# CS162
# Operating Systems and Systems Programming
# Lecture 3

## Processes (con't), Fork, Introduction to I/O

January 28th, 2015
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

## Recall: Four fundamental OS concepts

- Thread
  - Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack
- Address Space w/ Translation
  - Programs execute in an *address space* that is distinct from the memory space of the physical machine
- Process
  - An instance of an executing program is *a process consisting of an address space and one or more threads of control*
- Dual Mode operation/Protection
  - Only the "system" has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses

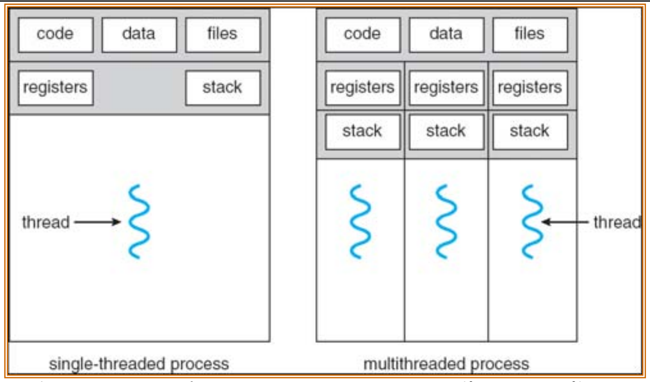


## Recall: Process

- Process: execution environment with Restricted Rights
  - Address Space with One or More Threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Encapsulate one or more threads sharing process resources
- Why processes?
  - Protected from each other!
  - OS Protected from them
  - Navigate fundamental tradeoff between protection and efficiency
  - Processes provides memory protection
  - Threads more efficient than processes (later)
- Application instance consists of one or more processes

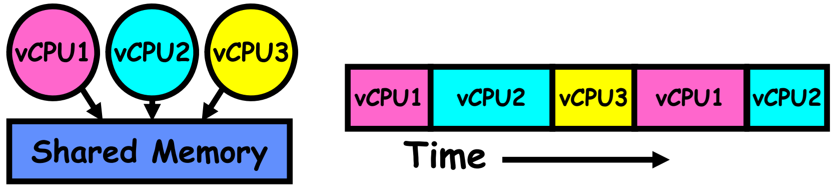


## Single and Multithreaded Processes




- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

## Recall: give the illusion of multiple processors?

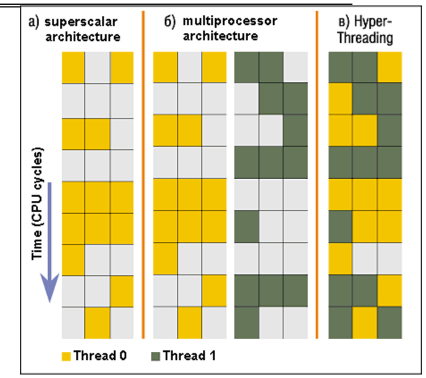



- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
- Each virtual "CPU" needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one virtual CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block
- What triggers switch?
  - Timer, voluntary yield, I/O, other things



## Simultaneous MultiThreading/Hyperthreading

- Hardware technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading duplicates register state to make a second "thread," allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!

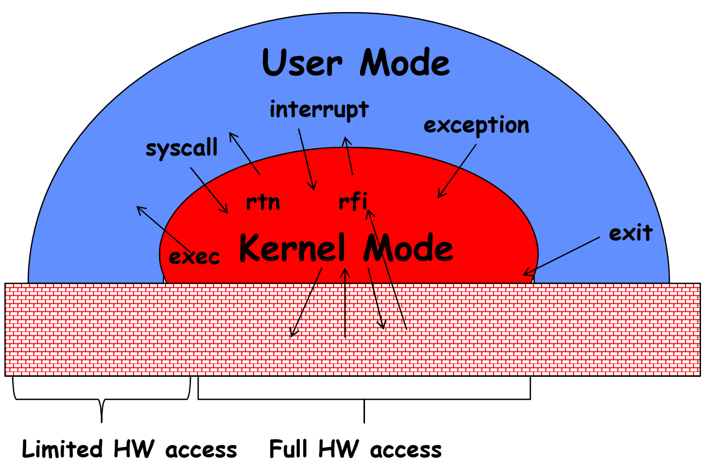



Colored blocks show instructions executed

- Original technique called "Simultaneous Multithreading"
  - http://www.cs.washington.edu/research/smt/index.html
  - SPARC, Pentium 4/Xeon ("Hyperthreading"), Power 5

