OS 2025 Lab 3

Multithreading Program & Linux Kernel Module

Due Date: 2025/12/26 17:00 (before lab3 course finish)

TA: 陳煒勛、徐健翔、鄭煦霖

Outline

A. Objectives

B. Background & Requirements

   1. Lock
   2. Multithread
   3. Kernel Module

C. Grading

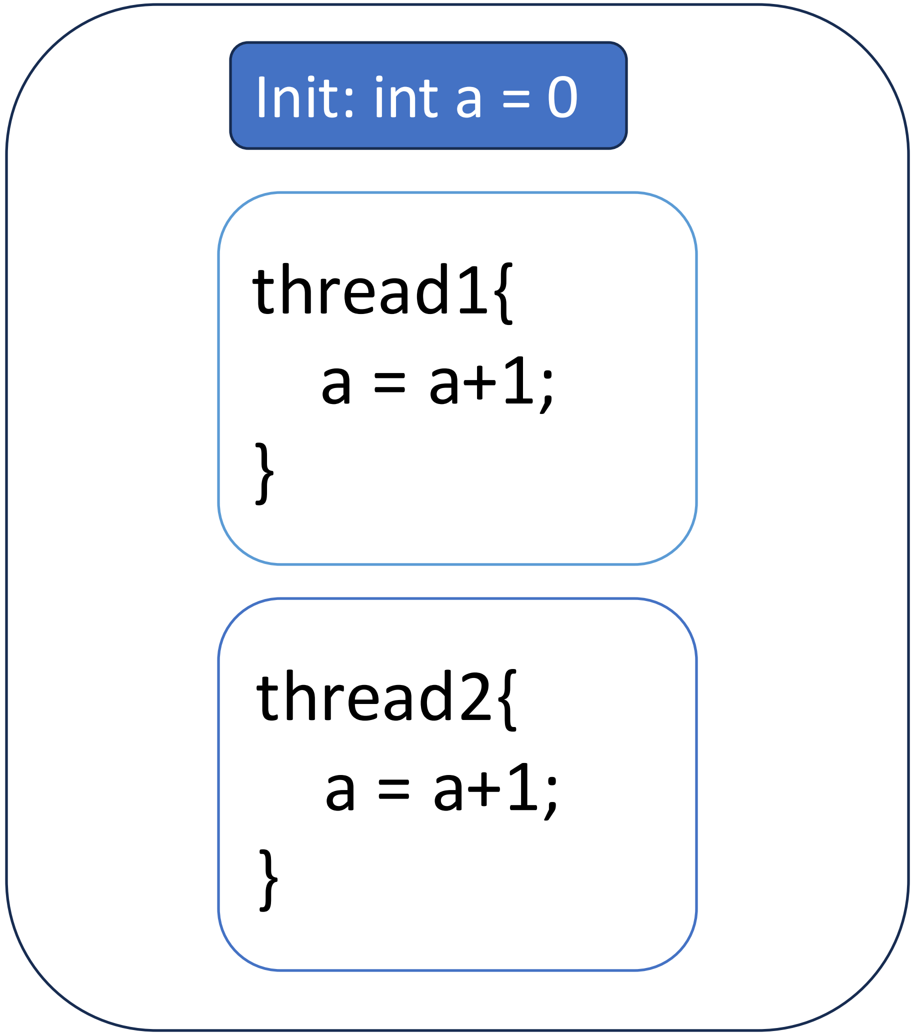
Objectives

- Process Synchronization
  - Learn how to protect critical section

- Multithreading
  - Take advantage of multi-core systems
  - Beware potential synchronization problem

- Linux Kernel Module with Proc File System
  - Communicate with kernel using Proc file

