Lecture 7

Topic1: Threads

Topic 2: Context Switch

Topic 3 : Process state diagram

Multithreading Models

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

2.2.6 Hybrid Implementations

Various ways have been investigated to try to combine the advantages of user-level threads with kernel-level threads. One way is use kernel-level threads and then multiplex user-level threads onto some or all of them, as shown in Fig. 2-17.

When this approach is used, the programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one. This model gives the ultimate in flexibility.

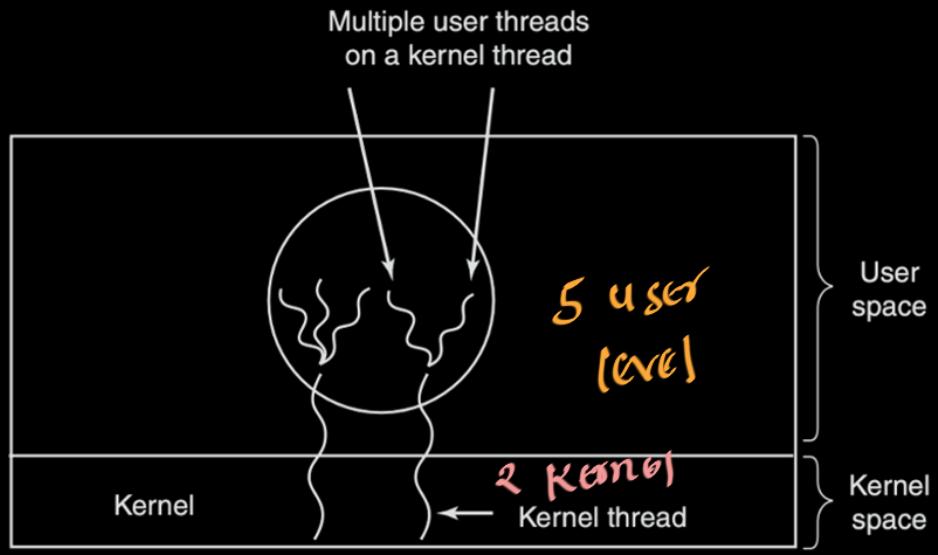
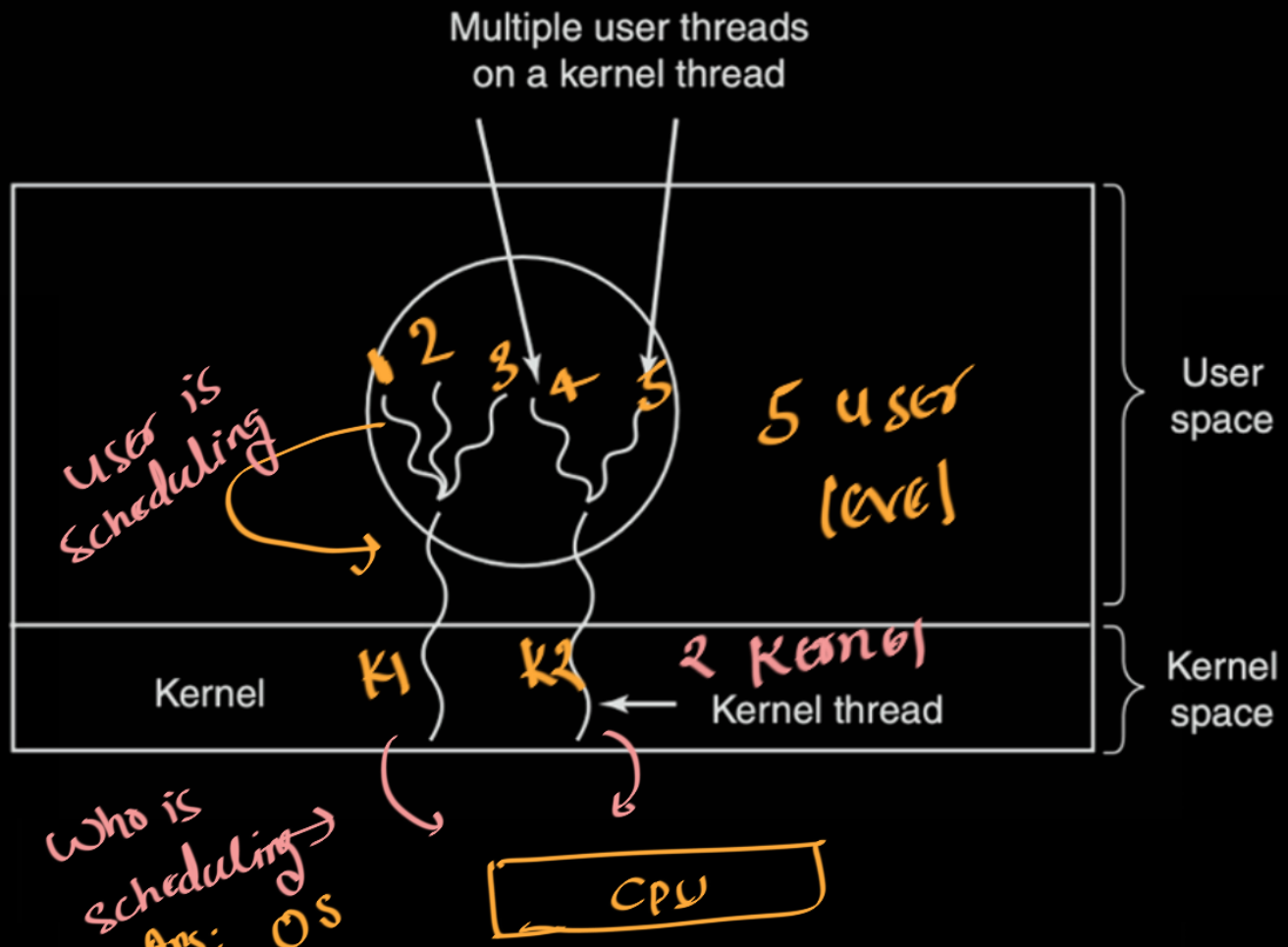


Figure 2-17. Multiplexing user-level threads onto kernel-level threads.

Source: Tanenbaum



User is scheduling

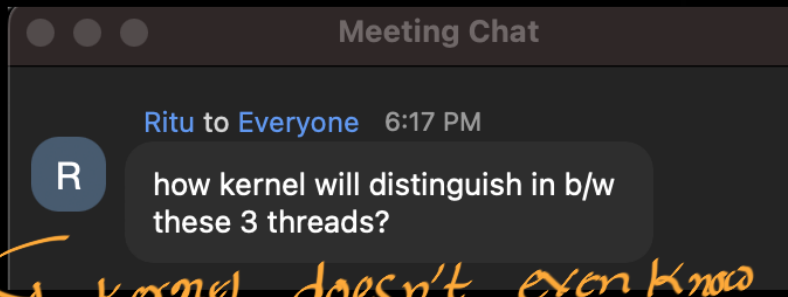
5 user level

2 Kernel

Who is scheduling
Ans: OS

K1 is running on CPU

One of the U1, U2 or U3 is running



Kernel doesn't even know

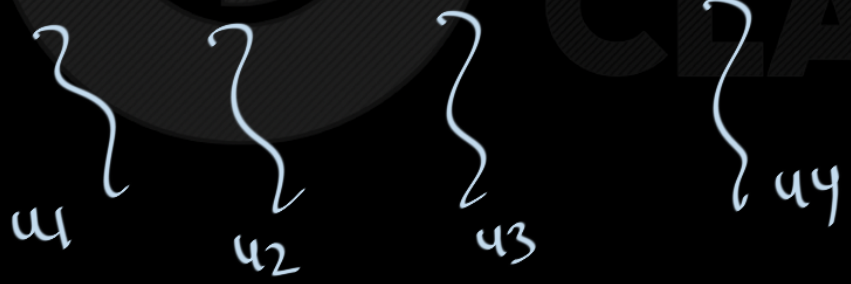
COMBINED APPROACHES Some operating systems provide a combined ULT/KLT facility (see Figure 4.5c). In a combined system, thread creation is done completely in user space, as is the bulk of the scheduling and synchronization of threads within an application. The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs. The programmer may adjust the number of KLTs for a particular application and processor to achieve the best overall results.