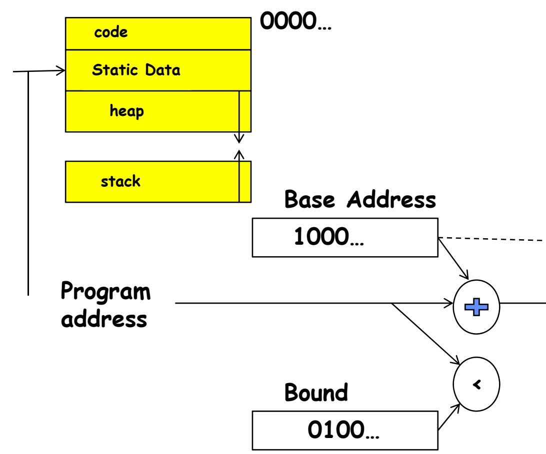


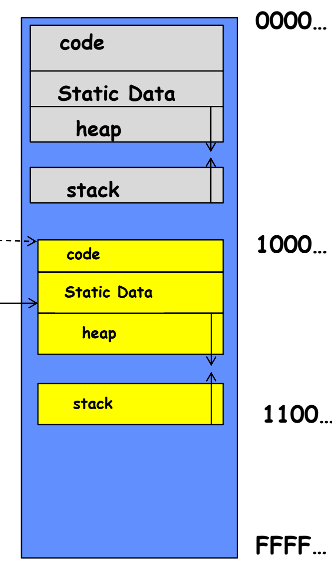
## Recall: User/Kernal(Priviledged) Mode





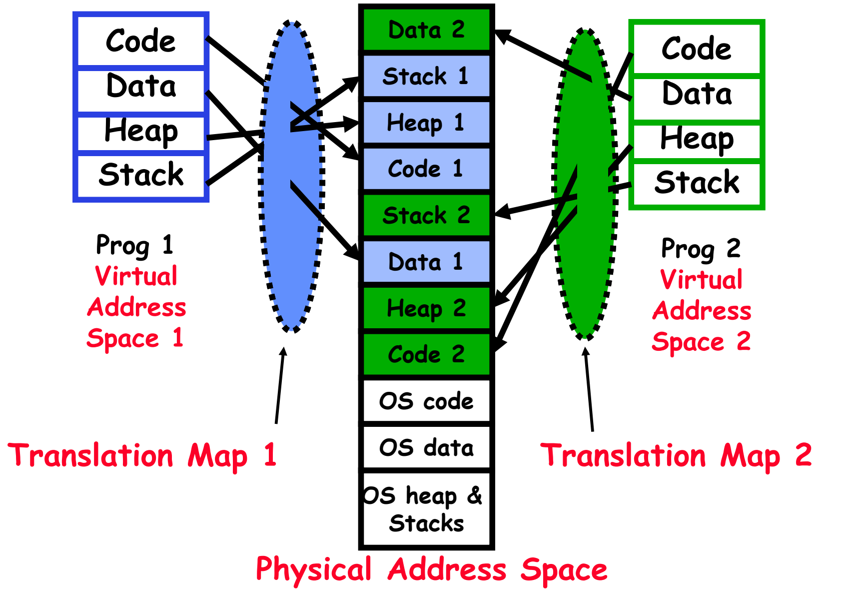

## Recall: A simple address translation (B&B)



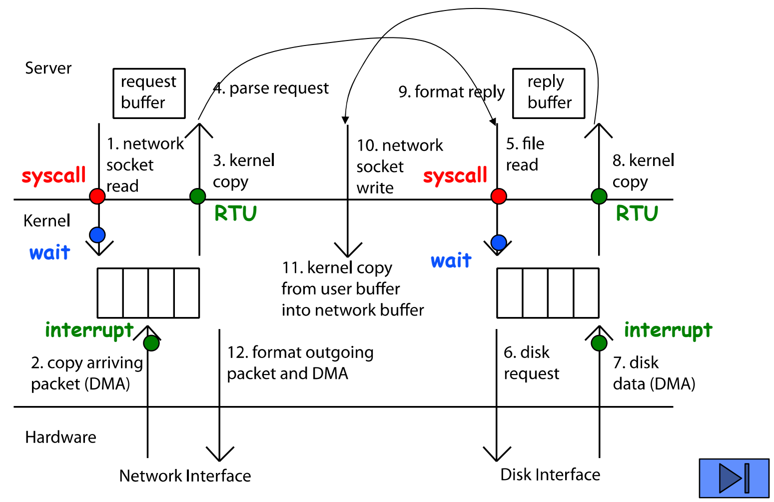

- Can the program touch OS?
- Can it touch other programs?

## Recall: Address Mapping






## Putting it together: web server






## Running Many Programs

- We have the basic mechanism to
  - switch between user processes and the kernel,
  - the kernel can switch among user processes,
  - Protect OS from user processes and processes from each other
- Questions ???
  - How do we represent user processes in the OS?
  - How do we decide which user process to run?
  - How do we pack up the process and set it aside?
  - How do we get a stack and heap for the kernel?
  - Aren't we wasting are lot of memory?
  - …



## Process Control Block

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation, …
- Kernel Scheduler maintains a data structure containing the PCBs
- Scheduling algorithm selects the next one to run

## Scheduler

```
if ( readyProcesses(PCBs) ) {
      nextPCB = selectProcess(PCBs);
      run( nextPCB );
} else {
      run_idle_process();
}
```

- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..