Race Condition



Init: int a = 0

```
thread1{
    a = a+1;
}
```

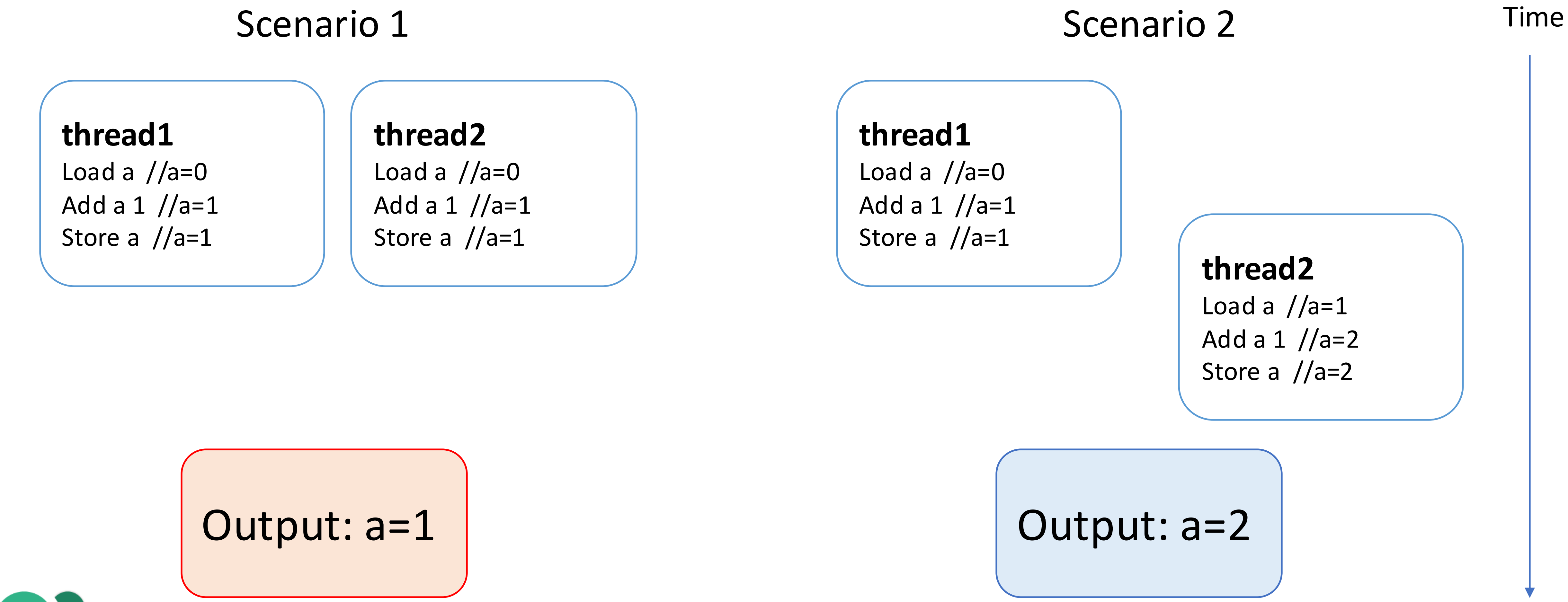
```
thread2{
    a = a+1;
}
```

Program

Expect: a=2

Output: a=2 or a=1

Race Condition



Scenario 1

thread1
Load a //a=0
Add a 1 //a=1
Store a //a=1

thread2
Load a //a=0
Add a 1 //a=1
Store a //a=1

Output: a=1

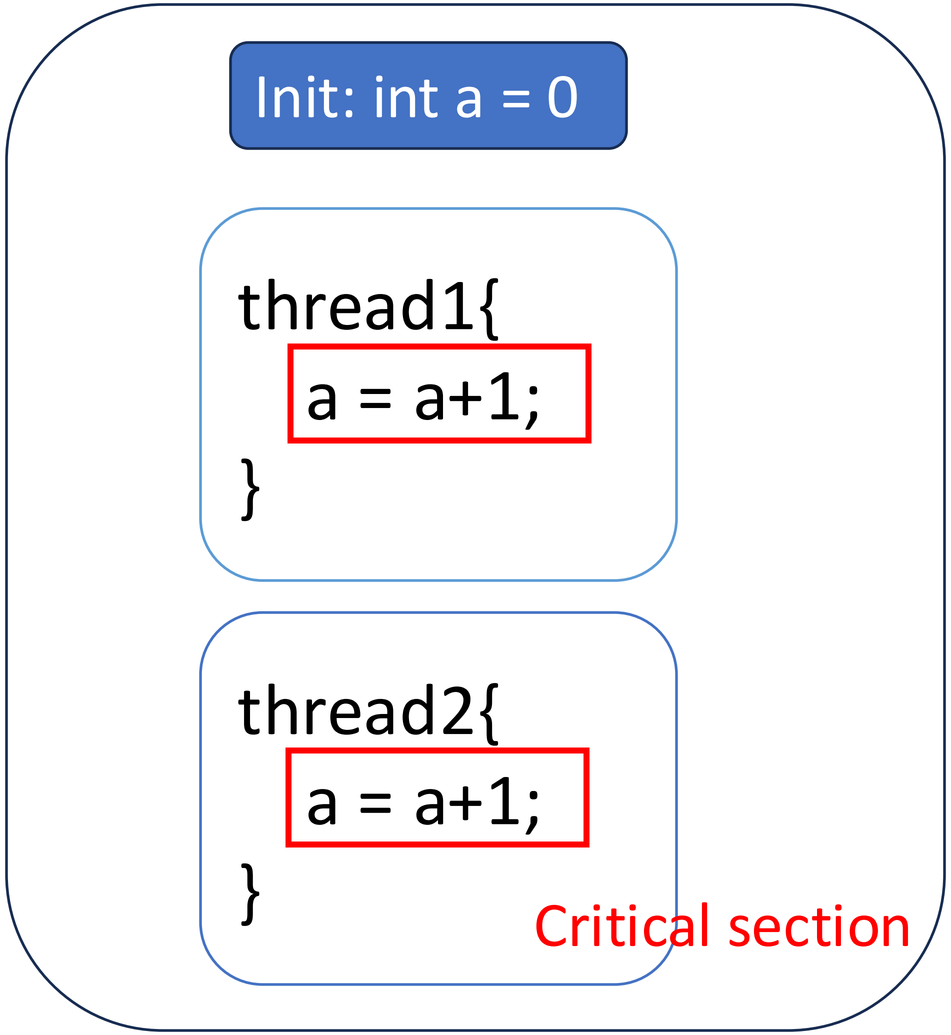
Scenario 2

thread1
Load a //a=0
Add a 1 //a=1
Store a //a=1

thread2
Load a //a=1
Add a 1 //a=2
Store a //a=2

Output: a=2

Race Condition



Init: int a = 0

```
thread1{
    a = a+1;
}
```

```
thread2{
    a = a+1;
}
```

