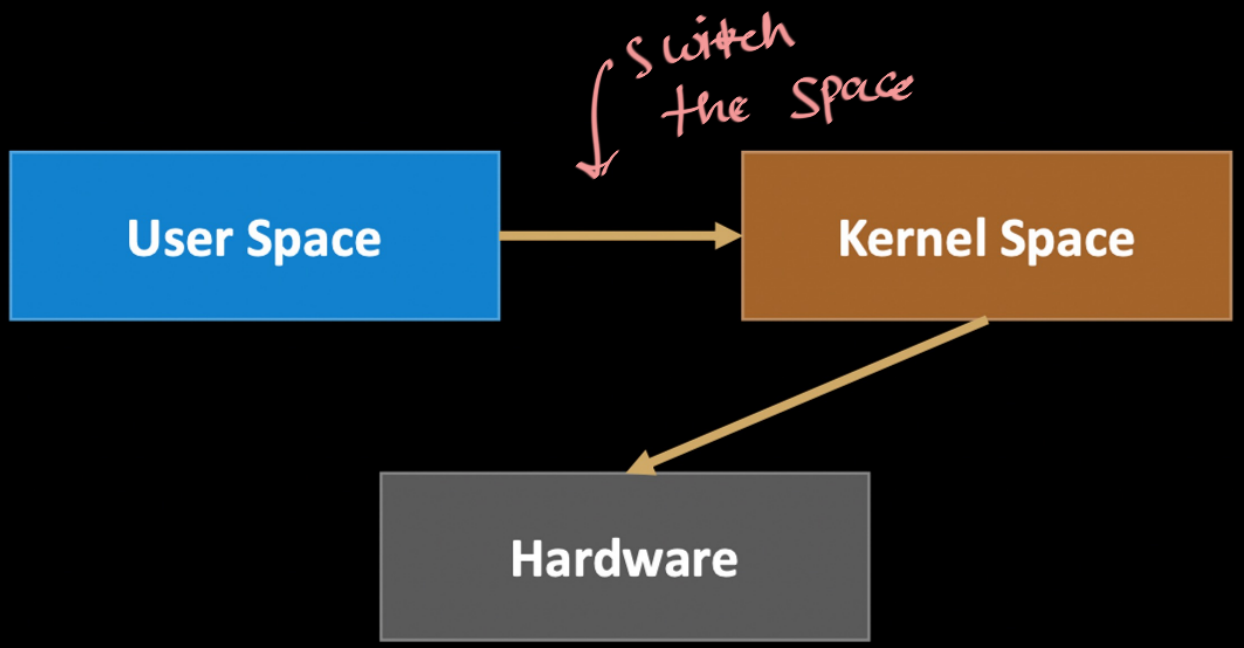
Lecture 5

Topic1: System Call Execution

Topic 2: Threads

Kernel Space vs User Space

- User space
  - Less privileged memory space where user processes execute
- Kernel space
  - Privileged memory space where the OS main process resides
  - No User application should be able to change


Switch the Space
User Space
Kernel Space
Hardware

What is a system call?

What is a system call?

- Various possible answers, from different perspectives

- Various possible answers, from different perspectives
- Answer 1: **request to kernel to perform a service**
  - Open a file
  - Execute a new program
  - Create a new process
  - Send a message to another process

Operating Systems

- Various possible answers, from different perspectives
- Answer 1: **request to kernel to perform a service**
  - Open a file
  - Execute a new program
  - Create a new process
  - Send a message to another process
- Answer 2 (programmer's perspective): "call a function"
  - `fd = open("myfile.txt");`
  - System call **looks like** any other function call

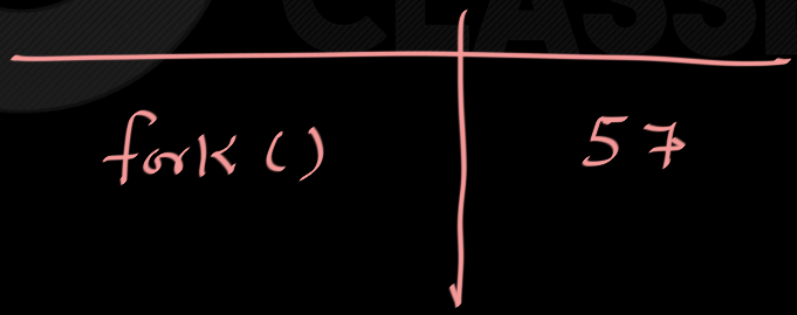
https://man7.org/conf/openfest2015/system_call_tracing_with_strace-OpenFest_2015-Kerrisk.pdf

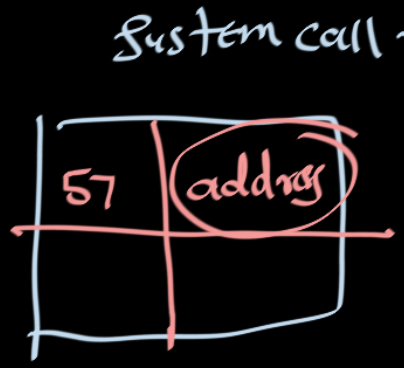
- Answer 3: entry point providing **controlled mechanism to execute kernel code**
- User-space programs **can't** call functions inside kernel
- Syscall = one of few mechanisms by which program can ask to execute kernel code
- Set of system calls is:
  - Operating-system specific
    - Can't run Linux binaries on another OS, and vice versa
  - Limited/strictly defined by OS
    - Linux kernel provides 400+ syscalls

System Call Execution

System Call Execution

- Each system call has a unique numeric identifier
- OS has a system call table that maps numbers to functionality requested

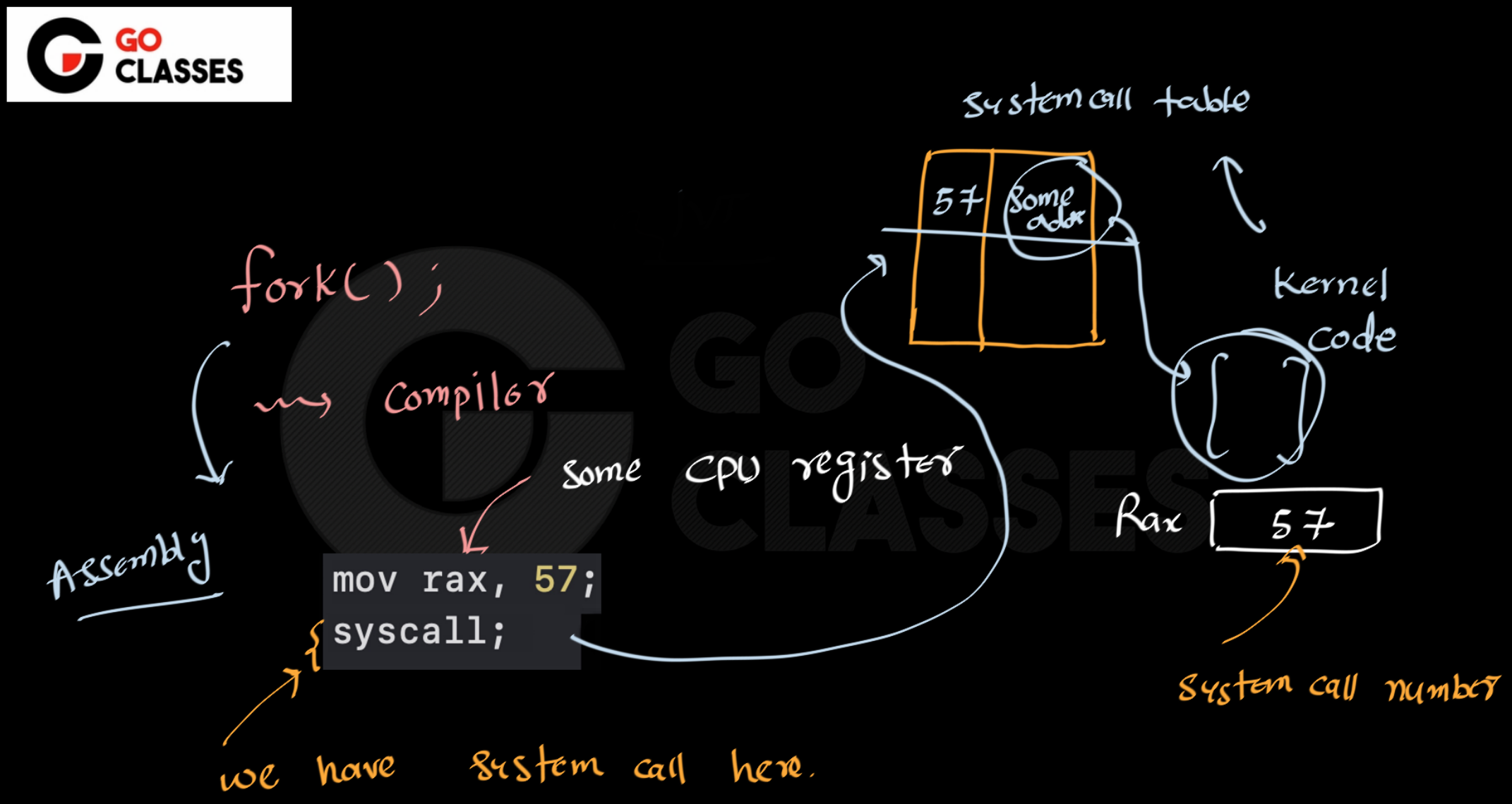




```
; System call numbers are listed in "unistd.h"
```