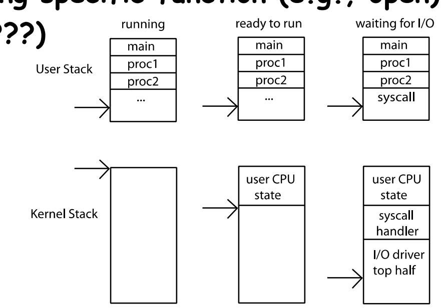
## Implementing Safe Kernel Mode Transfers

- Important aspects:
  - Separate kernel stack
  - Controlled transfer into kernel (e.g. syscall table)
- Carefully constructed kernel code packs up the user process state an sets it aside.
  - Details depend on the machine architecture
- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself.



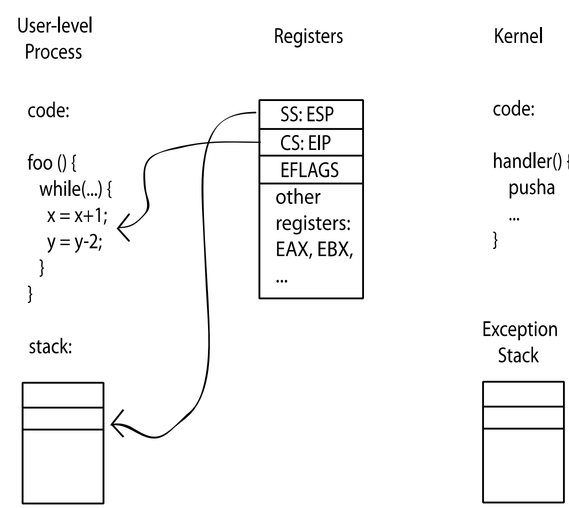
## Need for Separate Kernel Stacks

- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
  - OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
  - Syscall handler copies user args to kernel space before invoking specific function (e.g., open)
  - Interrupts (???)

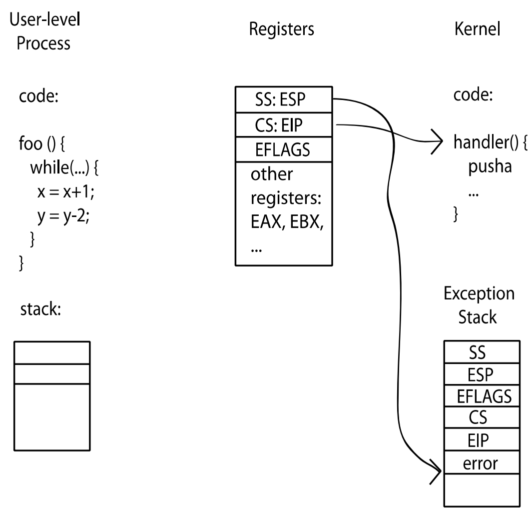





## Before



## During






## Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user(!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - into user memory

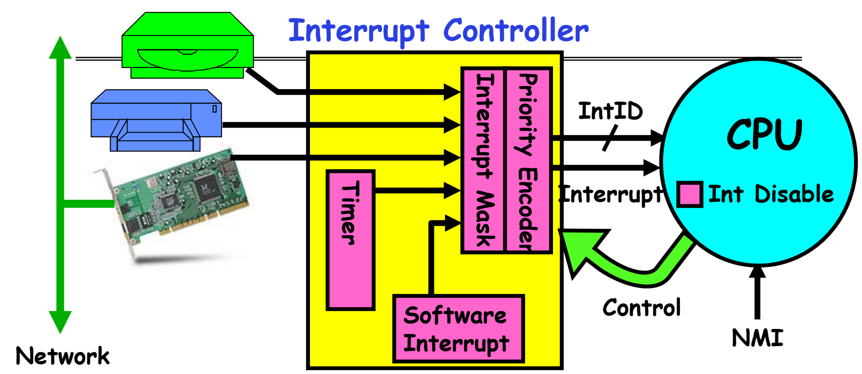


## Hardware support: Interrupt Control

- Interrupt processing not be visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?
- Interrupt Handler invoked with interrupts 'disabled'
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack up in a queue and pass off to an OS thread for hard work
    - » wake up an existing OS thread
- OS kernel may enable/disable interrupts
  - On x86: CLI (disable interrupts), STI (enable)
  - Atomic section when select next process/thread to run
  - Atomic return from interrupt or syscall
- HW may have multiple levels of interrupt
  - Mask off (disable) certain interrupts, eg., lower priority
  - Certain non-maskable-interrupts (nmi)
    - » e.g., kernel segmentation fault



## Interrupt Controller




- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled

## How do we take interrupts safely?

- Interrupt vector
  - Limited number of entry points into kernel
- Kernel interrupt stack
  - Handler works regardless of state of user code
- Interrupt masking
  - Handler is non-blocking
- Atomic transfer of control
  - "Single instruction"-like to change:
    » Program counter
    » Stack pointer
    » Memory protection
    » Kernel/user mode
- Transparent restartable execution
  - User program does not know interrupt occurred



## Administrivia: Getting started

- Kubiatowicz Office Hours:
  - 1pm-2pm, Monday/Wednesday
- Homework 0 immediately ⇒ Due on Monday!
  - Get familiar with all the cs162 tools
  - Submit to autograder via git
- Should be going to section already!
- Participation: Get to know your TA!
- Friday is Drop Deadline!
- Group sign up form out next week (after drop deadine)
  - Get finding groups ASAP
  - 4 people in a group!
- Finals conflicts: Tell us now
  - Must give us a good reason for providing an alternative
  - No alternate time if the conflict is because of an overlapping class (e.g. EE122)!



