Lecture 6

Topic: Threads

Threads

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

Why we do not want different processes for "similar tasks"?

→ interprocess communication
→ context switch

Thread: Concurrency vs. Parallelism

- Threads are about concurrency and parallelism

Thread: Concurrency vs. Parallelism

- Threads are about concurrency and parallelism

- **Concurrent execution on single-core system**

(sequential execution)

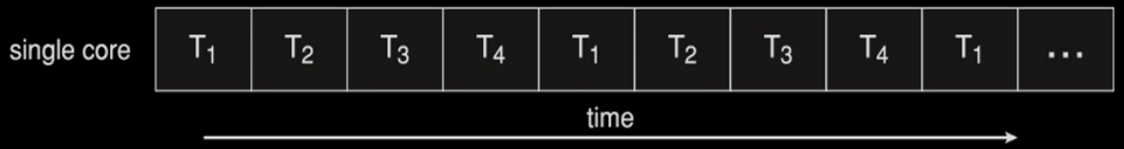
Thread: Concurrency vs. Parallelism

- Threads are about concurrency and parallelism

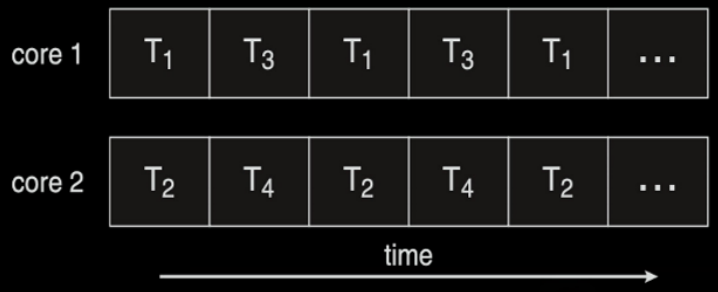
- **Parallelism on a multi-core system**

Thread: Concurrency vs. Parallelism

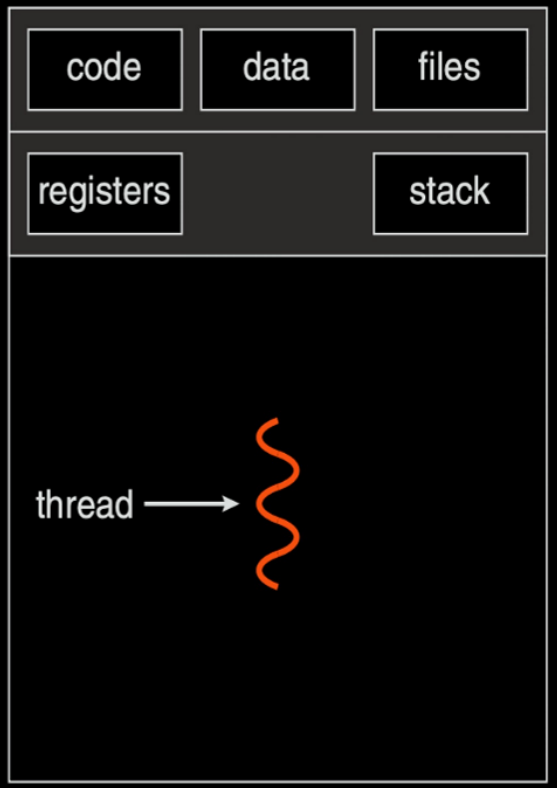
- Threads are about concurrency and parallelism
- **Concurrent execution on single-core system:**



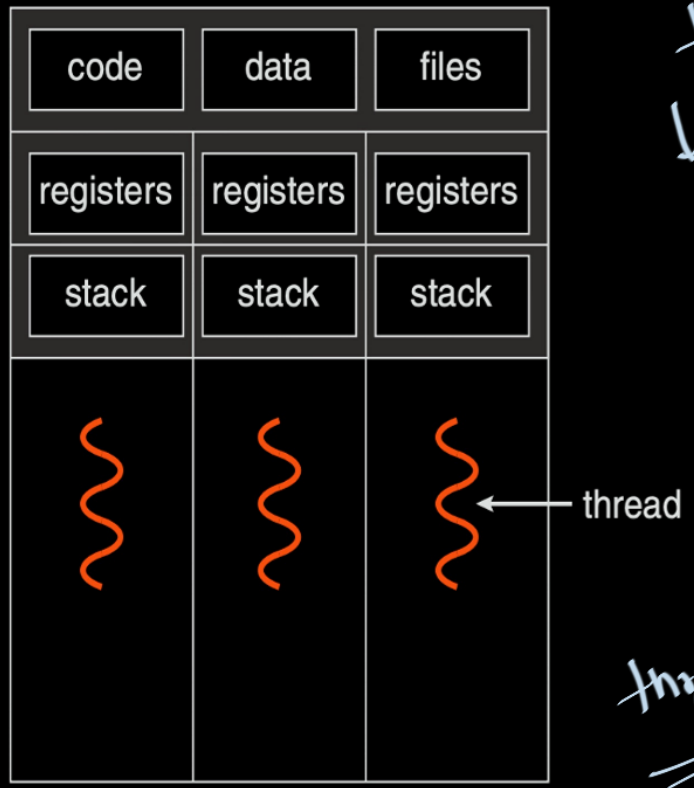
- **Parallelism on a multi-core system:**



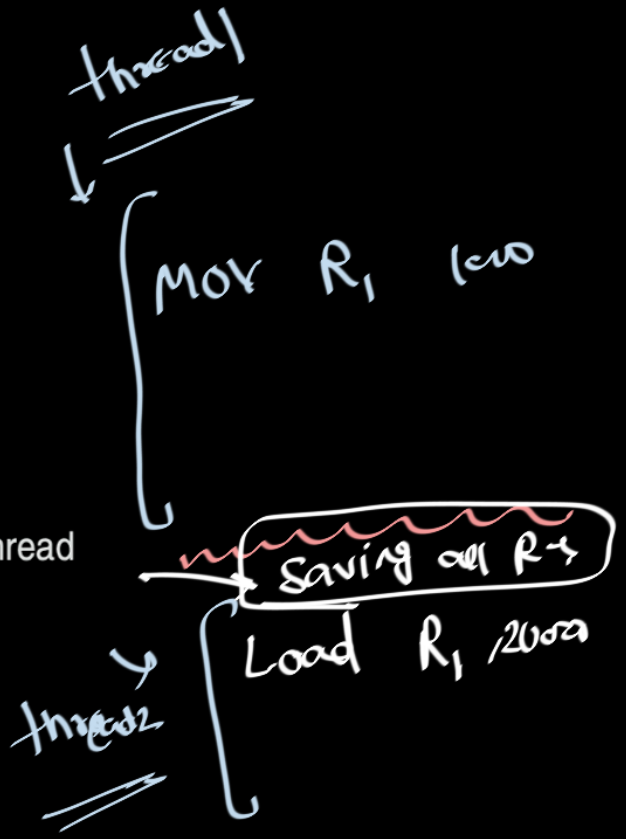
Single and Multi threaded process



single-threaded process

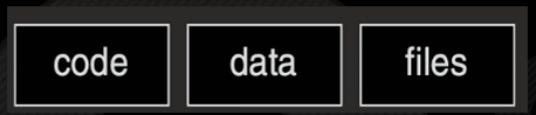


multithreaded process



GO CLASSES

What are the things that threads share?

