OS Condensed Summary Notes For Quick In-Exam Strategic Fact Deployment

Maksymilian Mozolewski

April 26, 2021

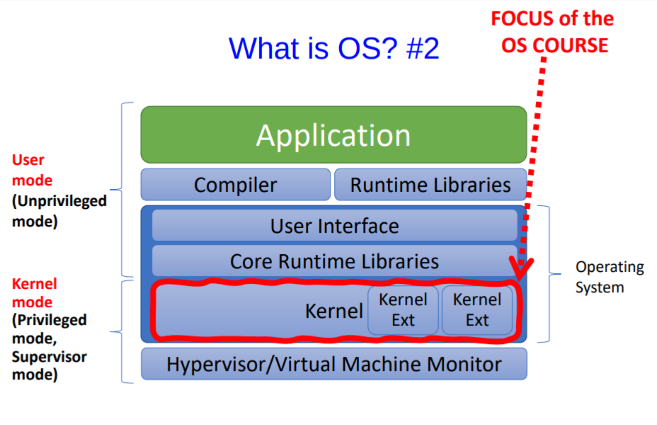
Contents

Introduction

Introduction

OS

An intermediary between the user of a computer and computer hardware. A program itself most intimately connected to the hardware. Everything you don't need to write in order to run your application. Library, all operations on I/O, syscalls. The OS can be an invisible intermediary



Main benefits of OS abstraction:

- Application benefits
  - programming simplicity - see high level abstraction (files) instead of low level hardware details (device registers)
  - abstractions are **reusable** across many platforms
  - **portability** (across machine configurations or architectures) - device independence: 3com or intel card?
- User benefits
  - safety: program sees its own virtual machine, can believe it owns the computer
  - OS **protects** programs from each other
  - OS **fairly multiplexes** resources across programs
  - efficiency - **share** one computer across many users. **Concurrent** execution of multiple programs

Basic OS Concepts

Multiprogramming : ability to keep multiple programs running in parallel (almost). This is done by keeping all possible jobs in the **job pool** on the disk, when a job is ready to take its turn to execute, it's brought to main memory and run for a bit untill the OS decides to switch it for another (because of an interrupt) or it completes.

Multitasking/timesharing : while multiprogramming makes teh OS efficient, it does not faciliate for user interaction! For the user to be able to run multiple tasks simultaneously and have a smooth experience, the CPU needs to actually switch jobs much more frequently so as it appears that all of them are being processed simultaneously. A system which faciliates **timesharing** is **interactive -** it frequently awaits user input and has short response time. A time-shared system enables the system to be used by **multiple** users simultaneously.

Process : a program which is loaded in main memory. This may be a full on program, or a printer job. Processes may spawn sub-processes if they wish to using syscalls. This is the **unit of work in a system**.

Job scheduling : prioritising which jobs should be in main memory at any time.

CPU scheduling : prioritising which jobs in main memory should be executed first.

Virtual memory : a technique that allows each program to see the entire memory as theirs and not mess with other processes as well as running programs which require more memory than is **physically available**

Logical memory : memory as seen by the programmer, abstracted away from the nitty-gritty mechanical details of the OS.

Interrupt driven : if there's no demand for action from the OS, it will IDLE. Events are almost always signalled by the occurence of an interrupt or a trap

Trap : (exception) is a software generated interrupt caused by either an error or a syscall.

Dual-mode/multimode operation : in a multiprogramming system processes are run in parallel, hence protections must be put into place so that processes cannot impede other processes. Most commonly this is done via introduction of **user mode** and **kernel mode**. In user mode certain possibly harmful **priviliged instructions** are forbidden by the **hardware itself** which sends a trap signal (switch to kernel mode, timer management, I/O control)

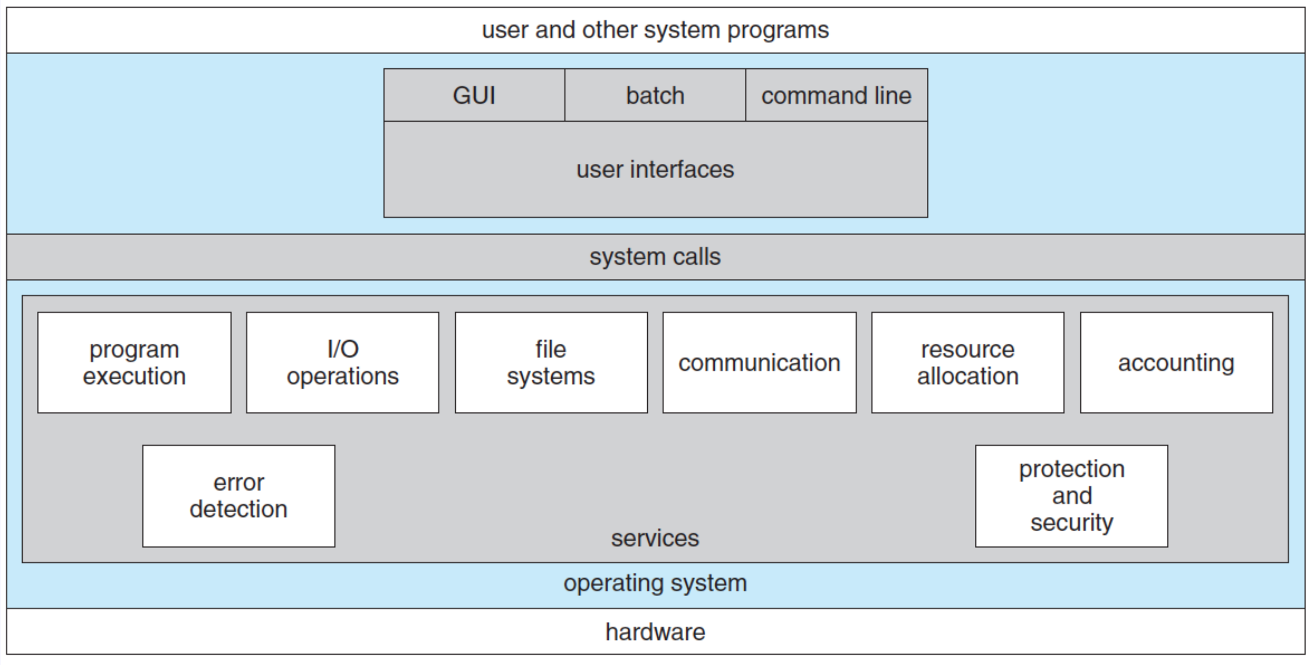
Virtual machine manager : virtual machine management software which can be set to run on a third mode (which requires more mode bits), giving it less power than the kernel but more than the user. Virtualisation does not necessarily require its own privilege level

Mode bits : reserved bits in the hardware which signify which mode we're in. Typically 0 = user mode and 1 = kernel mode. Always set

before passing control to the user program.

File : abstract memory concept which is mapped to physical storage space. Each file may contain absolutely any data. The OS is responsible for creating/removing files, creating removing directories to organise files, suporting primitives for manipulating files and directories, mapping files onto secondary storage, backing up files on stable (nonvolatile) storage media.

OS Services



User Interface : can appear in many different forms, CLI, GUI or batch - where commands and directives run directly from files.

Program execution : the OS loads programs into memory and runs their instructions, then halts them.

IO Operations : interactions with external devices such as the keyboard or mouse.

File-system manipulation : search through files and directories, creation and deletion of files, permission management.

Communications : facilities for exchanging information between different processes on the same computer or via network. I.e. **Shared memory** or perhaps **message passing**.

Error detection : the OS needs to be aware of errors which occur and correct them as they appear. These can happen anywhere in the system, including hardware and software.

Resource allocation : distribution of resources available to different jobs and users at the same time efficiently.

Accounting : keeping track of who used what resources for either economic purposes or analytics.

Protection and security : all data needs to be securely stored and only available to the users who have the correct perissions. Several processes cannot interfere with each other or harm the system.

