CMSC421: Principles of Operating Systems

Nilanjan Banerjee

Assistant Professor, University of Maryland
Baltimore County
nilanb@umbc.edu
http://www.csee.umbc.edu/~nilanb/teaching/421/

Acknowledgments: Some of the slides are adapted from Prof. Mark Corner and Prof. Emery Berger's OS course at Umass Amherst

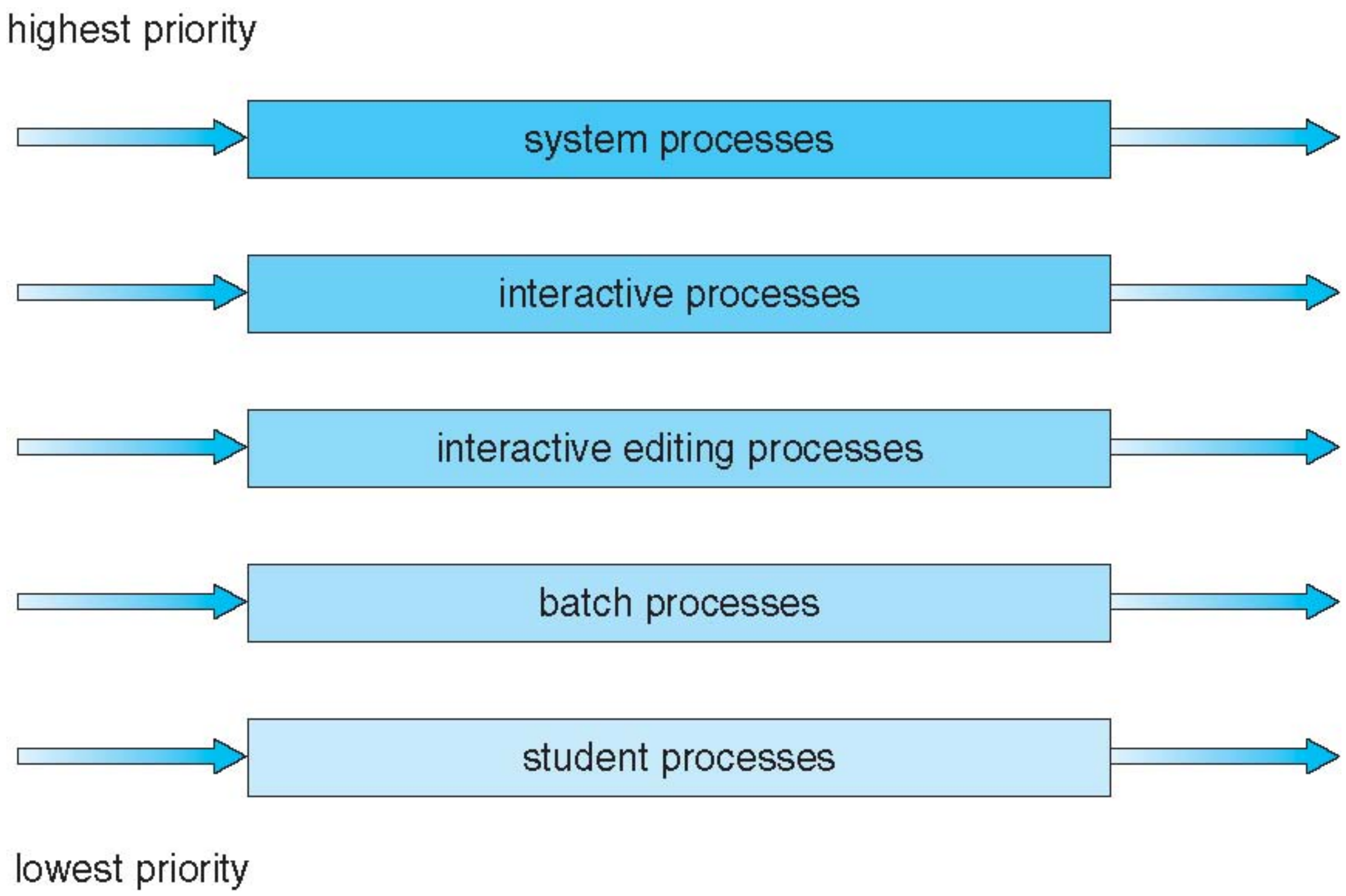
Announcements

- Midterm (29th of October in class)
- Project 2 is out (there are several submission dates)
- Readings from Silberchatz [5th chapter]
- Towards the end of class today--- I will talk about project 2

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - foreground (interactive)
  - background (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling



Multilevel Feedback Queue