## Question

- Process is an instance of a program executing.
  - The fundamental OS responsibility
- Processes do their work by processing and calling file system operations
- Are their any operations on processes themselves?
- exit ?



## pid.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  int c;

  pid_t pid = getpid();   /* get current processes PID */

  printf("My pid: %d\n", pid);

  c = fgetc(stdin);
  exit(0);
}
```

ps anyone?

## Can a process create a process ?

- Yes
- Fork creates a copy of process
- Return value from Fork: integer
  - When > 0:
    - » Running in (original) Parent process
    - » return value is pid of new child
  - When = 0:
    - » Running in new Child process
  - When < 0:
    - » Error! Must handle somehow
    - » Running in original process
- All of the state of original process duplicated in both Parent and Child!
  - Memory, File Descriptors (next topic), etc…



## fork1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 1024
int main(int argc, char *argv[])
{
  char buf[BUFSIZE];
  size_t readlen, writelen, slen;
  pid_t cpid, mypid;
  pid_t pid = getpid();         /* get current processes PID */
  printf("Parent pid: %d\n", pid);
  cpid = fork();
  if (cpid > 0) {                  /* Parent Process */
    mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
  }  else if (cpid == 0) {         /* Child Process */
    mypid = getpid();
    printf("[%d] child\n", mypid);
  } else {
    perror("Fork failed");
    exit(1);
  }
  exit(0);
}
```



## UNIX Process Management

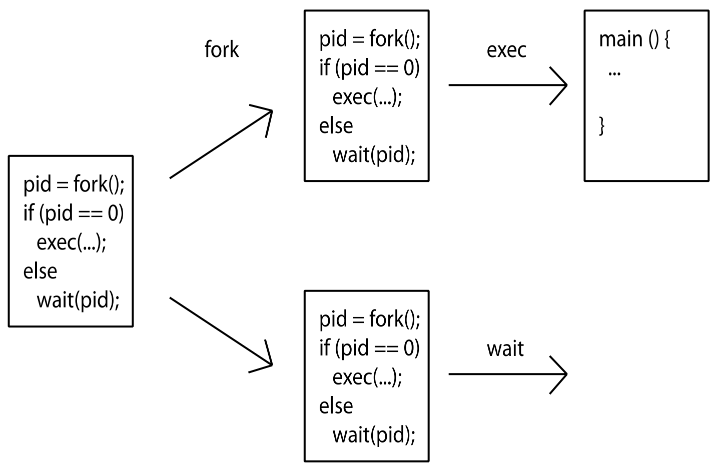
- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec – system call to *change the program* being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send a notification to another process



## fork2.c

```
…
cpid = fork();
if (cpid > 0) {                /* Parent Process */
  mypid = getpid();
  printf("[%d] parent of [%d]\n", mypid, cpid);
  tcpid = wait(&status);
  printf("[%d] bye %d\n", mypid, tcpid);
}  else if (cpid == 0) {      /* Child Process */
  mypid = getpid();
  printf("[%d] child\n", mypid);
}
…
```

## UNIX Process Management






## Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells
- Example: to compile a C program

  cc –c sourcefile1.c

  cc –c sourcefile2.c

  ln –o program sourcefile1.o sourcefile2.o

  ./program

HW1



## Signals – infloop.c

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>

#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum)
{
  printf("Caught signal %d - phew!\n",signum);
  exit(1);
}

int main() {
  signal(SIGINT, signal_callback_handler);

