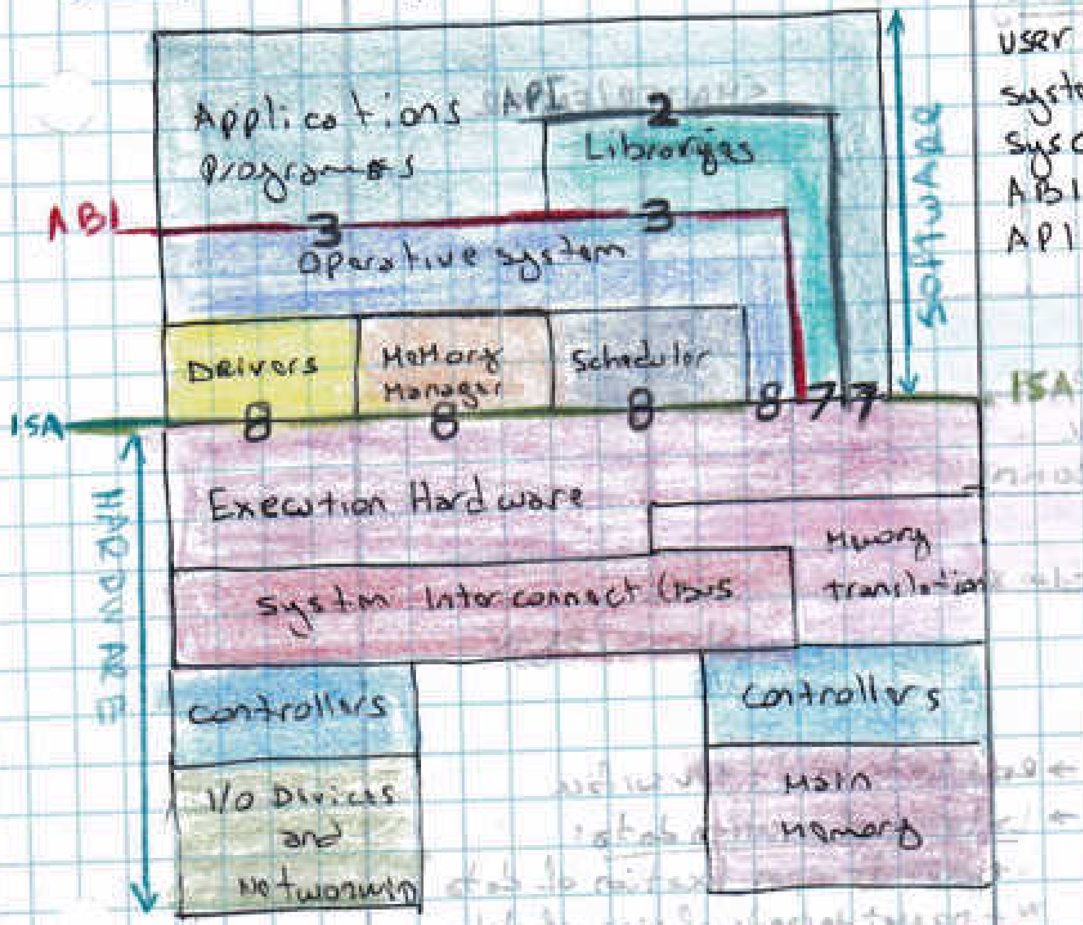
[redacted] self study



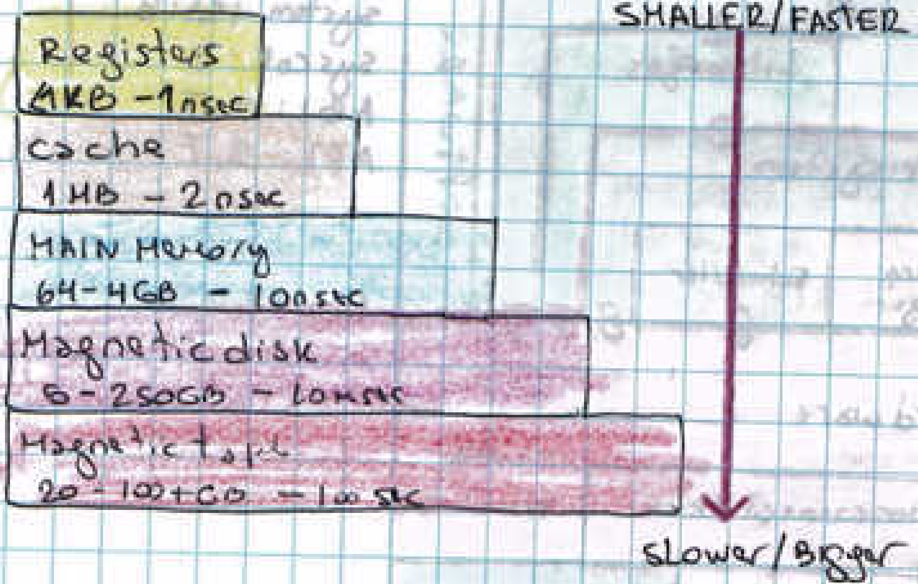
User ISA: 7
System ISA: 8
Syscall: 3
ABI: 3,7
API: 2,7