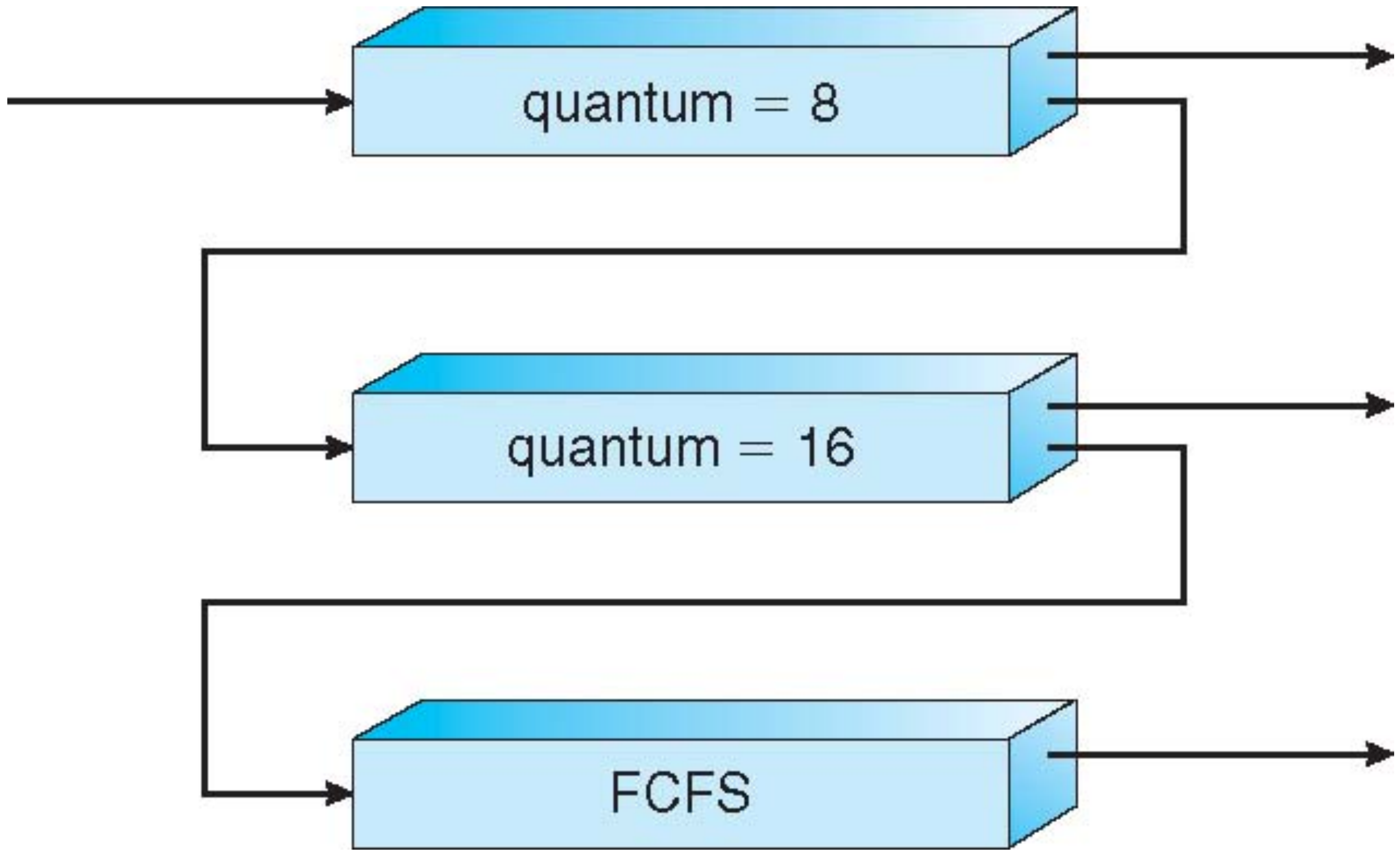
- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

Multilevel Feedback Queues



Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
   int i;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* set the scheduling algorithm to PROCESS or
   SYSTEM */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* set the scheduling policy - FIFO, RT, or OTHER */
   pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
   /* create the threads */
   for (i = 0; i < NUM THREADS; i++)
       pthread create(&tid[i],&attr,runner,NULL);
```

Pthread Scheduling API

```
   /* now join on each thread */
   for (i = 0; i < NUM THREADS; i++)
       pthread_join(tid[i], NULL);
}