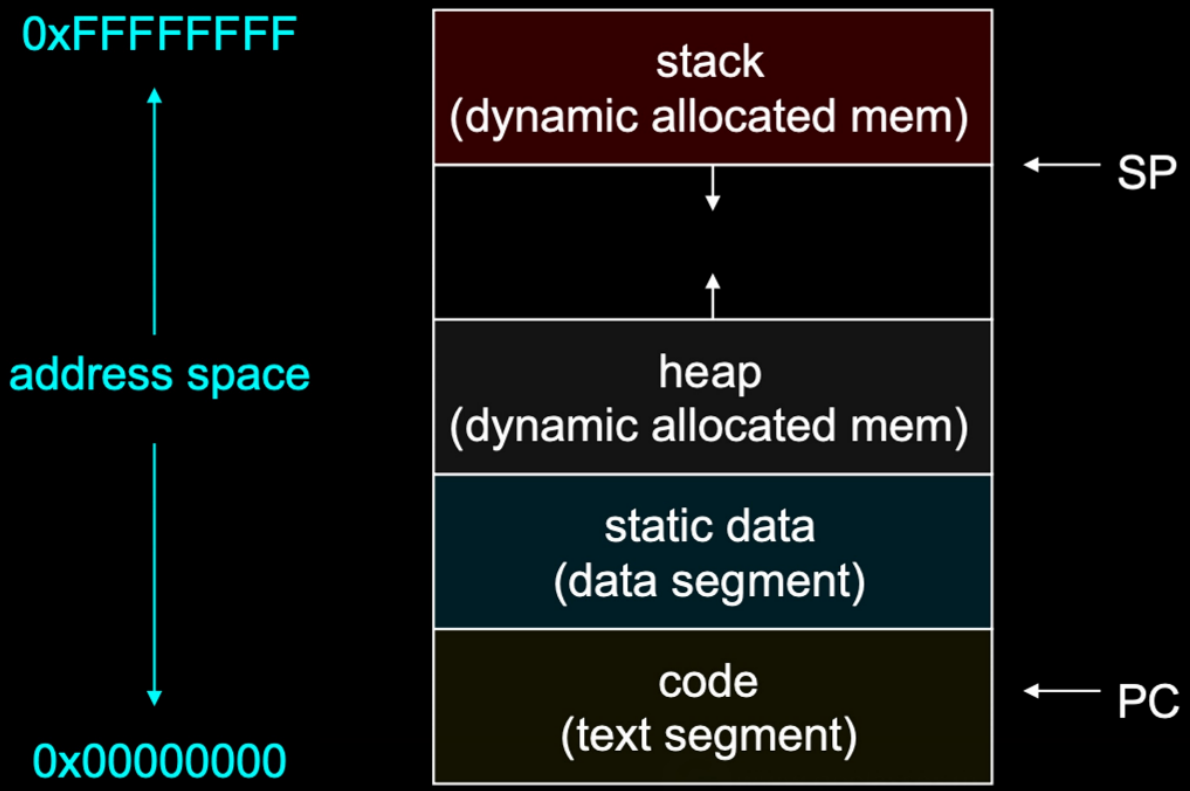


, heap.

Does not share

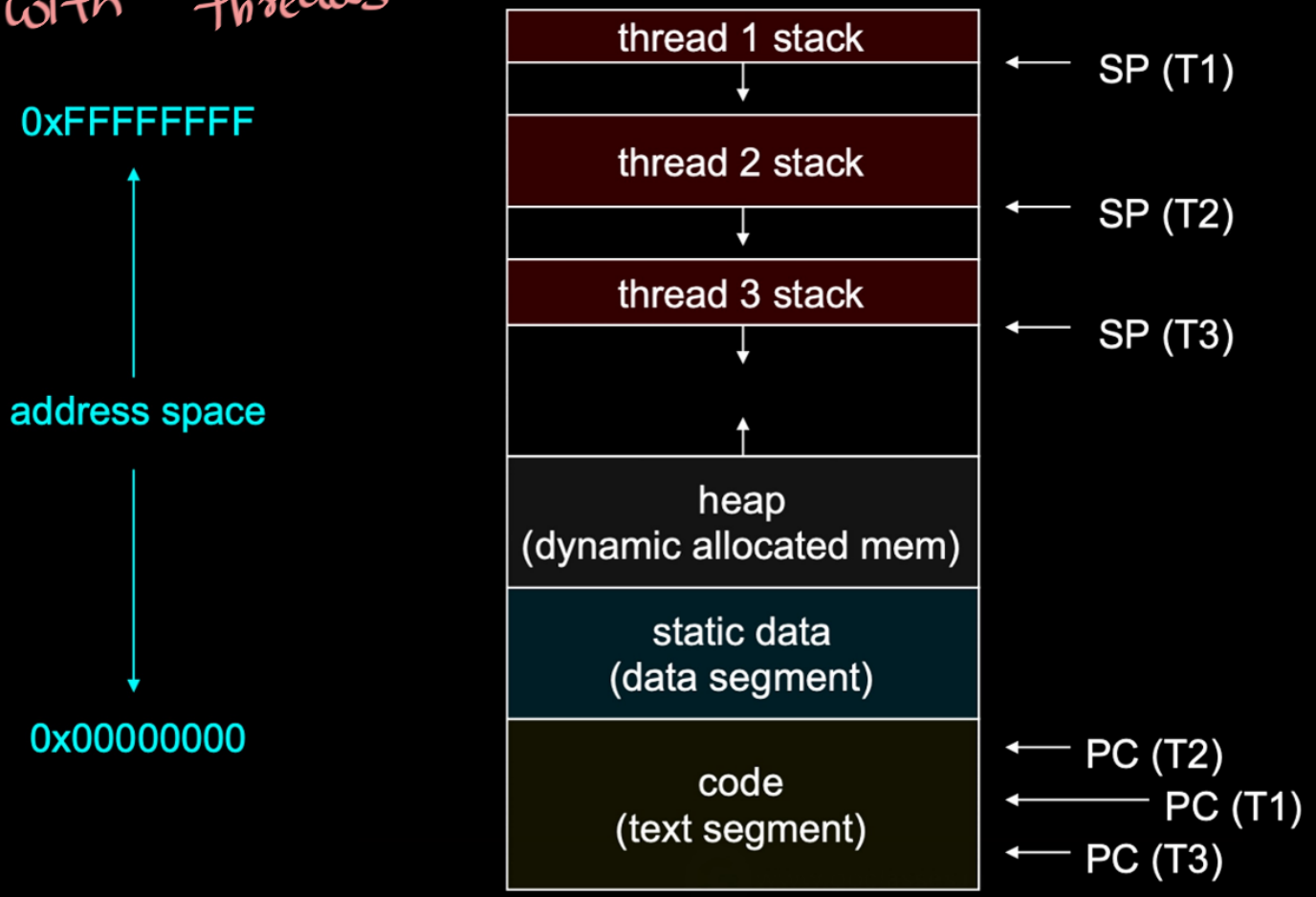
(old) Process address space

↳ without any threads



(new) Process address space with threads

↳ 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 thread control block (TCB)
- Kernel level
  - When we make threads using System calls
  - For each thread there is new PCB or TCB

How to initiate threads ?