- Instruction Set Architecture (ISA)
- Application Binary Interface (ABI)
- Application Programming Interface (API)

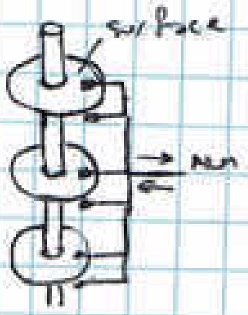
- Application Binary Interface (ABI):
  + It provides a program with access to the hardware resource and services available in a system
  + Consist of User ISA and System Call Interface

- Application Programming Interface (API):
  + IT is a source code interface that an OS, library or service provide to support requests made by computer programs

Memory Hierarchy

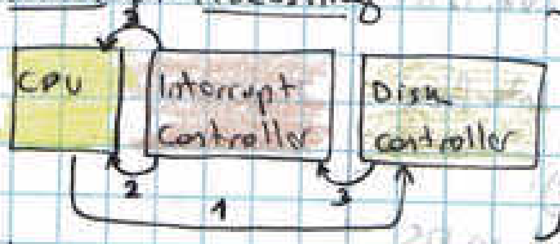


Hard Drives

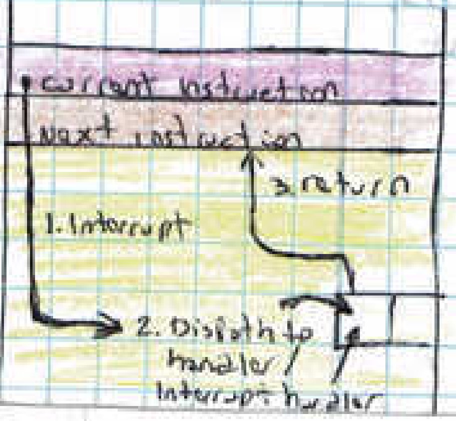


→ Read/write head : 1 per surface
→ Latency of accessing data:
+ depends upon location of data
+ Do Not depends of size of data
→ Disk Performance:
+ Measure in I/O operation per second (IOPS)
+ Not measure in Bytes per seconds (Bps
→ Began to be replaced by flash drives:
+ Almost constant access latency

Interrupt Processing



Steps in starting an I/O device and getting interrupt



How the CPU is interrupted

System Call Interface hide all the complexity of hardware and decide which process use the CPU for how long

[redacted] self study

→ Mega Bytes (MB)
→ Mega bits per seconds (Mbps)
→ I/O operations per second (IOPS)

What is an OS?

1. Hide details of interacting with hardware resoraes
   + Present user with a virtual execution environment, which is easier to use

2. Control and arbitrates users' access to hardware resources
   + Decides which programs get "how much time for using different resources and "when it get that time" ie: cpu, ~~[illegible]~~, etc
   + Decide which program gets "how much space" on different resources and "where" ie: Main memory, HD

CPU Privilege levels

→ Instruction Set Architecture (ISA):
   + Multiple privilege levels in which a software code can execute.

→ x86 Isa
   + specifies four privilege levels 0, 1, 2, 3 (0: highest, 3: lowest) (also called privilege rings)
   + level 0: OS code
   + level 3: Application and libraries
   + level 1 and 2: unused except in virtual machine envir
   + Invoking a system call from a process causes "change of privilege level" from 3 to 0

• Page table there is a place where indicate in which level a part of the code should be executed.