 /* Each thread will begin control in this
   function */
void *runner(void *param)
{
   printf("I am a thread\n");
   pthread exit(0);
}
```

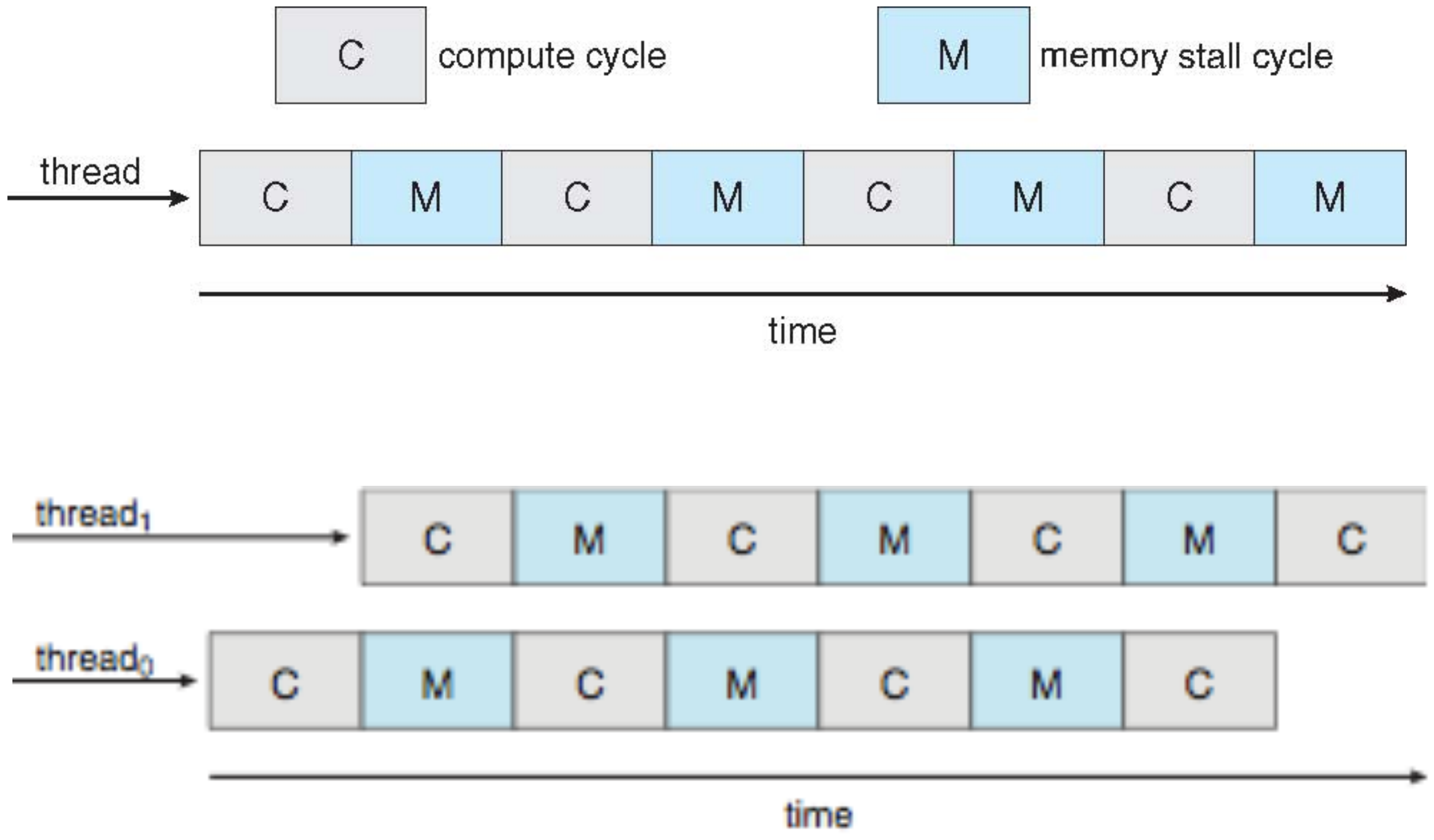
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**

Multicore Processors (Scheduling is an open problem)

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

Multithreaded Multicore System (Heuristic)



Linux Kernel Scheduler (CFS - completely Fair)

- CFS a very simple scheduler
- Intuition behind the scheduler “ideal, precise multitasking CPU” – one that could run multiple processes simultaneously, giving each equal processing power.

CFS scheduling

- CFS measures how much runtime each task has had and try and ensure that everyone gets their fair share of time.

