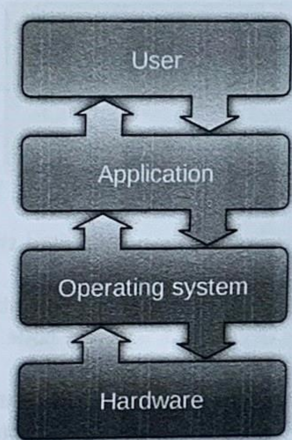
13

OS SECURITY:
OS CONCEPTS

Operating systems

- An OS provides the interface between the users of a computer and that computer's hardware.
- The OS handles the management of low-level hardware resources:
  - disk drives,
  - CPU,
  - RAM,
  - I/O devices, and
  - network interfaces

OS provides a lvl. of abstraction for developers to write programs w/o having to handle low-lvl. details



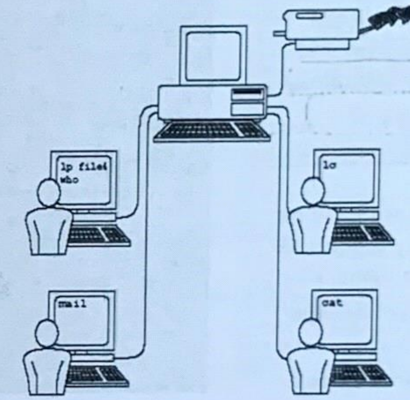
2 REASONS WHY WE NEED OS SECURITY:

1. Multi-users

OSes must allow for multiple users with potentially different levels of access to the same computer.

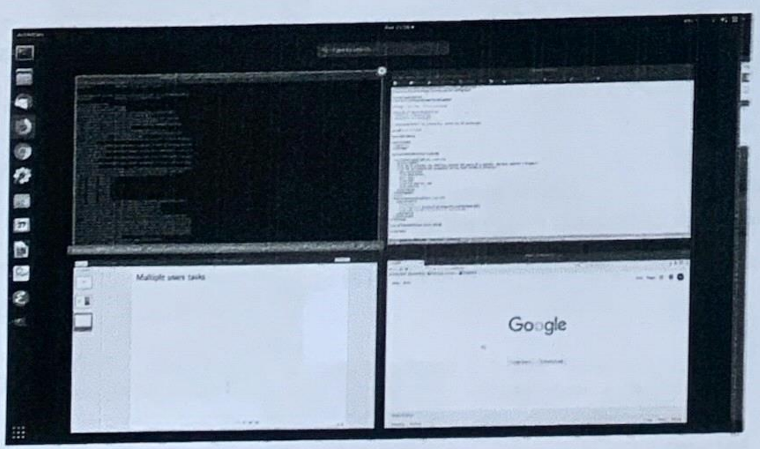
Hence...

The OS needs to have a mechanism to isolate users from one another



Multi-tasking 2.

OSes must allow multiple application programs to run at the same time.



Hence the OS needs to ensure tht. the execution of one process must not be able to alter the exec. of another process

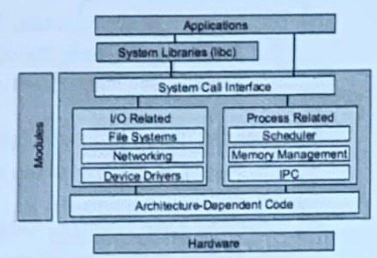
Essential Unix architecture

Execution modes:

- User mode - access to resources through syscall to kernel
- Kernel mode - direct access to resources

System calls are usually contained in a collection of programs, *eg.* a library such as the C library `libc`

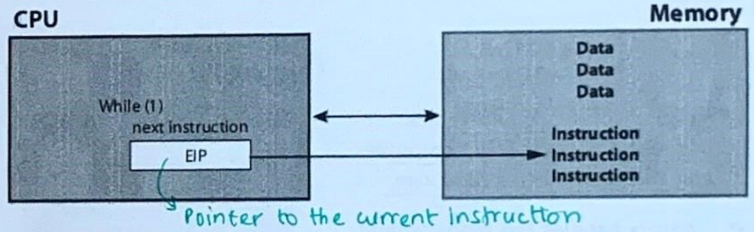
allow apps. to request tht the kernel performs actions on their behalf