flag
Register
Privilege level

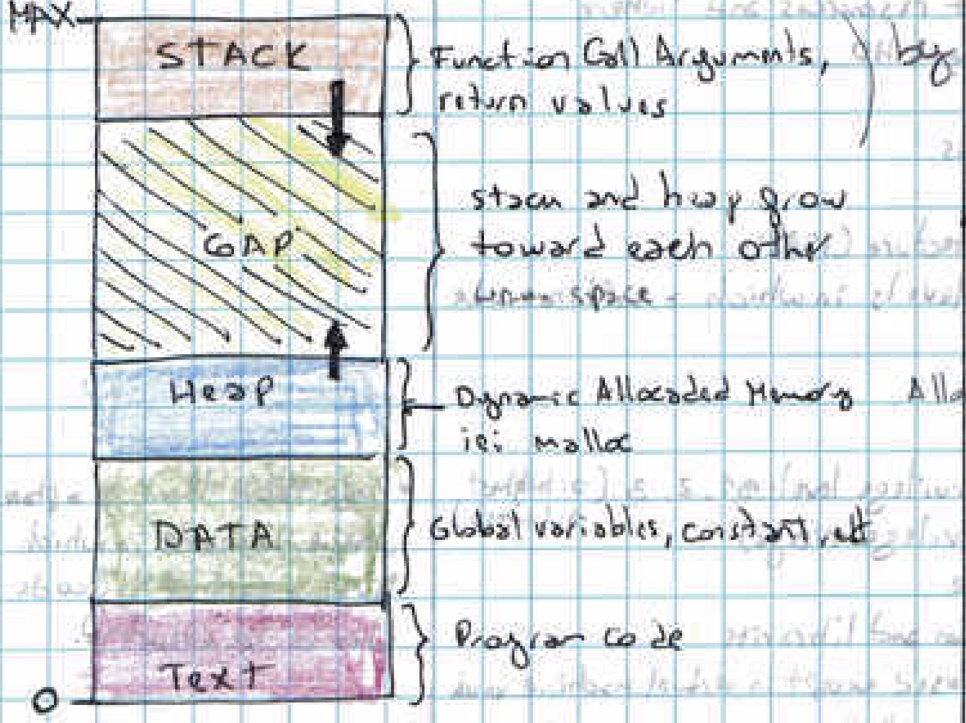
Process

Library 13 o[redacted]
front desk

+ A process is a program in execution but it is not the same as a program
  → It is not a passive entity stored in the disk
  → It is an actively executing entity
  → A program is just one component of a process

+ Memory space (static & dynamic)
+ Produce call stack
+ Register and counters (Program counter, stack pointer, General purpose registers)
+ open files, connection
+ ...