- This value is held in the **vruntime** variable for each task, and is recorded at the nanosecond level. A **lower** vruntime indicates that the task has had less time to compute, and therefore has more need of the processor.

- Furthermore, instead of a queue, CFS uses a Red-Black tree to store, sort, and schedule tasks.

Red Black Trees

- A red-black tree is a binary search tree, which means that for each node, the left subtree only contains keys less than the node's key, and the right subtree contains keys greater than or equal to it.

- A red-black tree has further restrictions which guarantee that the longest root-leaf path is at most twice as long as the shortest root-leaf path. This bound on the height makes RB Trees more efficient than normal BSTs.

- Operations are in O(log n) time.

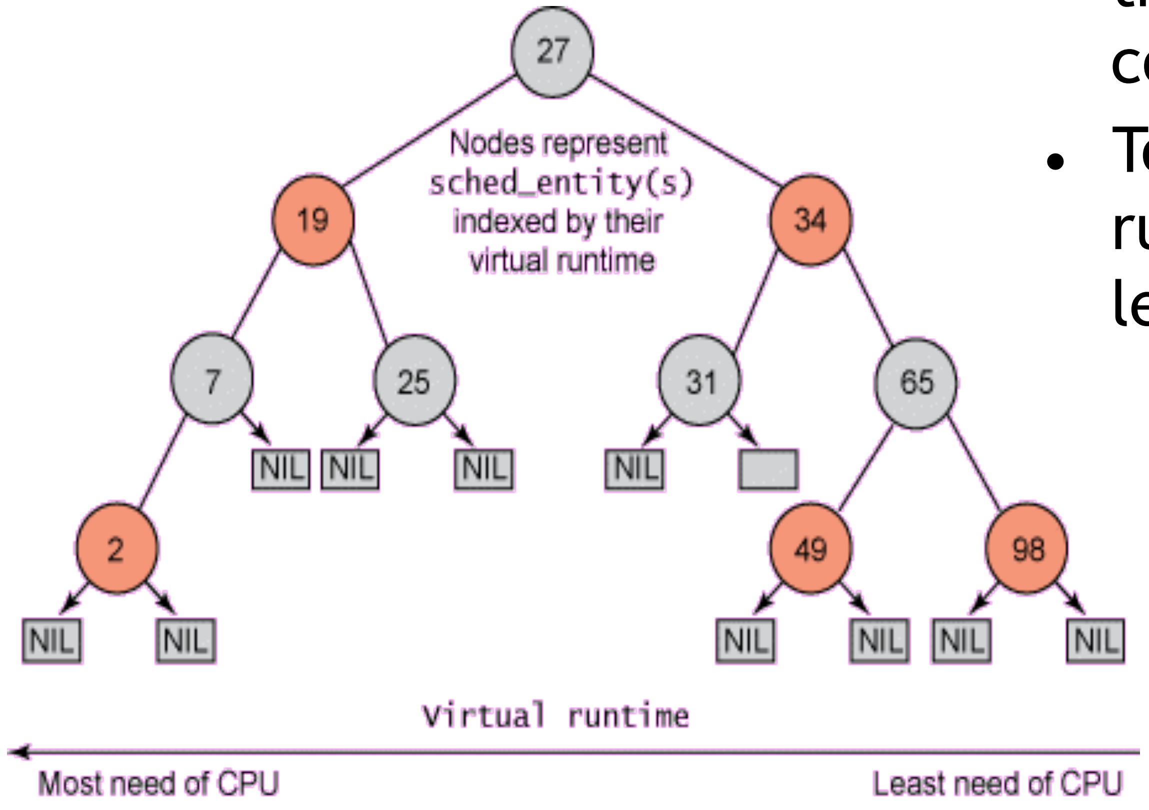
Red Black Trees

- A red-black tree is a binary search tree, which means that for each node, the left subtree only contains keys less than the node's key, and the right subtree contains keys greater than or equal to it.

- A red-black tree has further restrictions which guarantee that the longest root-leaf path is at most twice as long as the shortest root-leaf path. This bound on the height makes RB Trees more efficient than normal BSTs.

- Operations are in O(log n) time.