Critical section

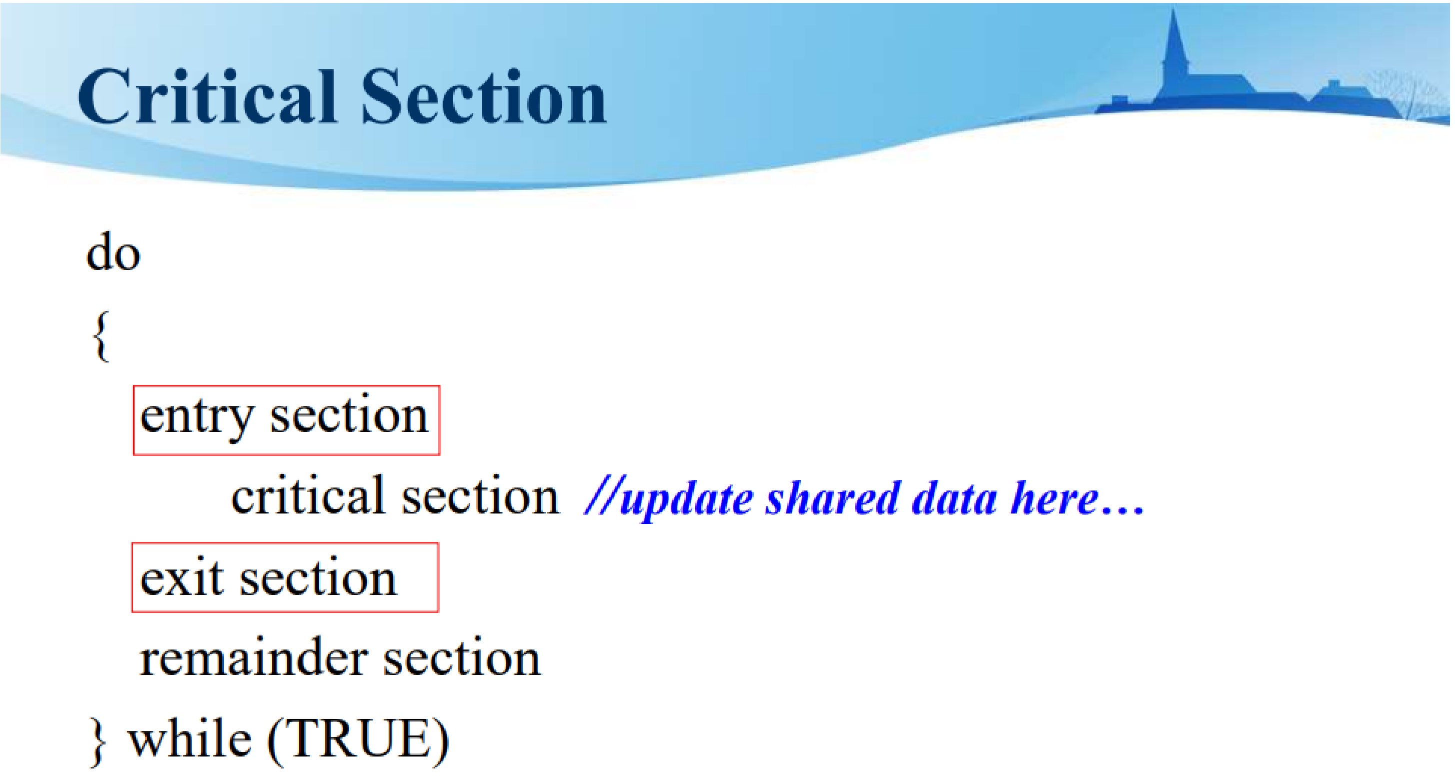
Program

Expect: a=2

Output: a=2 or a=1

Background – 1.1

How to protect critical section ?



Critical Section

```
do
{
    entry section
        critical section  //update shared data here…
    exit section
    remainder section
} while (TRUE)
```

Pthread Spin lock

- pthread_spin_lock
  - The pthread_spin_lock() function locks the spin lock referred to by lock. If the spin lock is currently unlocked, the calling thread acquires the lock immediately.
- Pthread_spin_unlock
  - The pthread_spin_unlock() function unlocks the spin lock referred to lock. If any threads are spinning on the lock, one of those threads will then acquire the lock.

Assignment 1.1

- You will get ***1_1.c, 1_ans.txt, judge.out, and Makefile*** in Assignment 1.1

- Fill /*YOUR CODE HERE*/ in ***1_1.c .***

- Test your code with Makefile.

Requirements – 1.1

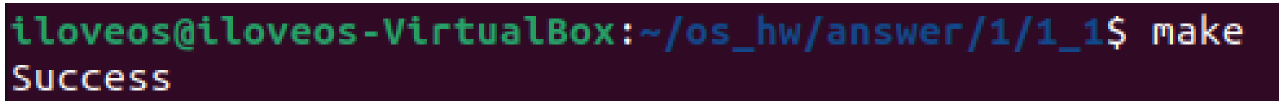
Assignment 1.1

Two threads which increment a by 1 are given.

Protect the critical section with spin lock.

Expectation: a=20000

Use Makefile to validate your code!



```
iloveos@iloveos-VirtualBox:~/os_hw/answer/1/1_1$ make
Success
```

Complete your code insides this area!

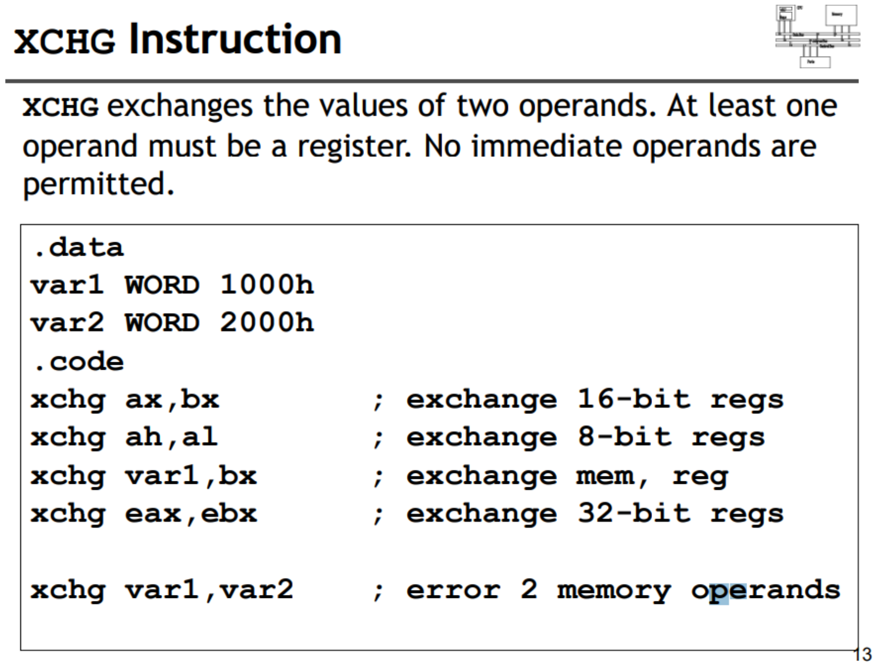
```
/*YOUR CODE HERE*/
   critical section
/***************/
```

Spin Lock

- Atomic operation XCHG(exchange)
  - XCHG instructions swaps the contents of two operands.
  - When a memory operand is used with the XCHG instruction, the processor’s LOCK signal is automatically asserted.

Atomic Instruction

- xchg
  - **Exch**ange values of register or memory
  - Atomic operation



XCHG Instruction

XCHG exchanges the values of two operands. At least one operand must be a register. No immediate operands are permitted.