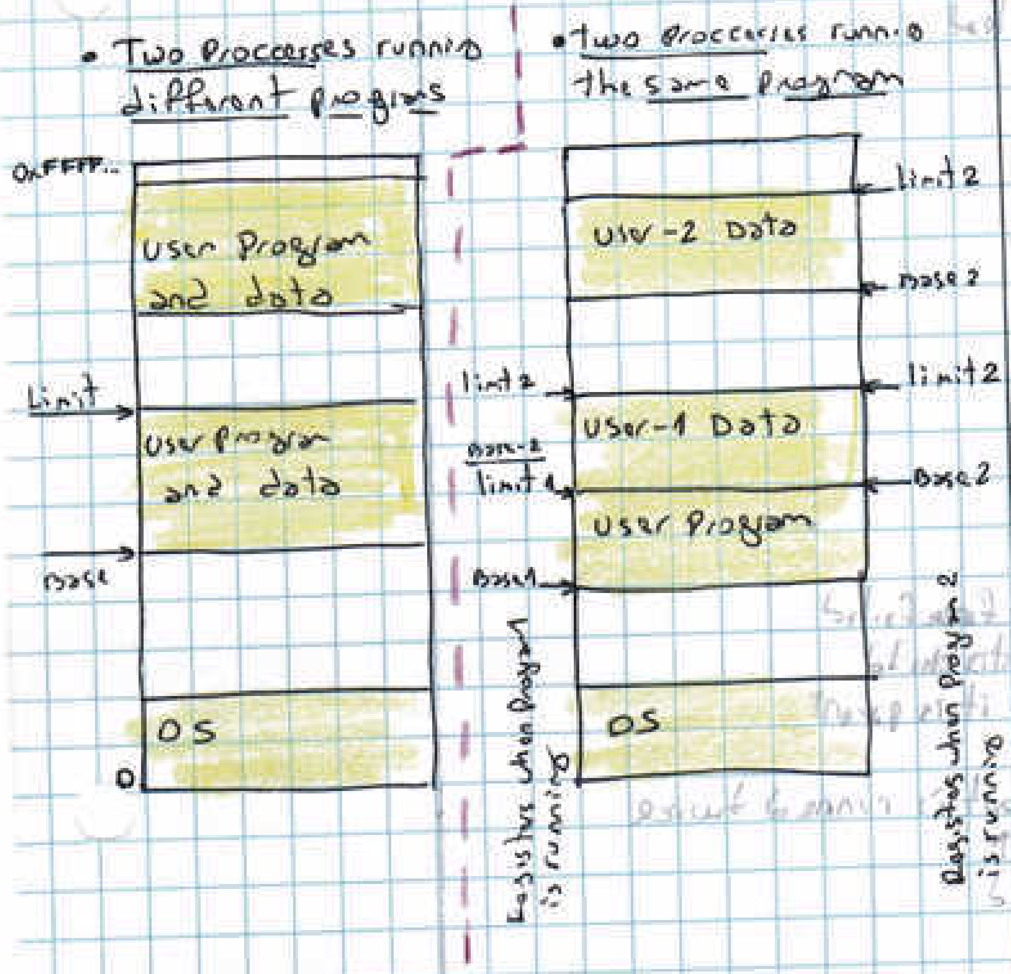
Memory Layout of a typical process


MAX
STACK
GAP
Heap
DATA
Text
0
Function Call Arguments, return values
stack and heap grow toward each other
Dynamic Allocated Memory ie: malloc
Allocal
Global variables, constant, etc
Program code

[redacted] self study

Multiple Processes Sharing Main Memory

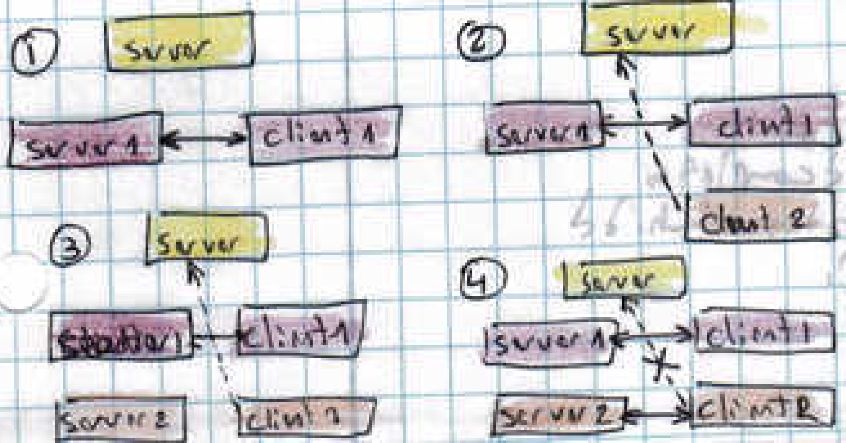
- Two Processes running different programs
- two processes running the same program



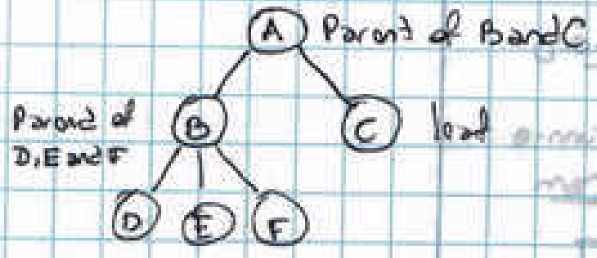
Process creation

+ fork() system call
+ one process start another process (ie server)
+ OS creates a process to provide a service

+ Example



Process Hierarchy Tree



Examinto Process

+ ps - elf
+ /proc
+ top

Creating new Process

```
pid = fork()
 if pid == -1 then fork failed
 if pid == 0 then it's child
 if pid > 0 then it is parent
```

+ fork is called once but is runned twice
  → once for parent
  → once for child

Exec

+ replace current process image with new program image
+ All I/O descriptors open before exec remain open after exec (file descriptors, socket descriptors pipe descriptors, ect)
+ man 2 exec

wait () system call

+ Help parent process to
  → know when child completes
  → check return status of child
(→ check wait pid()

[redacted] selfstudy

<u>example</u>

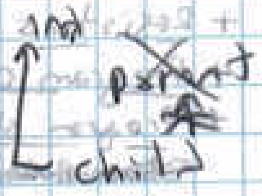
```
if ((pid = fork()) == 0) {
    //child
    ret = execvp( —, — , myfile, — );
    if (ret) == −1
        ↳ failed
} else {
    //parent
    while ( wait(Null) != pid );

    //remove file
    unlink ("myfile")
}
```

+ waitpid()
+ sleep()
+ exit
  → 0 = normal
  → NON 0 = viol
  → status returned to forking parent

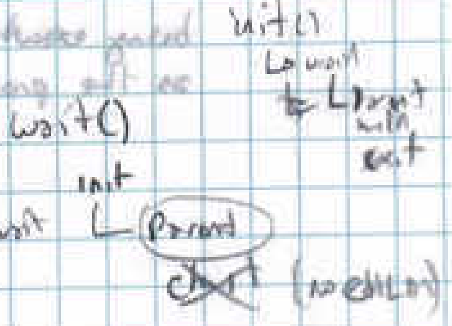
<u>Orphan Process</u>

+ when parent dies, child process become orphan
  + init process (pid = 1) become parent of orphan child
+ kill orphans: "kill -9 <pids>"
+ ps -f shows parent process id of orphan