CFS Tree

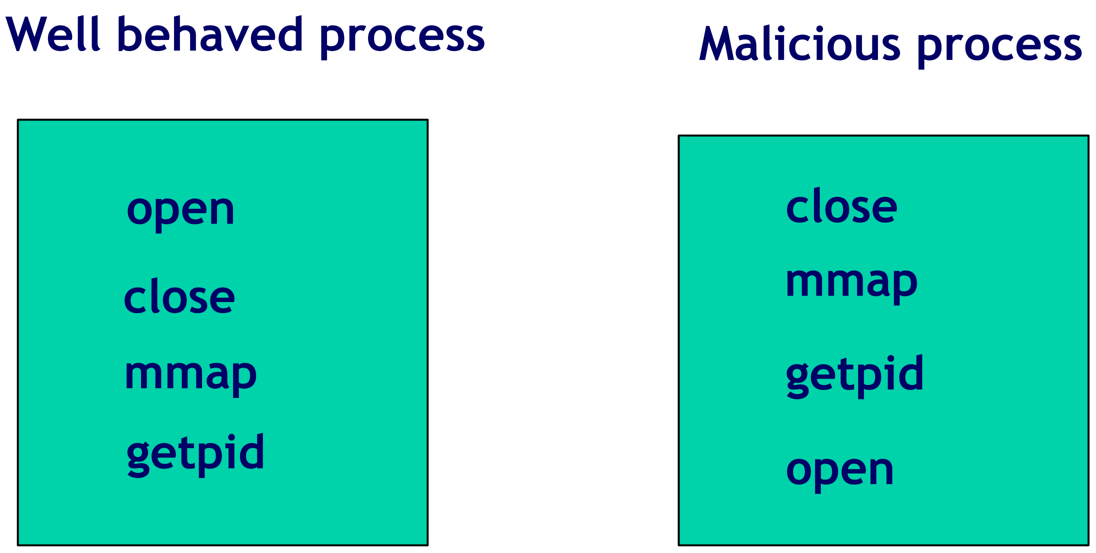


- The key for each node is the **vruntime** of the corresponding task.
- To pick the next task to run, simply take the leftmost node.

http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

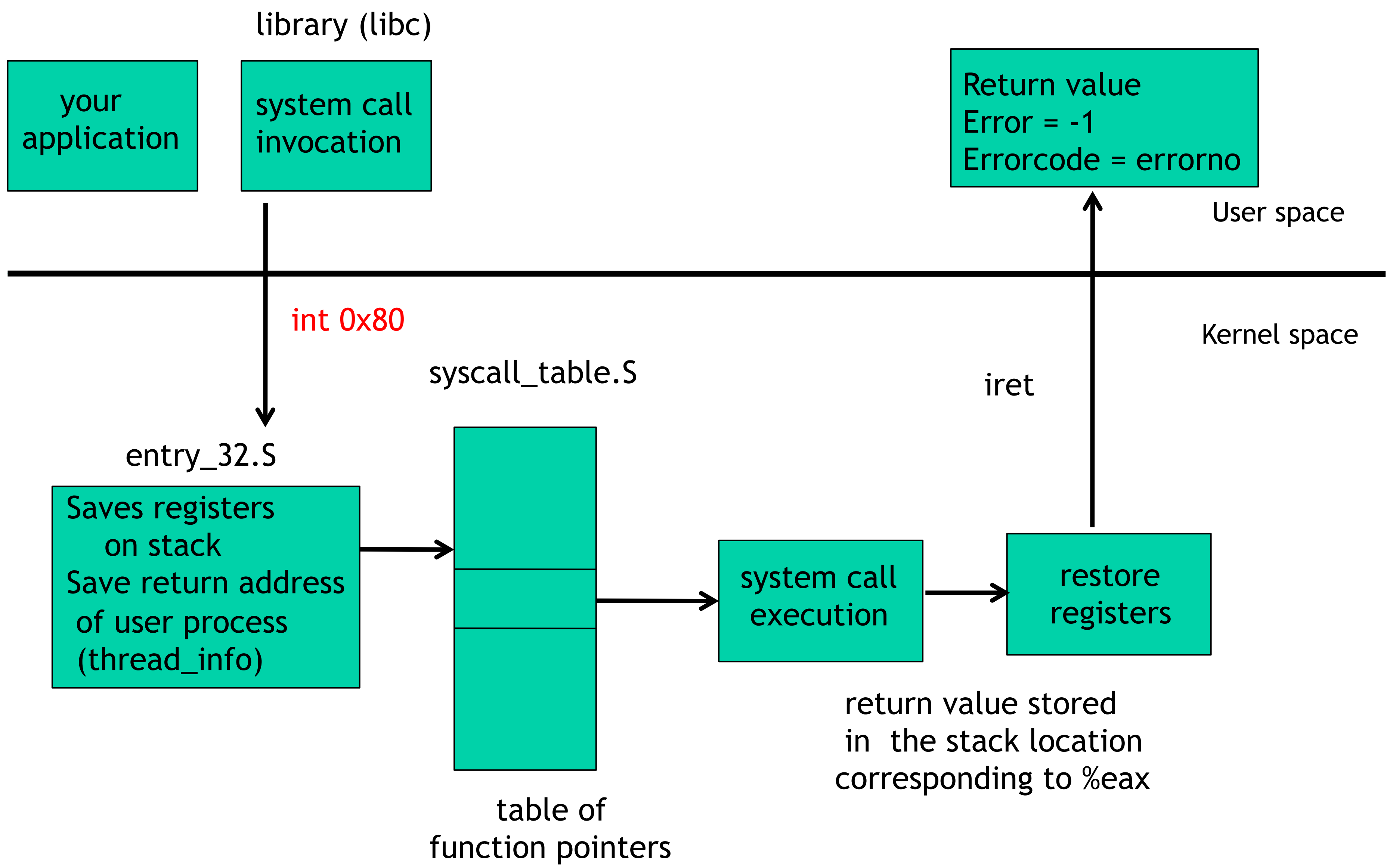