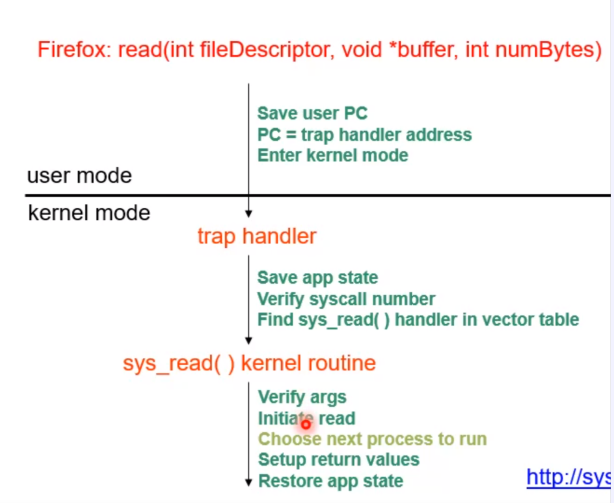
Command interpreter : allows the user to directly interface with the system via commands either in the form of a GUI or CLI or in other forms. When multiple are available these are shown as **shells**. The commands themselves may be stored by the shell, or the shell might simply direct the appropriate loading of file-stored directives which run the appropriate commands (i.e. PATH resolution)

System calls : calls to the OS to develop certain specific functions such as opening files or starting sub-processes. Many OS use API's on top of system calls to make portability more achievable.

System-call interface : the interface provided by programming languages to interface with the system calls in different OS' which the compiler knows the specifics of.

Syscalls

The user cannot perform IO operations by himself, he must ask the OS to do it for them.
Syscalls define procedures which the OS performs for the user in privileged mode.
Usually implemented as vectors, where each syscall is simply an offeset to a base address.
Mechanically just a procedure call (but is not one! in a normal procedure call the caller knows the location of the procedure, in this case a syscall is just an ID), the caller puts arguments in a place the callee expects, then retrieves output from known place. Usually each system call will have wrapper functions provided by each programming language, i.e. the **system call interface**, which the compiler understands.



Firefox: read(int fileDescriptor, void *buffer, int numBytes)

Save user PC
PC = trap handler address
Enter kernel mode

user mode

kernel mode

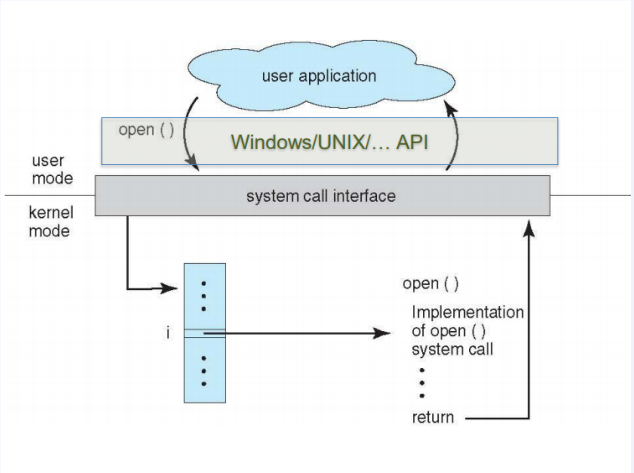
trap handler

Save app state
Verify syscall number
Find sys_read( ) handler in vector table

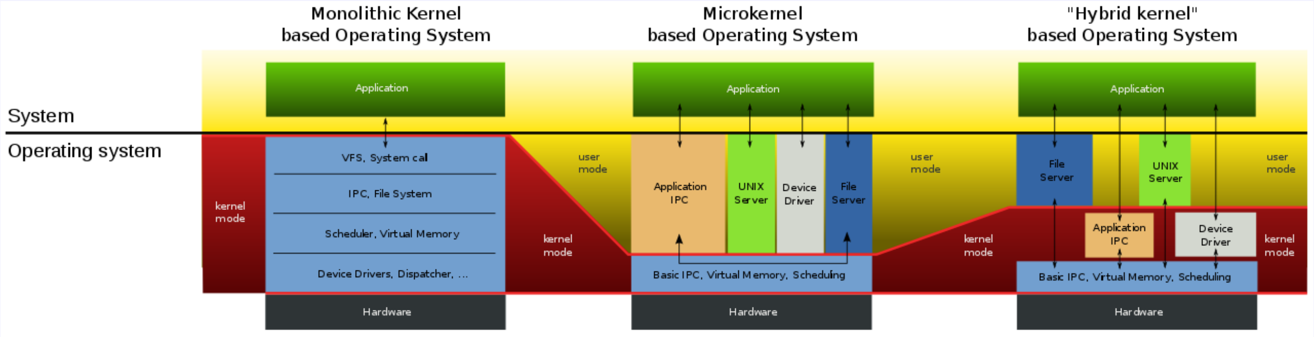
sys_read( ) kernel routine

Verify args
Initiate read
Choose next process to run
Setup return values
Restore app state

http://sys



OS Structures



Monolithic Kernel

All major subsystems implemented in kernel.

+ low system interaction cost (procedure call)
- hard to understand and modify
- unrelible (no isolation between system modules)
- hard to maintain

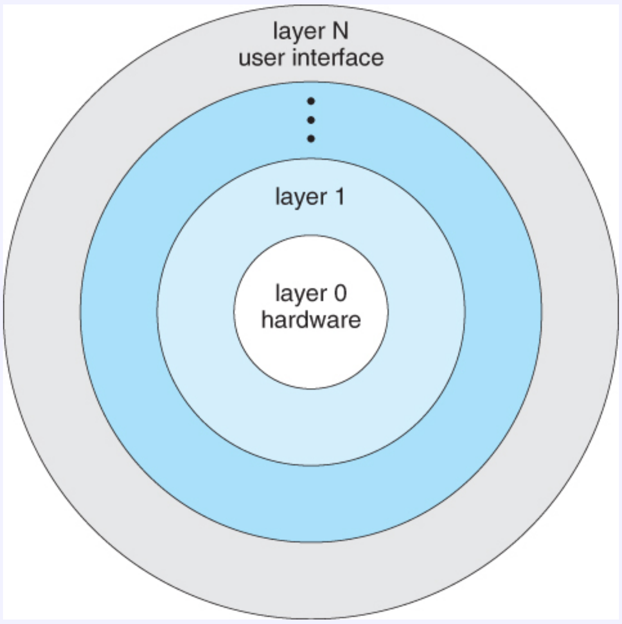
Microkernel

Minimize what goes into kernel, implement rest of OS as user-level processes

+ better reliability (isolation between components)
+ ease of extension and customization (easy to replace parts)
- poor performance (a lot of user/kernel switches)

Layered Kernel

Implement OS as a set of layers, each layer interacts only with layer below.



+ more reliable (separation of components)
- strict layering isn't flexible enough - in real life modules might need to communicate with not only nearby layers.
- poor performance, each layer crossing has overhead associated with it (due to API generalization)
- Disjunction between model and reality - system modelled as layers, but not really built that way

Dynamically loadable kernel modules

Core services in the kernel, others dynamically loaded.

Common in modern implementations:

- **Monolithic**: load the code in kernel space (Solaris, Linux, etc.)
- **Microkernel**: load the code in user space (any)

+ Convenient: no need for rebooting for newly added modules
+ Efficient: no need for message passing unlike microkernel
+ Flexible: any module can call any other module unlike layered model

- Memory fragmentation: fragments OS memory which is normally unfragmented when loaded initially