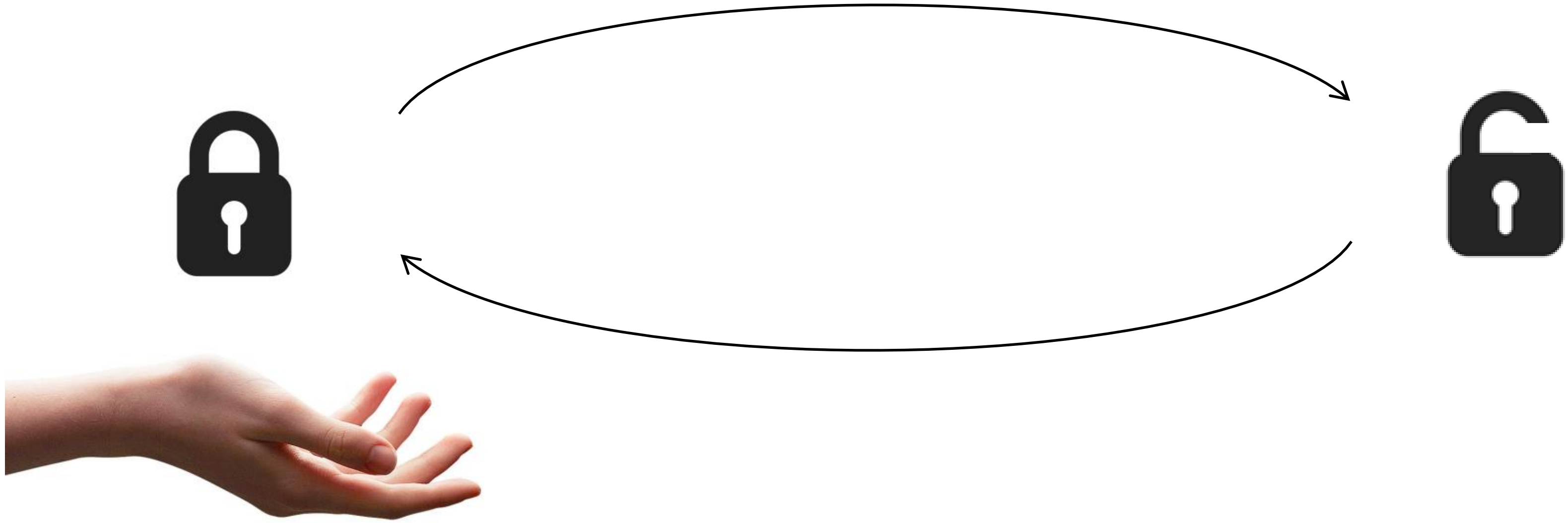
```
.data
var1 WORD 1000h
var2 WORD 2000h
.code
xchg ax,bx          ; exchange 16-bit regs
xchg ah,al          ; exchange 8-bit regs
xchg var1,bx        ; exchange mem, reg
xchg eax,ebx        ; exchange 32-bit regs

xchg var1,var2      ; error 2 memory operands
```

13

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/10fall/lectures/handouts/lec14_x86isa.pdf

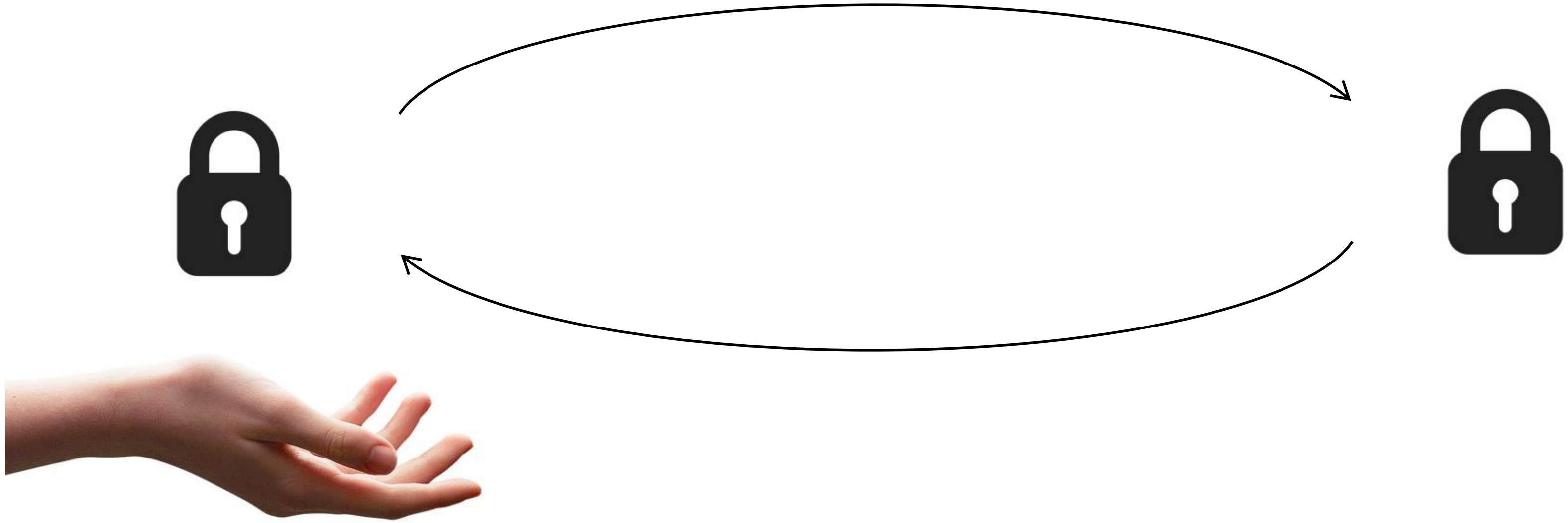
Spin Lock

- Atomic operation
  - xchg



Spin Lock

- Atomic operation
  - xchg



Assignment 1.2

- You will get ***1_2.c, 1_ans.txt, judge.out, and Makefile*** in Assignment 1.2
- Fill /*YOUR CODE HERE*/ in ***1_2.c .***
- Test your code with Makefile.

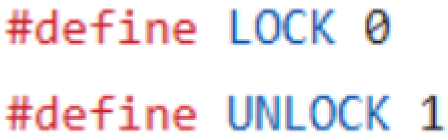
Assignment 1.2

Two threads which increment a by 1 are given.

Protect the critical section with spin lock.

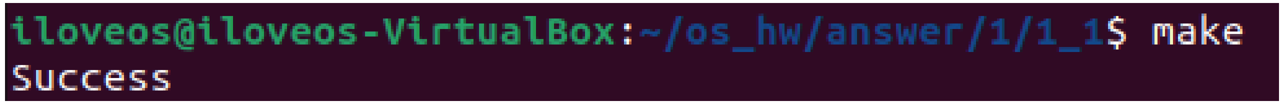
Using **xchg** instruction to complete the spin lock by yourself.