Kernel Dive to understand entry points for the scheduler

A Bit about Project 2 (Goal)

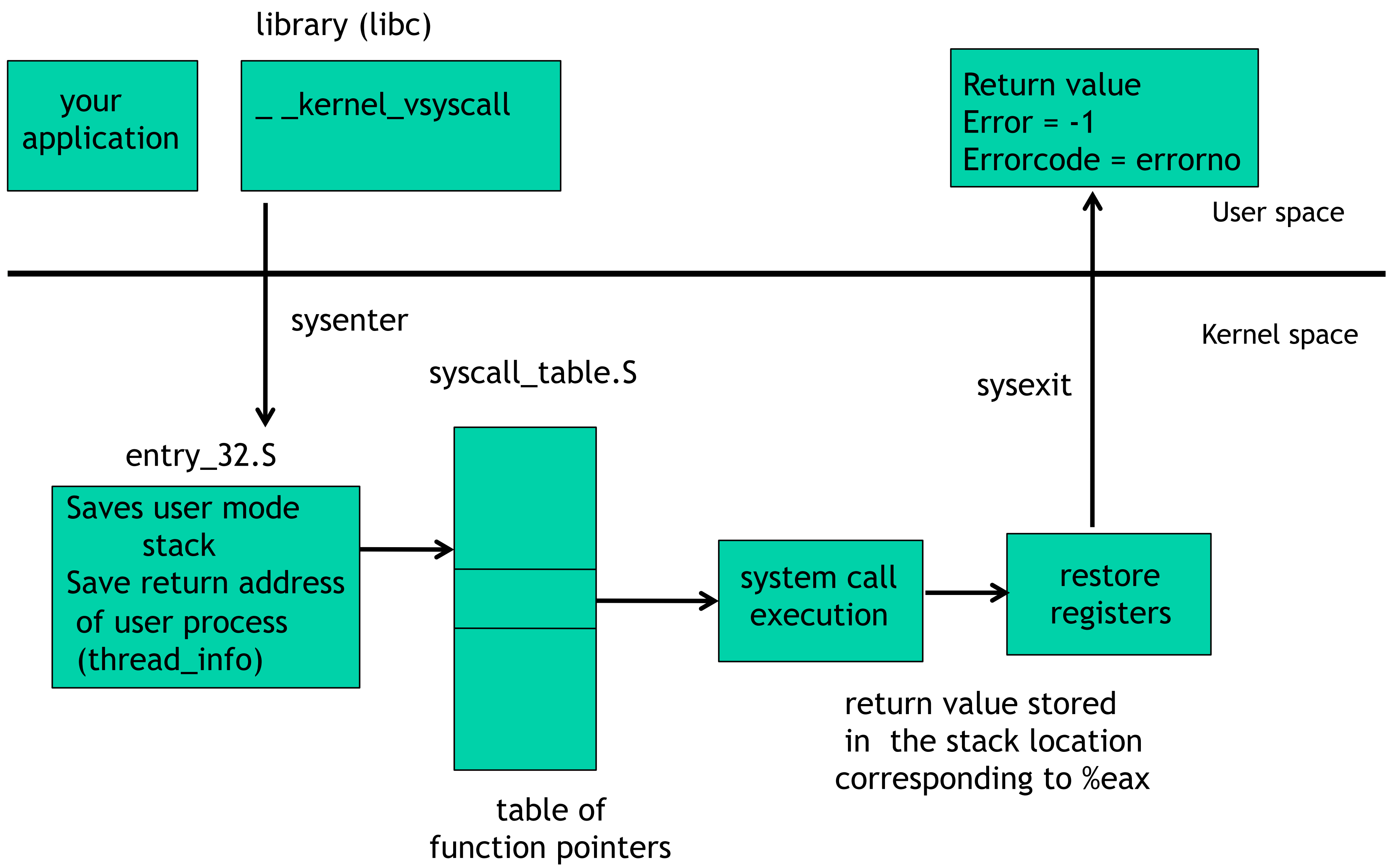


System call sequence = <open, close, mmap, getpid>
Well behaved case = <1, 1, 1, 1>
Malicious case = <0, 0, 0, 0>
Distance between these two = 4

Flow of control during a system call invocation