Here we're calling the "fork();", which creates a new process.

```
mov rax, 57;    the system call number of "fork".
syscall;    Issue the system call
```


GO CLASSES
System call table
57
Some addr
Kernel code
fork();
Compiler
Some CPU register
Rax
57
Assembly
mov rax, 57;
syscall;
we have System call here.
System call number

GO CLASSES

Printf("hello");

we will put in the stack

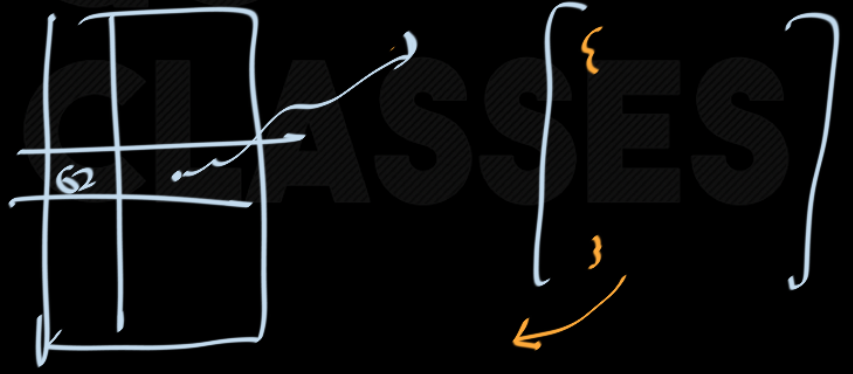
ARGUMENTS

sys_write( )

```
mov rax, 62;
```

Move R1 "hello"

```
syscall;
```



Satyam Naik to Everyone 8:12 PM

SN

store hello in some other register before doing syscall

Put the return value in specific location

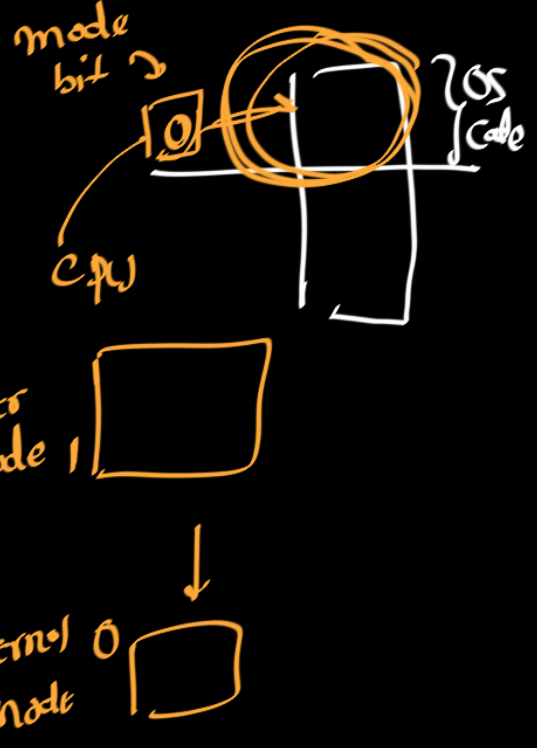
System Call Execution (Contd..)

- Each system call has a unique numeric identifier
- OS has a system call table that maps numbers to functionality requested
- When invoking a system call, user places system call number and associated parameters in an specific register, then executes the syscall instruction

Steps in system call execution

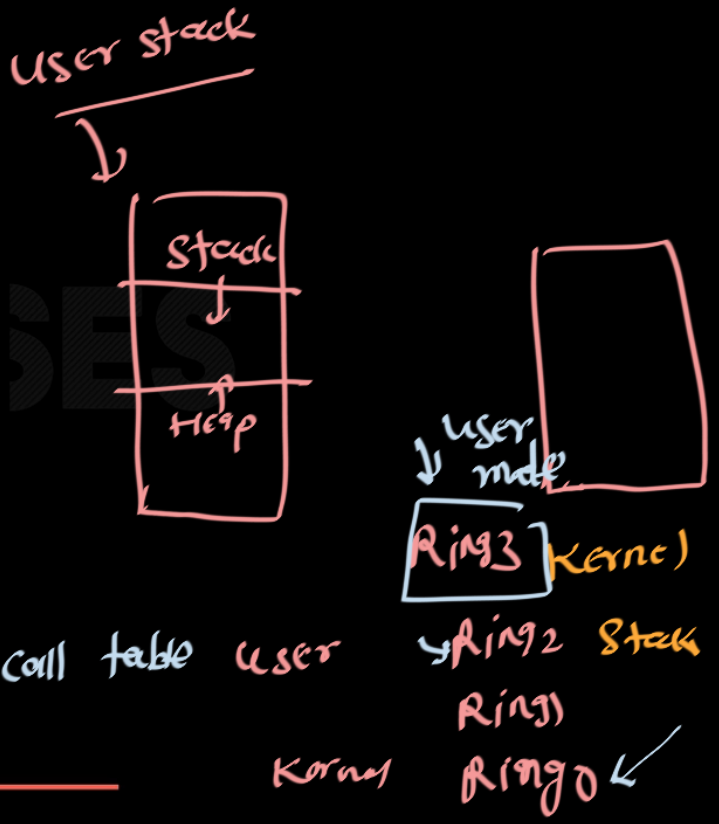
Operating Systems

1. Program calls wrapper function in C library
2. Wrapper function
   - Packages syscall arguments into registers
   - Puts (unique) syscall number into a register
3. Wrapper flips CPU to kernel mode (user-mode ⇒ kernel-mode)
   - Execute special machine instruction (e.g., syscall)
   - Main effect: CPU can now touch memory marked as accessible only in kernel mode
4. Kernel executes syscall handler:
   - Invokes **service routine** corresponding to syscall number
     - **Do the real work**, generate result status
   - Places return value from service routine in a register
   - Switches back to user mode, passing control back to wrapper
     - (kernel-mode ⇒ user-mode)



Inside the SYSCALL instruction

- Saves the old (user) SP value
- Switches the SP to the kernel stack
- Saves the old (user) PC value (= return addr)
- Saves the old privilege mode
- Sets the new privilege mode to
- Sets the new PC to the kernel syscall handler