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

How to initiate threads ?

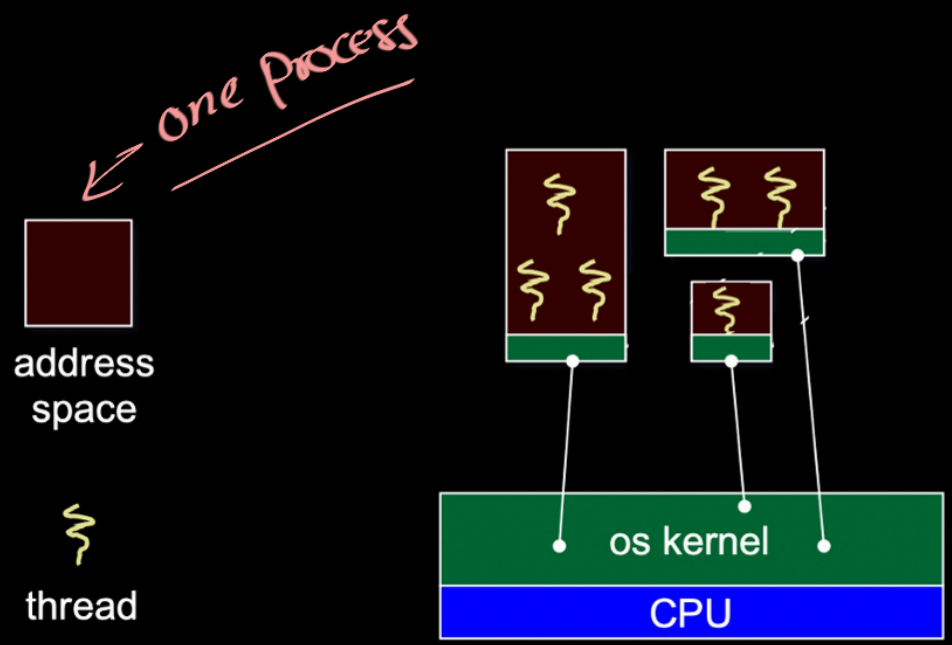
4.4 Thread Libraries

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.

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

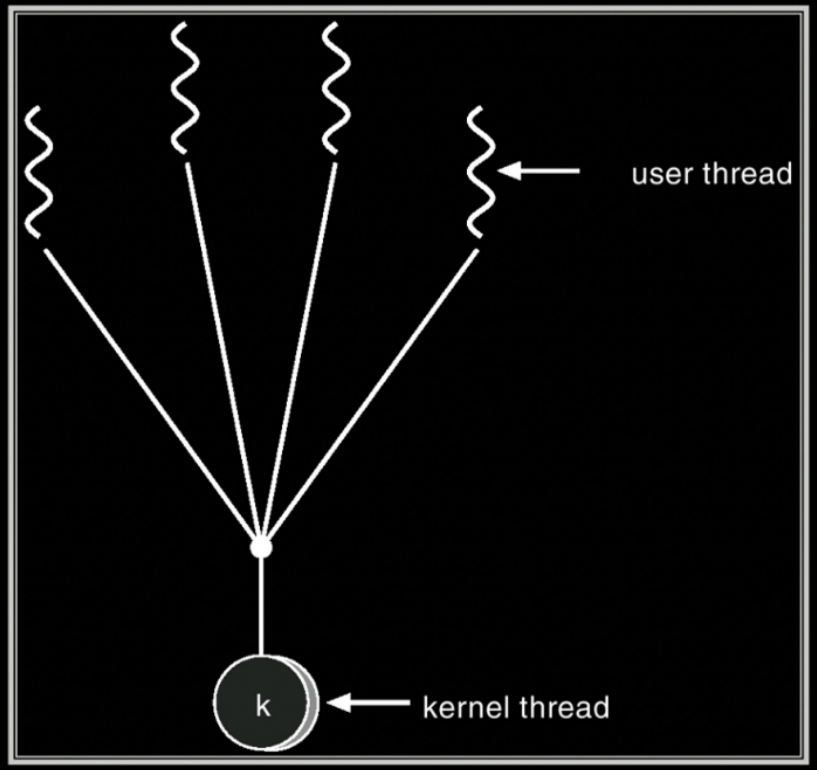
- User-level threads are small and fast
  - managed entirely by user-level library
    - E.g., pthreads (`libpthreads.a`)
  - each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB)
  - creating a thread, switching between threads, and synchronizing threads 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

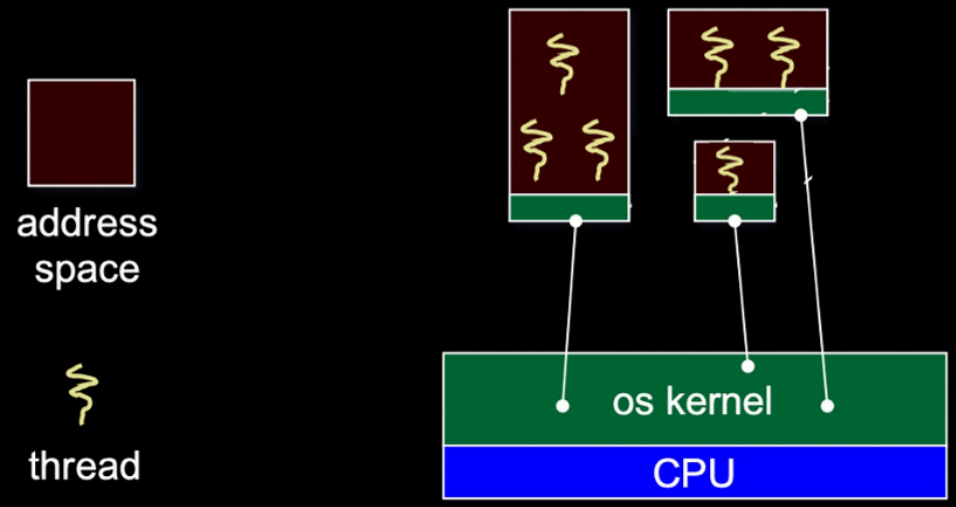
Many-to-One



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).