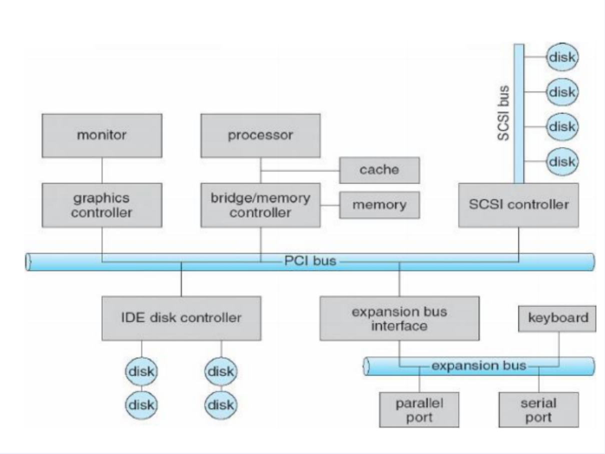
Hybrid OS Design

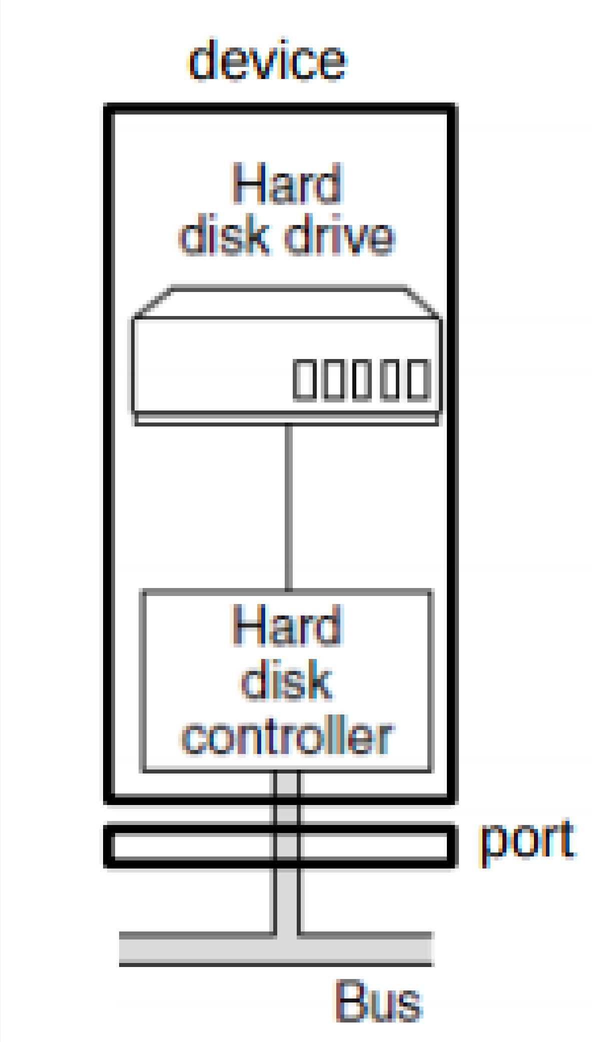
Many different approaches. Key idea: exploit the benefits of monolithic and microkernel designs. Extensibility via kernel modules.

I/O

Variety of I/O Devices

Port : Connection point for a device (e.g., USB, parallel, serial, ethernet)
Bus : Peripheral buses (e.g. PCI/PCIe), Expansion bus connects relatively slow devices
Device
Controller(host adapter) : electronics that operate port, bus, device (sometimes integrated). Contains processor, microcode, private memory, bus controller etc.



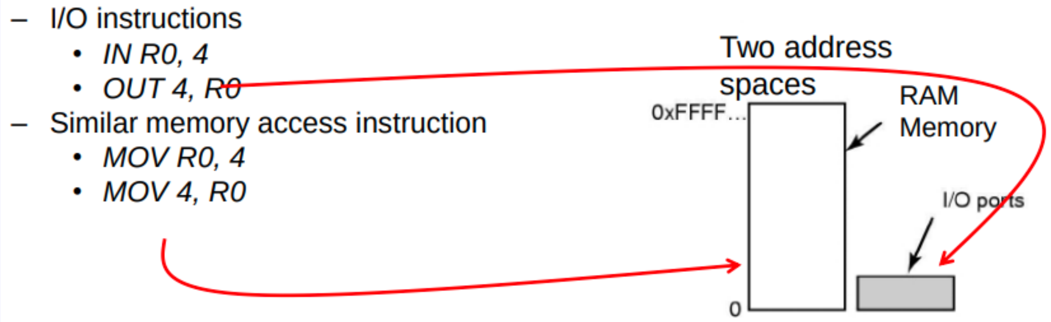


Controllers have **registers** for data and control as well as **buffers**, mostly for data. Communication Methods:

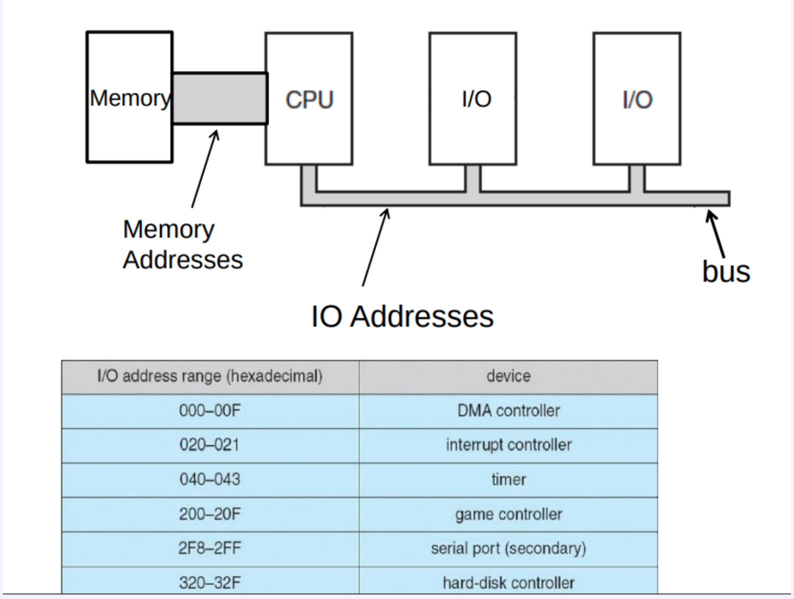
- **IO Ports**
- **Memory-mapped IO**
- Hybrid

I/O Ports

- Each control register has an I/O port number
- Special instructions exist to access the I/O port space
- CPU reads in from device I/O PORT to CPU register (IN REG, PORT)
- CPU writes to device I/O PORT from CPU register (OUT PORT, REG)
- Instructions are privileged (OS kernel only)
- Seperate I/O Port space and memory space:



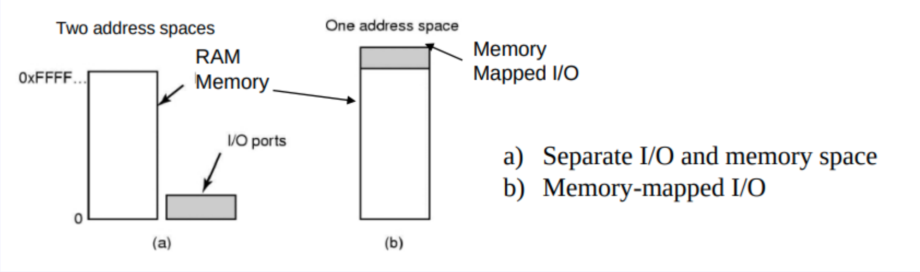
- I/O instructions
  - *IN R0, 4*
  - *OUT 4, R0*
- Similar memory access instruction
  - *MOV R0, 4*
  - *MOV 4, R0*

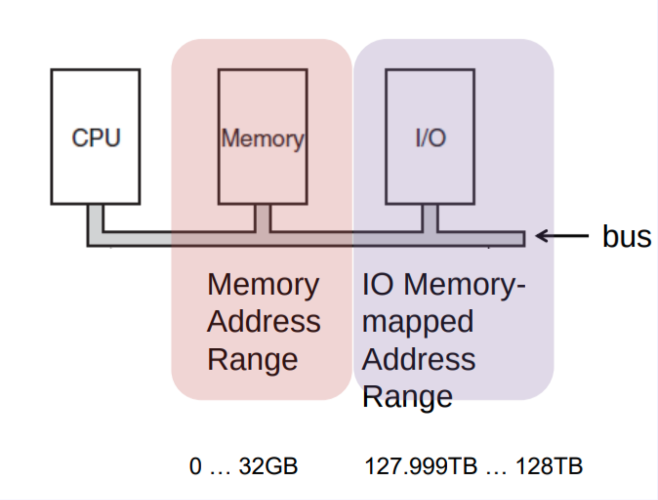


I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller

Memory Mapped I/O

- All control registers and buffers mapped into the memory space
- Each control register is assigned a unique memory address
- There is no actual RAM memory for that address
- Such addresses may be at the top of the physical address space





Hybrid I/O

Simply do both, use memory mapped I/O for the **data buffers**, and keep separate I/O ports for **control registers**.