↳ using the system call table

http://www.cs.cornell.edu/courses/cs3410/2018fa/schedule/slides/14-ecf.pdf

GO CLASSES

optional

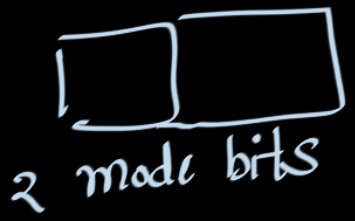
$a = b + c$

User mode ——— Ring3

Kernel mode ——— Ring2 open file

——— Ring1

——— Ring0 print on monitor



2 mode bits

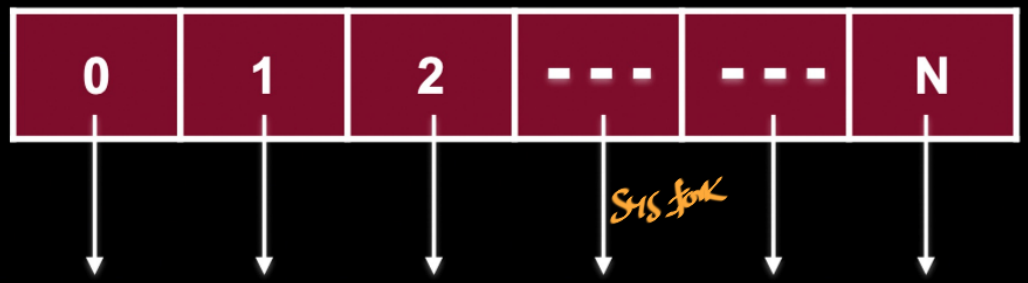
System Call table

- Protected entry points into the kernel for each system call

  We don't want application to randomly jump into any part of the OS code.

- Syscall table is usually implemented as an array of function pointers, where each function implements one system call

- Syscall table is indexed via system call number

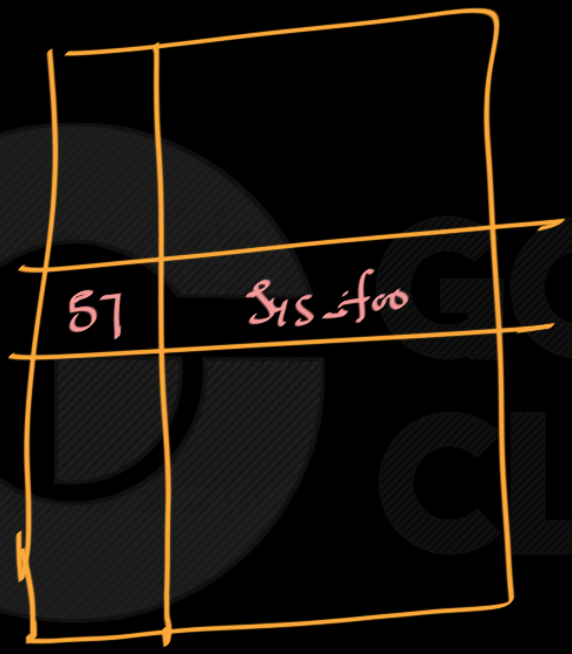


foo

foo ( )
{
}

int a[5]

a


67
Syntax

GO CLASSES

Summary

whenever we see "syscall" instruction then what are the steps?

Printf( ) ← wrapper function → essentially calling syscall / trap / int

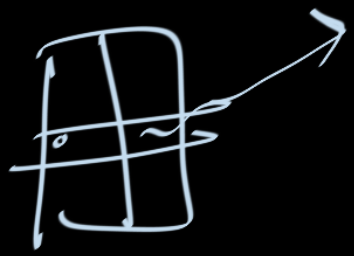
CPU is in infinite cycle of fetch - decode - execute

ISA

GO CLASSES

CPU will pause the current execution (after decoding "Syscall")

Interrupt Service routine



Page fault

} Exception

I/O

} Run time error (Segmentation fault)

CPU will pause the current execution (after decoding "Syscall") → in user code

⇒ Save the registers value

⇒ Switch the mode

⇒ load Rax value and go to System call table (IVT (interrupt vector table))

⇒ find the address of kernel code ← Interrupt service routine (ISR)

→ if kernel function requires the arguments then we check specific location for the arguments

→ Execute the Kernel Code (interrupt service routine code)

⇒ Placing the return values in some specific location

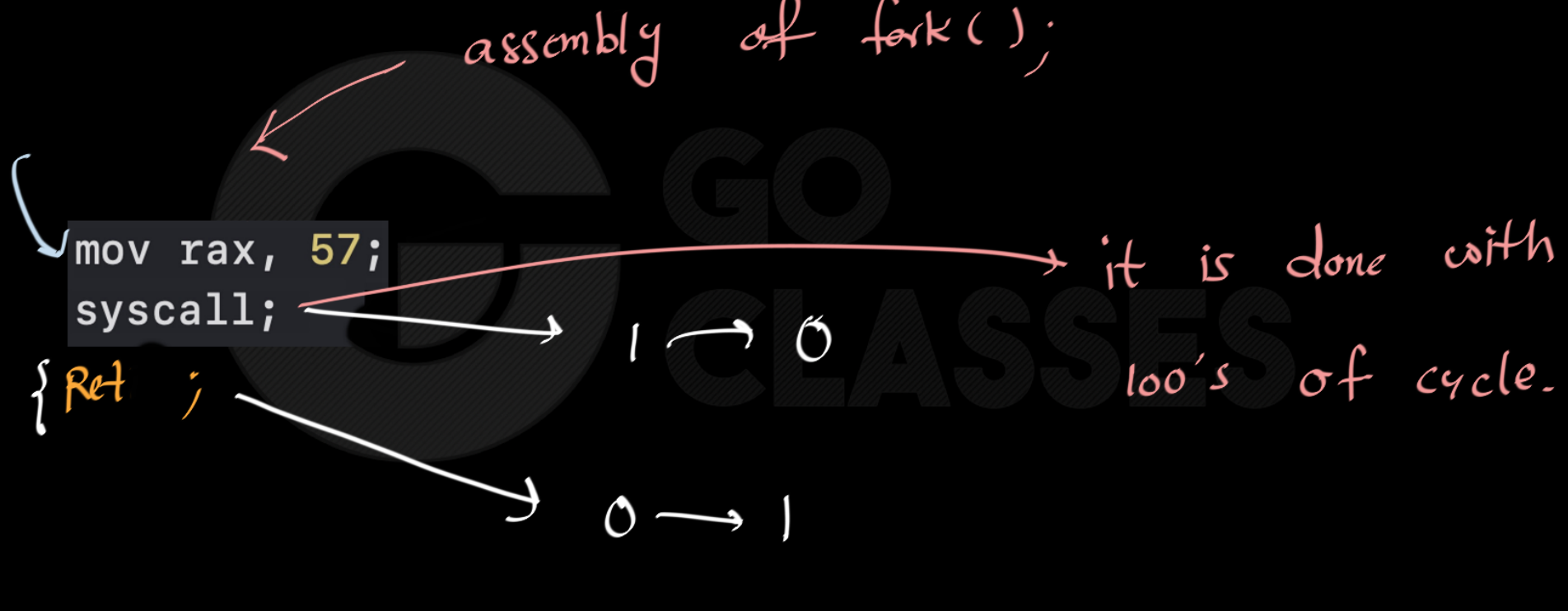
⇒ Reti ← it will switch mode bit from 0 to 1
in user code

GO CLASSES

array name of system call table is

sys_table

pc = sys_table [Rax]


assembly of fork();
mov rax, 57;
syscall;
{ Ret ;
it is done with
100's of cycle.
1 → 0
0 → 1
trap;

Overhead

System calls are very slow.

They can take tens to hundreds of thousands of clock cycles.

This is due to:

- Changing protection domains
- Validating arguments
- Adjusting memory mappings
- Cache effects
- …

Programs should make fewer system calls when practical.

https://cse.buffalo.edu/~eblanton/course/cse220-2022-2f/materials/37-kernel.pdf

QUESTION 13D. True or false? System calls are just as fast as function calls. Explain briefly.

https://cs61.seas.harvard.edu/site/2018/Midterm/

QUESTION 13D. True or false? System calls are just as fast as function calls. Explain briefly.