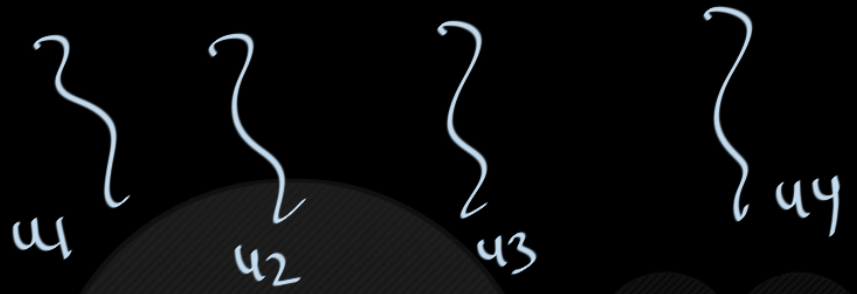
In a combined approach, multiple threads within the same application can run in parallel on multiple processors, and a blocking system call need not block the entire process. If properly designed, this approach should combine the advantages of the pure ULT and KLT approaches while minimizing the disadvantages.

Source: William Stallings

GO CLASSES

Suppose we only have user level threads in a Program (No Kernel level thread)

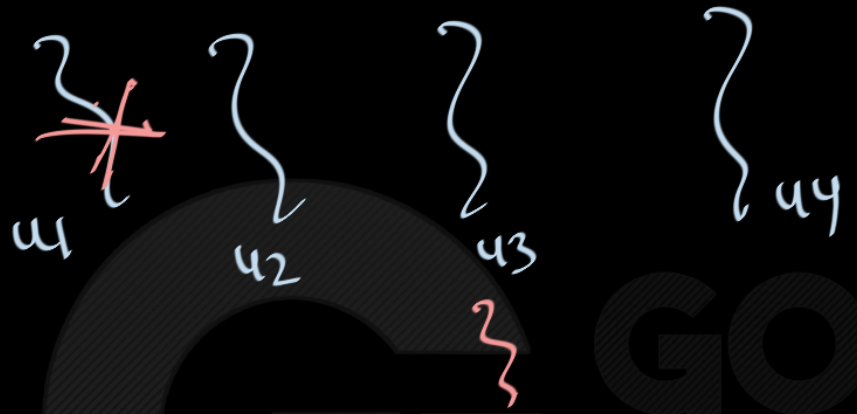




Suppose : we want to run u_3 even if u_4 is blocked then

a) we will map u_1, u_3 to one Kernel level thread

b) " " " different " "



Suppose : we want to run u_3 even if u_1 is blocked then

a) we will map u_1, u_3 to one Kernel level thread

b) " " " different " "

why we even mapping 2 user level threads to one kernel thread

(in other words, why we are not mapping every user level Thread to different kernel level thread ?)

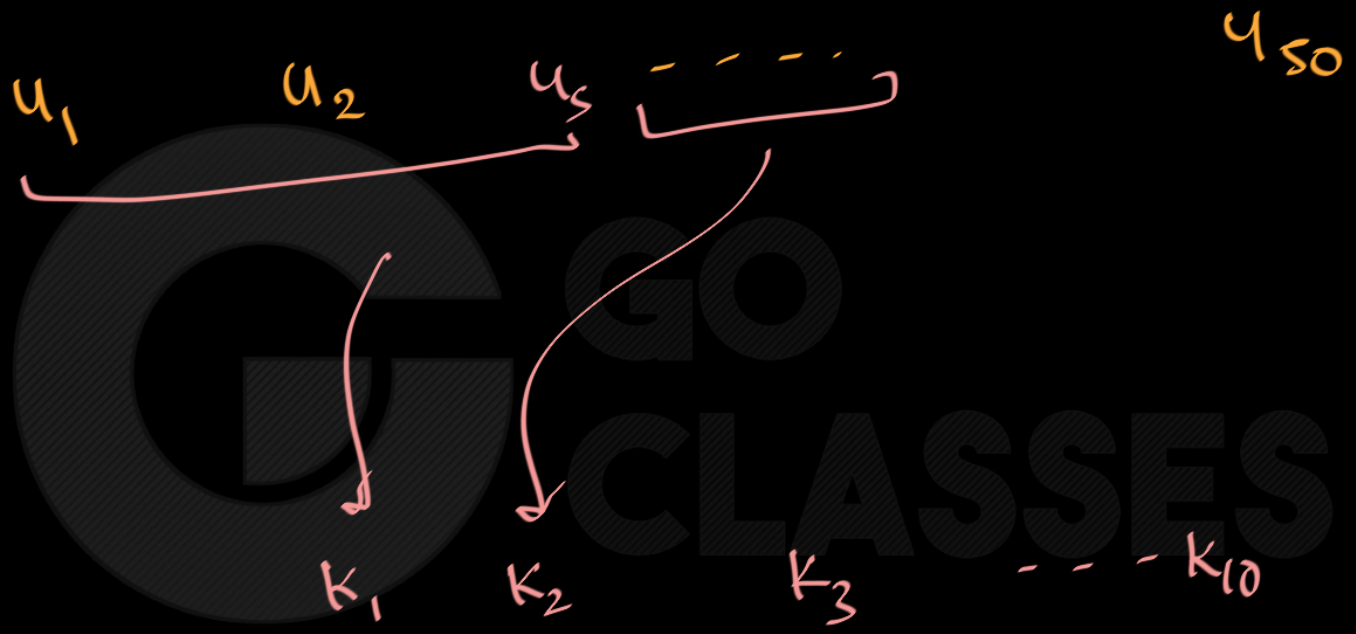
GO CLASSES

why we even mapping 2 user level threads to one kernel thread

(in other words, why we are not mapping every user level Thread to different kernel level thread ?)

⇒ it is more overhead for OS to mange.


many user threads
kernel thread
in which case this might not be desired?
u1
u2
u3
u4
when i want u3 to run even if u1 is blocked.