  while (1) {}
}
```

Got top?



## Process races: fork.c

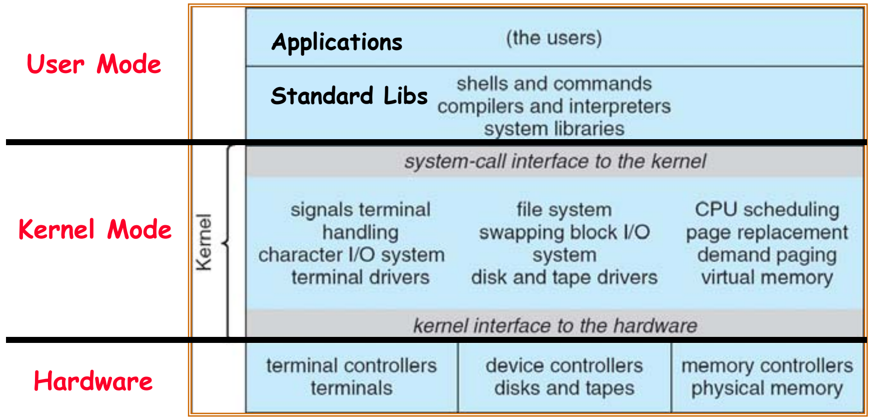
```
if (cpid > 0) {
   mypid = getpid();
   printf("[%d] parent of [%d]\n", mypid, cpid);
   for (i=0; i<100; i++) {
     printf("[%d] parent: %d\n", mypid, i);
     //      sleep(1);
   }
 }  else if (cpid == 0) {
   mypid = getpid();
   printf("[%d] child\n", mypid);
   for (i=0; i>-100; i--) {
     printf("[%d] child: %d\n", mypid, i);
     //      sleep(1);
   }
 }
```

- Question: What does this program print?
- Does it change if you add in one of the sleep() statements?

## Break



## Recall: UNIX System Structure





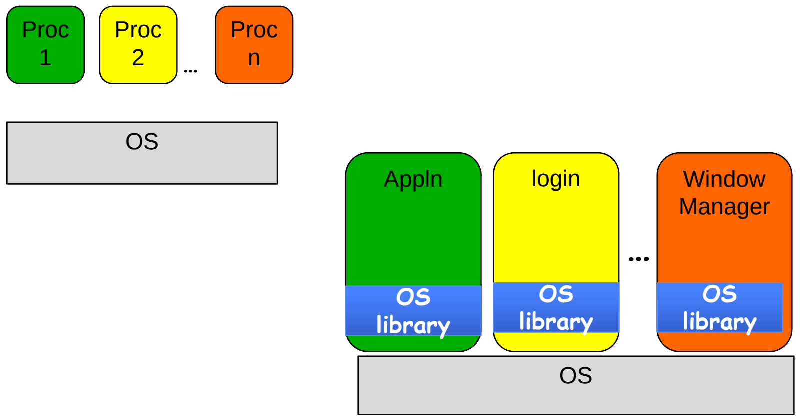

## How does the kernel provide services?

- You said that applications request services from the operating system via *syscall*, but …
- I've been writing all sort of useful applications and I never ever saw a "syscall" !!!

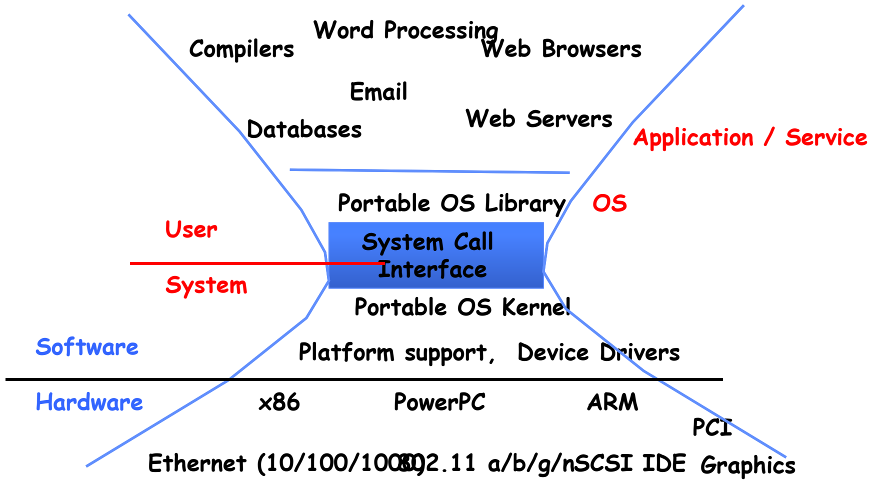
- That's right.
- It was buried in the programming language runtime library (e.g., libc.a)
- … Layering



## OS run-time library



## A Kind of Narrow Waist



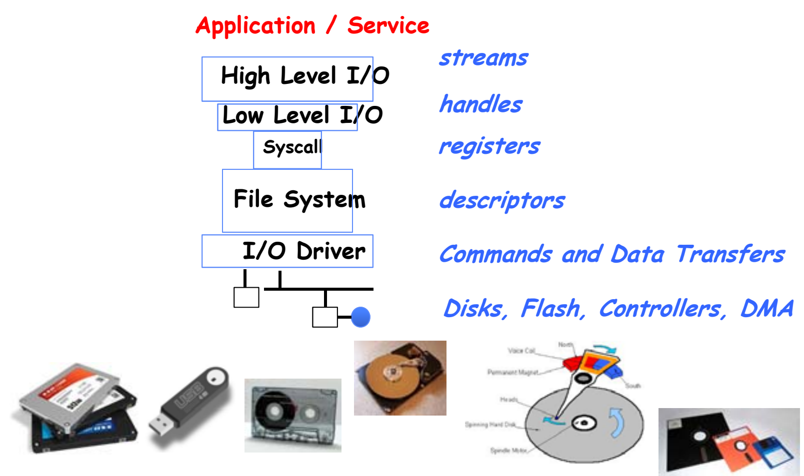



## Key Unix I/O Design Concepts

- Uniformity
  - file operations, device I/O, and interprocess communication through open, read/write, close
  - Allows simple composition of programs
    - » find | grep | wc …
- Open before use
  - Provides opportunity for access control and arbitration
  - Sets up the underlying machinery, i.e., data structures
- Byte-oriented
  - Even if blocks are transferred, addressing is in bytes
- Kernel buffered reads
  - Streaming and block devices looks the same
  - read blocks process, yielding processor to other task
- Kernel buffered writes
  - Completion of out-going transfer decoupled from the application, allowing it to continue
- Explicit close



## I/O & Storage Layers






## The file system abstraction

- File
  - Named collection of data in a file system
  - File data
    - » Text, binary, linearized objects
  - File Metadata: information about the file
    - » Size, Modification Time, Owner, Security info
    - » Basis for access control
- Directory
  - "Folder" containing files & Directories
  - Hierachical (graphical) naming
    - » Path through the directory graph
    - » Uniquely identifies a file or directory
      - /home/ff/cs162/public_html/fa14/index.html
  - Links and Volumes (later)

## C high level File API – streams (review)

- Operate on "streams" - sequence of bytes, whether text or data, with a position

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
int fclose( FILE *fp );
```