Expectation: a=20000



```
#define LOCK 0
#define UNLOCK 1
```

Use Makefile to validate your code!



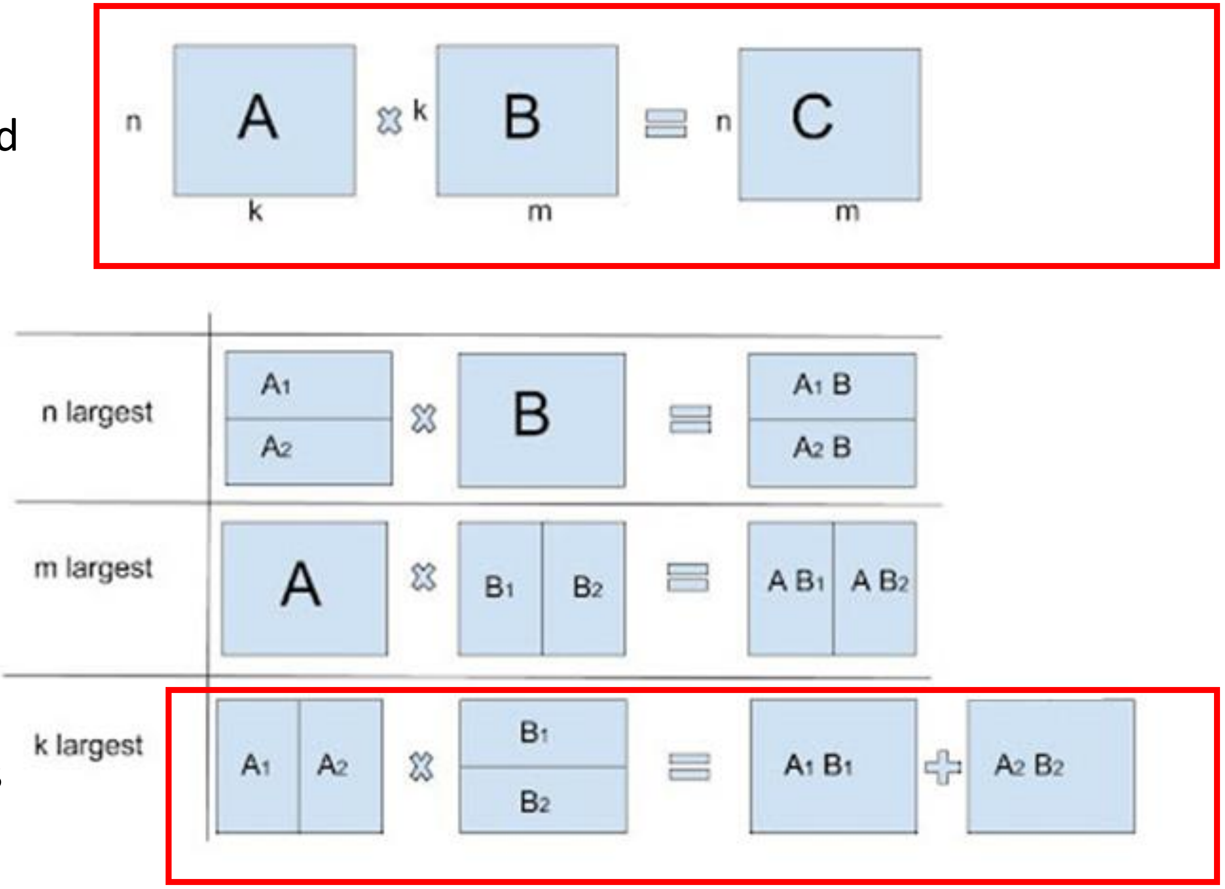
```
iloveos@iloveos-VirtualBox:~/os_hw/answer/1/1_1$ make
Success
```

Background – 2.1 & 2.2

Multithreading Program

Single thread

Two threads



Assignment 2.1

Assignment 2.2

Your work!

Beware race condition problem!

Assignment 2

- You will get ***2_1.c, 2_2.c, m1.txt, m2.txt and Makefile*** in Assignment 2.1 and Assignment 2.2.

- Fill /*YOUR CODE HERE*/ in ***2_1.c and 2_2.c.***

- Test your code with Makefile.

Requirement – 2.1 & 2.2

Assignment 2

2.1

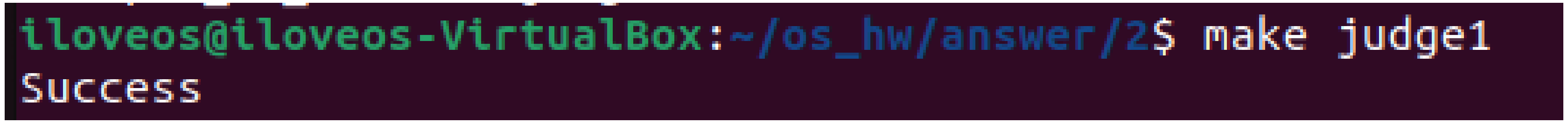
Complete the matrix multiplication with single thread.

2.2

Complete the matrix multiplication with two threads.

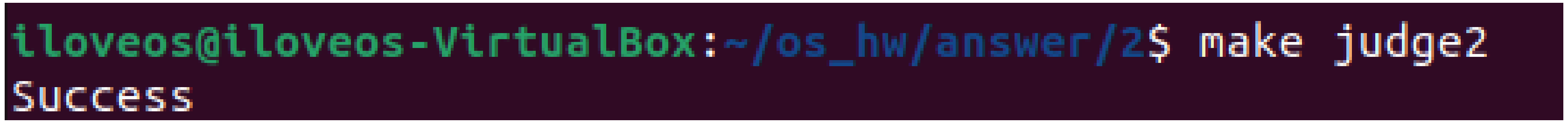
Use Makefile to validate your code!

2.1



```
iloveos@iloveos-VirtualBox:~/os_hw/answer/2$ make judge1
Success
```

2.2

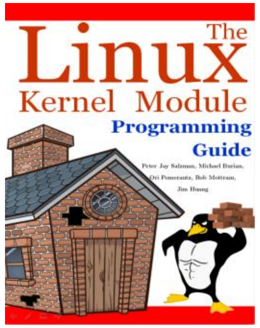


```
iloveos@iloveos-VirtualBox:~/os_hw/answer/2$ make judge2
Success
```

Linux Kernel Module

- The linux kernel has the ability to load and unload arbitrary sections of kernal code on demand. These loadable kernel modules run in priviledged kernal mode. In theory, there is no restriction on what a kernael module is allowed to do.