u_1
u_2
u_5
u_{50}
k_1
k_2
k_3
k_{10}

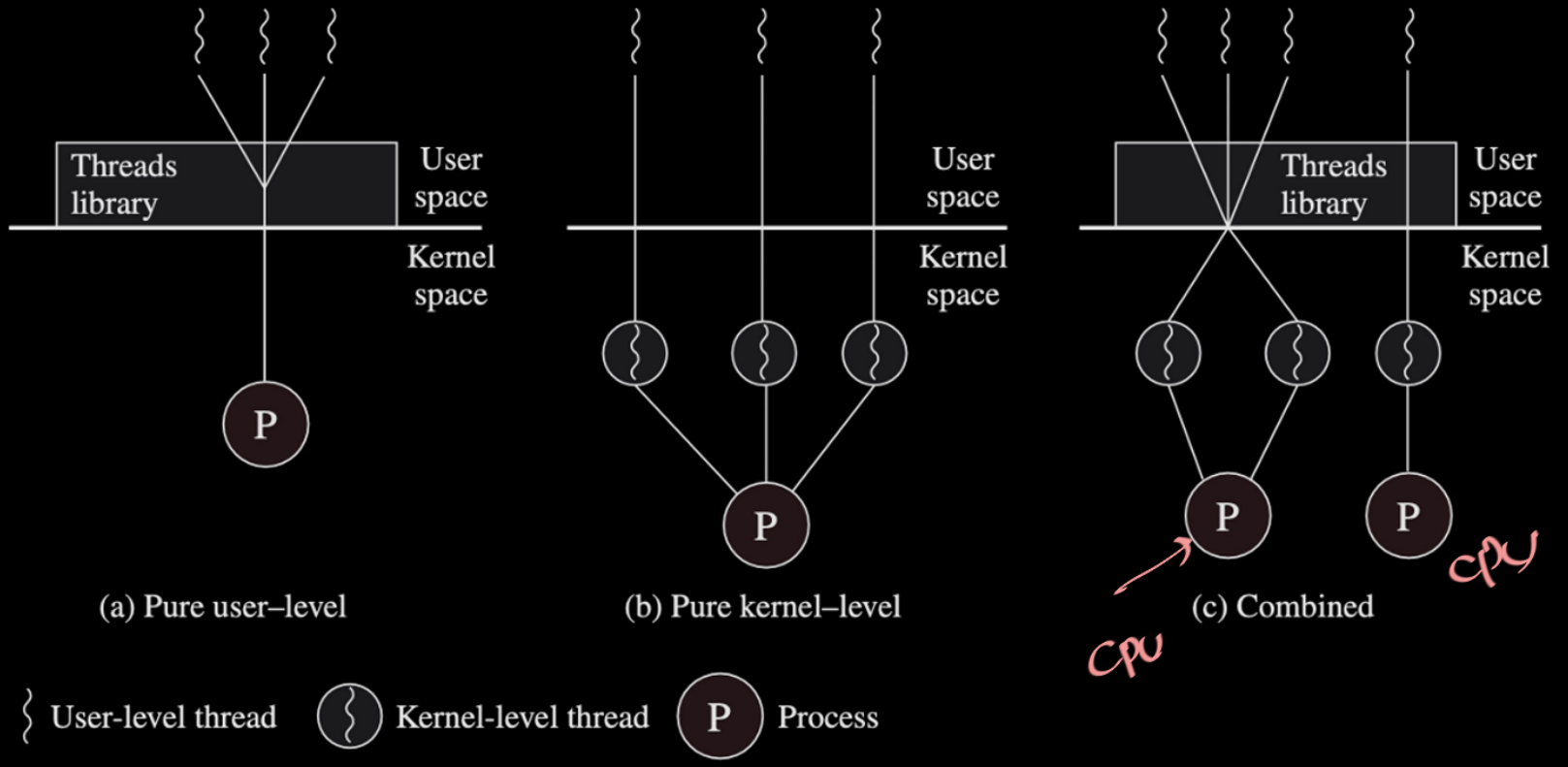


Figure 4.5 **User-Level and Kernel-Level Threads**

Multithreading Models

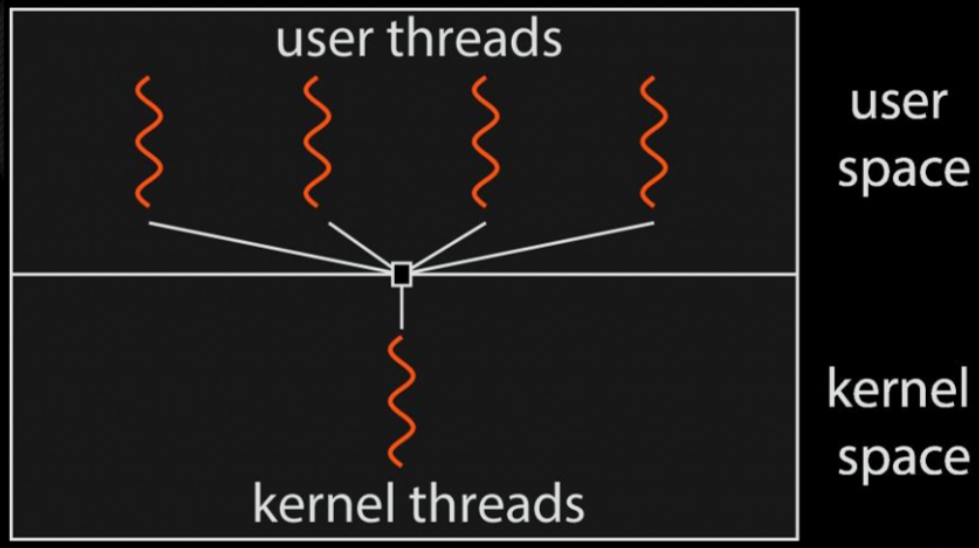
Our discussion so far has treated threads in a generic sense. However, support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**. User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, Mac OS X, and Solaris—support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads. In this section, we look at three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to-many model.

Source: Galvin

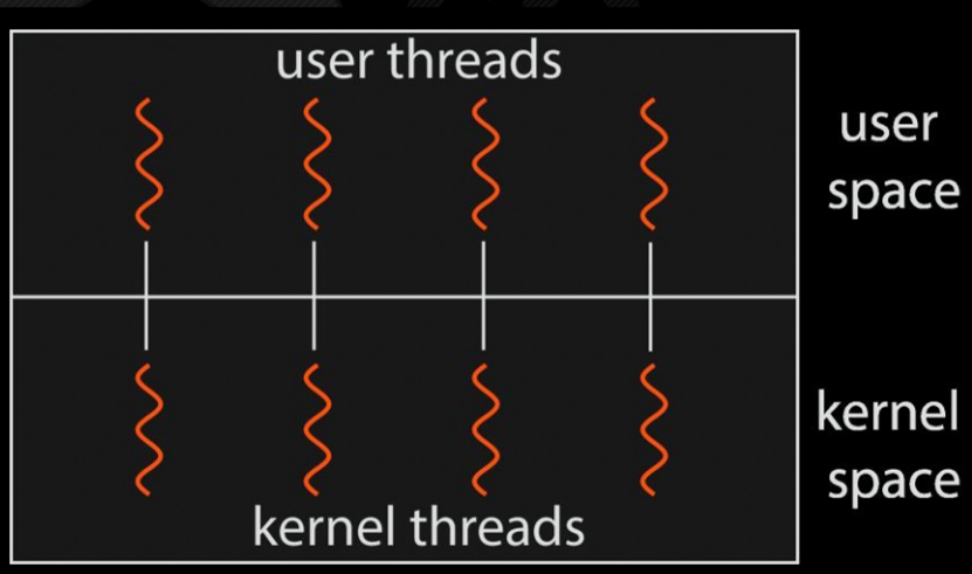
Many-to-One

- Many user-level threads mapped to single kernel thread



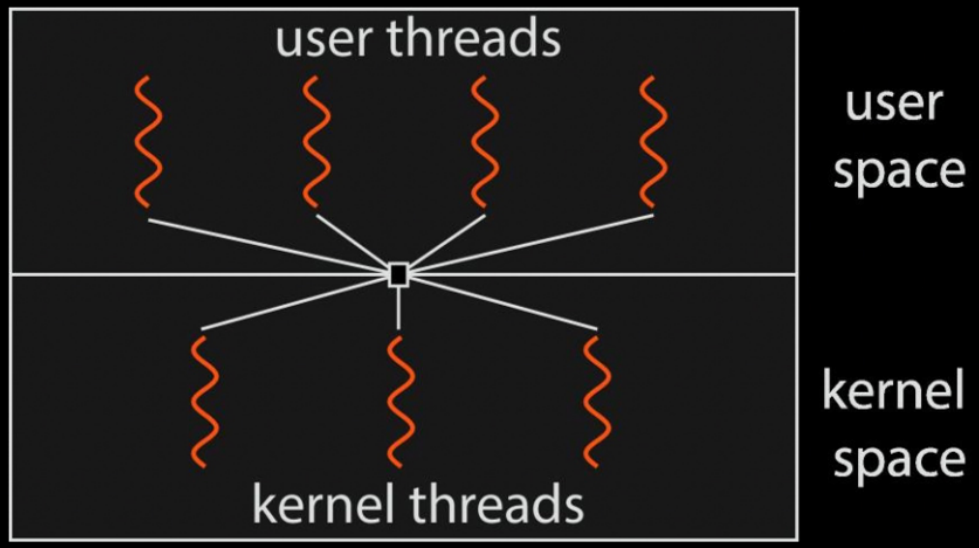
One-to-One

- Each user-level thread maps to kernel thread

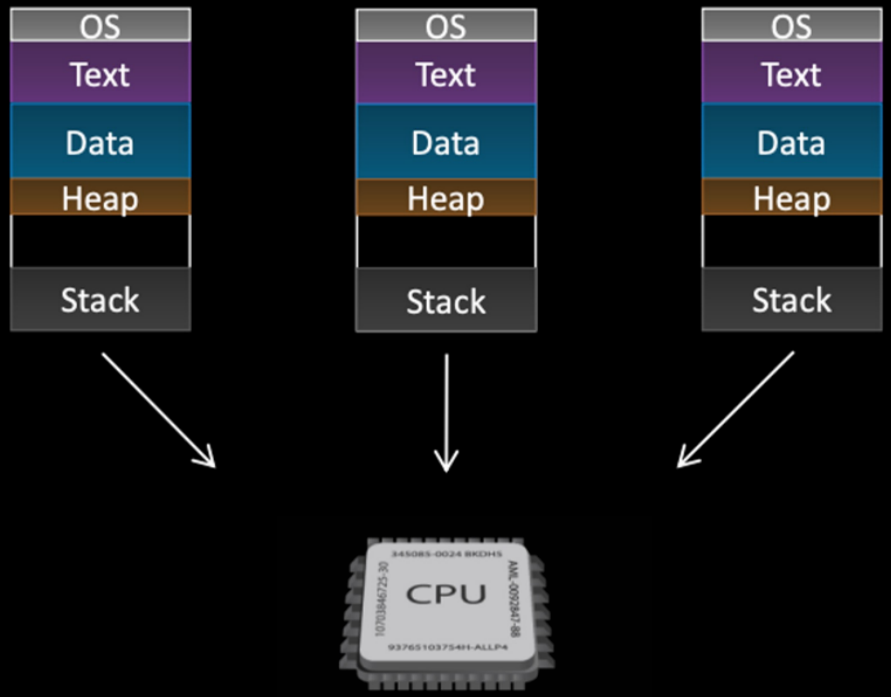


Many-to-Many

- 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



New Topic:
Context Switching


OS
Text
Data
Heap
Stack
OS
Text
Data
Heap
Stack
OS
Text
Data
Heap
Stack
CPU

Program 1

```
mov     r1, #100
mov     r2, #200
add     r3, r2, r1
mov     r1, #111
mov     r2, #222
sub     r4, r3, #1
...
```