Offloaded Communication

The CPU can request data from an I/O controller one byte at at time (Programmed IO). This wastes CPU time for large data transfers, small data transfers are ok.
CPU can instead offload data transfers using DMA:

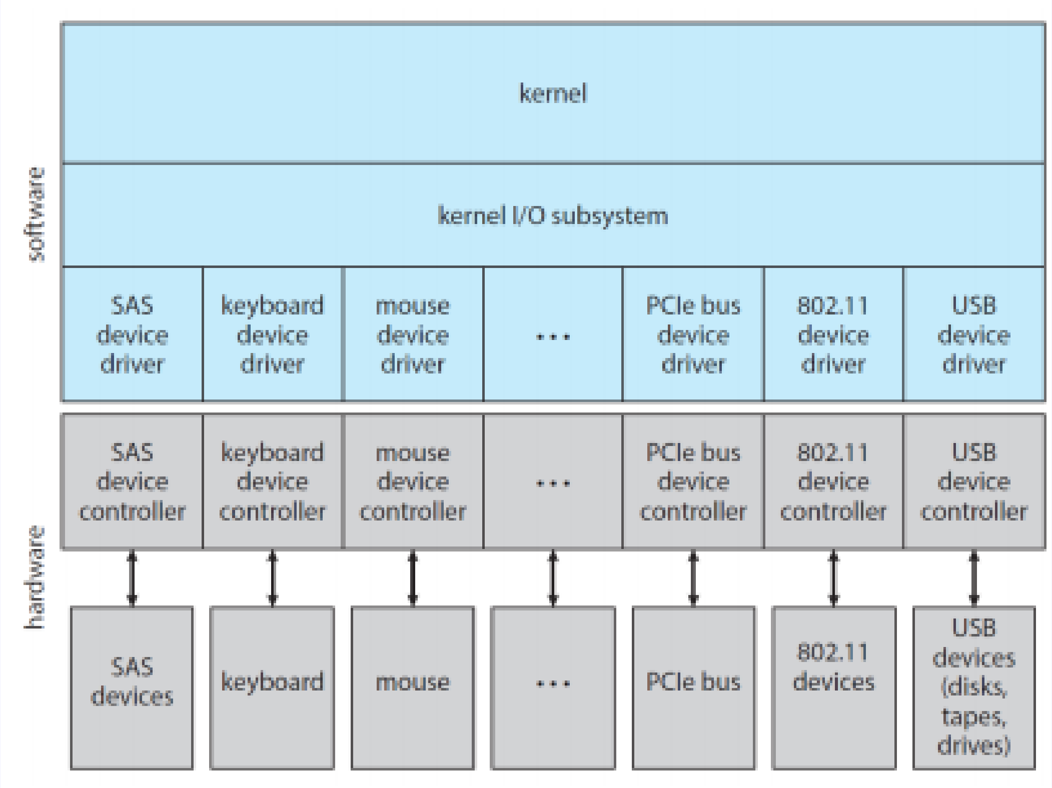
DMA (Direct Memory Access) controller transfers data fro the CPU, either from/to IO or between IO devices.

This requires a DMA controller either on the device's host controller, and the motherboard. This controller contains registers to be read/written by the software:

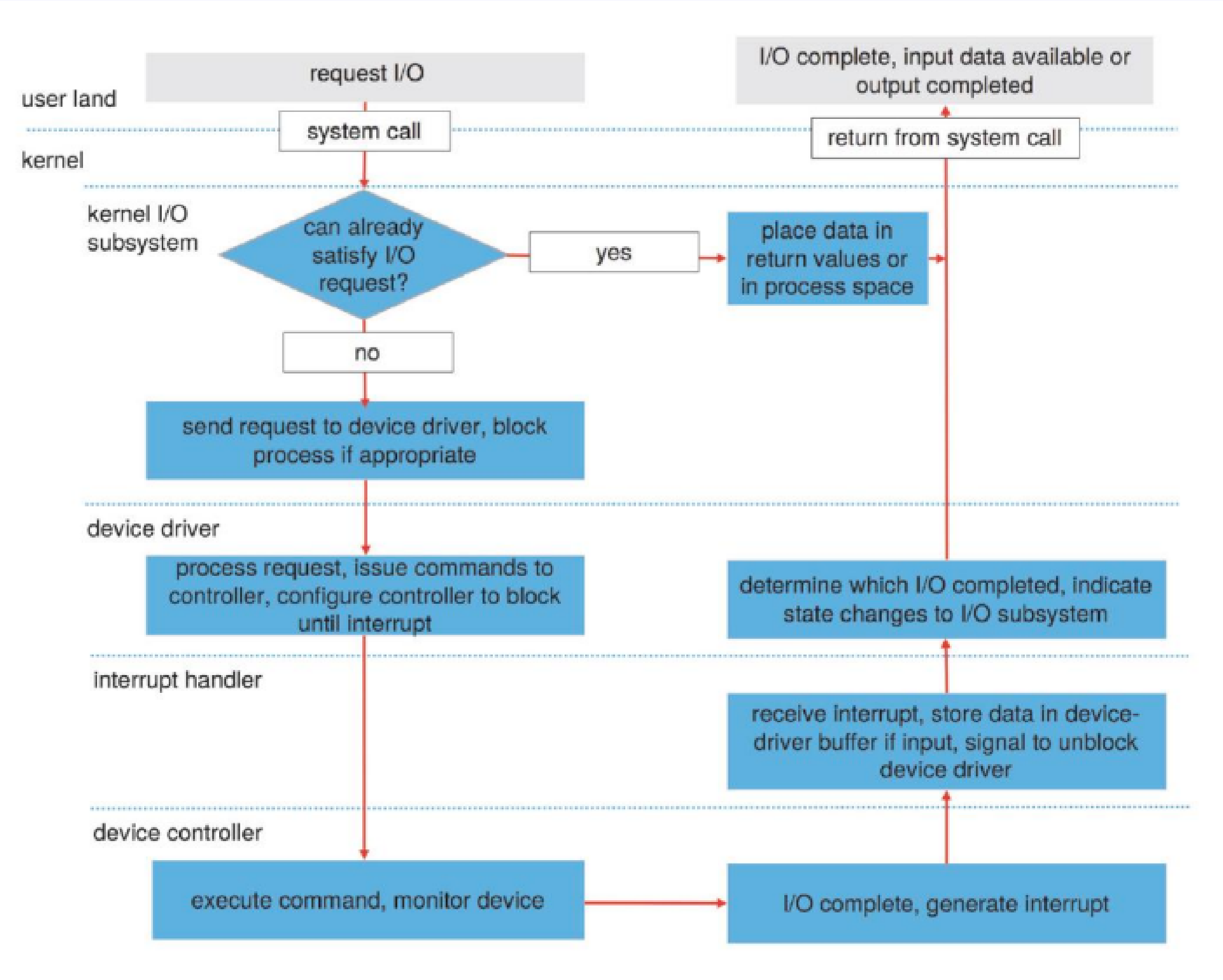
- Memory address register
- byte count register
- Control registers: direction, unit, byte burst size etc..

OS Device drivers

Great variety of devices, each one has very different specs.
OS Deals with IO devices in a standard and uniform way. Each type of driver is an interface which the vendor can implement as a class. Each OS has its own standard.