Processes and process management

- A process is an instance of a program that is currently executing.
- To actually be executed the program must be loaded into RAM and uniquely identified.
- Each process running is identified by a unique process ID (pid).
- To a pid, we can associate its CPU time, memory usage, user ID (uid), program name, etc.
- A process might control other processes (fork).
- Child process inherits context from parent process.

x86 CPU/Memory



- To actually be executed the program must be loaded into RAM and uniquely identified
- The RAM memory allocated to a process is its address space
- It contains both the code for the running program, its input data, and its working memory
- Memory stores instructions and data
- CPU interprets instructions
- %eip points to next instruction
- %eip incremented after each instruction
- %eip modified by call, ret, jmp, and conditional jmp

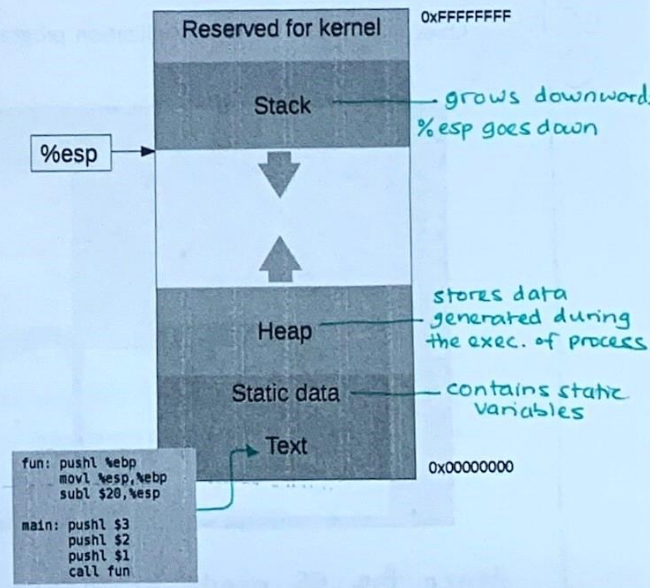
x86 registers

- Temporary registers: %eax, %ebx, %ecx, %edx, %edi, %esi
  - These registers are like variables built in the processor
  - Most of the instructions perform on these registers
- Extended stack pointer: %esp
  - Points at the top of the stack
- Extended base pointer: %ebp
  - Points to the base of the stack frame of the current function call

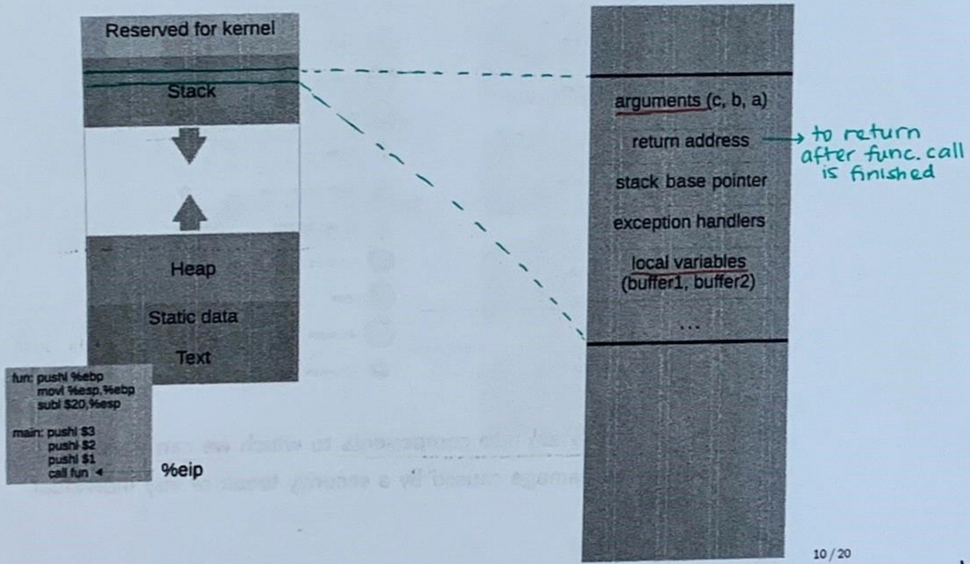
x86 process memory layout (simplified)