3pts False. System calls are slower because they transfer to kernel. and many other over heads

Hide solution

https://cs61.seas.harvard.edu/site/2018/Midterm/

Types of System Calls

- Process control
  - create process, terminate process
  - end, abort
  - load, execute
  - get process attributes, set process attributes

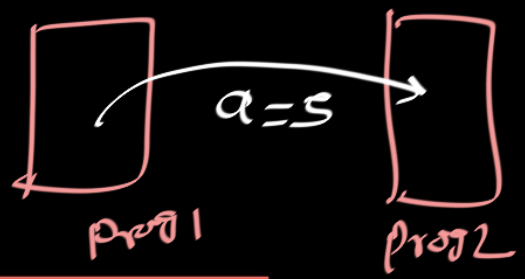
- File management
  - create file, delete file
  - open, close file
  - read, write

Types of System Calls (Contd..)

- Device management
  - request device, release device
- Information maintenance
  - get time or date, set time or date

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

→ Process id of current executing process



GO CLASSES

we can check COA interrupt questions and try to solve all questions related to interrupt execution.

New Topic: Threads

Threads

In certain situations, a single application may be required to perform several similar tasks.

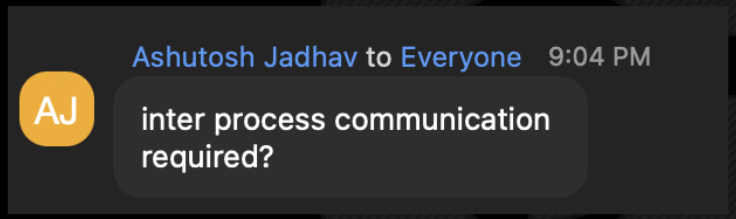
in these situations we may not want to have different processes.

word processor
- Spell checker
- write something

if we have different processes then -
- context switching
- different add. space

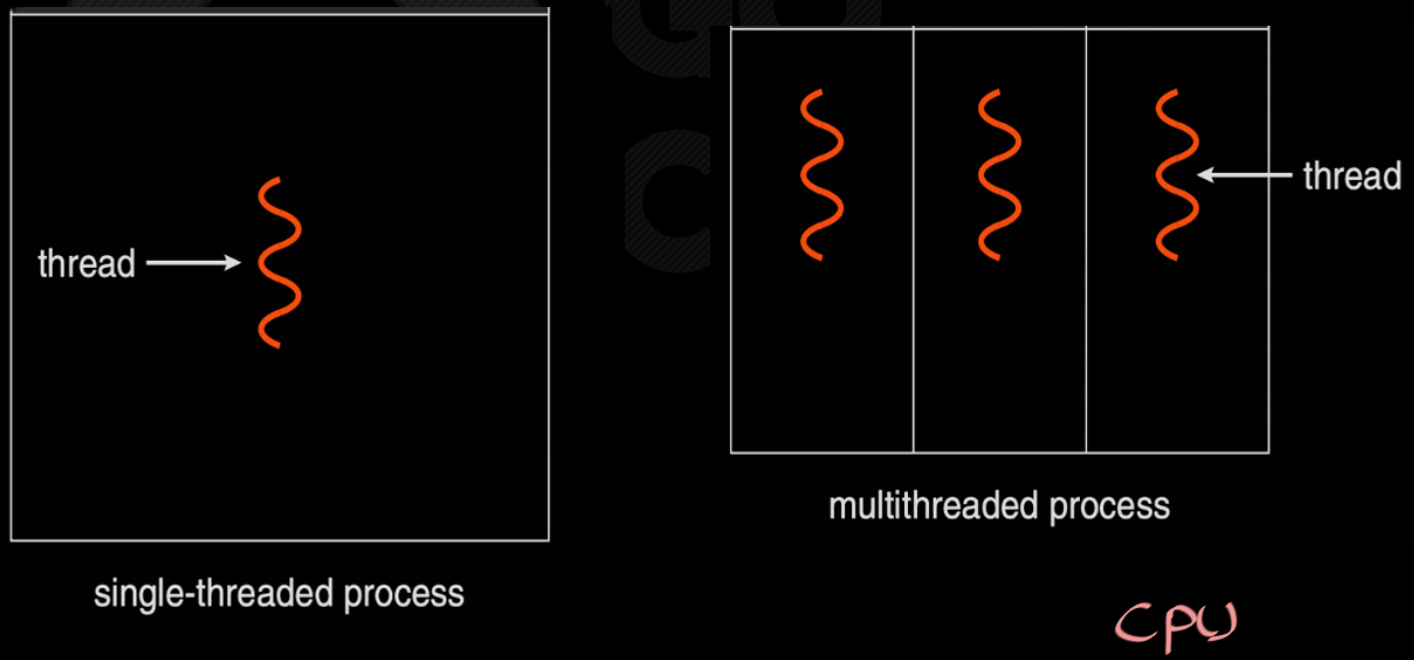
GO CLASSES

for the simmillar tasks, if we have different processes (instead of threads) then



→ IPC (inter process comm) is required

→ process contact switching is bit costlier than thread context switching


thread
thread
single-threaded process
multithreaded process
CPU

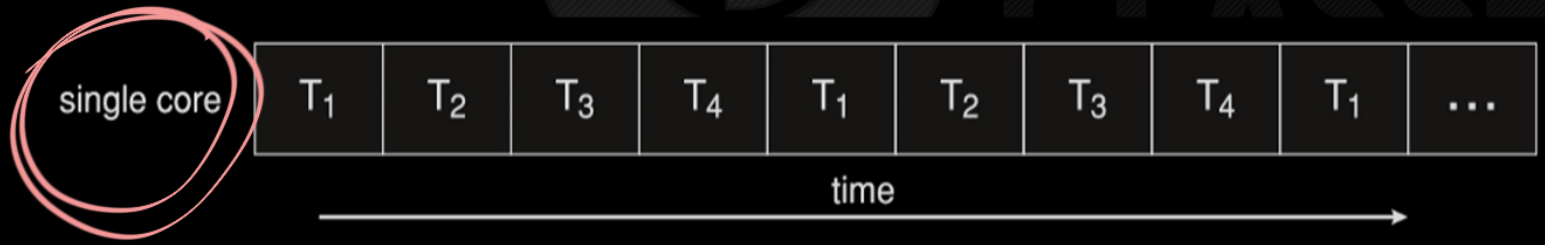
Thread: Concurrency vs. Parallelism

- Threads are about concurrency and parallelism

Thread: Concurrency vs. Parallelism

- Threads are about concurrency and parallelism

T_1, T_2, T_3, T_4



- Concurrent execution on single-core system

Thread: Concurrency vs. Parallelism

- Threads are about concurrency and parallelism

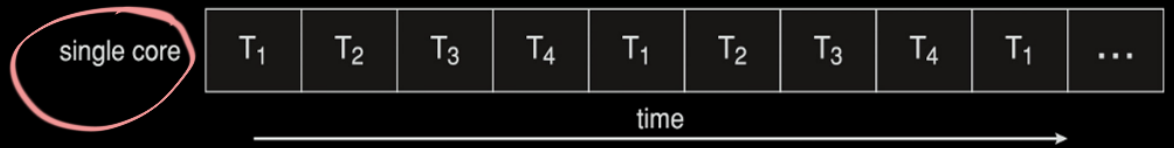
Processor
CPU

"if we have two Processors then we can run atmost 2 threads at a time

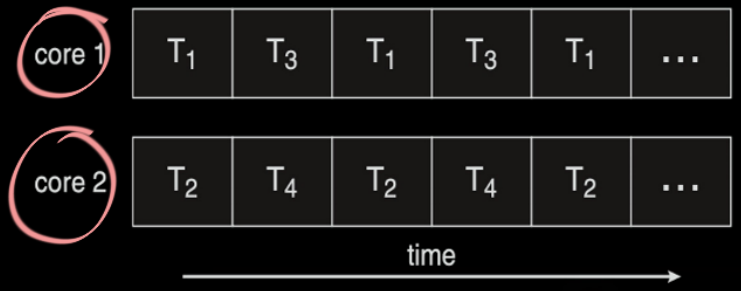
- **Parallelism on a multi-core system**