| Mode Text | Binary | Descriptions |
|---|---|---|
| r | rb | Open existing file for reading |
| w | wb | Open for writing; created if does not exist |
| a | ab | Open for appending; created if does not exist |
| r+ | rb+ | Open existing file for reading & writing. |
| w+ | wb+ | Open for reading & writing; truncated to zero if exists, create otherwise |
| a+ | ab+ | Open for reading & writing. Created if does not exist. Read from beginning, write as append |

Don't forget to flush



## Connecting Processes, Filesystem, and Users

- Process has a 'current working directory'
- Absolute Paths
  - /home/ff/cs152
- Relative paths
  - index.html, ./index.html - current WD
  - ../index.html - parent of current WD
  - ~, ~cs152 - home directory



## C API Standard Streams

- Three predefined streams are opened implicitly when the program is executed.
  - FILE *stdin – normal source of input, can be redirected
  - FILE *stdout – normal source of output, can too
  - FILE *stderr – diagnostics and errors
- STDIN / STDOUT enable composition in Unix
  - Recall: Use of pipe symbols connects STDOUT and STDIN
    - » find | grep | wc …

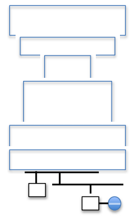


## C high level File API – stream ops

```
#include <stdio.h>
// character oriented
int fputc( int c, FILE *fp );                     // rtn c or
EOF on err
int fputs( const char *s, FILE *fp );    // rtn >0 or EOF

int fgetc( FILE * fp );
char *fgets( char *buf, int n, FILE *fp );

// block oriented
size_t fread(void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements,
             size_t number_of_elements, FILE *a_file);

// formatted
int fprintf(FILE *restrict stream, const char *restrict
format, ...);
int fscanf(FILE *restrict stream, const char *restrict format,
... );
```

## C Stream API positioning

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



- Preserves high level abstraction of a uniform stream of objects
- Adds buffering for performance

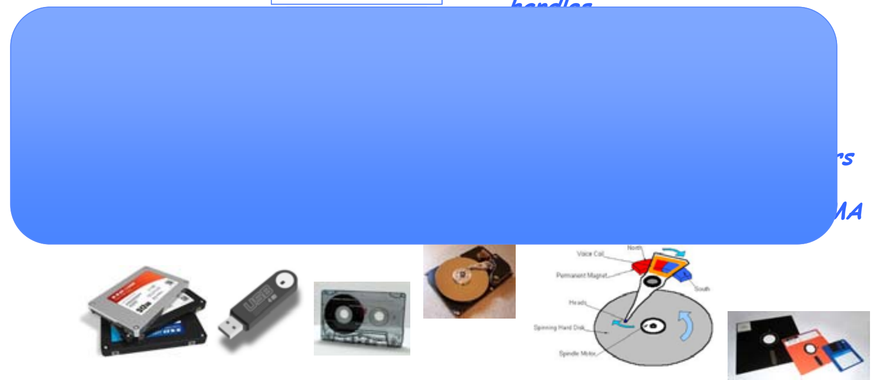


## What's below the surface ??

Application / Service

High Level I/O

*streams*





## C Low level I/O

- Operations on File Descriptors – as OS object representing the state of a file
  – User has a "handle" on the descriptor

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

int open (const char *filename, int flags [, mode_t mode])
int creat (const char *filename, mode_t mode)
int close (int filedes)
```

Bit vector of:
- Access modes (Rd, Wr, …)
- Open Flags (Create, …)
- Operating modes (Appends, …)

Bit vector of Permission Bits:
- User|Group|Other X R|W|X

http://www.gnu.org/software/libc/manual/html_node/Opening-and-Closing-Files.html



## C Low Level: standard descriptors

```
#include <unistd.h>

STDIN_FILENO -  macro has value 0
STDOUT_FILENO - macro has value 1
STDERR_FILENO - macro has value 2

int fileno (FILE *stream)