<u>Zombie Process</u>

+ when child dies, SIGCHLD signal is sent to parent
+ If parent doesn't wait () on child & child exit (), parent becomes a zombie (status "Z" in ps)
+ zombie hang around till parent calls wait() or waitpid()
+ They do not occupy much of system resources
  + Integer status kept in OS → All other resources freed up



Procers States

[redacted]

Possible process states

+ Running
+ Blocked
+ Ready

$$\begin{tikzcd}
 & \text{Running} \arrow[dl, bend right, "1"'] \arrow[dr, bend left, "2"] & \\
\text{Blocked} \arrow[rr, bend right, "4"'] & & \text{Ready} \arrow[ul, "3"']
\end{tikzcd}$$

ex:

Transitions
1. Process blocks for input
2. Scheduler picks another process
3. scheduler picks this process
4. Input become available

- task_struct (http://lxe.linux.no)

- sched.h (#1215)

- Ready: Ready to use the CPU but the CPU schedule stop it.

- I/O Intensive: Priority normally given by the scheduler.

- ps-elf | grep xxxx

Typical kernel-level data structure for each process

Process management

+ Registers
+ Program counter: contain mem address of next instruction
+ Program status word: describes fully the condition ~~[illegible]~~ of a processor at each instant. It indicate which classes of operations are allowed and which are forbidden, and the status of all interrupts associated with the processor. It will also contain the address of the instruction currently being executed. It is held in a register known as the processor status register

PSW: condition bits, which are set by comparison instructions, The CPU priority, The mode user or kernel, and others. user program can read it but write only some of it fields. ~~Psw~~ PSW play an important part in system call I/O

[redacted] self study

Process management (Continuation)

+ Stack Pointer: Register which points to the top current stack in memory — stack pointer 20
  ↳ The stack contains one frame for each procedure that has been entered but not yet exited
    ↳ A procedure's stack frame holds those input parameters, local variables, & temporal variables that are not kept in registers

+ Process States: — Process state 90
  ↳ 1. Running (Actually using the CPU at that instant
  ↳ 2. Ready (runnable; temporarily stopped to let another process to run)
  ↳ 3. Blocked (unable to run until external event happends).

+ Priority: — Priority inversion problem 126
Priority scheduling 155

  ↳ Priority scheduling: each process is assigned a priority and the runnable process with the highest priority is allowed to run

+ Scheduling Parameters: (Priority) of the newly created thread
·1

+ Process ID: — Process ID: 54
+ Parent Process: the leaving process that created a child process — Process Identifier: 740
+ Process Group: A process group consist of its parent (and furthermore ancestors), siblings, and children (and further descendants) — Parent Process: 740
Process group: 741
signals, mitts: 116
Signal 741

+ signals
+ time when process started:
~~+ CPU time~~

Process Management (continuation)

+ CPU time used: Amount of time for which a central processing unit (CPU) was used for processing instruction of a computer program

+ Children's CPU time:

+ time of next alarm

*:

Memory Management

+ Pointer to text segment:

↳ text segment: (ie. program code) contains the machine instructions that form the program's executable code.

+ Pointer Data segment:

↳ Data segment: contains storage for all the program variables

+ Pointer to stack segment:

↳ stack segment:

text segment: 55-56 75B

~~Data se~~

Data segment 55-56, 75B

stack segment 55-56

[redacted] self study

File Management

+ Root Directory: top ~~priority of~~ directory hierarchy
+ Working Directory: each process have a working directory
+ File descriptors:
+ User ID
+ Group ID

Root directory 42, 260
~~working directory~~
Working directory 42, 270-271, 781

System call: A system call is use by a user program to obtain services from the operative system which traps into the kernel and invokes the OS.

system call 21

Inter-Process Communication (IPC)

simple forms:

Parent/Child
+ Command line argument
+ wait(): await process completion
+ wait pid(): wait for process to change state
+ exit(); Cause the shell to exit

+ Reading/modifying common files
  - (servers commonly use pid file to determine other active servers)
+ Signals
  - event notification from one process to another
+ Shared Memory:
  - common piece of read/write memory
  - Needs synchronization for access

+ Semaphores: Locking and event signaling mechanism between processes

Semaphor[illegible]29-130

+ Pipes: Connect two processes
  - ↳ Uni-directional (if used clearly)
  - ↳ ps -aux | more
  - ↳ Can be used bi-directionally with some synchronization effort

Pipe: 43,741

+ sockets:
  - ↳ Bi-directional
  - ↳ Not just across the network, but also between processes.