Thread: Concurrency vs. Parallelism

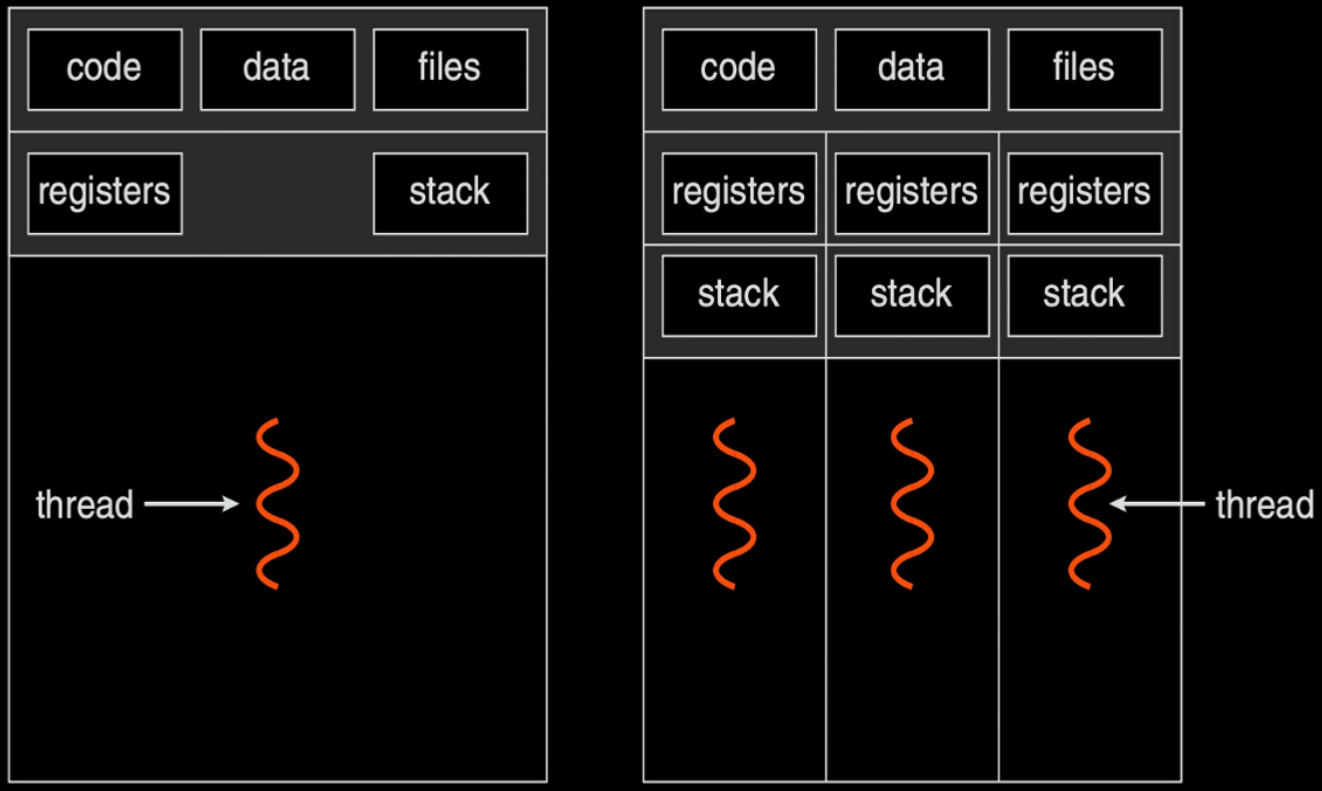
- Threads are about concurrency and parallelism
- **Concurrent execution on single-core system:**



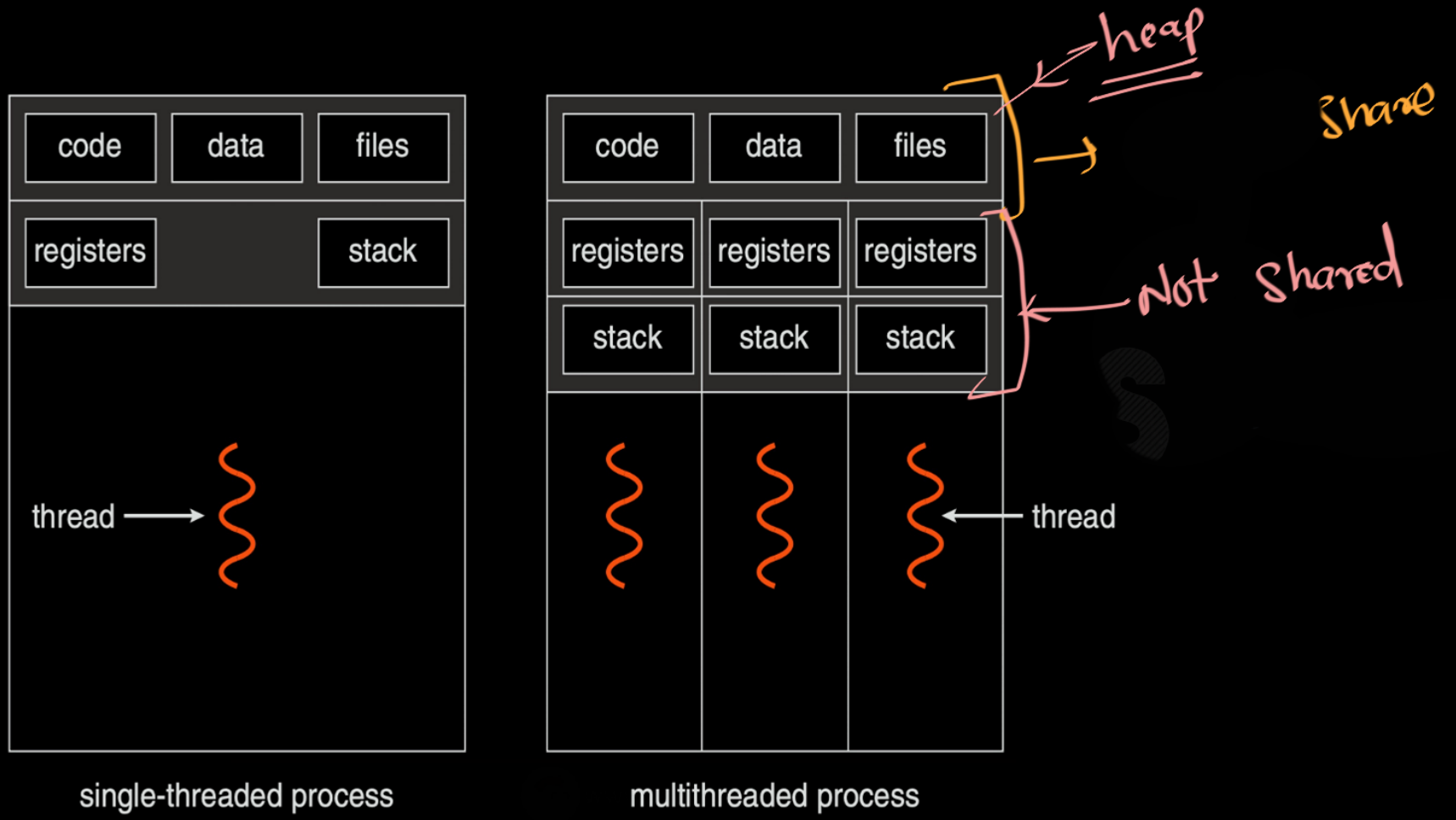
- **Parallelism on a multi-core system:**