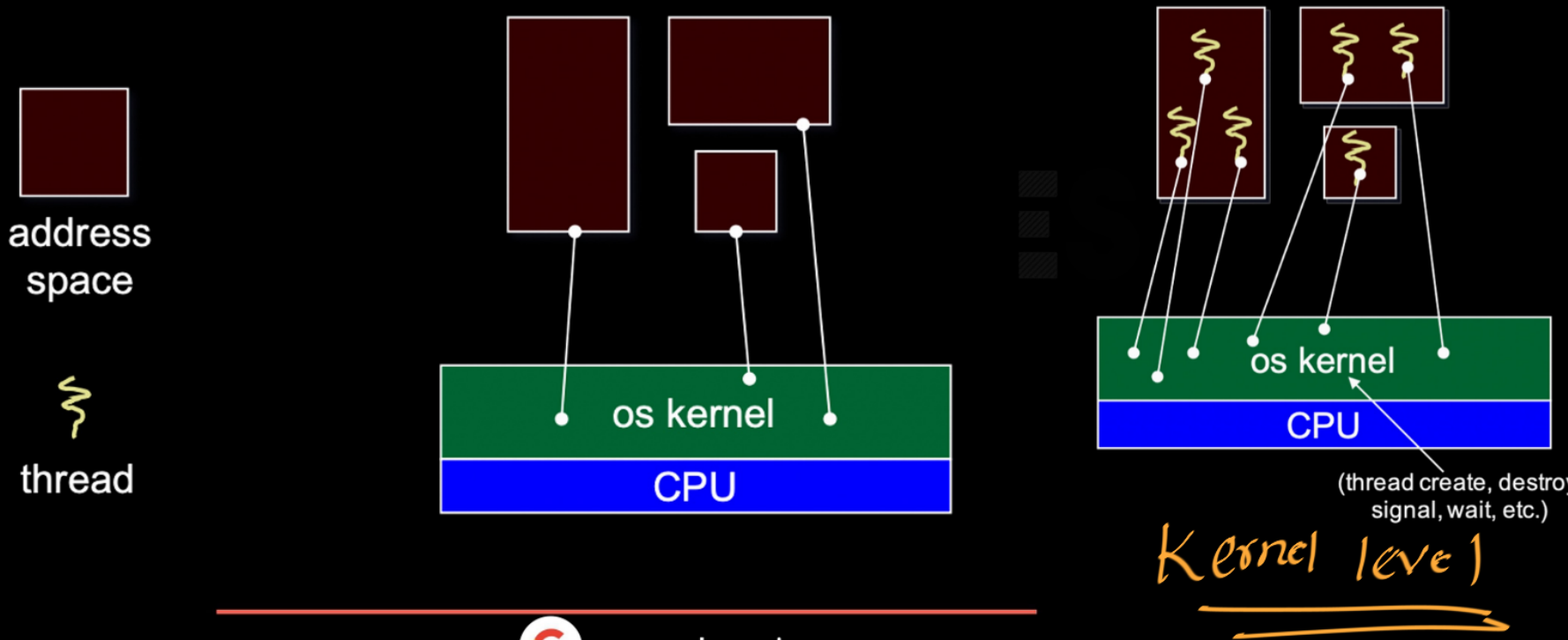
User-level threads



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

User-level threads: What the kernel sees

"scheduler.c"



GO CLASSES

[Operating System must have a "scheduler.c"

↳ scheduler ( )

P1

P2

P3

OS designer

{

> loop through all PCBs

> Picking one of the PCB based on some criteria

}

threads in user space.

Can we take help from kernel scheduler to schedule the threads? ⇒ No.

threads in user space.

Can we take help from kernel scheduler to schedule the threads? ⇒ No.

> i (user) might be having my own preferences to schedule.

GO CLASSES

Question: i can not take help from OS scheduler, so how i am supposed to schedule them?

Answer: i need to write my own scheduler

there are libraries to help us.

Mayukh Kundu to Everyone 7:31 PM
MK
user level thread manager

Sanjana to Everyone 7:32 PM
S
Thread scheduler implemented in user space

Sangeet Bhowmick to Everyone 7:32
SB
user's own scheduler.c may be

we have thread libraries, which has everything

- → scheduler
- → thread manager.

GO CLASSES

Question: Can user level threads run Kernel code?

GO CLASSES

Question: Can user level threads run Kernel code?

Answer: Yes, we can have system call in user thread.

> P_1 is running

> we want to schedule other process now

> we will first schedule / run "scheduler".

"syscall"

question : How to schedule the scheduler itself?

Answer : sched : ← instruction (this will call scheduler function)