Life cycle of an IO request


user land
kernel
request I/O
system call
I/O complete, input data available or output completed
return from system call
kernel I/O subsystem
can already satisfy I/O request?
yes
place data in return values or in process space
no
send request to device driver, block process if appropriate
device driver
process request, issue commands to controller, configure controller to block until interrupt
determine which I/O completed, indicate state changes to I/O subsystem
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
device controller
execute command, monitor device
I/O complete, generate interrupt

Processes

Process

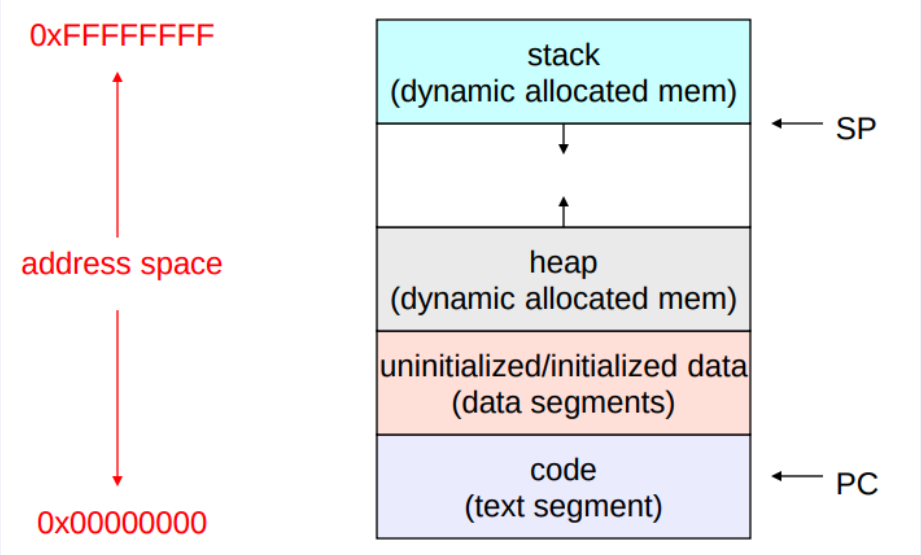
Process is the OS's abstraction for execution.

- Program is the list of instructions, initialized data, etc
- A process is a **program in execution**
- A single flow/sequence of instruction in execution
- An address space (an abstraction of the CPU)

Only one process can be running on a processor core at any instant
Contents:

- An address space, containing:
  - Code (instructions) for running program
  - Data for the running program (static data, heap data, stack)
- A CPU state, consisting of
  - Program counter, indicating the next instruction
  - Stack pointer, current stack position
  - Other general-purpose register values
- A set of OS resources
  - Open files
  - network connections
  - sounc channels

I.e. everything needed to run the program



Each process is identified by a process ID (**PID**). PID's are unique and global. With certain exceptions (cgroups)

Operations that create processes return a PID, and those which operate on processes accept PID's as arguments

Process representation

Each process is represented internally by the OS with a **Process control block (PCB)** or process/task descriptor, identified by the PID.
OS keeps all of a process's execution state in (or linked from) the PCB when the process isn't running:

- PC, SP, registers etc.
- when a process execution is stopped, its state is transferred out of the hardware into the PCB

When the process is running its state is spread between the PCB and the hardware (CPU regs)
PCB's contain:

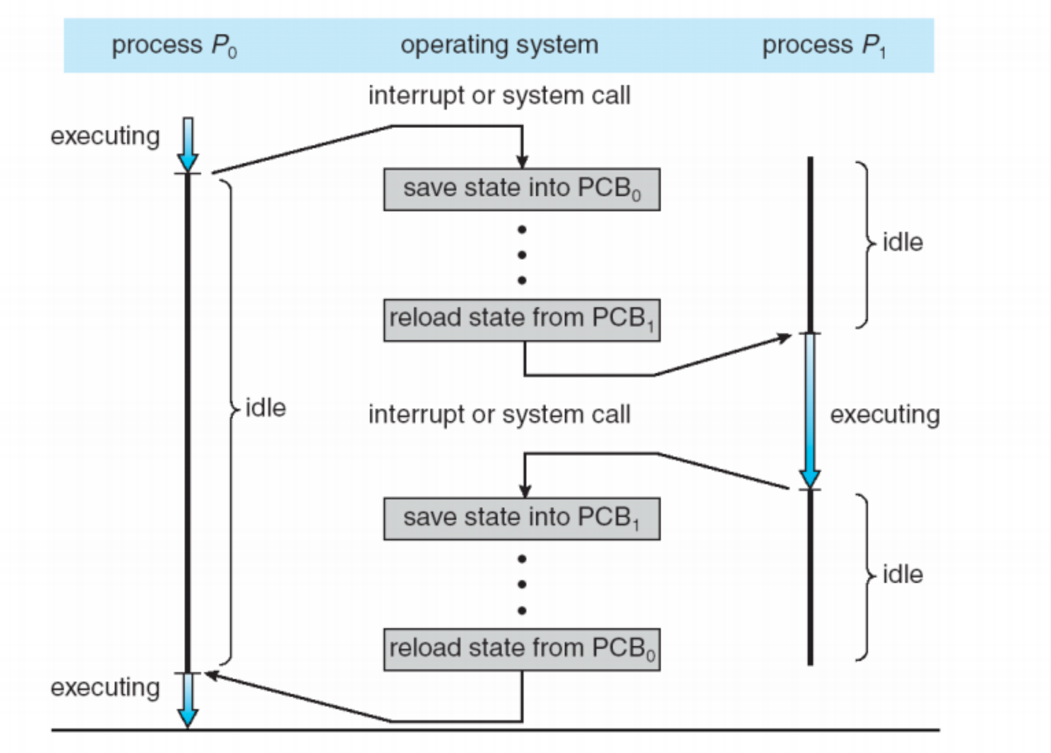
- Process ID (PID)
- Parent process ID
- Execution state
- PC, SP, registers
- Address space info
- UNIX user ID, group ID
- Scheduling priority
- Accounting info
- Pointers for state queues

and likely many more.
Whenever the OS gets control becase of :

- Syscalls
- Exceptions
- Interrupts

The OS then saves the CPU state into the PCB. Whenever the process is resumed into execution again, its PCB is loaded onto the machine registers SP, PC etc.
This is called a **Context switch**

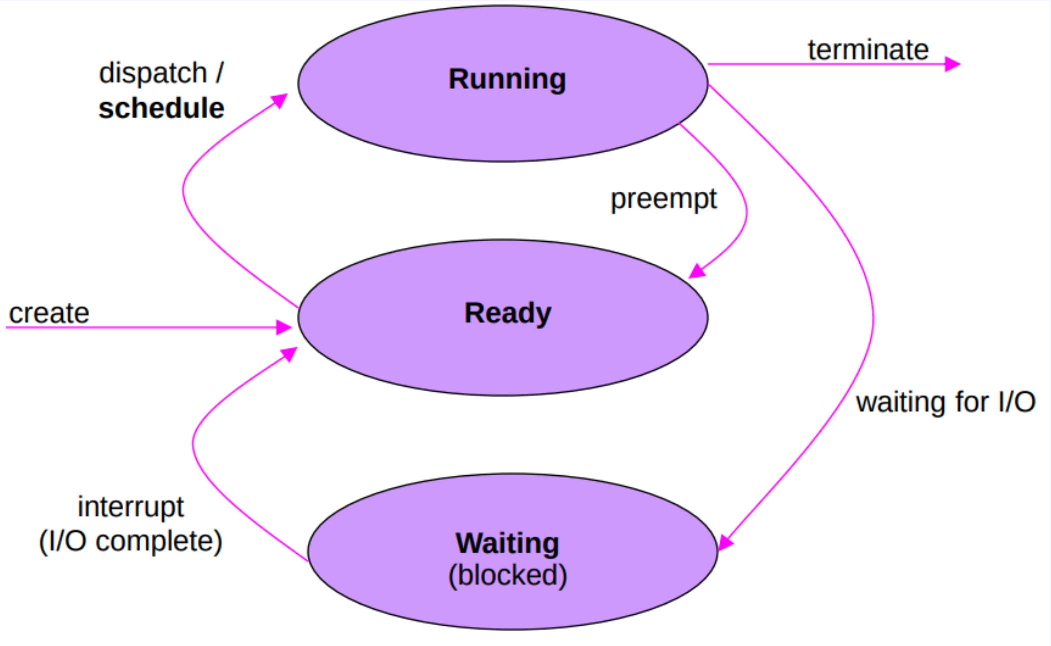
Context Switch



Execution states

Each process has an **Execution state**, which indicates what it's currently doing.

- **Ready**: waiting to be assigned to a CPU
- **Running**: executing on a CPU
- **Waiting**: Waiting for an event, e.g. IO completion, or a message from another process.