Simplified view of sysenter/sysexit in Linux > 2.5



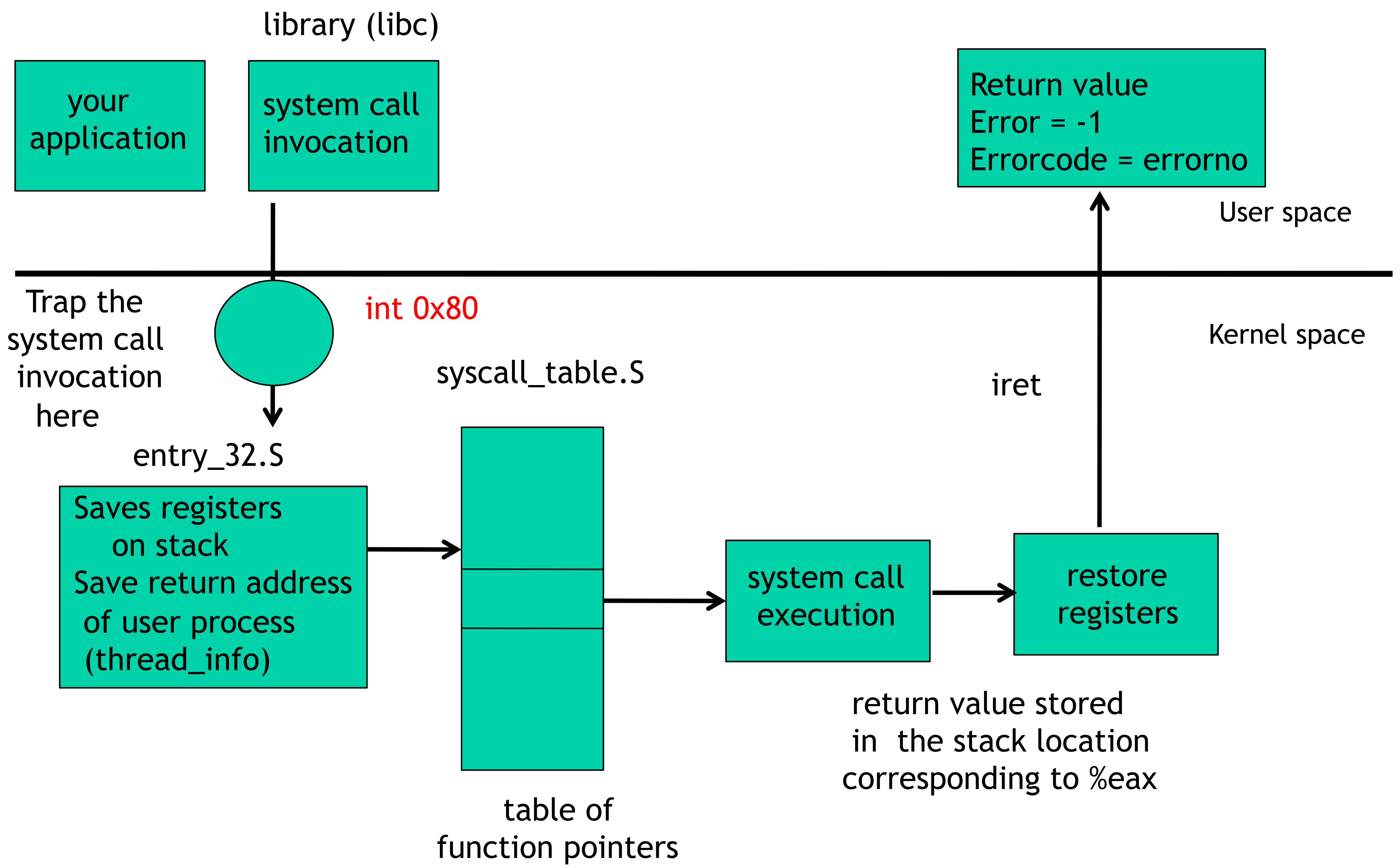
Key goal: how to trap the system calls in the kernel

To make your life easier:

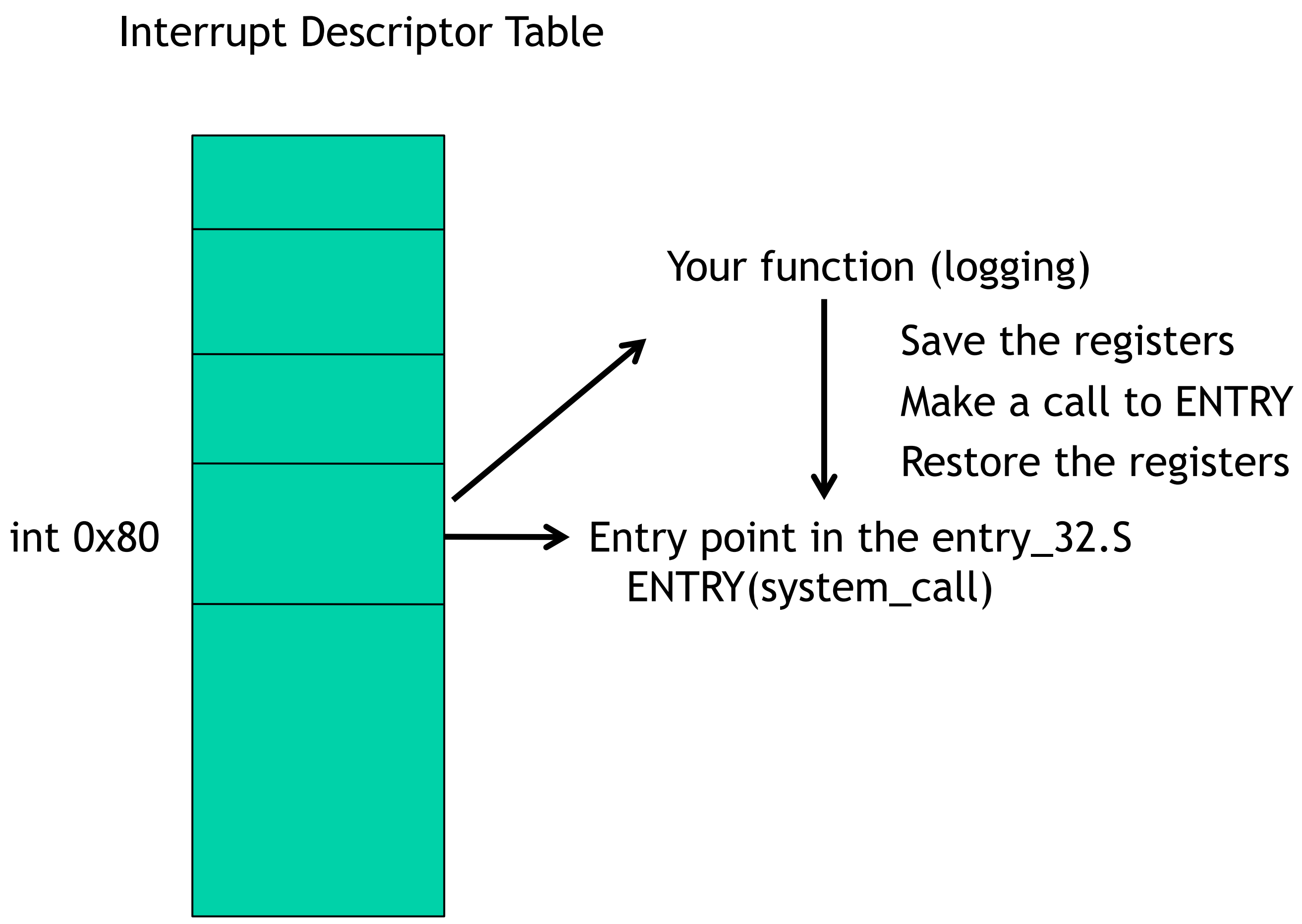
you need to disable one of the two system call invocation --- choose which ever you want

hint in the text on how to disable sysenter/sysexit

Flow of control during a system call invocation



What does trapping a sys call mean



What are the things you have to think about?

(1)Finding where the IDT is?
(2)Finding which entry in the IDT to hook?
(3)How to replace the entry with your function
(4)Proper saving/restoring of register