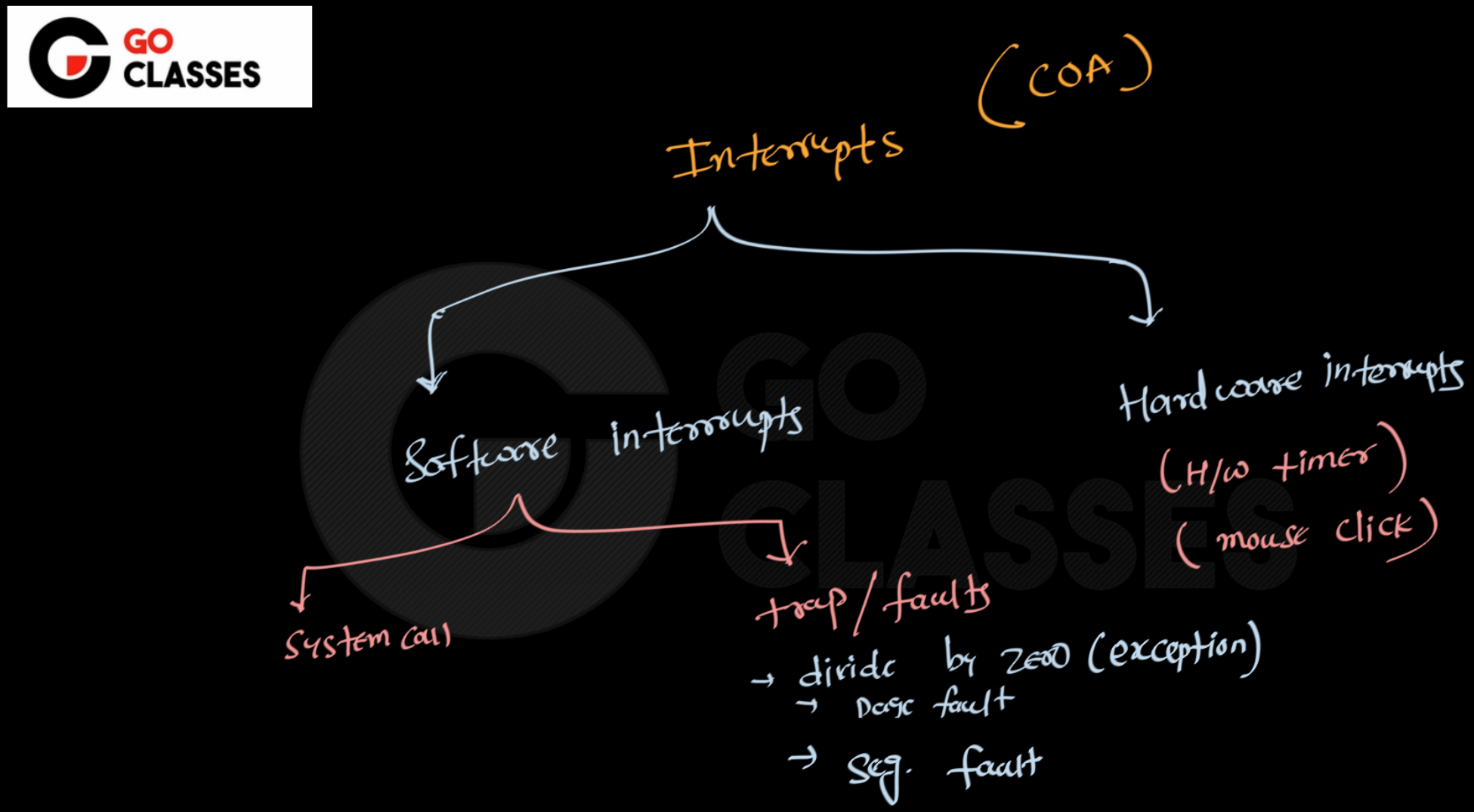
GO CLASSES

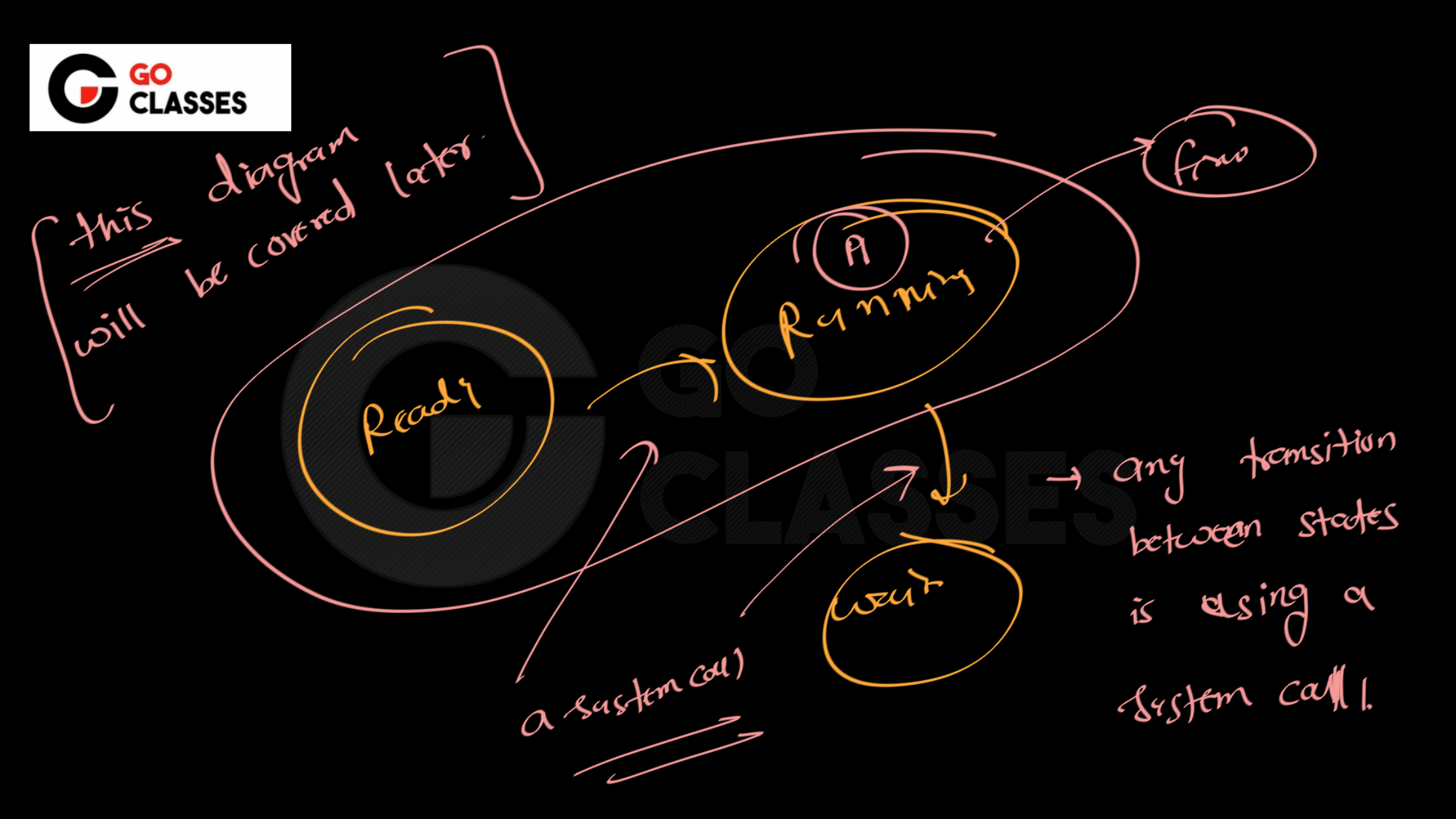
P1 timer is over → hardware interrupt (if timer is maintained)

P1 is accessing some I/O : write system call (using hardware)

↳ System call (Software interrupt) (if there is a variable inside the OS code then Software interrupt)

↳ Synchronous


GO CLASSES
(COA)
Interrupts
Software interrupts
Hardcoare interrupts
(H/w timer)
(mouse click)
System call
trap / faults
→ divide by zero (exception)
→ Page fault
→ Seg. fault


GO CLASSES
[this diagram will be covered later]
Ready
Running
A
Wait
free
a system call
→ any transition between states is using a system call.

→ P_1 is running on CPU

→ Now P_1 want to interact I/O device

→ (we are moving out P_1 from running to some other state)

→ we will be having System Call.

→ in that System call there will be instruction (ex sched) that will schedule the OS scheduler

→ OS scheduler will schedule some other process.

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 ?

Disadvantages :

Not visiable to kernel (hence OS Can not make decisions about scheduling based on available hardware (CPUs))

OS will put the complete process in the block state

GO CLASSES

Advantages :

- it is fast to create (because no system call needed)
- Kernel doesn't have to maintain different states of different threads (user code will do it) hence Context Switching is not needed. → No Context Switching → using libraries
- we have our own scheduler here.
- They can run on any OS.

Comparitively it is not advantage anymore.