State queues

The OS maintains a collection of queues, that represent the state of all processes in the system. Typically one queue for each state (executing, waiting etc..) Each PCB is queued onto a state queue according to the current state of the process it represents. As a process changes state, its PCB is unlinked from one queue, and then linked onto another.

There may be many wait queues, one for each type of wait (specific device, timer, message) etc.

Process creation

New processes are created by existing processes (parent-child)
The first process is started by the OS, everything else stems from it (init in linux)

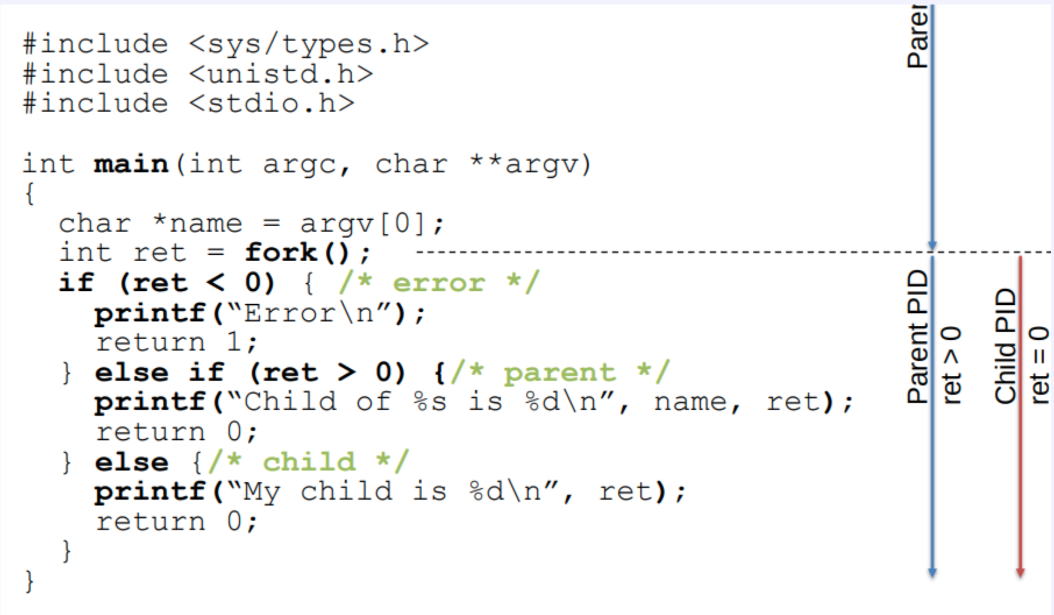
Depending on OS, child processes inherit certain attributes of parent, (i.e. open file table: implies stdin/stdout/stderr) Some systems divide resources of parent between children.

UNIX - fork()

On UNIX systems, process creation is done through the fork() system call:

- Creates and initializes a new PCB
- Initializes kernel resources of new process with resources of parent (e.g. open files)
- Initializes PC,SP to be same as parent
- Creates a new address space, which is an identical copy of this of the parent's (by value)

The fork call returns "twice" once in the parent, and once in the child. In the child the PID returned is 0 and in the parent, the child's PID is returned.

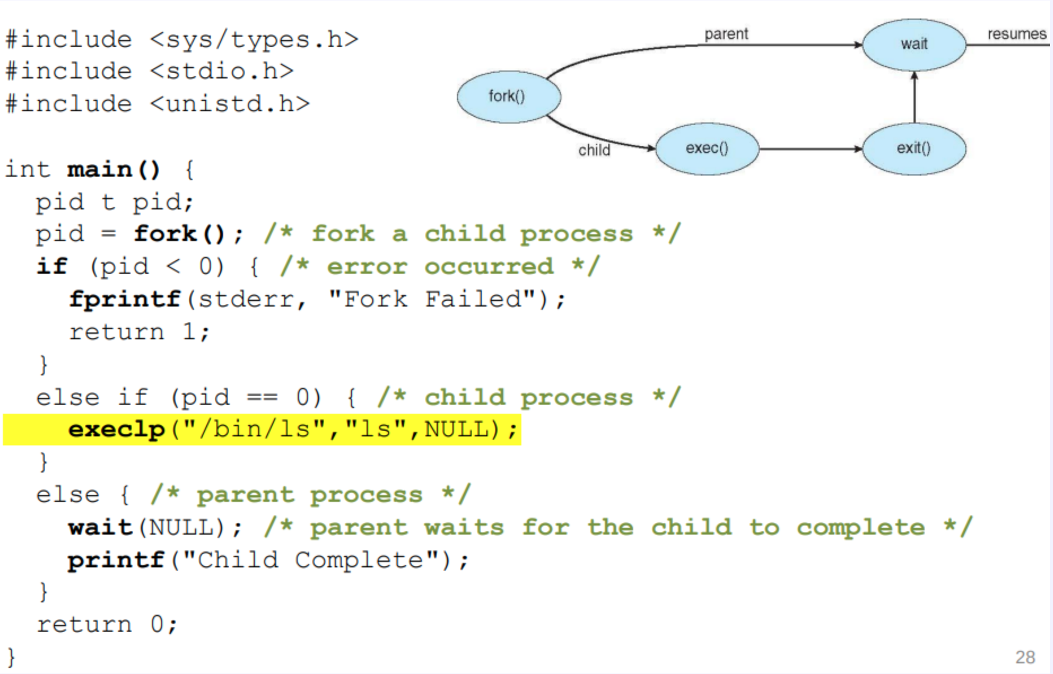


```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  char *name = argv[0];
  int ret = fork();
  if (ret < 0) { /* error */
    printf(“Error\n”);
    return 1;
  } else if (ret > 0) {/* parent */
    printf(“Child of %s is %d\n”, name, ret);
    return 0;
  } else {/* child */
    printf(“My child is %d\n”, ret);
    return 0;
  }
}
```

UNIX - exec()

In order to start a new program instead of just copying the old one, we must use exec(). Which is the call which stops the current process, loads a new program into the address space, initializes the hardware context and args for the new program and finally places the PCB onto the ready queue