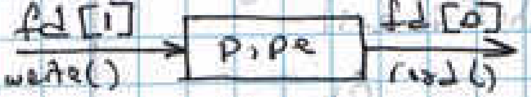
Socket: 773, 800

Pipes

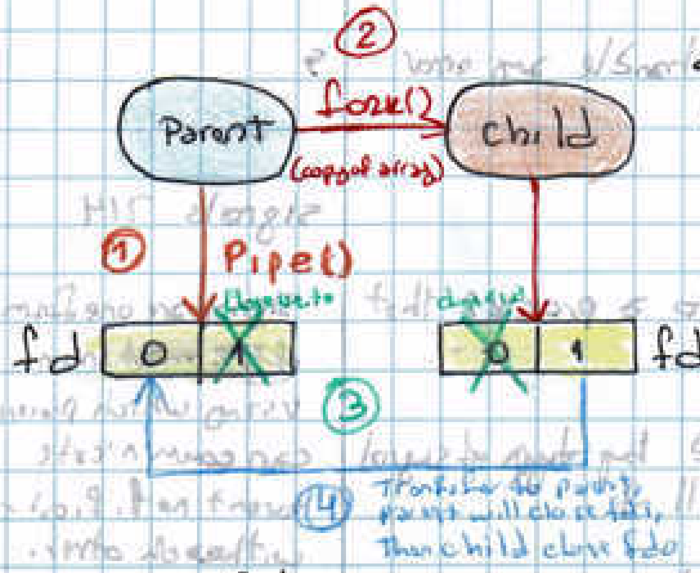
Pipe Abstraction

+ Write to one end, read from another
  - ↳ As for the receiver goes there in not boundaries but must read one message at the time.
  - ↳ Diference between byte-stream and message based is a diffrence ~~[illegible]~~ of the boundaries between messages

+ Pipe



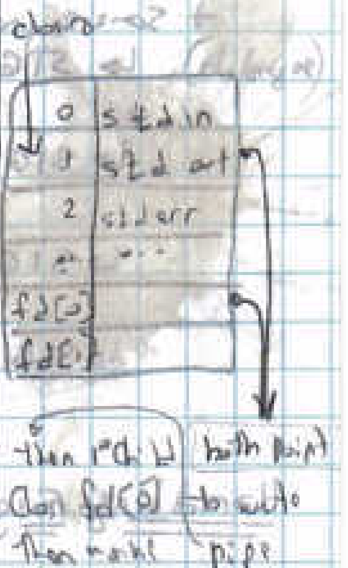
Parent-Child Communication using pipe



- Ones pipe is a fd close, it cannot be reopen until end of operation

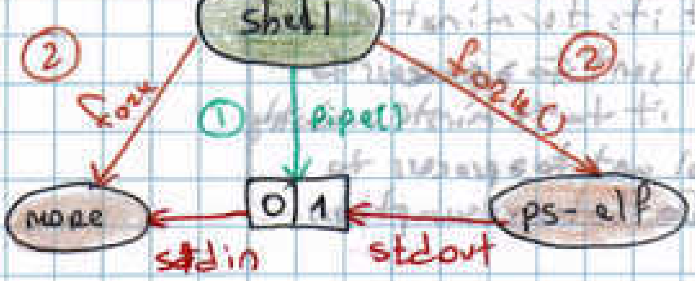
- ~~If fd donot close~~ ~~Then child doesnot~~ If a program doesn't close the pipe properly the program may not terminate

ex: parent → pipe → child
~~if child~~
if it is not able, the child doesn't know where the information is comming from because it cannot corroborate the origin of data.



ps

ex PS - elf | More



Being careful with read()/write()

+ read: read from file a file descriptor
  ↳ unistd.h
  ↳ ssize_t read(int fd, void *buf, size_t count);
  ↳
+ read (fds[0], buf, 6);
  ↳ Doesn't mean read will return with 6 bytes of data! why?
    ➡ read() could reach end of input stream (EOF)
    - Other end point may abruptly close the connection
    - read() could return on a signal)
+ You must incorporate error handling w/ every I/O call (signal recalls)

Error handling

④ Check return value of every read()/write() system call

② wait to read mandate or handle any error condition

signals

+ Signal is a ~~non-electrical~~ notification to a process that an event has occured
  ↳ from process or OS
+ type of event determined by type of signal
+ list of all signal type: kill -l
+ some signals: ~~SIGCHL~~

(signal.h) ↳ SIGCHLD: signal sent to a process when a child process terminates
↳ SIGTERM: signal sent to a process to request its termination
↳ SIGKILL: a signal sent to a process to cause it to terminate immediately
↳ SIGSTOP: a signal sent to a process to stop it for later resumption

signals 7.14

Signals are one form of IPC mechanism using which process can communicate event notifications with each other.