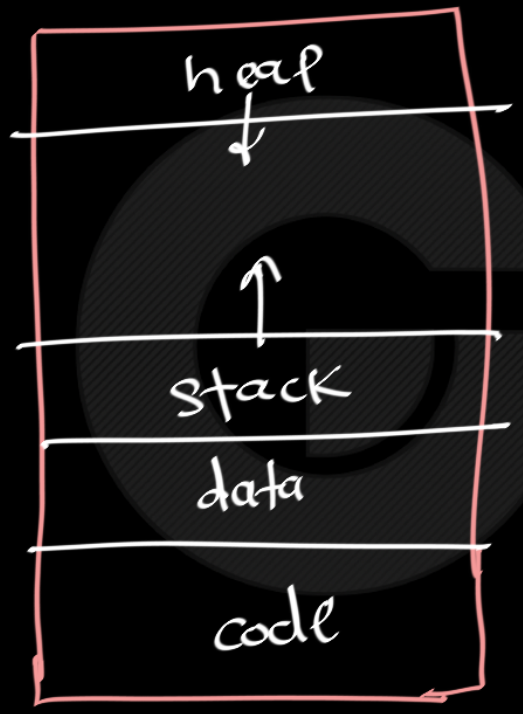


Single and Multi threaded process



Single and Multi threaded process





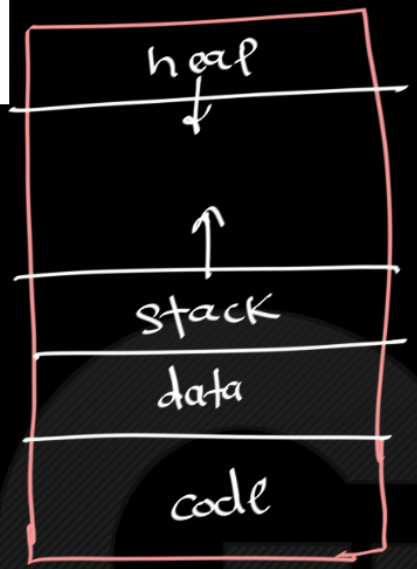
Process

Content

→ might have opened some file

→ some register values

PC, R_1, R_2



Process

Context

→ might have opened some file

→ some register values PC, R_1, R_2

Should Share

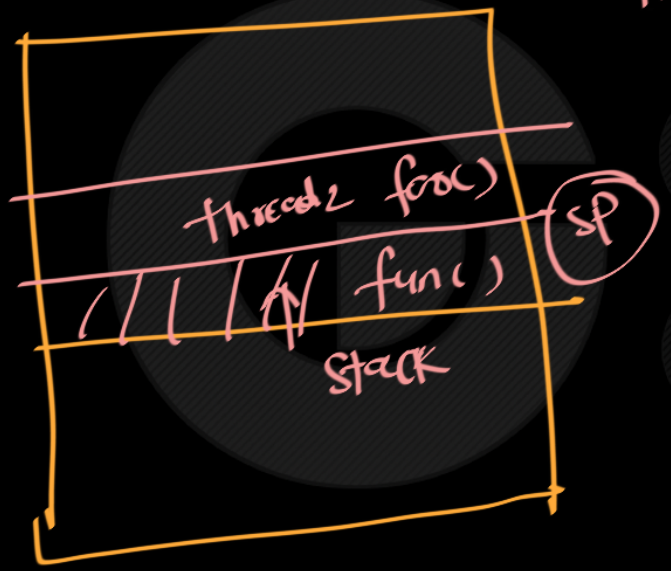
Code

data

⇒ what are the things that threads should NOT share?

→ Stack,

→ Register values (PC, SP, GPR)



thread1

```
int a = 5;
foo ( )
{
    a = a+1;
}
```

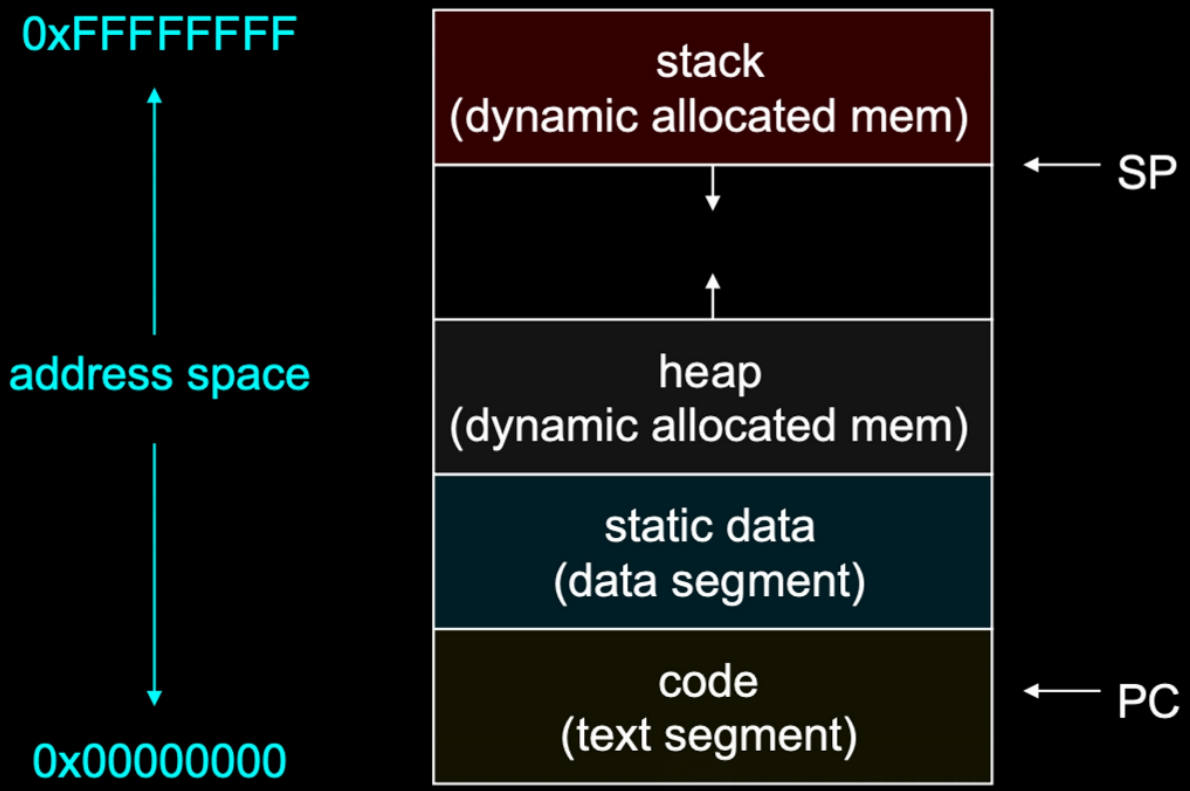
thread2

```
fun ( )
{
    a = 2*9
}
```