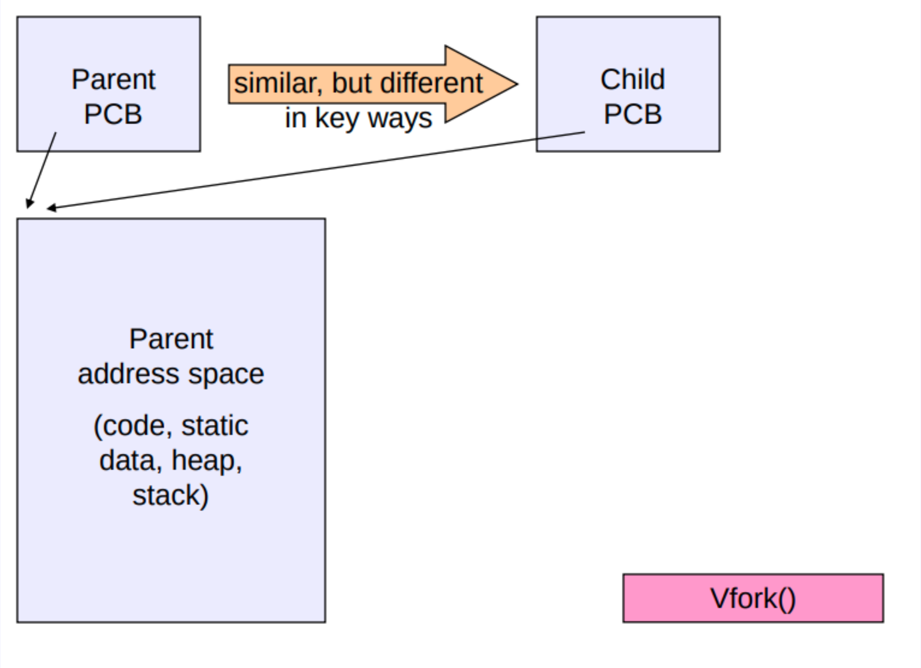
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  pid t pid;
  pid = fork(); /* fork a child process */
  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  else if (pid == 0) { /* child process */
    execlp("/bin/ls","ls",NULL);
  }
  else { /* parent process */
    wait(NULL); /* parent waits for the child to complete */
    printf("Child Complete");
  }
  return 0;
}
```

Alternatively use vfork() which is faster but less safe

UNIX - vfork()

Same as fork, but hte child's address space **IS** by address the same space as the parents.
Usage relies on the child not modifying the address space before doing an execve() call, otherwise bad things can happen.



Parent PCB
similar, but different in key ways
Child PCB
Parent address space
(code, static data, heap, stack)
Vfork()

Copy-on-write (COW) fork()

Retains original semantics, but copies ”only what is necessary” rather than the entire address space.
On fork():

- create a new address space
- initialize page tables with same mappings as parents (identical)
- Set both parent and child page tables to make all pages read-only
- if either parent or child writes to memory, an exception occurs
- On exception, OS copies the page, adjusts page tables, etc..

Interprocess communication

Shared memory

Allow processes to communicate and synchronize:

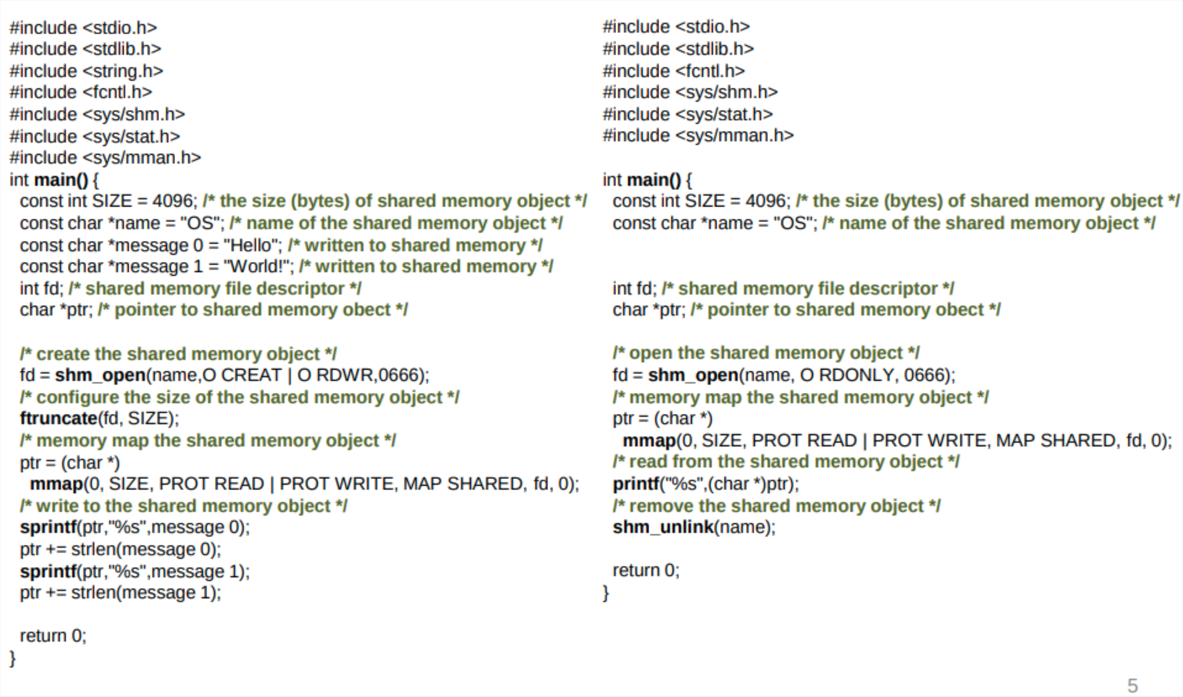
- Sharing part of address space
- OS doesn’t mediate communication (no overhead)
- Usually OS prevents processes from accessing each other’s memory
- Processes should agree to void this restriction

Data:

- Format decided by application
- Direct access (not mediated by the OS) - very fast
- Application programmer fully manages the data transfer - not trivial

Possible use cases:

- Passing of large (single) objects (image)
- Notification variable



```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
int main() {
  const int SIZE = 4096; /* the size (bytes) of shared memory object */
  const char *name = "OS"; /* name of the shared memory object */
  const char *message 0 = "Hello"; /* written to shared memory */
  const char *message 1 = "World!"; /* written to shared memory */
  int fd; /* shared memory file descriptor */
  char *ptr; /* pointer to shared memory obect */

  /* create the shared memory object */
  fd = shm_open(name,O CREAT | O RDWR,0666);
  /* configure the size of the shared memory object */
  ftruncate(fd, SIZE);
  /* memory map the shared memory object */
  ptr = (char *)
    mmap(0, SIZE, PROT READ | PROT WRITE, MAP SHARED, fd, 0);
  /* write to the shared memory object */
  sprintf(ptr,"%s",message 0);
  ptr += strlen(message 0);
  sprintf(ptr,"%s",message 1);
  ptr += strlen(message 1);

  return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main() {
  const int SIZE = 4096; /* the size (bytes) of shared memory object */
  const char *name = "OS"; /* name of the shared memory object */


  int fd; /* shared memory file descriptor */
  char *ptr; /* pointer to shared memory obect */

  /* open the shared memory object */
  fd = shm_open(name, O RDONLY, 0666);
  /* memory map the shared memory object */
  ptr = (char *)
    mmap(0, SIZE, PROT READ | PROT WRITE, MAP SHARED, fd, 0);
  /* read from the shared memory object */
  printf("%s",(char *)ptr);
  /* remove the shared memory object */
  shm_unlink(name);