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:

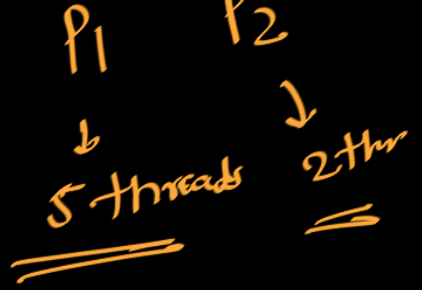
3. What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?

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.

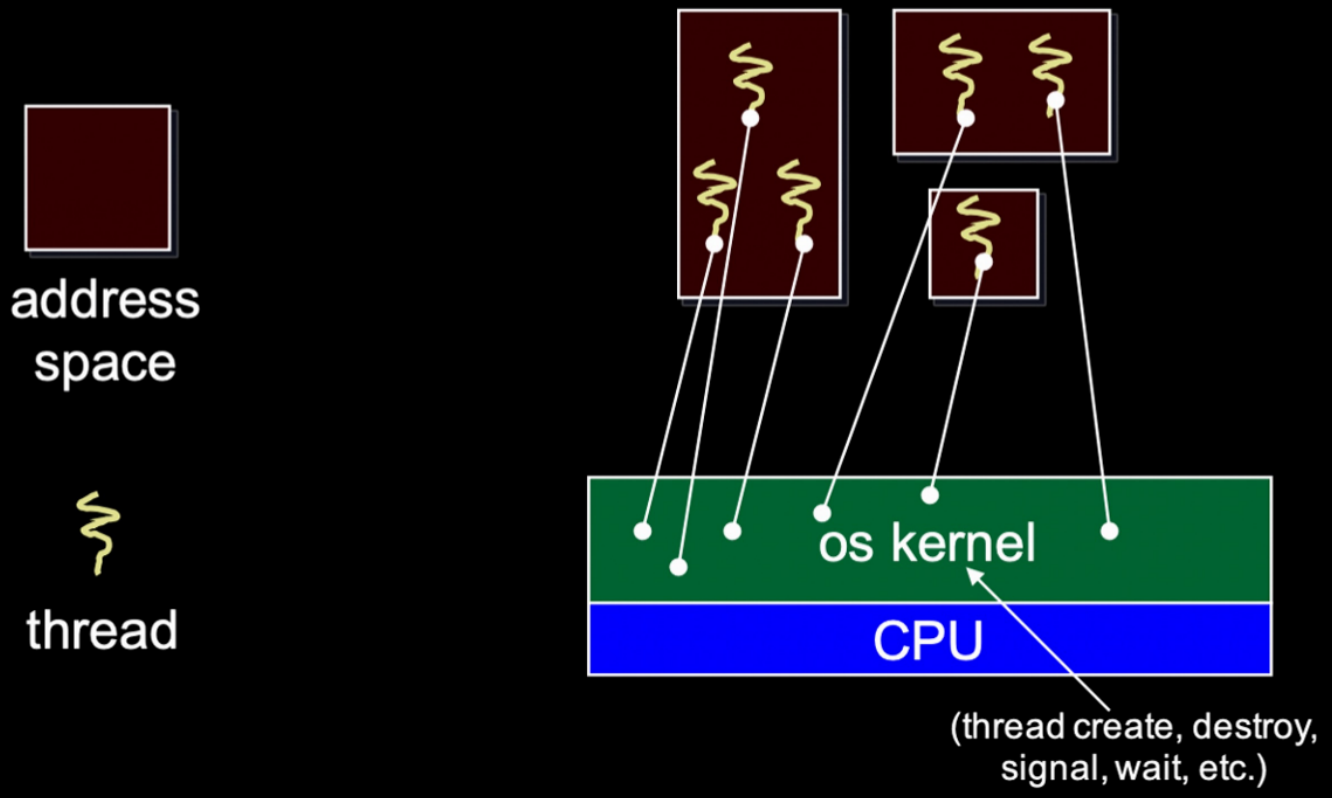
The biggest disadvantage is that if one thread blocks, the entire process blocks.

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



GO CLASSES

1) Create a process using fork - exec

2) Create kernel level thread

3) Create user level thread

Arrange them in the decreasing order of cost -

1 > 2 > 3

(one is most costly)

GO CLASSES

Creating entire address space

1) Create a process using fork - exec

2) Create kernel level thread

3) Create user level thread

Arrange them in the decreasing order of cost -

1 > 2 > 3

(one is most costly)

Kernel threads

- OS now manages threads *and* processes / address spaces
  - all thread operations are implemented in the kernel
  - OS schedules all of the threads in a system
    - if one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process
    - possible to overlap I/O and computation inside a process
- Kernel threads are cheaper than processes (But costlier than user threads)
  - less state to allocate and initialize
- But, they're still pretty expensive for fine-grained use
  - orders of magnitude more expensive than a procedure call
  - thread operations are all **system calls**
    - context switch
    - argument checks
  - must maintain kernel state for each thread

Summary

- Thread: a single execution stream within a process
- Switching between user-level threads is faster than between kernel threads since a context switch is not required.
- User-level threads may result in the kernel making poor scheduling decisions, resulting in slower process execution than if kernel threads were used.

Performance Example

- On a 700MHz Pentium running Linux 2.2.16 (only the relative numbers matter; ignore the ancient CPU!):
  - Processes
    - `fork/exit`: 251 µs
  - Kernel threads
    - `pthread_create()/pthread_join()`: 94 µs **(2.5x faster)**

Why?

  - User-level threads
    - `pthread_create()/pthread_join`: 4.5 µs **(another 20x faster)**

Why?

- When CPU transitions to supervisor mode and starts running kernel code (because of a syscall, exception or interrupt) it is still in the context of the current thread
  - Code that thread executes comes from kernel instructions

Question: T/F

Question: 1

The OS provides the illusion to each thread that it has its own address space.

Question: 2

With kernel-level threads, multiple threads from the same process can be scheduled on multiple CPUs simultaneously.

Question: T/F

Question: 1

The OS provides the illusion to each thread that it has its own address space.

No- OS doesn't say every thread is having it's own heap / code.

Question: 2

With kernel-level threads, multiple threads from the same process can be scheduled on multiple CPUs simultaneously.

True

The OS provides the illusion to each thread that it has its own address space.

False – Each process has its own address space, but threads in the same address space share that address space (e.g., they use the same code and heap).

With kernel-level threads, multiple threads from the same process can be scheduled on multiple CPUs simultaneously.

True – this is the benefit of kernel-level threads (true thread support from OS); we could not do this with user-level threads.

Question: T/F

If a user-level-thread is blocked for I/O operation, then kernel of OS will perform context switching to run another user-level-thread which is not blocked.

Question: T/F

If a user-level-thread is blocked for I/O operation, then kernel of OS will perform context switching to run another user-level-thread which is not blocked.

false

Question: T/F

Many-to-one is the most efficient model of multi-threading because it allows several threads to be assigned to different processors in a multi-processor computer system.

If a user-level thread is blocked for I/O operation, ~~the kernel of operating system will perform context switching to run another user-thread which is not blocked~~.