FILE * fdopen (int filedes, const char *opentype)
```

- Crossing levels: File descriptors vs. streams
- Don't mix them!

## C Low Level Operations

```
ssize_t read (int filedes, void *buffer, size_t maxsize)
 - returns bytes read, 0 => EOF, -1 => error
ssize_t write (int filedes, const void *buffer, size_t size)
 - returns bytes written

off_t lseek (int filedes, off_t offset, int whence)

int fsync (int fildes) – wait for i/o to finish
void sync (void) – wait for ALL to finish
```

- **When write returns, data is on its way to disk and can be read, but it may not actually be permanent!**



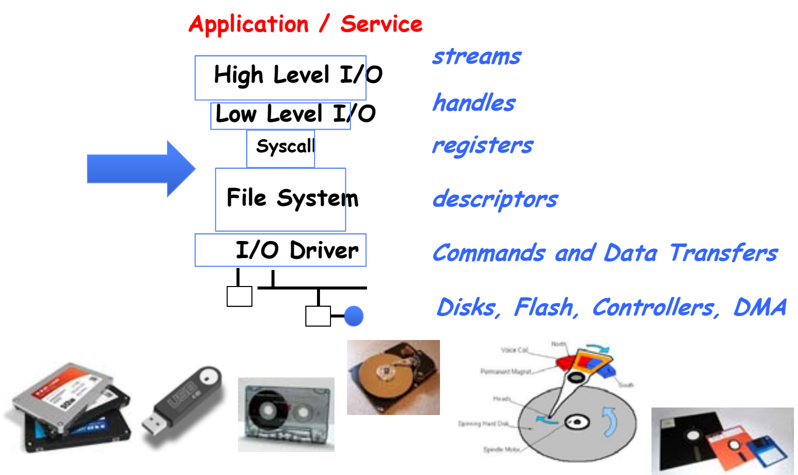
## And lots more !

- **TTYs versus files**
- **Memory mapped files**
- **File Locking**
- **Asynchronous I/O**
- **Generic I/O Control Operations**
- **Duplicating descriptors**

```
int dup2 (int old, int new)
int dup (int old)
```

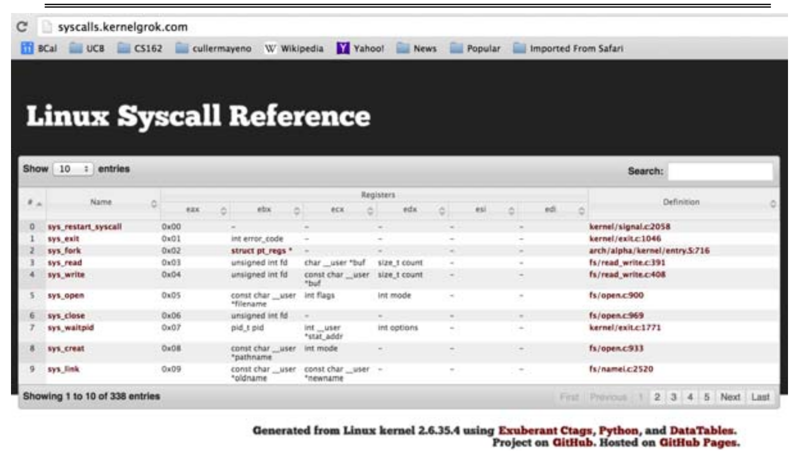


## What's below the surface ??






## SYSCALL




| # | Name | Registers eax | ebx | ecx | edx | esi | edi | Definition |
|---|---|---|---|---|---|---|---|---|
| 0 | sys_restart_syscall | 0x00 | - | - | - | - | - | kernel/signal.c:2058 |
| 1 | sys_exit | 0x01 | int error_code | - | - | - | - | kernel/exit.c:1046 |
| 2 | sys_fork | 0x02 | struct pt_regs * | - | - | - | - | arch/alpha/kernel/entry.S:716 |
| 3 | sys_read | 0x03 | unsigned int fd | char __user *buf | size_t count | - | - | fs/read_write.c:391 |
| 4 | sys_write | 0x04 | unsigned int fd | const char __user *buf | size_t count | - | - | fs/read_write.c:408 |
| 5 | sys_open | 0x05 | const char __user *filename | int flags | int mode | - | - | fs/open.c:900 |
| 6 | sys_close | 0x06 | unsigned int fd | - | - | - | - | fs/open.c:969 |
| 7 | sys_waitpid | 0x07 | pid_t pid | int __user *stat_addr | int options | - | - | kernel/exit.c:1771 |
| 8 | sys_creat | 0x08 | const char __user *pathname | int mode | - | - | - | fs/open.c:933 |
| 9 | sys_link | 0x09 | const char __user *oldname | const char __user *newname | - | - | - | fs/namei.c:2520 |

Showing 1 to 10 of 338 entries

Generated from Linux kernel 2.6.35.4 using Exuberant Ctags, Python, and DataTables.
Project on GitHub. Hosted on GitHub Pages.

- **Low level lib parameters are set up in registers and syscall instruction is issued**
  - **A type of synchronous exception that enters well-defined entry points into kernel**

## Internal OS File Descriptor

- Internal Data Structure describing everything about the file
  - Where it resides
  - Its status
  - How to access it