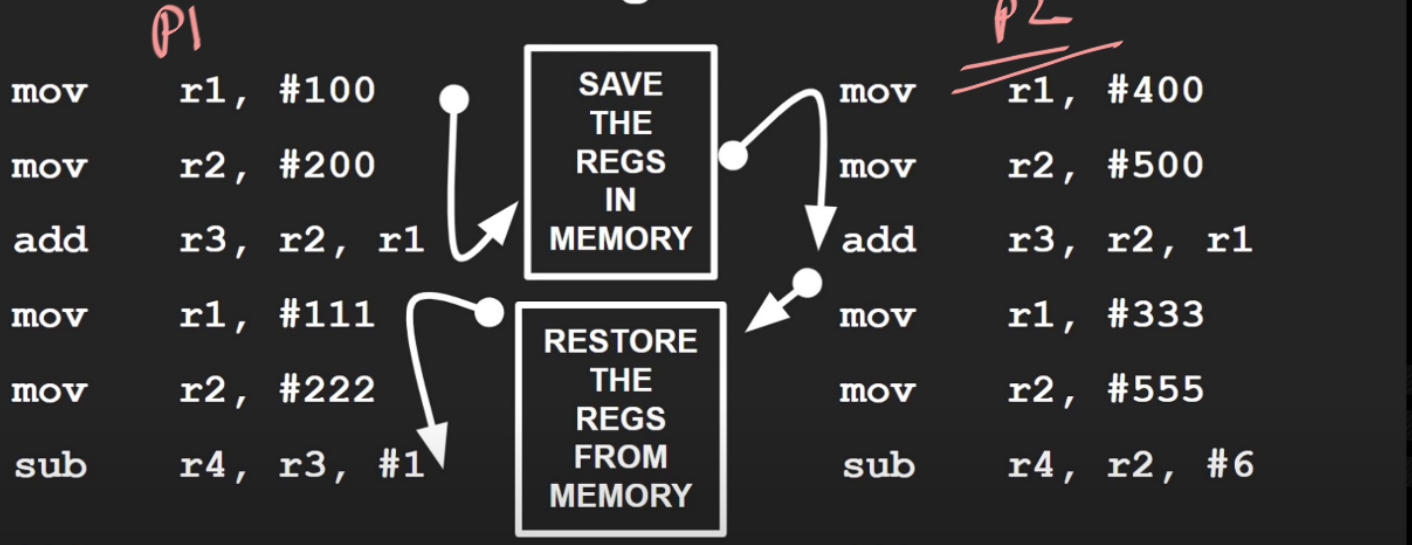
Program 2

```
mov     r1, #400
mov     r2, #500
add     r3, r2, r1
mov     r1, #333
mov     r2, #555
sub     r4, r2, #6
...
```

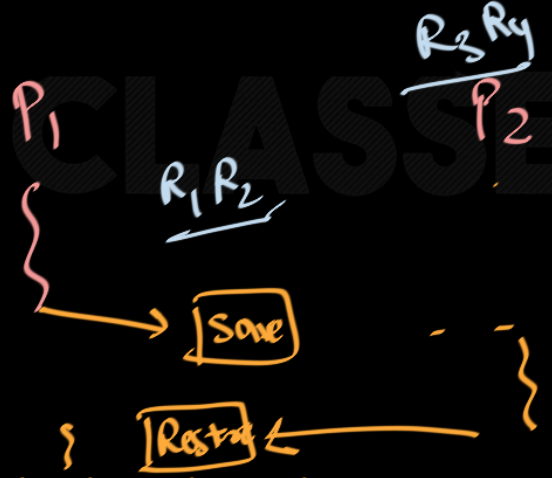
100 R1

200 R2

What if we save the register values?



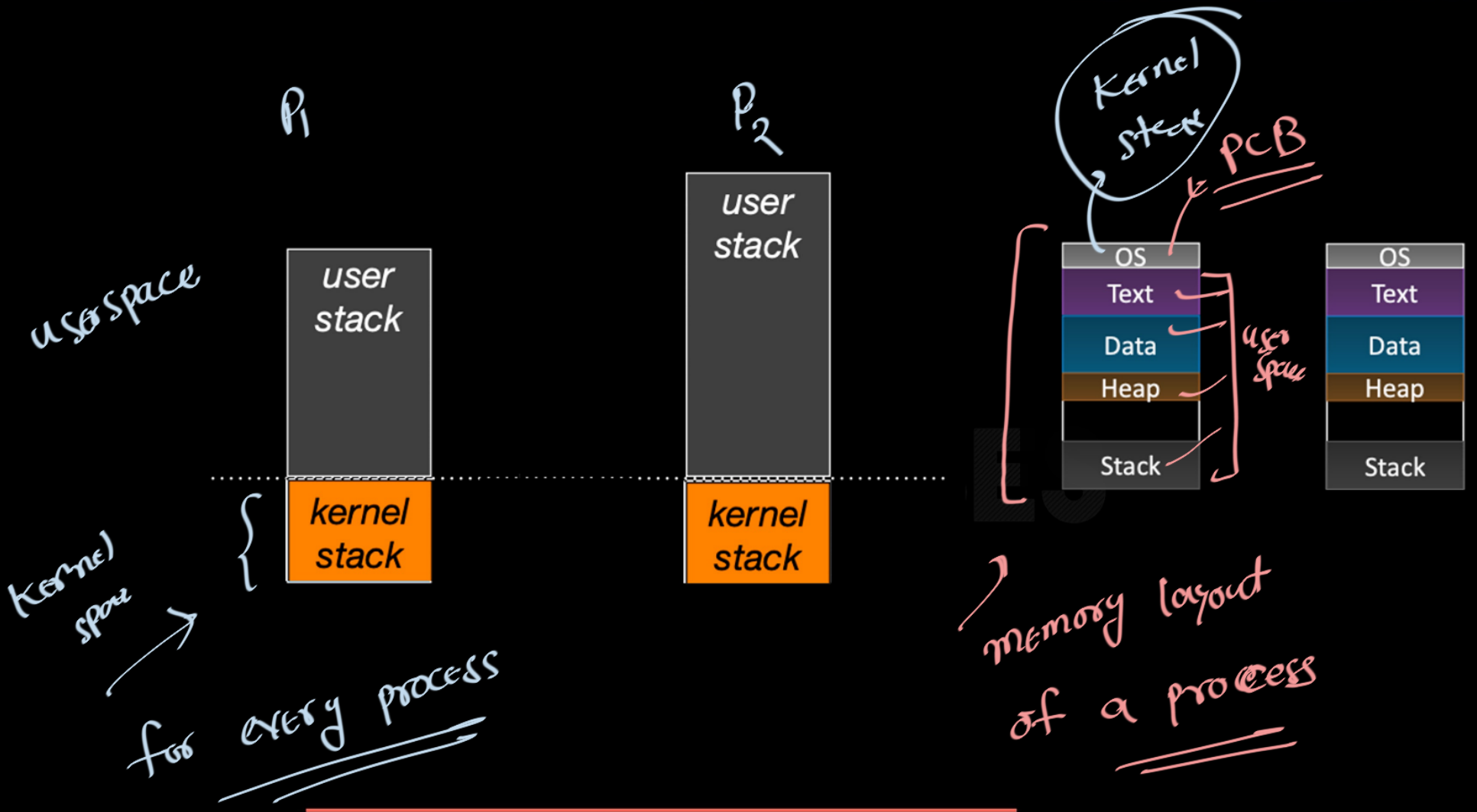
- Context switching
  - Saves state of a process before a switching to another process
  - Restores original process state when switching back



https://www.ccs.neu.edu/home/skotthe/classes/cs5600/fall/2015/notes/Processes.pdf

- Simple concept, but:
  - How do you save the state of a process?
  - How do you stop execution of a process?
  - How do you restart the execution of process that has been switched out?