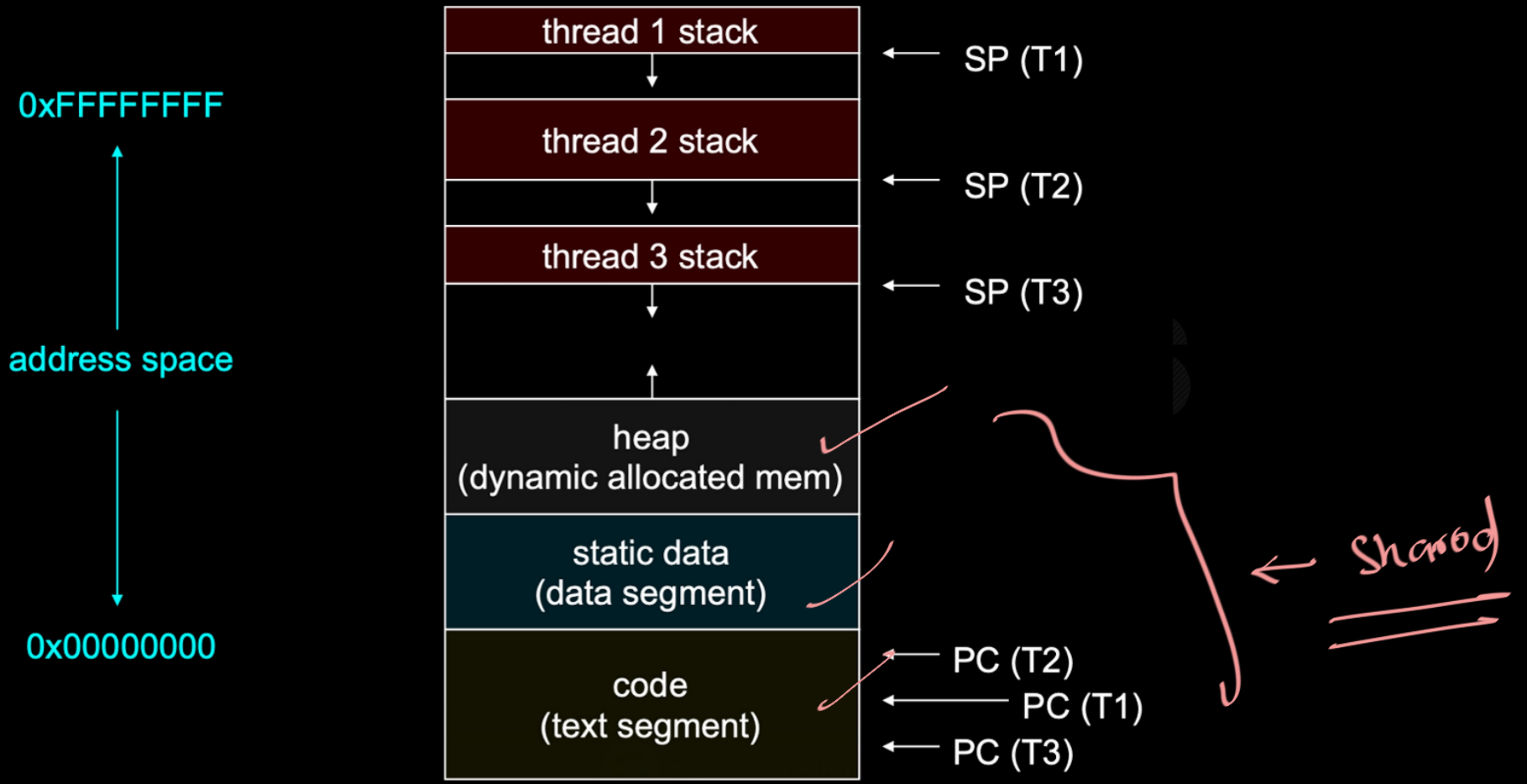
GO CLASSES

Distinguishing between these kinds of memory matters because each thread will have its own stack. However, all the threads in a process will share the heap. Some people call threads lightweight processes because they have their own stack but can access shared data. 23-Nov-2022

(old) Process address space



(new) Process address space with threads



Types of threads

- User level
- Kernel level

Types of threads

- User level
  - When we make threads using function calls
- Kernel level
  - When we make threads using System calls

Types of threads

- User level
  - When we make threads using function calls
  - we have TCB
- Kernel level
  - When we make threads using System calls
  - For each thread there is new PCB (TCB)
  - Since only Kernel can create PCB hence System call is needed

How to initiate threads ?

- User level
  - Some library help us
- Kernel level
  - Some library help us

How to initiate threads ?

4.4 Thread Libraries

A **thread library** provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

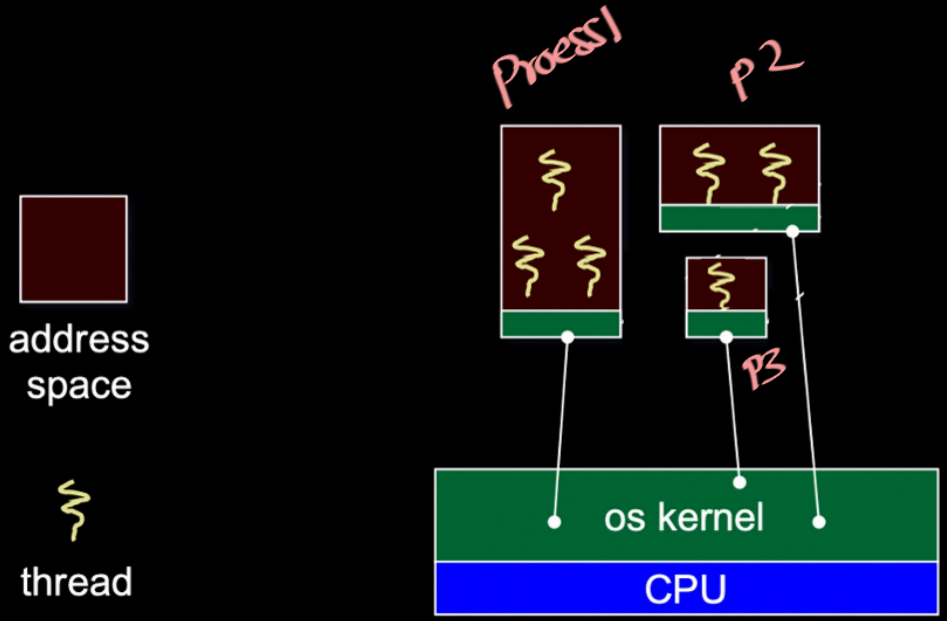
The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

User level

Kernel level

Optional read from Galvin

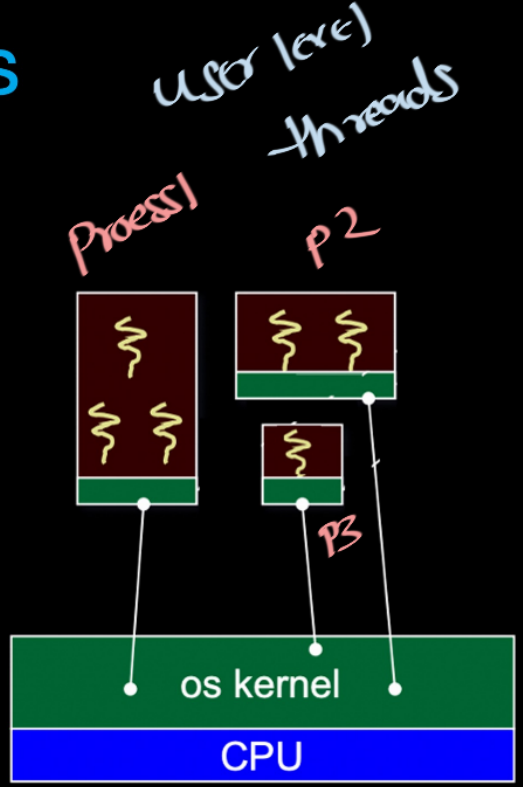
User-level threads



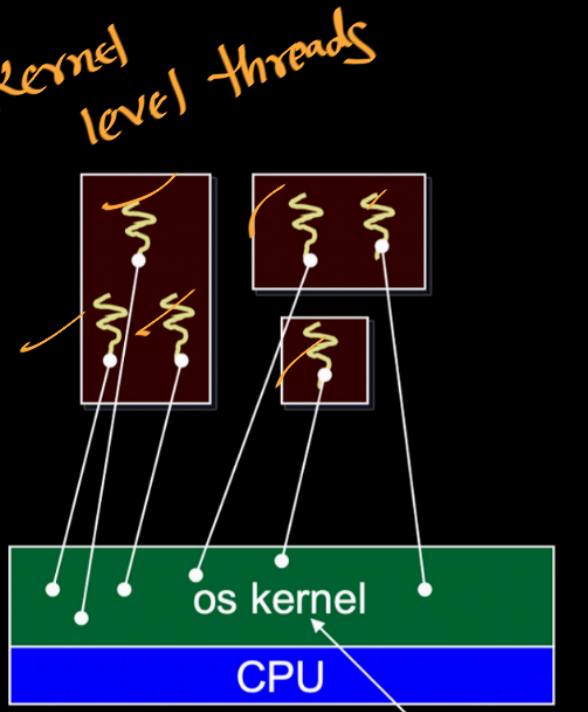
Now thread id is unique within the context of a process, not unique system-wide

