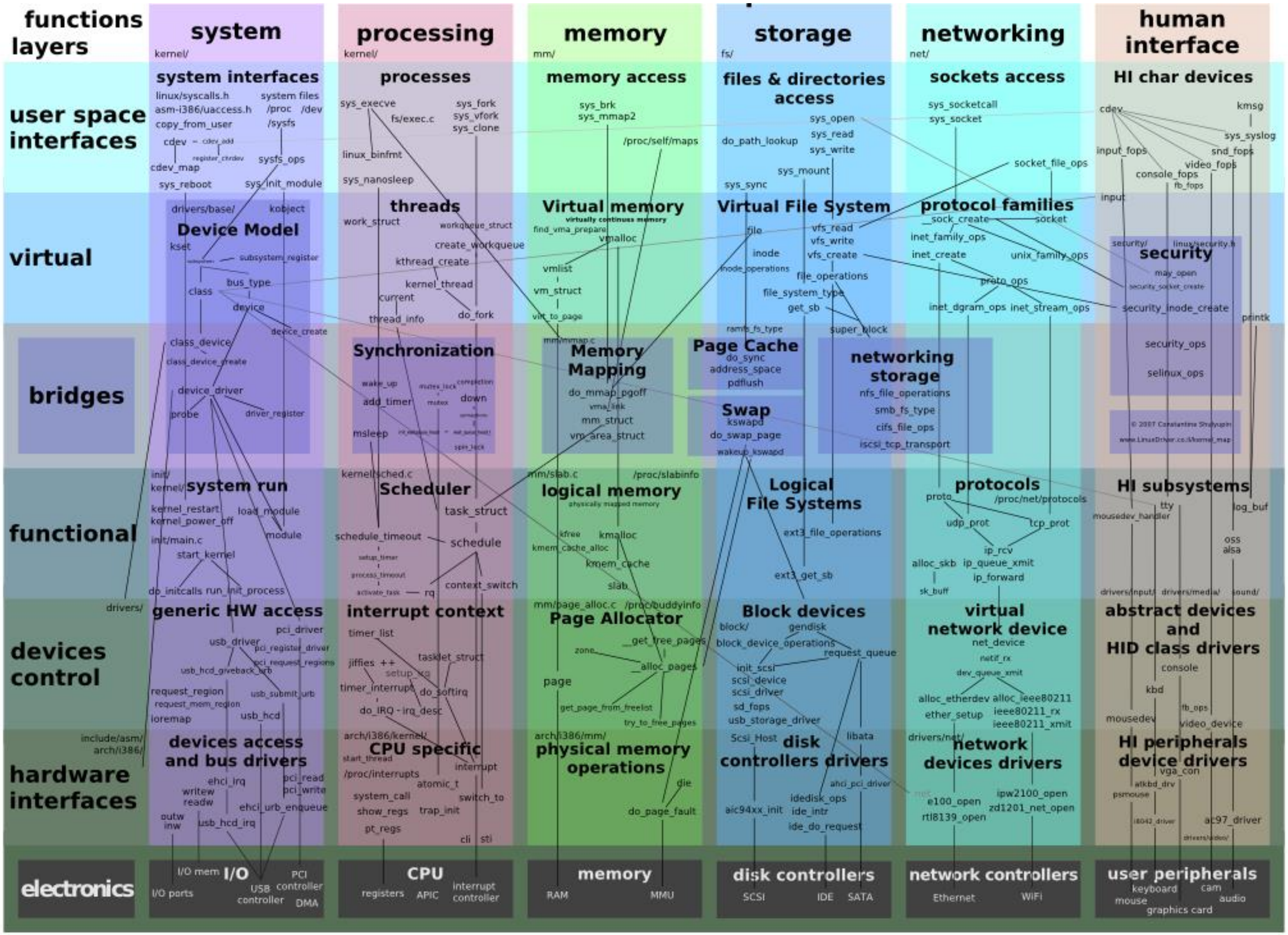
- **Example: Process Scheduling**

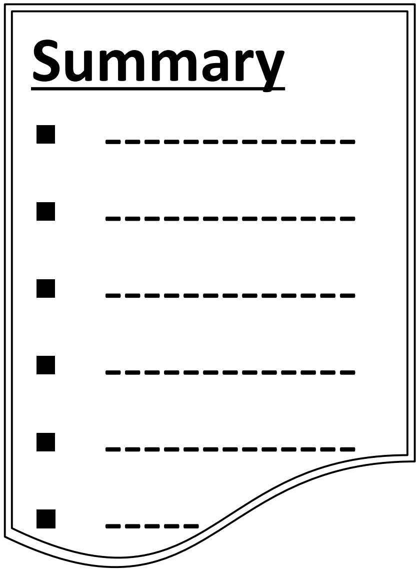
```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define N 100000000

int main(int argc, char *argv[])
{
  int cpuid = -1;
  cpu_set_t set;
  volatile int a;
  unsigned int i, c = 0;

  if (argc > 1) {
    cpuid = atoi(argv[1]);
    printf("[%5d] Pinning CPU %d\n", getpid(), cpuid);

    CPU_ZERO(&set);
    CPU_SET(cpuid, &set);

    if (sched_setaffinity(0, sizeof(cpu_set_t), &set) == -1) {
      perror("Error setting CPU affinity");
      return EXIT_FAILURE;
    }
  }

  while (1) {
    for (i=0; i<N; i++) a = 1;
    printf("[%5d pinned to: %2d running on: %2d] %8d\n", getpid(), cpuid, sched_getcpu(), c++);
  }

  return EXIT_SUCCESS;
}
```

runner.c

Linux Kernel Map

source: https://makelinux.github.io/kernel/map/





Module Summary

Summary

- Unix and derivates
  - 50 year old history
  - properly designed from the start
  - today, the vast majority of all operating systems have Unix roots
  - user programs interact with kernel space via system call interface
- IEEE POSIX
  - portable operating system interface standard (1988 ~)
  - guarantees interoperability between POSIX-compliant systems
    - (almost) all Unix-based systems and, these days, even Windows
- The Unix Philosophy
  - a system is a modular collection of programs, each doing one thing well
    - no bloated monster programs
  - program output to be used as input for other programs
    - no cluttered, binary output