- Linux kernel can load and unload modules dynamically at run time. It saves recompiling, relinking and reloading time.


The
Linux
Kernel Module
Programming
Guide
Peter Jay Salzman, Michael Burian,
Ori Pomerantz, Bob Mottram,
Jim Huang

Follow the guide. Try to write a HelloWorld first by yourself!

https://sysprog21.github.io/lkmpg/

Linux Kernel Module(In This Assignment)

- Program
  - Create 2 threads like Assignment 2.1 and 2.2.
- Linux Kernel Module
  - Create /proc file.
  - Complete the Read operation and Write operation.
  - Show the process information through Read and Write operations.

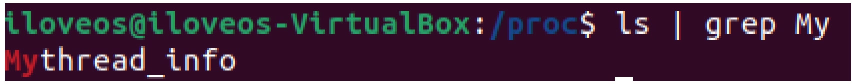
Linux Process File System

- Pseudo file system -> Data is not stored persistently.
- Provide a way for user programs to access process information as plain text files.
- For example, in the past, traditional UNIX **ps** command has been implemented as a privileged process that reads the process state directly from kernel’s virtual memory. Under Linux, this command is implemented as an entirely unprivileged program that simply parses the information from /proc.

Background – 3.1

Linux Process File System

- proc_create(name, mode, parent, proc_ops)
  - Create a proc file under proc file system.



```
iloveos@iloveos-VirtualBox:/proc$ ls | grep My
Mythread_info
```

- Struct proc_ops
  - We use 2 operations(function pointers) in this struct.

  1. proc_read
  2. proc_write

Read & Write

- When read(write) proc file, function proc_read(proc_write) will be executed.

- Therefore, our objectives are to complete the desired read and write operation.

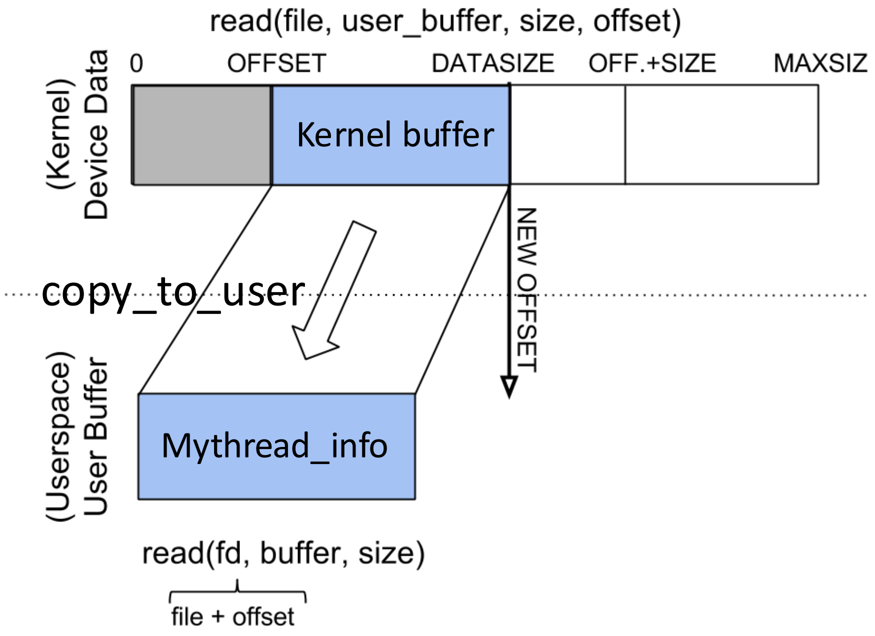
Access Kernel

- copy_to_user: copies n bytes from the kernel-space, from the address referenced by from in user-space.

- copy_from_user: copies n bytes from user-space from the address referenced by from in kernel-space.

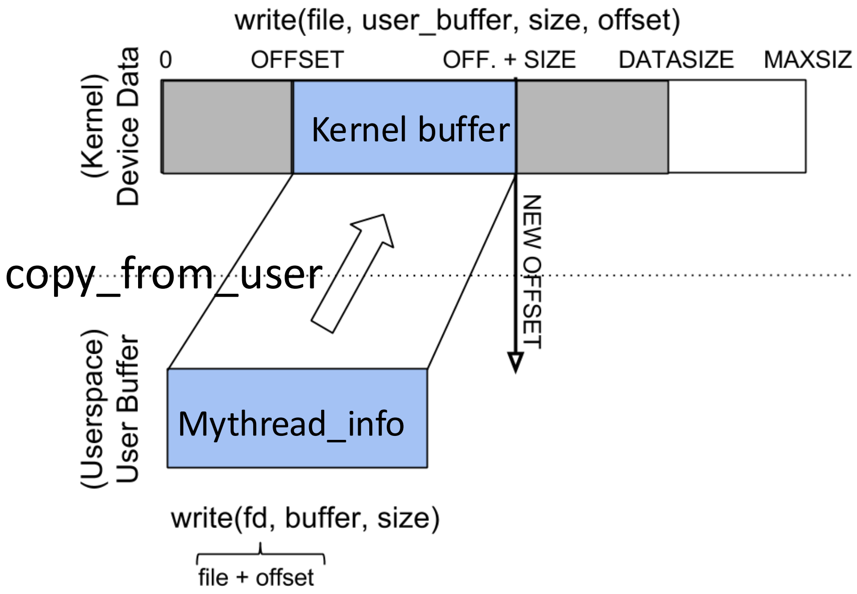
Read

- ssize_t read(file, ubuf, size, offset)
- The module is responsible for advancing the offset according to how much it reads and returning the read size.



Write

- ssize_t write(file, ubuf, size, offset)
- The module is responsible for advancing the offset according to how much it reads and returning the write size.



Assignment 3

- You will get ***3_1.c, My_Kernel.c, m1.txt, m2.txt and Makefile*** in Assignment 3.1 and ***3_2.c, My_Kernel.c, m1.txt, m2.txt and Makefile*** in Assignment 3.2.