Handling Signals

+ Signals can be caught and an actions can be associated with them
  ↳ Note: SIGKILL and SIGSTOP cannot be caught
+ Actions to signals can be customized using sigaction() which associate a signal handler with the signal.
+ Default action for most signals is to terminate the process
  ↳ Note: Except SIGCHLD and SIGURG are ignored by default
+ unwanted signals can be ignored
  ↳ Note: except SIGKILL and SIGSTOP

SIGURG: a signal sent to a process when a socket has urgent data available to read.

[redacted] self study

MORE on SIGCHLD

+ sent to parent when a child process terminates or stop
+ If act.sa_~~handler~~ flags is SIG_IGN then SIGCHLD will be ignored (default behavior
+ If act.sa_~~handler~~ flags is SA_NOCLDSTOP then SIGCHLD won't be generated when children stop.
+ If act.sa_flags is SA_NOCLDWAIT then children of the calling process will not be transformed into zombies when they terminate
+ These need to be set in sigaction() <u>before</u> parent calls fork().

How to avoid zombies?

+ Parent could install a signal handler for SIGCHLD
+ Call wait() / waitpid() inside the signal handler

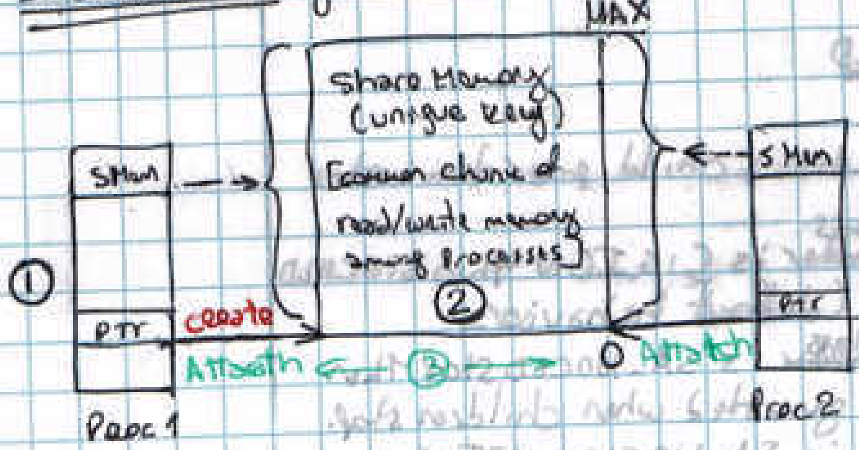
```
void handler_sigchld( int signo){
    pid_t pid;
    int status;
    pid = wait(&status);
    printf("child %d terminated\n", pid);
}
```

• Google "unix signals"
• Diff OS have different behaviors

Shared Memory, Semaphores

+ shmget: allocate a shared memory segment
+ shmat: shared memory operations (attach)
+ shmdt: " " " (detaches)
+ ~~[illegible]~~ shmctl: shared memory control
+ ~~sem~~ semget: get a semaphore set identifier
+ semop: semaphore operations
+ semctl: semaphore control operations

Shared Memory


MAX
Share Memory (unique key)
[Common chunk of read/write memory among processes]
②
SMem
PTR
①
create
Attach
③
0
Attach
SMem
PTR
Proc 1
Proc 2

① [Proc 1 create ~~Shared~~ Shared Memory ②

③ Attach proc 1 and any other process such as proc 2.

+ There is not restriction of which process can write to the same memory location, therefore there have to be a system that take care of the scheduling

+ At different of malloc(), which point return a pointer to a portion of the heap memory, we need to first create the portion of shared memory and then point to the end of it so we can read and write

Creating Shared Memory

```
int shmget (key_t key, size_t size int shmflg)

key_t key
int shmid
key = ftok ("file", 'A');
shmid = shmget (key, 1024, 0644 | IPC_CREAT)
```

+ ftok(): Convert a pathname and a project identifier to a System V IPC
  ↳ #include <sys/type.h> and <sys/ipc.h>
+ IPCS: to describe the share memory

[redacted] selfstudy

Attach and Detach Shared Memory

```
void *shmat (int shmid, void *shmaddr,
                int shmflg);
int shmdt (void *shmaddr);

 key_t key
 int shmid;
 char *data
  key = ftok ("somefile", 'A');
  shmid = shmget (key, 1024, 0644);
  data = shmat (shmid, (void*), 0, 0);
   ...
  shmdt (data);
```