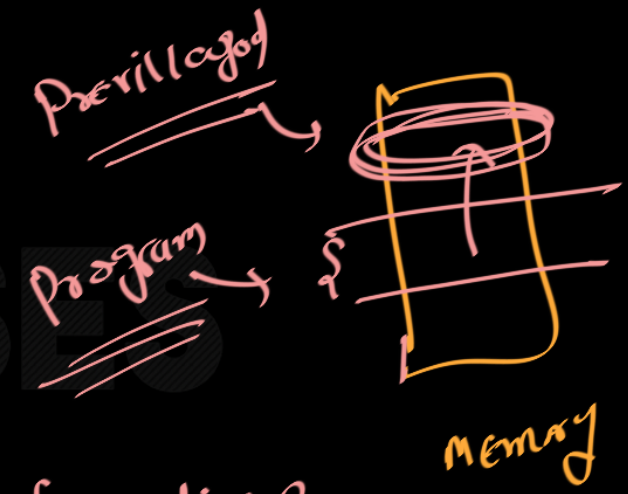
https://www.ccs.neu.edu/home/skotthe/classes/cs5600/fall/2015/notes/Processes.pdf


P1
P2
user stack
user stack
user space
kernel stack
kernel stack
kernel space
for every process
Kernel stack
PCB
OS
Text
Data
Heap
Stack
user space
OS
Text
Data
Heap
Stack
memory layout of a process

every process has some information to store

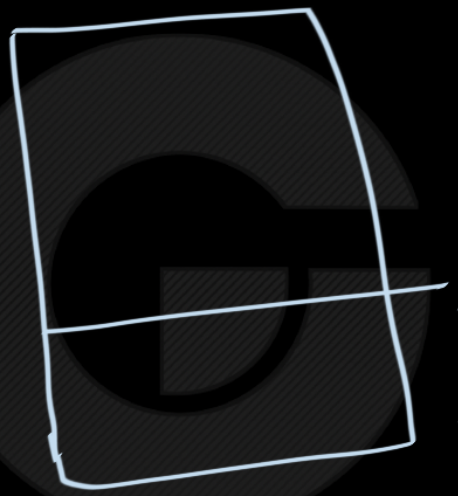
→ PCB

→ Registers in use



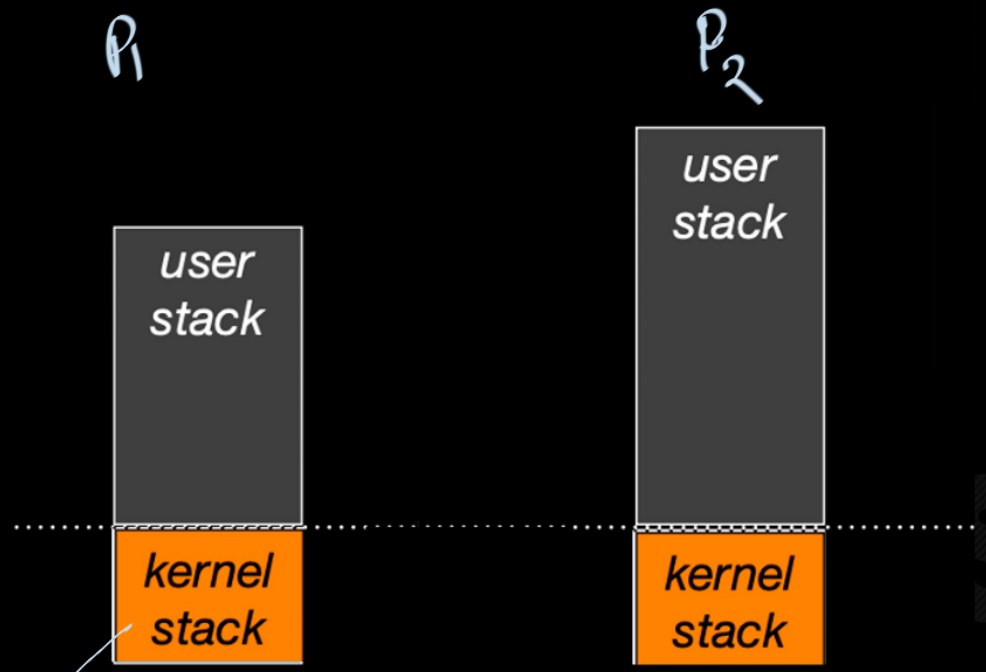
∴ where do we store this information?

⇒ in memory (privileged area of memory)

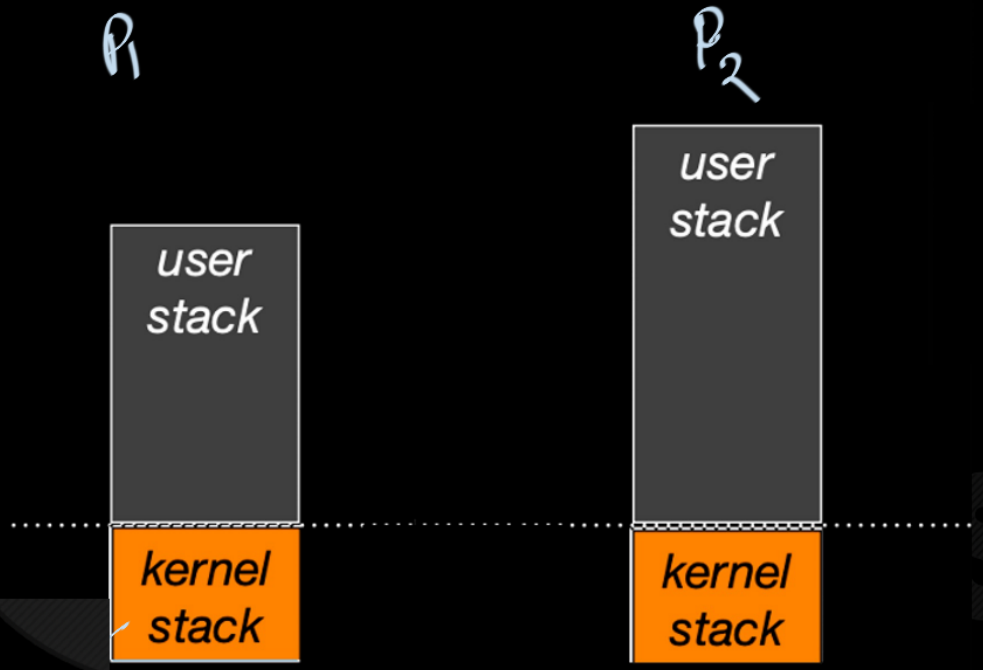


for every Process, i want to store something in the previllaged memory

Kernel stack → { → in that memory i am having some stack



we can push the registers value here at time of context switching.



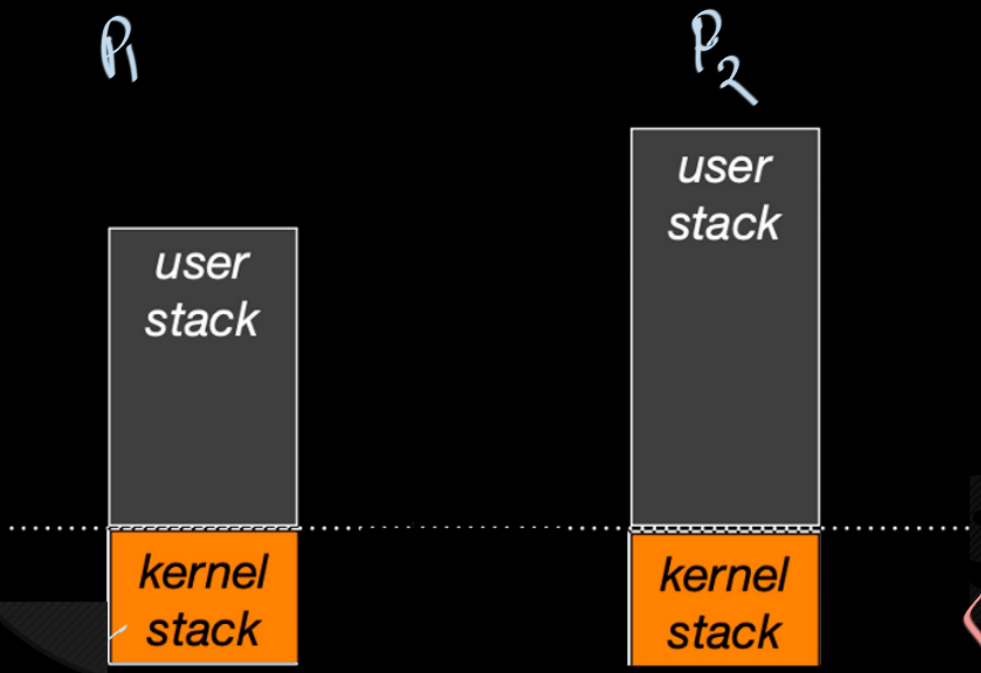
in user mode.

↳ P_1 is running

→ P_1 timer is over

→ P_1 wants to leave cpu by choice

→ P1 wants to go in block state because of I/O



in user mode.

