

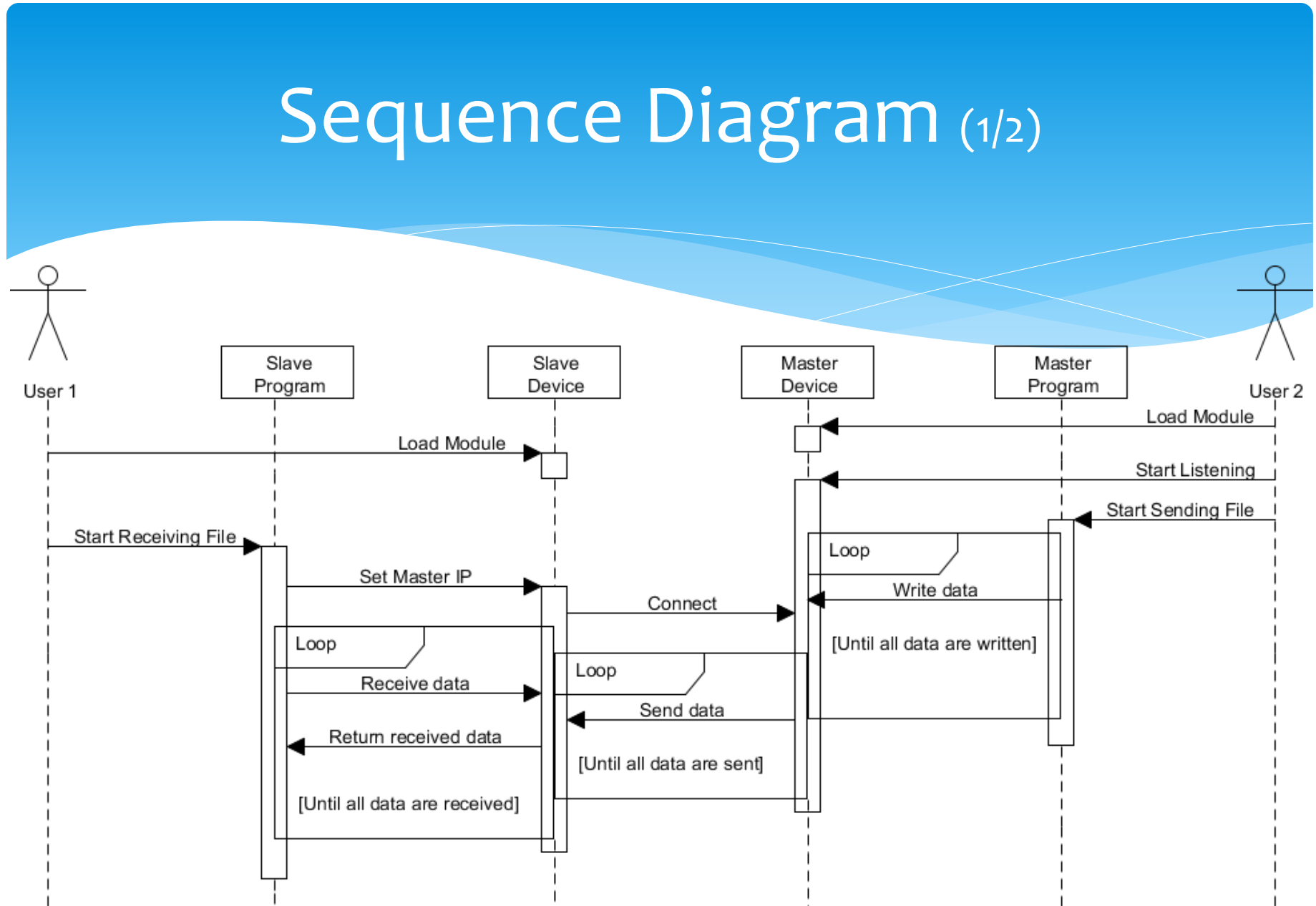
OS2019 Project 2

Supplemental Information

Outline

- * Sequence Diagram of Project 2
- * Kernel Modules
- * Kernel Sockets
- * Work Queues
- * Synchronization
- * Page Table Management

Sequence Diagram (1/2)



Sequence Diagram (2/2)

- * The diagram illustrates one possible situation.
 - * The actual sequence might differ from the diagram due to different manipulation.
- * Only the following orders are guaranteed.
 - * The slave program will be executed only after both device modules have been loaded.
 - * The master program will be executed only after the master device module has been loaded.
 - * In the slave program, it first set the IP address and then begin reading data from the slave device with blocking I/O.
- * Your design should not depend on any particular orders not defined here.

A Mini Example

- * In the following slides, we will go through the necessary kernel APIs for this project with a mini example.
- * In this example, it create a TCP server and read one line of texts from the client.
 - * The data is also cached for reading from user space.
 - * Please note that there are some possible race conditions in design.
- * All details can be found in the attached example.
 - * http://rswiki.csie.org/dokuwiki/_media/courses:102_2:miniex.tar.bz2

Kernel Modules (1/2)

- * A character device (cdev) is a device supporting data manipulation in bytes.
- * To create a character device, we need to do the following initialization.
 - * Requesting a range of device numbers by **alloc_chrdev_region**
 - * Initializing a cdev object and specifying the file operations of this device by **cdev_init**
 - * Adding the device to the kernel by **cdev_add**
- * The device file can be created as follows.
 - * Adding a new device class by **class_create**
 - * Creating a device file with the specified device number and file name by **device_create**

Kernel Modules (2/2)

- * The file operations, such as open, read, write, etc., of the character device is specified by the structure **file_operations**.
- * For example, the function pointer **release** in the structure should point to the function which will be called when the system call close is invoked over the device file in user space.

Kernel Sockets ^(1/2)

- * Kernel sockets have the similar interface as the BSD-style sockets.
- * Since it will be blocked when waiting incoming connections, we should put it in a separated thread through work queues.
- * In this example, the entry function of the work is **miniex_work_handler**.
 - * The incoming connections will be handled in the infinite loop.

Kernel Sockets (2/2)

- * You can use **miniex_rcv/miniex_send** to receive and send data with a socket.
- * The data buffer must be in kernel space, which means you need to copy the data from/to user space by **copy_from_user/copy_to_user**.

Work Queues

- * In the Linux kernel, a work queue is used for deferred works.
 - * Each work will be executed asynchronously in a kernel thread.
- * A work queue can be created by the function **create_workqueue**.
 - * It takes an input parameter specifying the name of this work queue, which can be an arbitrary text string.
- * A work can be defined by the macro **DECLARE_WORK**.
- * A work can be put in the work queue by the function **queue_work**.
 - * It takes two input parameters specifying the work queue and the defined work.

Synchronization

- * In the Linux kernel, a wait queue implements a conditional variable.
- * A work queue can be defined by the macro **DECLARE_WAIT_QUEUE_HEAD**.
- * To wait on a conditional variable, use the function **wait_event_interruptible**.
 - * The execution will be blocked until the specified condition evaluates to a true value or some signals occur.
- * To wake the tasks wait on a conditional variable, use the function **wake_up_interruptible**.

Page Table Management

- * Linux maintains the concept of three-level page table
 - * **Page Global Directory, Page Middle Directory, Page Table Entry**
 - * Address will be split into parts to yield offsets within these three page table levels and an offset within the actual page
 - * Can be access by, for example, **pgd_offset()**
- * Describing Page Table Entry
 - * Each entry is described by structure, for example, **pgd_t**
 - * They are defined as structure for reasons
 - * Type protecting
 - * Addressing protection