- Assignment 3.1:

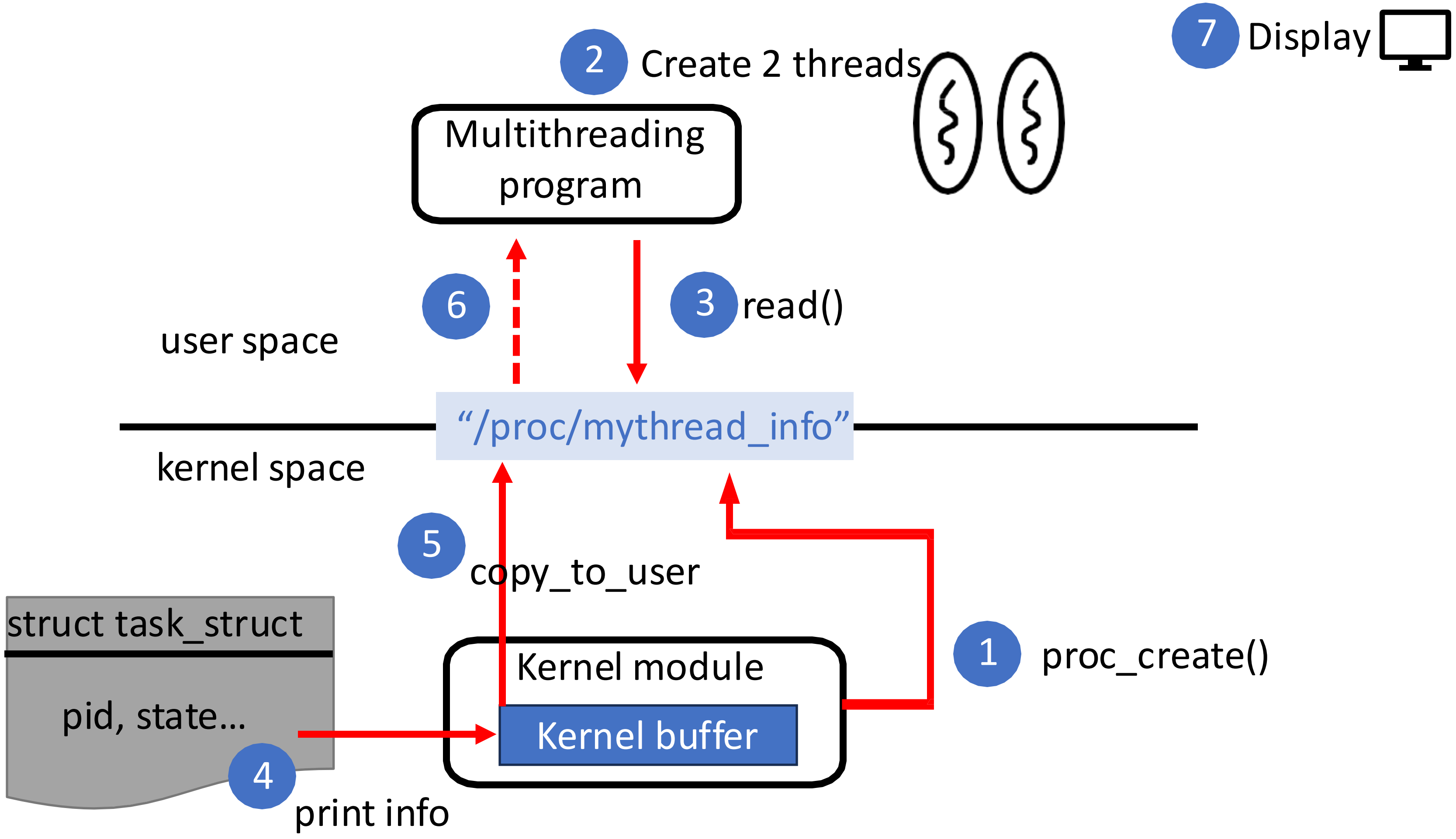
  Fill /*YOUR CODE HERE*/ in ***My_Kernel.c***

- Assignment 3.2:

  Fill /*YOUR CODE HERE*/ in ***3_2.c My_Kernel.c***

- Test your code with Makefile.

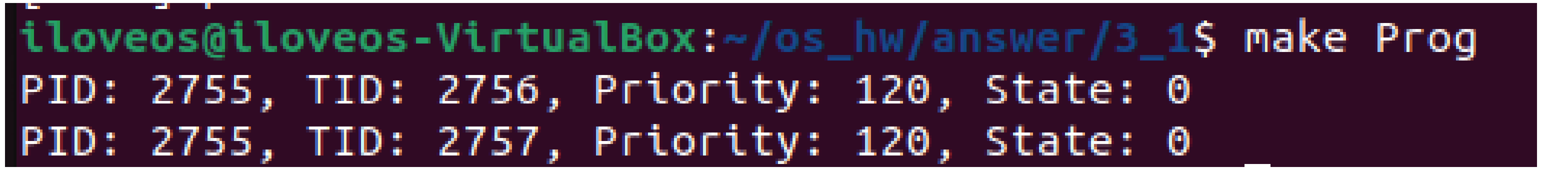
Overall Flow(Assignment 3.1)



Requirement – 3.1

Assignment 3.1

- Through proc_read, Show process ID, thread ID, priority and state.



```
iloveos@iloveos-VirtualBox:~/os_hw/answer/3_1$ make Prog
PID: 2755, TID: 2756, Priority: 120, State: 0
PID: 2755, TID: 2757, Priority: 120, State: 0
```

current->pid

for_each_thread(current, thread)