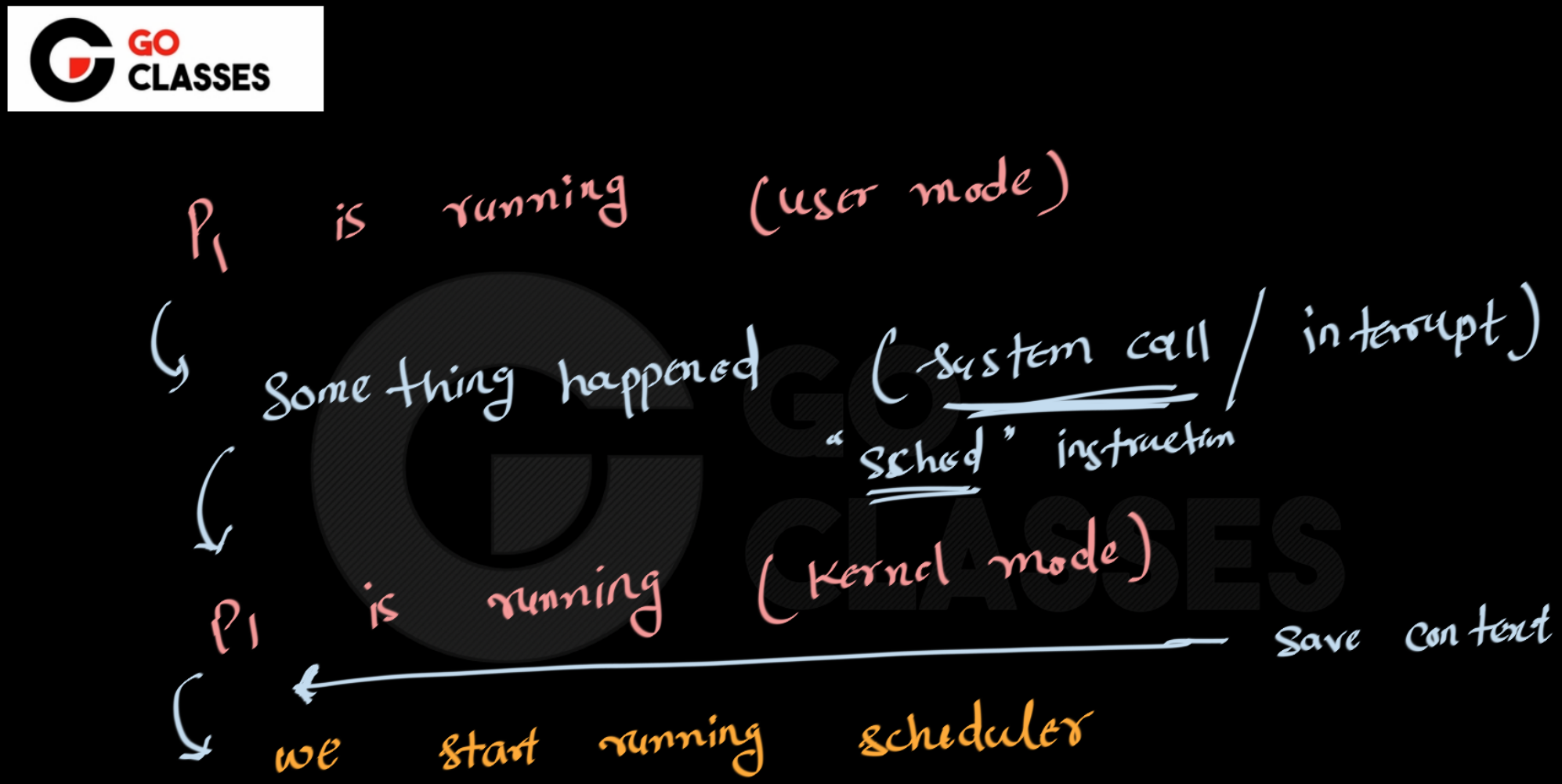
↳ P_1 is running

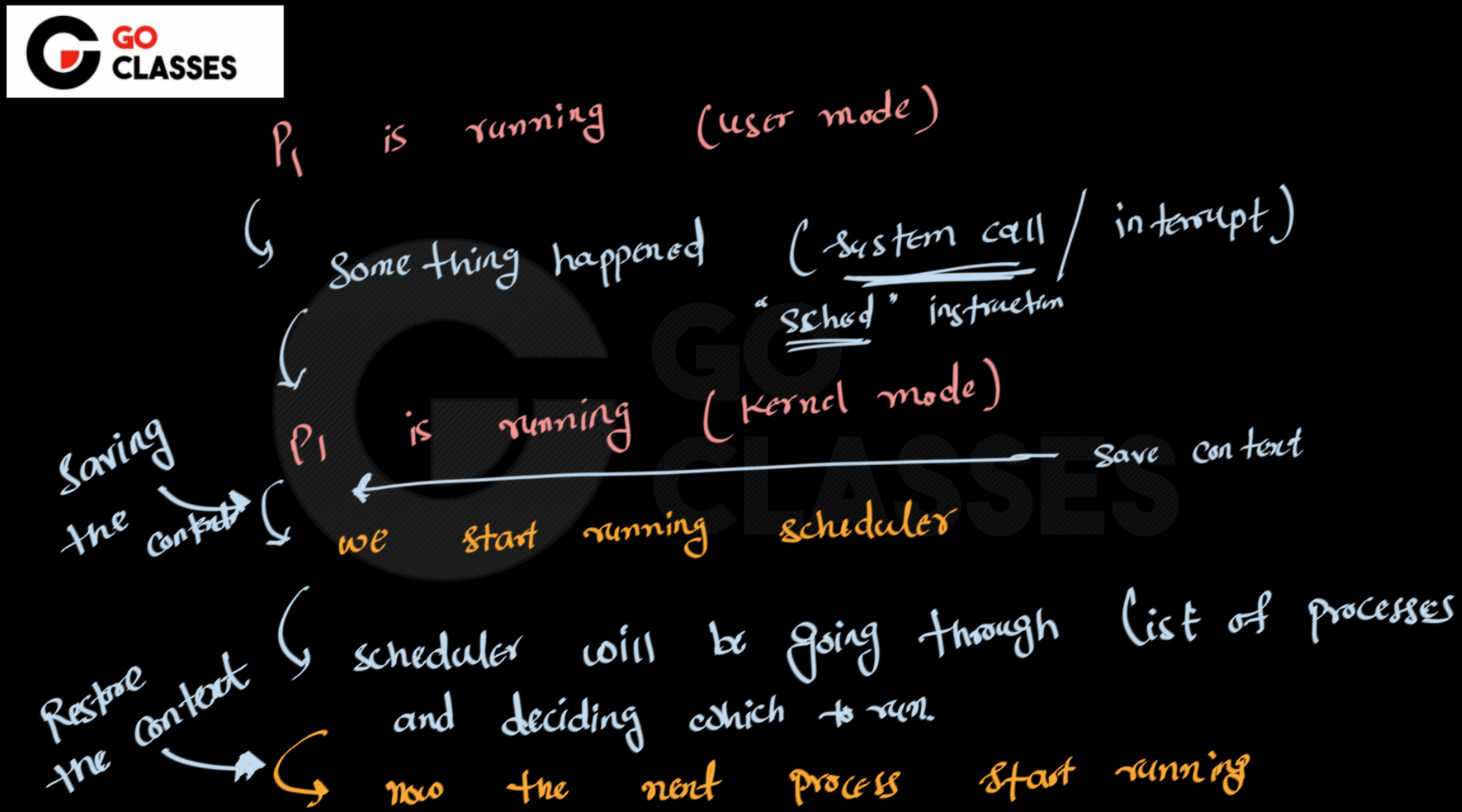
→ P_1 timer is over

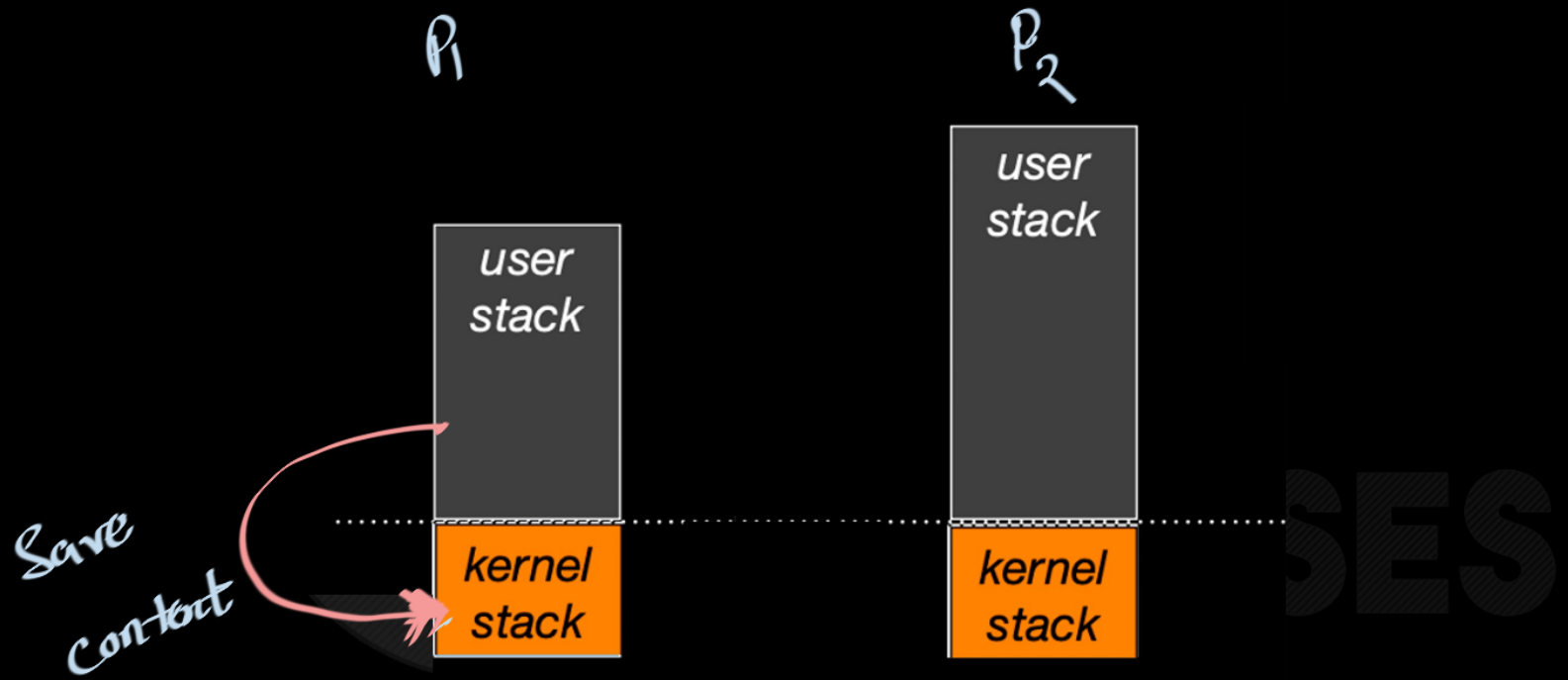
→ P_1 wants to leave cpu by choice