```
struct file {
        union {
                struct llist_node       fu_llist;
                struct rcu_head         fu_rcuhead;
        } f_u;
        struct path             f_path;
#define f_dentry        f_path.dentry
        struct inode            *f_inode;       /* cac
        const struct file_operations    *f_op;

        /*
         * Protects f_ep_links, f_flags.
         * Must not be taken from IRQ context.
         */
        spinlock_t              f_lock;
        atomic_long_t           f_count;
        unsigned int            f_flags;
        fmode_t                 f_mode;
        struct mutex            f_pos_lock;
        loff_t                  f_pos;
        struct fown_struct      f_owner;
        const struct cred       *f_cred;
        struct file_ra_state    f_ra;

        u64                     f_version;
#ifdef CONFIG_SECURITY
        void                    *f_security;
#endif
        /* needed for tty driver, and maybe others */
        void                    *private_data;

#ifdef CONFIG_EPOLL
        /* Used by fs/eventpoll.c to link all the hook
        struct list_head        f_ep_links;
        struct list_head        f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
        struct address_space    *f_mapping;
} __attribute__((aligned(4)));  /* lest something weir
```



## Device Drivers

- **Device Driver:** Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call
- Device Drivers typically divided into two pieces:
  - Top half: accessed in call path from system calls
    - » implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    - » This is the kernel's interface to the device driver
    - » Top half will *start* I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    - » Gets input or transfers next block of output
    - » May wake sleeping threads if I/O now complete



## File System: from syscall to driver

In fs/read_write.c

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
  ssize_t ret;
  if (!(file->f_mode & FMODE_READ)) return -EBADF;
  if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
    return -EINVAL;
  if (unlikely(!access_ok(VERIFY_WRITE, buf, count))) return -EFAULT;
  ret = rw_verify_area(READ, file, pos, count);
  if (ret >= 0) {
    count = ret;
    if (file->f_op->read)
      ret = file->f_op->read(file, buf, count, pos);
    else
      ret = do_sync_read(file, buf, count, pos);
    if (ret > 0) {
      fsnotify_access(file->f_path.dentry);
      add_rchar(current, ret);
    }
    inc_syscr(current);
  }
  return ret;
}
```



## Low Level Driver

- Associated with particular hardware device
- Registers / Unregisters itself with the kernel
- Handler functions for each of the file operations

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*flock) (struct file *, int, struct file_lock *);
    [...]
};
```

## Life Cycle of An I/O Request

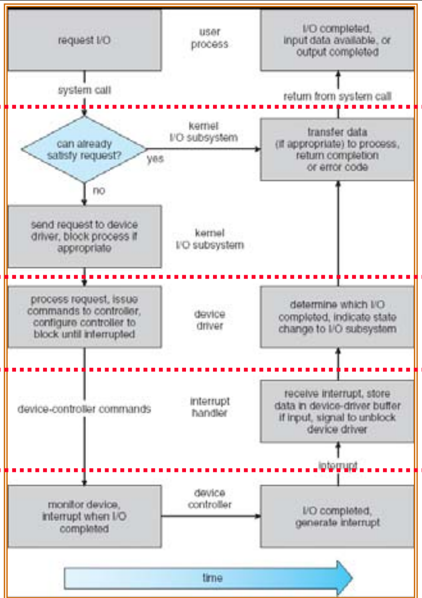
User Program

Kernel I/O Subsystem

Device Driver Top Half

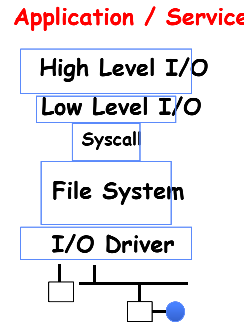
Device Driver Bottom Half

Device Hardware






## So what happens when you fgetc?




*streams*

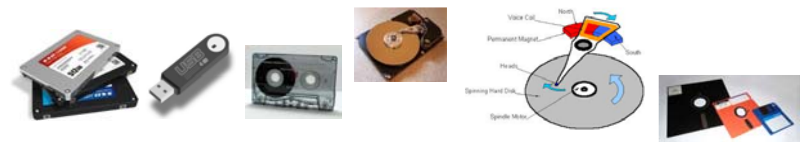
*handles*

*registers*

*descriptors*

*Commands and Data Transfers*

*Disks, Flash, Controllers, DMA*





## Summary

- Process: execution environment with Restricted Rights
  - Address Space with One or More Threads
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Encapsulate one or more threads sharing process resources
- Interrupts
  - Hardware mechanism for regaining control from user
  - Notification that events have occurred
  - User-level equivalent: Signals
- Native control of Process
  - Fork, Exec, Wait, Signal
- Basic Support for I/O
  - Standard interface: open, read, write, seek
  - Device drivers: customized interface to hardware