thread->pid

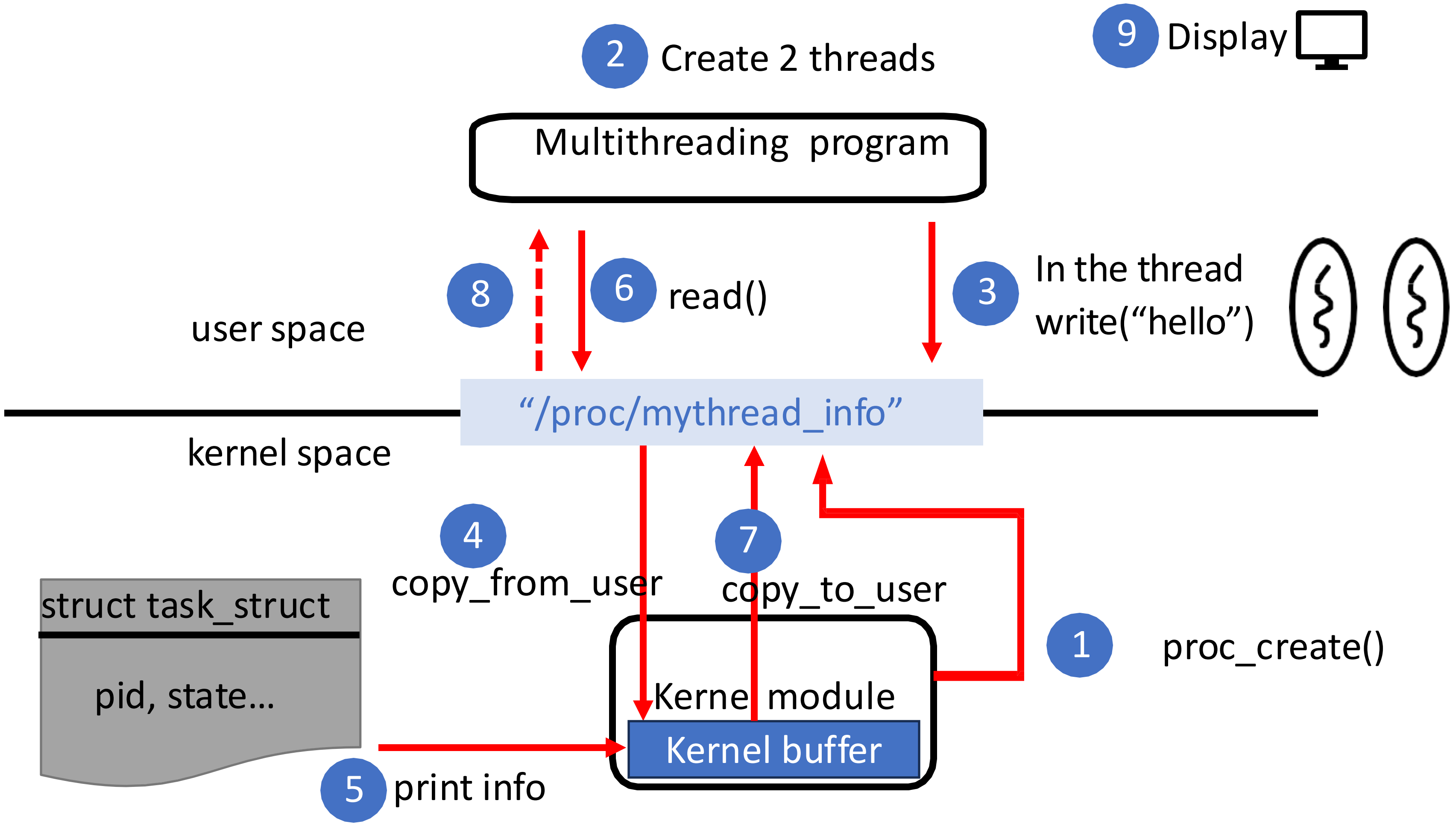
thread_prio

thread->__state

Before make Prog:

1. make all
   Build the modules.
2. make load/unload
   Load/unload the module to kernel.

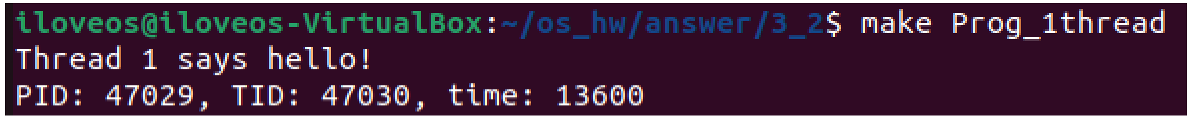
Overall Flow(Assignment 3.2)



Requirement – 3.2

Assignment 3.2

1. Through proc_read and proc_write, show string, process ID, thread ID and time(ms) in 1 thread.



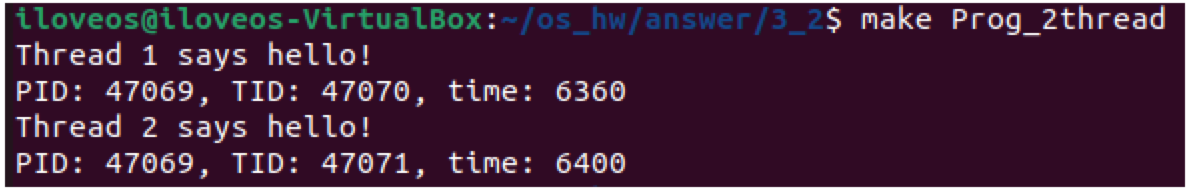
```
iloveos@iloveos-VirtualBox:~/os_hw/answer/3_2$ make Prog_1thread
Thread 1 says hello!
PID: 47029, TID: 47030, time: 13600
```

current->tgid

current->pid

current->utime/100/1000

2. Through proc_read and proc_write, show string, process ID, thread ID and time(ms) in 2 threads.



```
iloveos@iloveos-VirtualBox:~/os_hw/answer/3_2$ make Prog_2thread
Thread 1 says hello!
PID: 47069, TID: 47070, time: 6360
Thread 2 says hello!
PID: 47069, TID: 47071, time: 6400
```

Before make Prog:

1. make all
   Build the modules.
2. make load/unload
   Load/unload the module to kernel.

Grading

- 1 Lock
  - 1.1 (1 points) pthread spin lock
  - 1.2 (1 points) write a spin lock
- 2 Multithreading(pthread)
  - 2.1 (0.5 points) matrix multiplication(1 thread)
  - 2.2 (1 points) matrix multiplication(2 threads)
- 3 Kernel module
  - 3.1 (1.5 points) proc file read
  - 3.2 (2 points) proc file read and write
- Answer 3 questions
  - Each question (1 points)

Reference

- Operating System Concepts 8th Edition
  - Chapter 4 Multithreaded Programming
  - Chapter 6 Synchronization
  - Chapter 21 The Linux System
    - 21.3 Kernel modules
    - 21.7.4 The Linux Process File System
- Intel® 64 and IA-32 Architectures Software Developer Manuals
- https://sysprog21.github.io/lkmpg/
- https://en.wikipedia.org/wiki/Spinlock
- https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html