User-level threads

No free lunch



8 CPUs



8 cores

Now thread id is unique within the context of a process, not unique system-wide

User-level threads

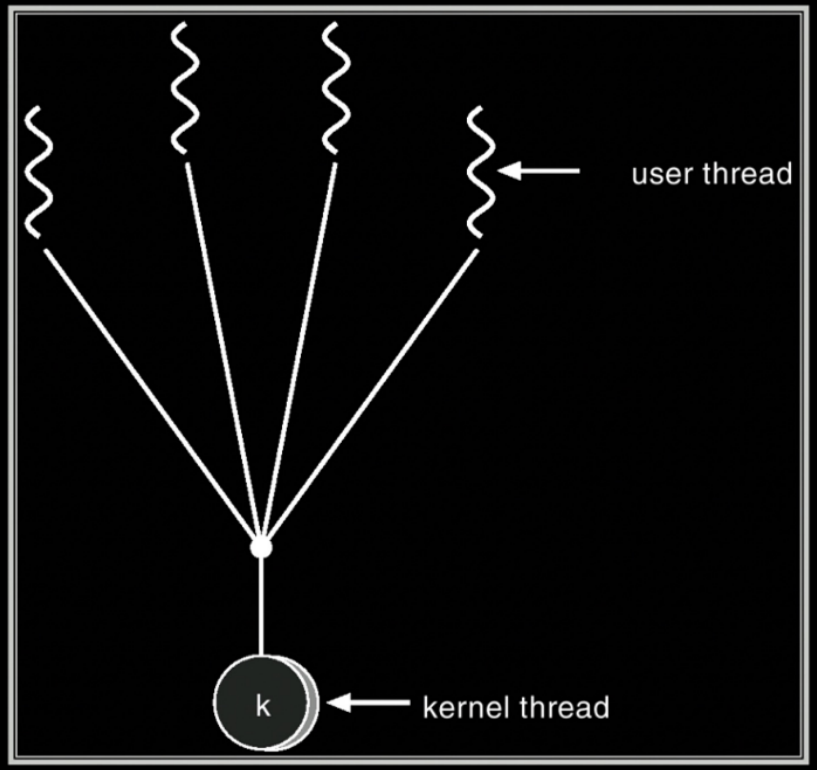
- User-level threads are small and fast
  - managed entirely by user-level library
    - E.g., pthreads (`libpthreads.a`)
  - each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
  - creating a thread, switching between threads, and synchronizing threads are done via procedure calls
    - no kernel involvement is necessary!
- User-level thread operations can be 10-100x faster than kernel threads as a result

THREADS

User-level threads

- Many user-level threads mapped to single kernel thread.
- Used on systems that do not support kernel threads.
- Examples:
  Solaris Green Threads
  GNU Portable Threads

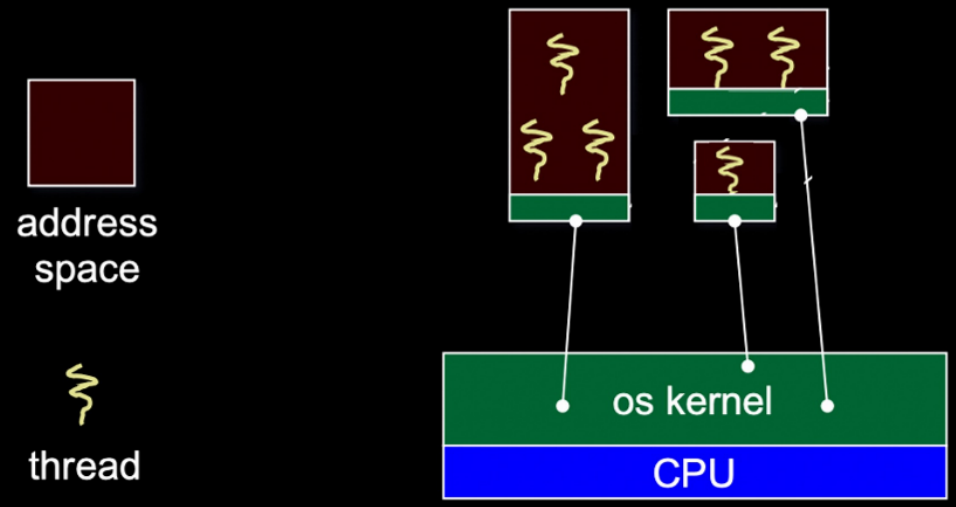
Many-to-One



User-Level Threads

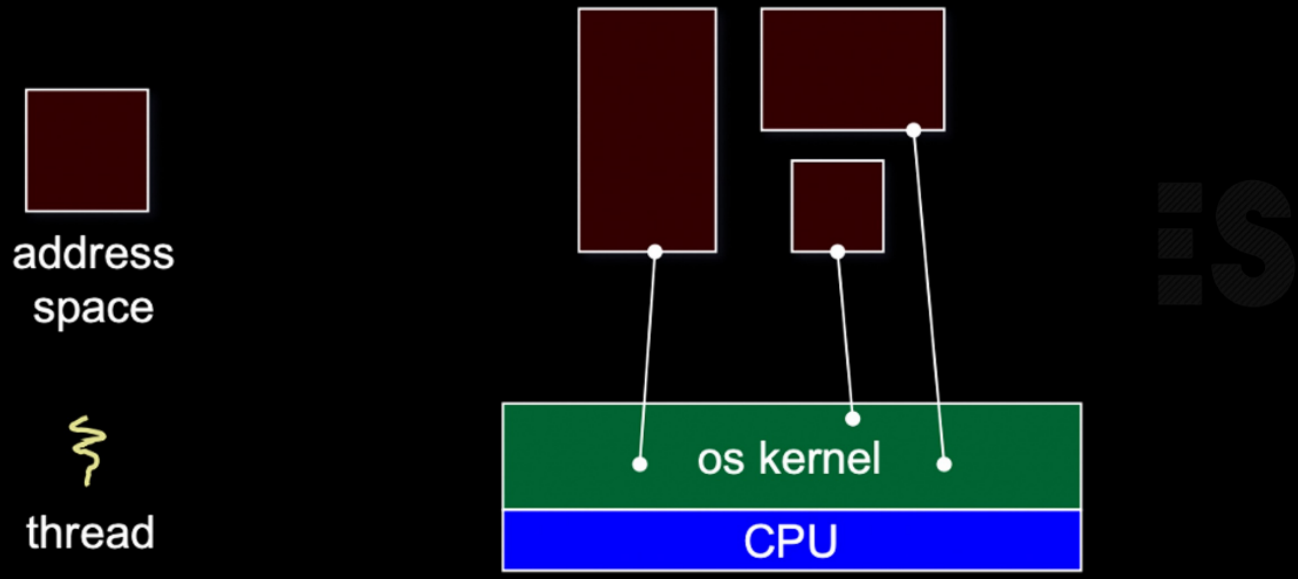
- A **user-level thread** is a thread that the OS does *not* know about.
- The OS only knows about the process containing the threads.
- The OS only schedules the process, not the threads within the process.
- The programmer uses a *thread library* to manage threads (create and delete them, synchronize them, and schedule them).

User-level threads



Now thread id is unique within the context of a process, not unique system-wide

User-level threads: What the kernel sees



Question:

What is one major advantage and major disadvantage of User level threads ?

Question:

What is one major advantage and major disadvantage of User level threads ?

Advantage: They are fast since no OS involvement (no context switch and all)

Disadvantage: OS doesn't know about threads so if one threads block because of I/O then all threads get block

Question:

3. What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?

   The biggest advantage is efficiency. No traps to the kernel are needed to switch threads. The ability of having their own scheduler can also be an important advantage for certain applications. The biggest disadvantage is that if one thread blocks, the entire process blocks.

User-Level Threads: Disadvantages

- Since the OS does not know about the existence of the user-level threads, it may make poor scheduling decisions:
  - It might run a process that only has idle threads.
  - If a user-level thread is waiting for I/O, the entire process will wait.
  - Solving this problem requires communication between the kernel and the user-level thread manager.
- Since the OS just knows about the process, it schedules the process the same way as other processes, regardless of the number of user threads.
- For kernel threads, the more threads a process creates, the more time slices the OS will dedicate to it.

Kernel threads