ADDRESS SPACE OF A PROCESS: divided into 5 segments

```
void function(int a, int b, int c) {
    char buffer1[5];
    char buffer2[10];
}
void main() {
    function(1,2,3);
}
```



Stack frame



x86 runtime memory - an example

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

Stack frame

		...	
SF_{main}	0xbfffeffc :	0xb7e31a83	$@ret_0$ → return address
	0xbfffeff8 :	0x00000000	$\%ebp_0$ → ebp
	0xbfffeff4 :	0x00000000	→ local variable $x = 0$
$SF_{fact(4)}$	0xbfffefd0 :	0x00000004	→ argument
	0xbfffefcc :	0x08048474	$@ret_m$ → return address
	0xbfffefc8 :	0xbfffeff8	$\%ebp_m$ → ebp of prev. stack frame
	0xbfffefbc :	0x00000001	→ local variable $y = 1$
$SF_{fact(3)}$	0xbfffefa0 :	0x00000003	
	0xbfffef9c :	0x08048449	$@ret_4$
	0xbfffef98 :	0xbfffefc8	$\%ebp_4$
	0xbfffef8c :	0x00000001	
$SF_{fact(2)}$	0xbfffef70 :	0x00000002	
	0xbfffef6c :	0x08048449	$@ret_3$
	0xbfffef68 :	0xbfffef98	$\%ebp_3$
	0xbfffef5c :	0x00000001	
$SF_{fact(1)}$	0xbfffef40 :	0x00000001	
	0xbfffef3c :	0x08048449	$@ret_2$
	0xbfffef38 :	0xbfffef68	$\%ebp_2$
	0xbfffef2c :	0x00000001	
$SF_{fact(0)}$	0xbfffef10 :	0x00000000	
	0xbfffef0c :	0x08048449	$@ret_1$
	0xbfffef08 :	0xbfffef38	$\%ebp_1$
	0xbfffeefc :	0x00000001	

Stack and functions: Summary

Calling function

1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction to run after control returns
3. Jump to the function's address

Called function

4. Push the old frame pointer onto the stack (%ebp)
5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
6. Push local variables onto the stack

Returning function

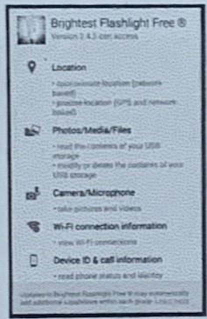
7. Reset the previous stack frame: %esp = %ebp, %ebp = (%ebp)
8. Jump back to return address: %eip = 4(%ebp)

5 CORE SECURITY PRINCIPLES

1. Defence-in-depth

- Security protections built in multiple layers of the system: if one mechanism fails, another steps up immediately behind to thwart attacks
- Firewalls, intrusion detection and protection systems, network segmentation, anti-virus, least privilege, strong passwords, patch management

2. Least privilege

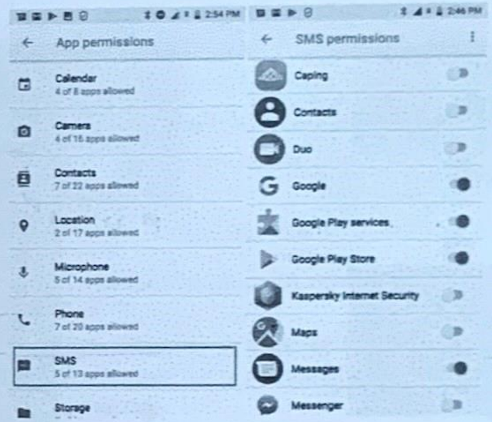


RESTRICT ACCESSES

e.g.

- Users and programs should only access the data and resources required to perform its function
- A torch application does not need access to your location, photos, camera, microphone, wifi, device id, to perform its intended task!

3. Privilege separation



This will allow you to implement the least privilege principle

- Segment the system into components to which we can limit access
- Will limit the damage caused by a security break of any individual component

4. Open design

(Kerckhoff's principle)

- The security of a mechanism should not depend on its secrecy
- The design and implementation details always get leaked (!)

5. Economy of mechanism

- When designing a security mechanism keep it simple!

hence

- It will facilitate the job of security researchers and allow verification
- It will facilitate the task of developers and avoid bugs
- It will facilitate the life of users and avoid misuses