  return 0;
}
```

5

Message Passing

Allow processes to communicate and synchronize:

- Without sharing part of address space
- OS mediates communication (overhead likely)

Works with processes on the same machine and also those on different inter-networked machines! This is not possible with shared memory.
Message passing facility provides at least two operations:

- Send(message)
- Receive(message)

Communication link:

- Several implementation tradeoffs, .e.g. messages size
- Fixed
- Variable

Communicating processes must refer to each other:

- Direct communication:
  - Symmetric: explicit name of sender and receiver
    - send(P, message) - send message to P
    - receive(P, message) - receive message from P
  - Assymetric: Explicit at least on one end:
    - send(P, message) - send to p
    - receive(id, message) - receive from any process, sender saved in id
- Indirect communication:
  - No need to know/explicitly in advance sender and/or receiver
  - Mailboxes (e.g., POSIX mailbox)
    - send(A, message) - send message to mailbox A
    - receive(A, message) - receive message from mailbox A
  - A mailbox can be accesed by more than two processes
  - Multiple mailboxes might exist between processes

send() and receive() calls might be implemented as **blocking** or **synchronous** as well as **nonblocking** or **asynchronous**. different combinations of these might be offered:

- Blocking send
- Nonblocking send
- Blocking receive
- Nonblocking receive

Rendezvous - When both send and receive are blocking

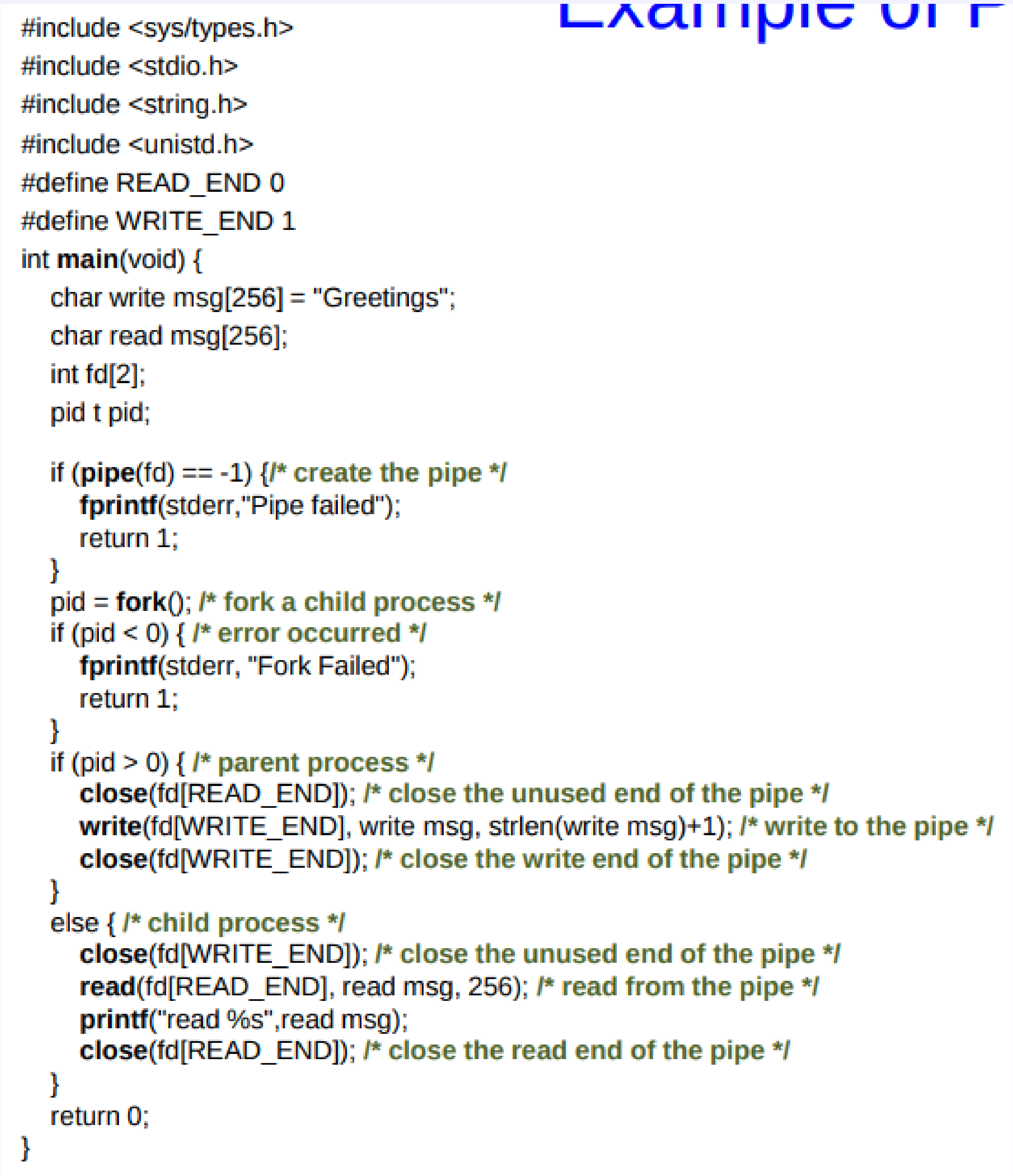
Buffering

Messages exchanged reside in temporary buffers/queues with either:

- Zero capacity (no buffering) - no message waiting, sender must block until recipient receives message
- Bounded capacity - n messages may reside. If the queue is not full, then nonblocking, otherwise blocking
- Unbounded - never blocks

Example implementation: Pipes

A pipe acts as a conduit allowing two processes to communicate **one-way** only and either **Annonymously** or **Named**.



```
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define READ_END 0
#define WRITE_END 1
int main(void) {
   char write msg[256] = "Greetings";
   char read msg[256];
   int fd[2];
   pid t pid;

   if (pipe(fd) == -1) {/* create the pipe */
      fprintf(stderr,"Pipe failed");
      return 1;
   }
   pid = fork(); /* fork a child process */
   if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   if (pid > 0) { /* parent process */
      close(fd[READ_END]); /* close the unused end of the pipe */
      write(fd[WRITE_END], write msg, strlen(write msg)+1); /* write to the pipe */
      close(fd[WRITE_END]); /* close the write end of the pipe */
   }
   else { /* child process */
      close(fd[WRITE_END]); /* close the unused end of the pipe */
      read(fd[READ_END], read msg, 256); /* read from the pipe */
      printf("read %s",read msg);
      close(fd[READ_END]); /* close the read end of the pipe */
   }
   return 0;
}
```

Mechanically act like file descriptors (streams)

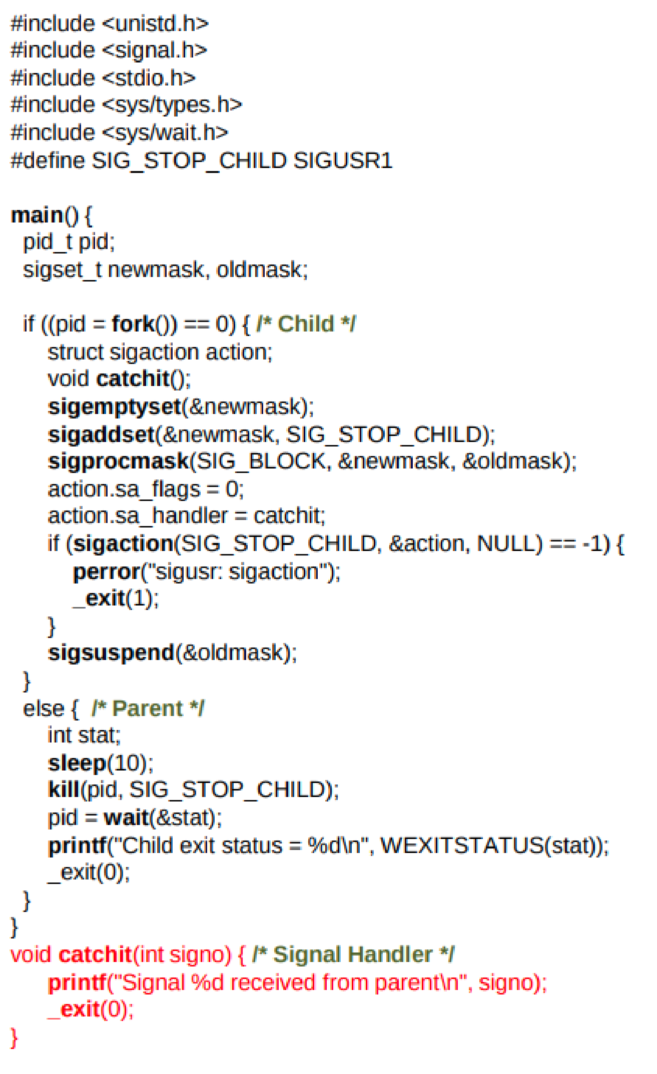
Client-Server communication

- **Sockets** abstraction
  - endpoint for communication
  - identified by an IP address concatenated with a port number
  - servers implementing specific services (SSH, FTP, HTTP) listen to well-known ports
  - an SSH server listens to port 22, an FTP server listens to port 21, a web or http server listens to port 80
- **Remove procedure call** (RPC)
  - Abstract the procedure-call mechanism
  - for use between systems with network connections
  - similar in many respects ot the IPC
  - uses message-based communication to provide remote service

Signals

- OS mechanism to notify a process (one way)
- From the OS POV can be thought as a software-generated interruption/exception (synchronous or asynchronous)
- From other processes POV, it is only a notification, no data is transferred. A communication method for: management, synchronization etc.

UNIX: signal handlers must be registered, and provide code for handling the signal.



```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#define SIG_STOP_CHILD SIGUSR1

main() {
  pid_t pid;
  sigset_t newmask, oldmask;

  if ((pid = fork()) == 0) { /* Child */
      struct sigaction action;
      void catchit();
      sigemptyset(&newmask);
      sigaddset(&newmask, SIG_STOP_CHILD);
      sigprocmask(SIG_BLOCK, &newmask, &oldmask);
      action.sa_flags = 0;
      action.sa_handler = catchit;
      if (sigaction(SIG_STOP_CHILD, &action, NULL) == -1) {
          perror("sigusr: sigaction");
          _exit(1);
      }
      sigsuspend(&oldmask);
  }
  else {  /* Parent */
      int stat;
      sleep(10);
      kill(pid, SIG_STOP_CHILD);
      pid = wait(&stat);
      printf("Child exit status = %d\n", WEXITSTATUS(stat));
      _exit(0);
  }
}
void catchit(int signo) { /* Signal Handler */
      printf("Signal %d received from parent\n", signo);
      _exit(0);
}
```