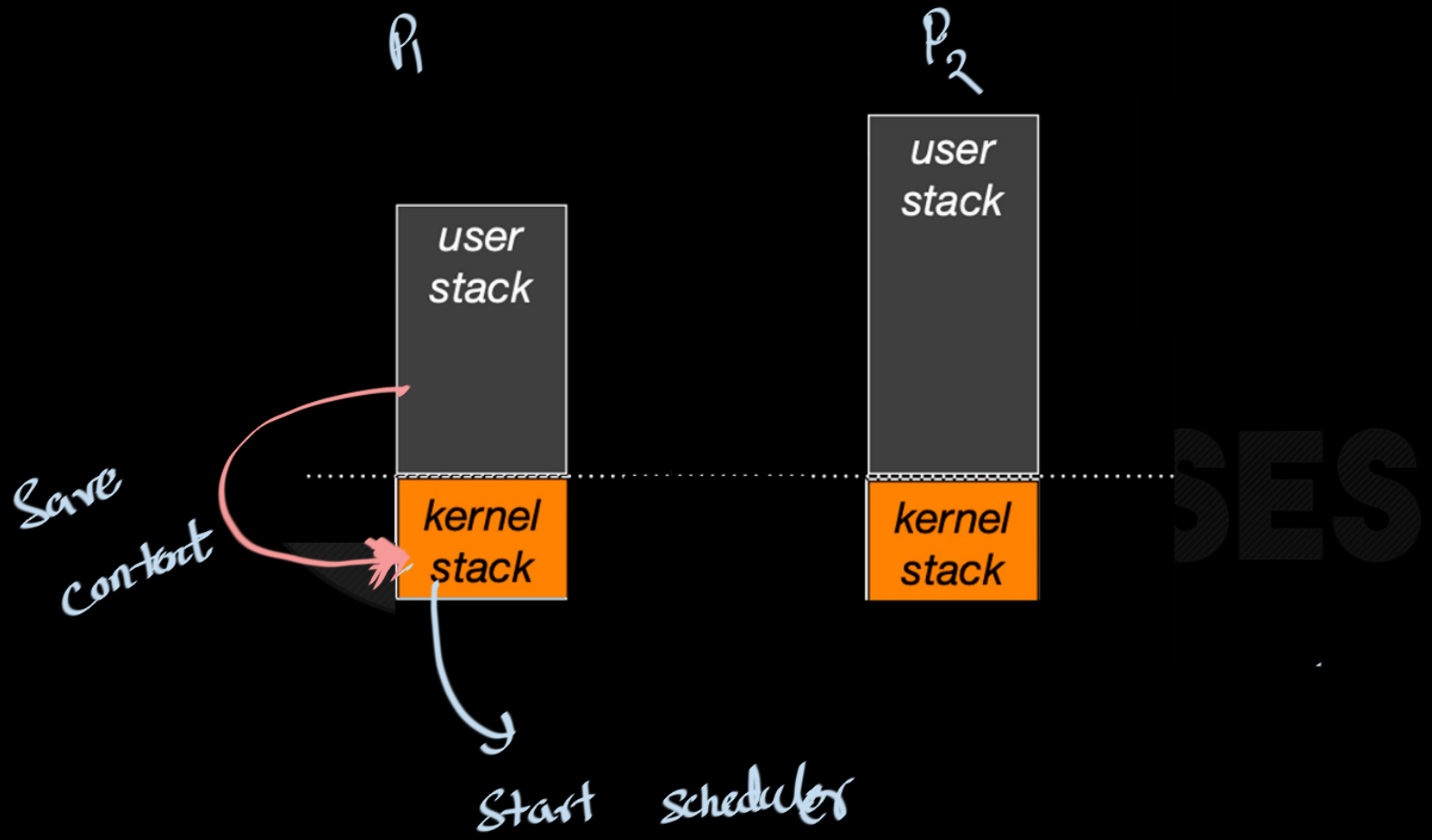
→ P_1 wants to go in block state because of I/O

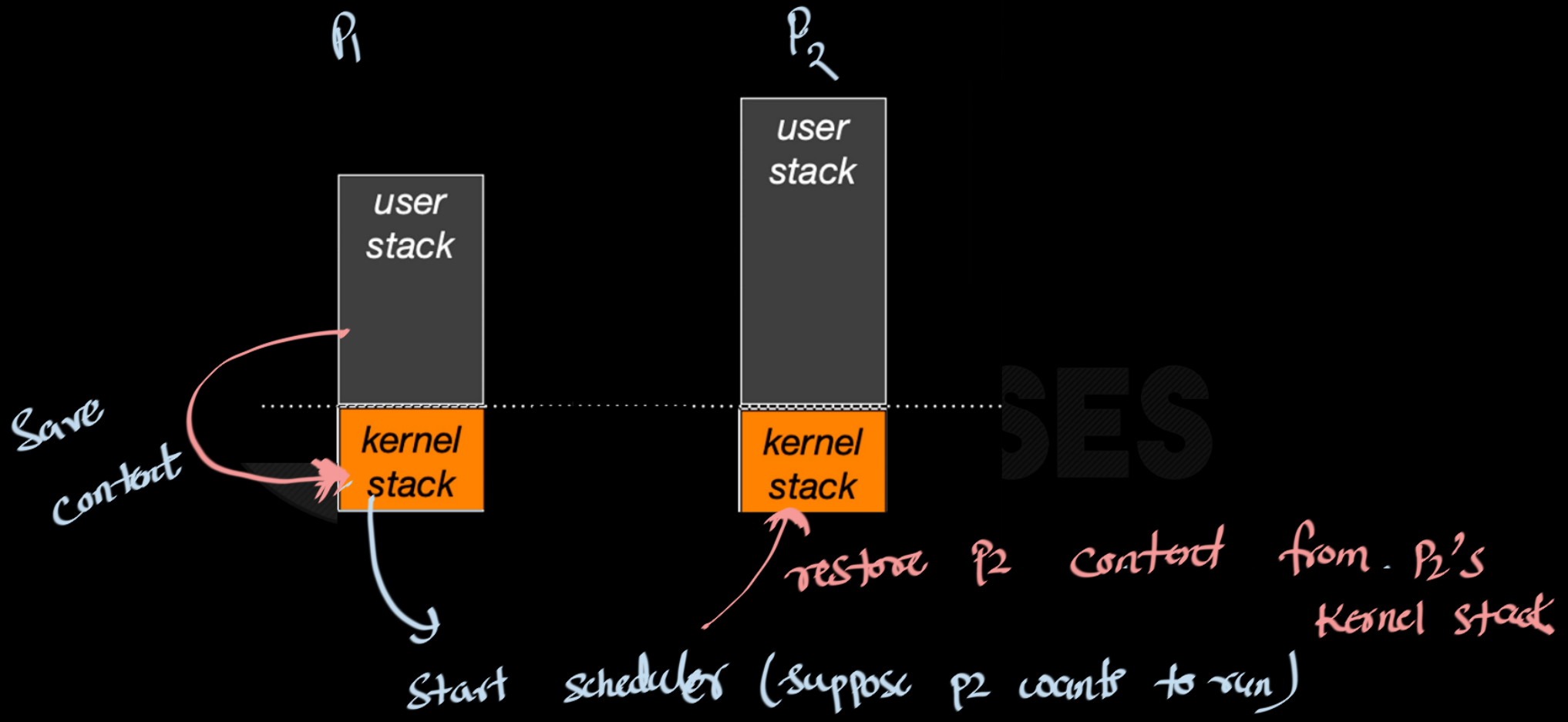
Becoz of one of this reason (system call / interrupt) P_1 will be in Kernel mode