Command-line IPC Control

+ IPC: Inter Process Communication
+ ipcs: List all IPC objects owned by the user
+ ipcrm: Remove specific IPC object

CPU scheduling (sect 2.4)

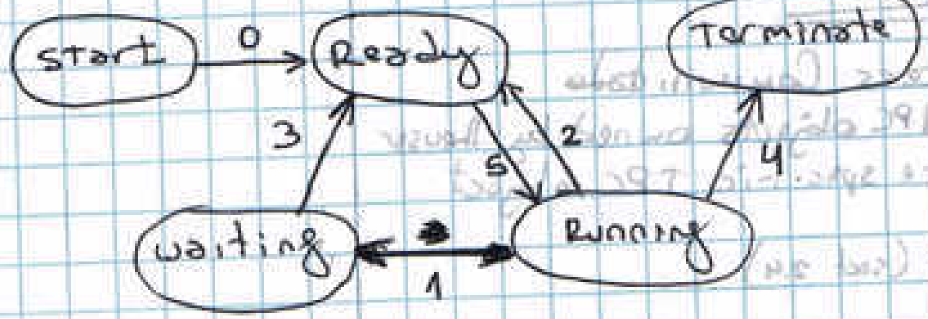
+ selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them

+ CPU scheduling can be non preemptive and/or preemptive algorithms:
  ↳ non preemptive: picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or until it voluntarily releases the CPU. Even if the process runs for hours, it will not forcibly suspended. No scheduling decisions are made during clock interruptions. After clock interrupt processing has been completed, the process that was running before the interrupt is resumed, unless a higher-priority was waiting for a now-satisfied timeout)

↳ Preemptive: picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available) ~~Doing preemptive scheduling~~

→ Doing preemptive scheduling requires having a clock interrupt occur at the end of an interval to give control of the CPU back to the scheduler. If no clock is available, nonpreemptive scheduling is the only option.

<u>Process life cycle</u>



CPU scheduling decisions may take place when a process:

Non-preemptives
1. Switches from running to waiting state
2. Switches from running to ready state
3. Switches from waiting to ready state
4. terminates

+ other scheduling are preemptive

[redacted] self study

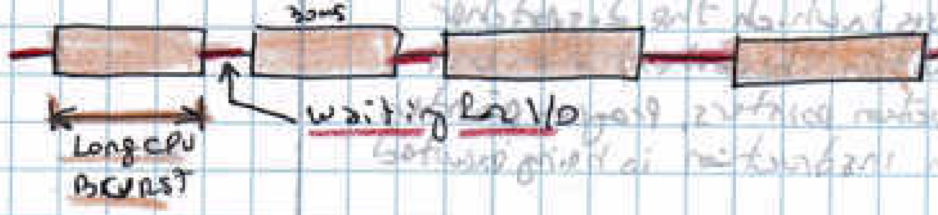
Alternating Sequence of CPU and I/O Bursts in a typical program

ex:
```
...
Load store
add store            ↑ CPU Burst
read from file       ↓
- WAIT for I/O -     ↕ I/O Burst
store increment      ↑
index                  CPU Burst
write to file        ↓
- WAIT for I/O -     ↕ I/O Burst
...
```

+ **A CPU-Bound process:**

↳ A CPU-Bound (compute-bound) processes have long CPU burst and thus infrequent I/O waits.

→ Note: The key factor is the length of the CPU burst, not the length of the I/O burst.



~~Compute bound to CPU~~	
CPU Bound	P 147
I/O Bound	P 147

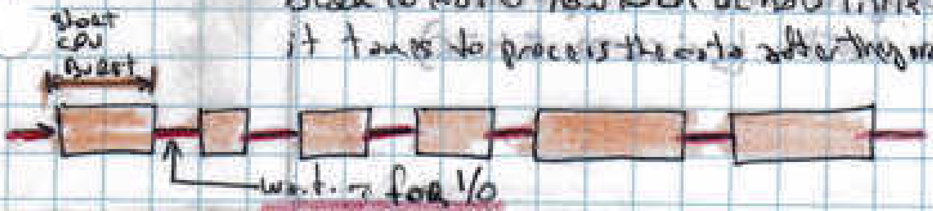
+ **An I/O-bound process:**

↳ I/O bound do not compute much between I/O request, not because they have especially long I/O requests.

→ It takes the same amount of time to issue the hardware request to read a disk block no matter how much or how little it takes to process the data after they arrive

I/O Burst longer than CPU burst

• Most desktop give more importance to I/O process than CPU processes ie: login



Short CPU Burst
waiting for I/O