The Kernel does not have any information about the user-level threads, therefore, it can not do task switching for user-level threads.

Many-to-One is the most efficient model of multi-threading since ~~it allows several user-level threads to be assigned to different processors in a multi-processor computer system.~~

Many-to-One uses one kernel thread therefore it cannot use multi-processor facility.

Question: T/F

In multi-threaded processing with Kernel-Level threads, the various threads associated with a single process must share a common Thread State (e.g., Ready, Blocked, or Running).

6. In multi-threaded processing with Kernel-Level threads, the various threads associated with a single process must share a common Thread State (*e.g.*, Ready, Blocked, or Running). *Answer*: **F**alse

Question: T/F

12. In multi-threaded processing, the various threads associated with a single process share a common User Stack.

Answer: **False**

Question:

2. **Processes and threads.** Suppose that three processes exist in a system, as described table below. Suppose that the system uses preemptive, round-robin scheduling, and that T_{11} is running when the quantum expires.

Process	Threads within the process
P_1	T_{11}, T_{12}, T_{13}
P_2	T_{21}, T_{22}
P_3	T_{31}

T_{11} ↓ P_1

Optional

MSQ

(a) If the threads are implemented entirely at the user level (with no support from the operating system), which threads might possibly execute at the beginning of the next quantum?

(b) If threads are supported by the operating system (i.e., lightweight processes), which threads might possibly execute at the beginning of the next quantum?

(c) How does your answer change if the system has two processors?

T_{12}
T_{21} ✓
T_{22} ✓
T_{31} ✓

2. **Processes and threads.** Suppose that three processes exist in a system, as described table below. Suppose that the system uses preemptive, round-robin scheduling, and that T_{11} is running when the quantum expires.

Process	Threads within the process
P_1	T_{11}, T_{12}, T_{13}
P_2	T_{21}, T_{22}
P_3	T_{31}

Answers

(a) (4) If the threads are implemented entirely at the user level (with no support from the operating system), which threads might possibly execute at the beginning of the next quantum?
T_{21}, T_{22} or T_{31} The OS is unaware of the threads within process P_1

(b) (4) If threads are supported by the operating system (i.e., lightweight processes), which threads might possibly execute at the beginning of the next quantum?
any except T_{11}

(c) (2) How does your answer change if the system has two processors?
It actually doesn't change. In the first case, the kernel will not know about the user's threads, so no other choices can be made. In the second case, the kernel will have full knowledge of the user's threads and will make the same decision (of course, realizing that the OS should not run the same thread on simultaneously on multiple processors).

Question: T/F

- A blocking kernel-scheduled thread blocks all threads in the process
- Threads are cheaper to context switch than processes
- A blocking user-level thread blocks the process

Question: T/F

- A blocking kernel-scheduled thread blocks all threads in the process NO
- Threads are cheaper to context switch than processes True
- A blocking user-level thread blocks the process True

- A blocking kernel-scheduled thread blocks all threads in the process – False
- Threads are cheaper to context switch than processes – True
- A blocking user-level thread blocks the process - True

Question:

P_1

Indicate which statements are **T**rue, and which are **F**alse, regarding a single process and its threads:

38. In an operating system environment that operates with User-Level threads only (no Kernel-Level threads), the various threads associated with a single process all share a common context, including a common PC.

39. In an operating system environment that operates with Kernel-Level threads only (no User-Level threads), the various threads associated with a single process all share a common context, including a common PC.

40. The different threads associated with a single process all share a single process**or** context., *i.e.*, the value of the **P**rocessor **S**tatus **W**ord is the same for all such threads.

41. Thread creation takes more time than process creation.

42. Each thread associated with a single process has exclusive ownership of any files that it has opened; they are not available to other threads associated with the same process.

43. Each thread associated with a single process has a separate text (program code) area.

https://users.cs.jmu.edu/abzugcx/Public/Operating-Systems/Answers-to-Mid-Term.pdf

Item vii: Indicate which statements are **T**rue, and which are **F**alse, regarding a single process and its threads:

2 pts each

38. In an operating system environment that operates with User-Level threads only (no Kernel-Level threads), the various threads associated with a single process all share a common context, including a common PC. *Answer*: **F**alse

39. In an operating system environment that operates with Kernel-Level threads only (no User-Level threads), the various threads associated with a single process all share a common context, including a common PC. *Answer*: **F**alse

40. The different threads associated with a single process all share a single process**or** context., *i.e.,* the value of the **P**rocessor **S**tatus **W**ord is the same for all such threads. *Answer*: **F**alse

41. Thread creation takes more time than process creation. *Answer*: **F**alse

42. Each thread associated with a single process has exclusive ownership of any files that it has opened; they are not available to other threads associated with the same process. *Answer*: **F**alse

43. Each thread associated with a single process has a separate text (program code) area. *Answer*: **F**alse

https://users.cs.jmu.edu/abzugcx/Public/Operating-Systems/Answers-to-Mid-Term.pdf

a. (4 marks)

Suppose that two long running processes, P_1 and P_2, are running in a system. Neither program performs any system calls that might cause it to block, and there are no other processes in the system. P_1 has 2 threads and P_2 has 1 thread.

(i) What percentage of CPU time will P_1 get if the threads are *kernel threads*? Explain your answer.

(ii) What percentage of CPU time will P_1 get if the threads are *user threads*? Explain your answer.

Sample Answer:

(i) If the threads are kernel threads, they are independently scheduled and each of the three threads will get a share of the CPU. Thus, the 2 threads of P_1 will get 2/3 of the CPU time. That is, P_1 will get 66% of the CPU.

(ii) If the threads are user threads, the threads of each process map to one kernel thread, so each *process* will get a share of the CPU. The kernel is unaware that P_1 has two threads. Thus, P_1 will get 50% of the CPU.

GATE CSE 2004 | Question: 11

asked in **Operating System** Sep 19, 2014 • edited Jul 12, 2018 by **kenzou** 21,296 views

44

Consider the following statements with respect to user-level threads and kernel-supported threads

I. context switch is faster with kernel-supported threads

II. for user-level threads, a system call can block the entire process

III. Kernel supported threads can be scheduled independently

IV. User level threads are transparent to the kernel

Which of the above statements are true?

A. (II), (III) and (IV) only
B. (II) and (III) only
C. (I) and (III) only
D. (I) and (II) only

https://gateoverflow.in/1008/gate-cse-2004-question-11

69

Best answer

Answer: (**A**)

I. User level thread switching is faster than kernel level switching. So, (I) is false.

II. is true.

III. is true.

IV. User level threads are transparent to the kernel

In case of Computing transparent means functioning without being aware. In our case user level threads are functioning without kernel being aware about them. So (IV) is actually correct.

GATE CSE 2007 | Question: 17

asked in **Operating System** Sep 22, 2014

16,706 views

32

Consider the following statements about user level threads and kernel level threads. Which one of the following statements is FALSE?