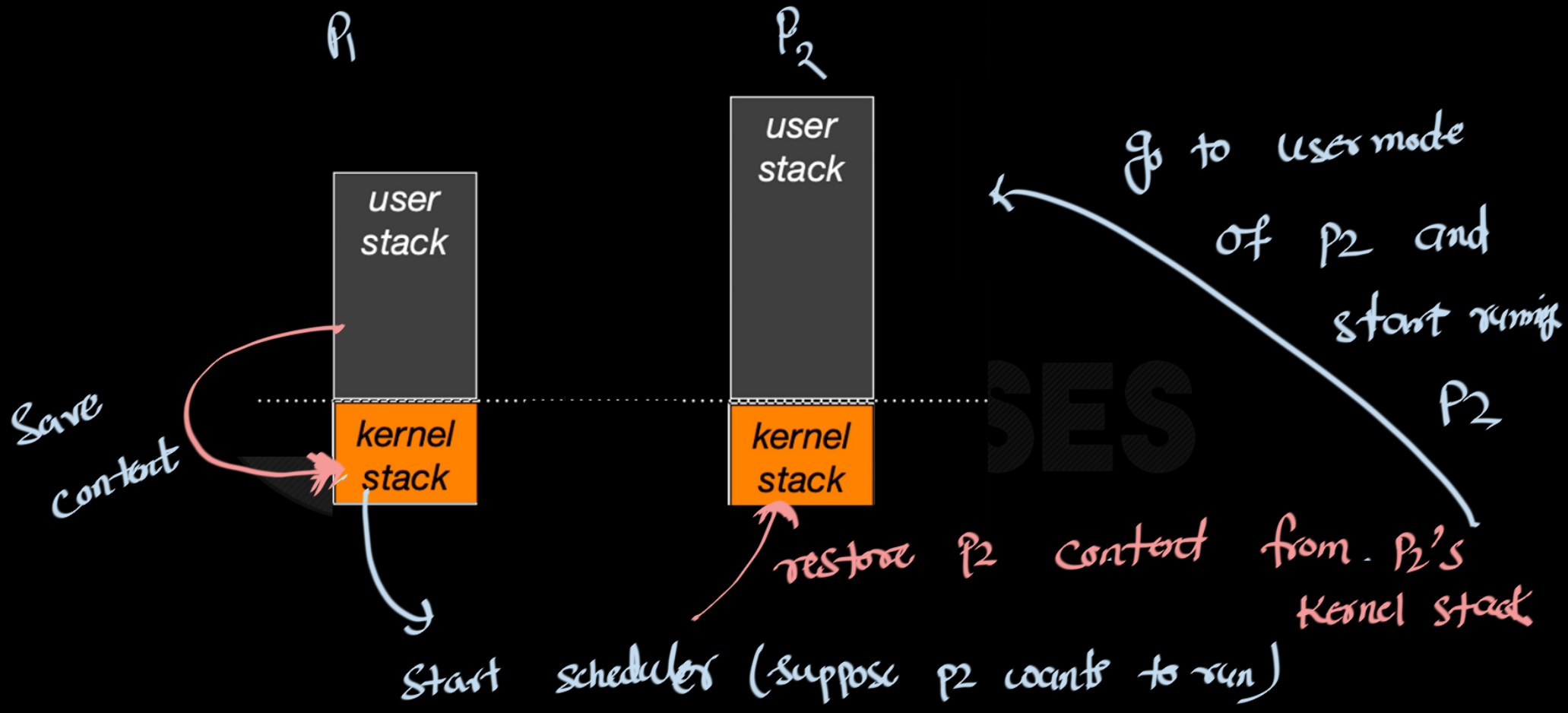

GO CLASSES
P1 is running (user mode)
Something happened (system call / interrupt)
"sched" instruction
P1 is running (kernel mode)
Save context
we start running scheduler


GO CLASSES
P1 is running (user mode)
Something happened (system call / interrupt)
"sched" instruction
P1 is running (kernel mode)
Saving the context
save context
we start running scheduler
Restore the context
scheduler will be going through list of processes
and deciding which to run.
now the next process start running


P1
P2
user stack
user stack
kernel stack
kernel stack
Save
Context


P1
P2
user stack
user stack
kernel stack
kernel stack
Save
Context
Start
scheduler


P1
P2
user stack
user stack
kernel stack
kernel stack
Save context
restore P2 context from P2's kernel stack
Start scheduler (suppose P2 wants to run)


P1
P2
user stack
user stack
kernel stack
kernel stack
Save context
go to user mode of P2 and start running P2
restore P2 context from P2's kernel stack
Start scheduler (suppose P2 wants to run)

Context switching

At a low level, xv6 performs two kinds of context switches: from a process's kernel thread to the current CPU's scheduler thread, and from the scheduler thread to a process's kernel thread. xv6 never directly switches from one user-space process to another; this happens by way of a user-kernel transition (system call or interrupt), a context switch to the scheduler, a context switch to a new process's kernel thread, and a trap return. In this section we'll example the mechanics of switching between a kernel thread and a scheduler thread.

https://www.cs.columbia.edu/~junfeng/11sp-w4118/lectures/sched.pdf

P1 User mode

P1 Kernel mode (i want to save P1 content in previlleged area)

Scheduler

P2 Kernel mode (i want to restore P2 context)

P2 User mode

Question : T/F

(Based on the System that we discussed)

Context switch only happens when process is already in kernel mode.

Question : T/F

(Based on the System that we discussed)

Context switch only happens when process is already in kernel mode.

TRUE

Context Switch Details

- Every System has a scheduler process
- Scheduler goes over list of processes and switches to one of the runnable ones
- Context switch only happens when process is already in kernel mode.

  Example: P1 running, timer interrupt,
  P1 moves to kernel mode, switches to scheduler thread,
  scheduler switches to P2, P2 returns to user mode

Summary of context switching

- What happens during context switch from process P1 to P2?
  - P1 goes to kernel mode and gives up CPU (timer interrupt or exit or sleep)
  - P2 is another process that is ready to run (it had given up CPU after saving context on its kernel stack in the past, but is now ready to run)
  - P1 switches to CPU scheduler thread
  - Scheduler thread finds runnable process P2 and switches to it
  - P2 returns from trap to user mode
- Process of switching from one process/thread to another
  - Save all register state (CPU context) on kernel stack of old process
  - Update context structure pointer of old process to this saved context
  - Switch from old kernel stack to new kernel stack
  - Restore register state (CPU context) from new kernel stack, and resume new process

Thread context switch Vs. process context switch

Thread context switch Vs. process context switch

Asked 12 years, 4 months ago Modified 3 years ago Viewed 110k times

155

Could any one tell me what is exactly done in both situations?
What is the main cost each of them?

multithreading process context-switch

238

The main distinction between a thread switch and a process switch is that during a thread switch, the virtual memory space remains the same, while it does not during a process switch. Both types involve handing control over to the operating system kernel to perform the context switch. The process of switching in and out of the OS kernel along with the cost of switching out the registers is the largest fixed cost of performing a context switch.

A more fuzzy cost is that a context switch messes with the processors cacheing mechanisms. Basically, when you context switch, all of the memory addresses that the processor "remembers" in its cache effectively become useless. The one big distinction here is that when you change virtual memory spaces, the processor's Translation Lookaside Buffer (TLB) or equivalent gets flushed making memory accesses much more expensive for a while. This does not happen during a thread switch.

Share Follow

edited Mar 22, 2019 at 2:05
Nan Xiao
16.6k • 18 • 102 • 164

answered Mar 26, 2011 at 3:18
Abhay Buch
4,528 • 1 • 21 • 26

Few Questions

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}

(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?

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}

(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:

- 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

- 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:

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.

New Topic:
Process States