A. Context switch time is longer for kernel level threads than for user level threads.

B. User level threads do not need any hardware support.

C. Related kernel level threads can be scheduled on different processors in a multi-processor system.

D. Blocking one kernel level thread blocks all related threads.

https://gateoverflow.in/1215/gate-cse-2007-question-17

GATE CSE 2011 | Question: 16, UGCNET-June2013-III: 65

asked in **Operating System** Sep 29, 2014 • recategorized Oct 22, 2018 by **Pooja Khatri**

12,935 views

46

A thread is usually defined as a light weight process because an Operating System (OS) maintains smaller data structure for a thread than for a process. In relation to this, which of the following statement is correct?

A. OS maintains only scheduling and accounting information for each thread
B. OS maintains only CPU registers for each thread
C. OS does not maintain virtual memory state for each thread
D. OS does not maintain a separate stack for each thread

92

Best answer

Answer to this question is (**C**).

Many of you would not agree at first So here I explain it how.

OS , on per thread basis, maintains ONLY TWO things : CPU Register state and Stack space. It does not maintain anything else for individual thread. Code segment and Global variables are shared. Even TLB and Page Tables are also shared since they belong to same process.

A. option (A) would have been correct if 'ONLY' word were not there. It NOT only maintains register state BUT stack space also.
B. is obviously FALSE
C. is TRUE as it says that OS does not maintain VIRTUAL Memory state for individual thread which isTRUE
D. This is also FALSE.

57

Best answer

Answer: (**D**)

A. Context switch time is longer for kernel level threads than for user level threads. — This is True, as Kernel level threads are managed by OS and Kernel maintains lot of data structures. There are many overheads involved in Kernel level thread management, which are not present in User level thread management !

B. User level threads do not need any hardware support.— This is true, as User level threads are implemented by Libraries programmably, Kernel does not sees them.

C. Related kernel level threads can be scheduled on different processors in a multi-processor system.— This is true.

D. Blocking one kernel level thread blocks all related threads. — This is false. If it had been user Level threads this would have been true, (In One to one, or many to one model !) Kernel level threads are independent.

GATE CSE 2014 Set 1 | Question: 20

asked in **Operating System** Sep 26, 2014

16,637 views

47

Which one of the following is **FALSE**?

A. User level threads are not scheduled by the kernel.
B. When a user level thread is blocked, all other threads of its process are blocked.
C. Context switching between user level threads is faster than context switching between kernel level threads.
D. Kernel level threads cannot share the code segment.

54

(D) is the answer. Threads can share the Code segments. They have only separate Registers and stack.

User level threads are scheduled by the thread library and kernel knows nothing about it. So, **A** is TRUE.

Best answer

When a user level thread is blocked, all other threads of its process are blocked. So, **B** is TRUE. (With a multi-threaded kernel, user level threads can make non-blocking system calls without getting blocked. But in this option, it is explicitly said 'a thread is blocked'.)

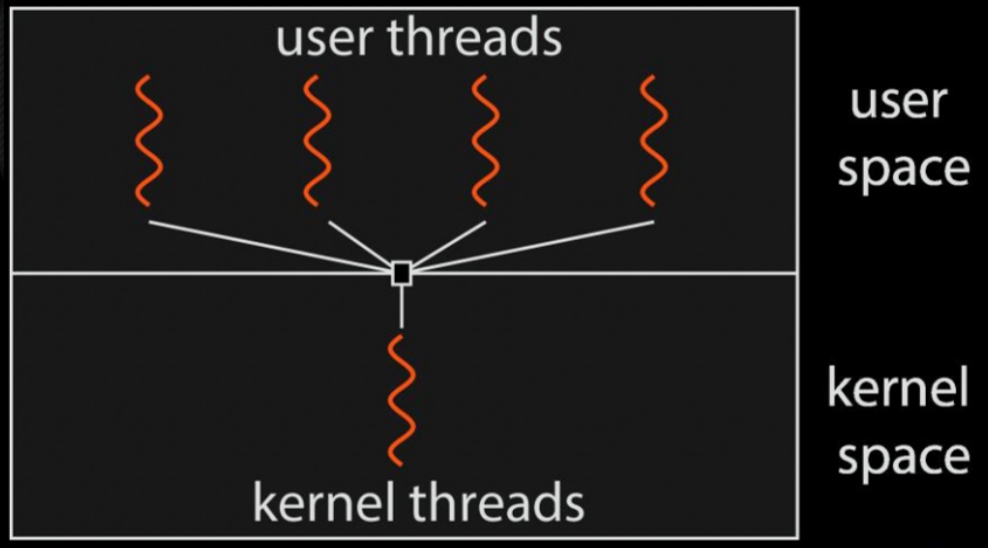
Context switching between user level threads is faster as they actually have no context-switch-nothing is saved and restored while for kernel level thread, Registers, PC and SP must be saved and restored. So, **C** also TRUE.

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

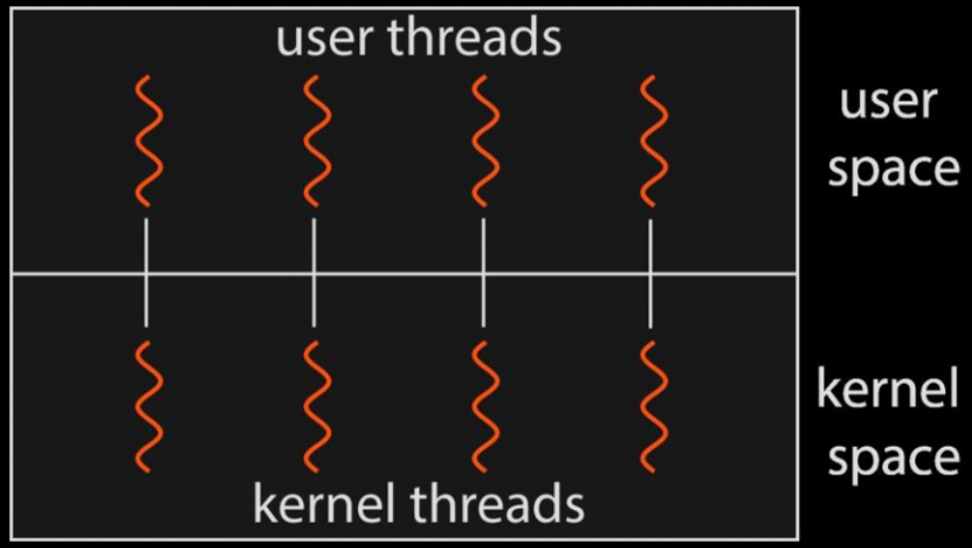
- Many user-level threads mapped to single kernel thread



if we have only user threads then this "many to one" model is by default

One-to-One

- Each thread maps to kernel thread



if you have only kernel level threads then it is a default case.

Many-to-Many

- Allows many user level threads to be mapped to many kernel threads Allows the operating system to create a sufficient number of kernel threads
- not very common