Lab 1: Setting up your Raspberry Pi
(參考資料來源：https://www.raspberrypi.com/documentation/computers/getting-started.html)

1 Objective
- Prepare cross development environment for Raspberry Pi.
- Be familiar with your Raspberry Pi.

2 Prerequisite
- Get familiar with basic Linux and gcc compiler commands.

3 Hardware (without using HTMI monitors and input peripherals)
To get started with your Raspberry Pi computer you'll need the following accessories:
- **A power supply.** We recommend the official Raspberry Pi Power Supply which has been specifically designed to consistently provide +5.1V despite rapid fluctuations in current draw.
- **A minimum of 8GB micro SD card.** You can use the https://www.raspberrypi.com/software/[Raspberry Pi Imager] to install an operating system onto it.
- **An Ethernet cable.** Connect your Raspberry Pi to your local network and the Internet.

4 Software
- **Raspberry Pi Imager.** We can write the image to SD card using this tool on the host PC.
- **putty** or similar tools. We need to connect the host PC to Raspberry Pi with this tool.
- **nmap.** We need to scan the whole network with this tool.

5 Setting Up
1. Download and Flash Image.
   - Download and install Raspberry Pi Imager to your Host PC.
   - Install an SD card reader to your Host PC.
   - Put the SD card into the reader and run Raspberry Pi Imager in your Host PC.
     - Choose **Raspberry Pi OS with desktop** as your *Operating System*.
     - Choose the SD card you'll use with your Raspberry Pi for the *Storage*.

2. Connect your Raspberry Pi to your local network using the Ethernet cable.

3. Install the SD card with image to your Raspberry Pi.
4. Power on your Raspberry Pi.
5. Find your Raspberry Pi in your network.
    - Connect your Host PC to the same local network.
    - Install nmap.
    - Install putty.
    - In your Host PC, launch the command console, and type:

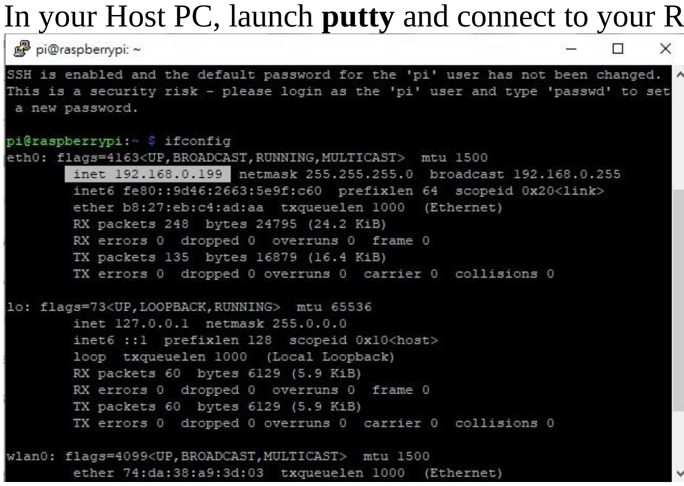
      `ipconfig`

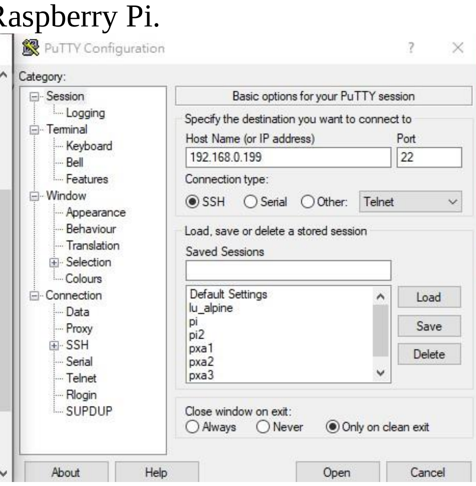
      Since Raspberry Pi support DHCP, when it connects to a network device supporting DHCP, it will be assigned with an IP automatically. The problem is you do not know which IP your Raspberry Pi is running with. `ipconfig` discloses the possible IP ranges for your Host PC and Raspberry Pi.

      If your local network is assigned with IPs ranging from 192.168.0.0 – 192.168.0.255, then type:

      `nmap 192.168.0.0-255 -p22`

      `nmap` shows a list of devices and their IPs running in this network.
    - In your Host PC, launch **putty** and connect to your Raspberry Pi.





    - Login to your Raspberry Pi using `pi` and `raspberry` as the username and password.

- Change the IP of your Raspberry Pi.
  `vi /etc/network/interfaces`

```
iface eth0 inet static
address 192.168.0.199
gateway 192.168.0.1
netmask 255.255.255.0
network 192.168.0.0
broadcast 192.168.0.255
allow-hotplug eth0
```

6. Reboot your Raspberry Pi and you can use the new IP, 192.168.0.199.

6 Discover your Raspberry Pi

- Q1: Describe your observation of Raspberry Pi OS (from any point of view, internal, external, …)
  If you refer to information from Internet, please remember to cite your reference.
- Q2: What's inside your Raspberry Pi?
- Q3: What does your Raspberry Pi OS support?

Lab 2: Build Kernel Image

(參考資料來源：https://www.raspberrypi.com/documentation/computers/getting-started.html)

1 Objective

- Be familiar with the development environment and cross-compilation.

2 Prerequisite

- Create a VM running with Ubuntu in your Host PC.

3 Development Environment

Raspberry Pi OS is a free operating system based on Debian, optimised for the Raspberry Pi hardware. The Raspberry Pi kernel is stored in GitHub and can be viewed at github.com/raspberrypi/linux: please use the `master` branch.

There are many reasons you may want to put something into the kernel:

- You've written some Raspberry Pi-specific code that you want everyone to benefit from
- You've written a generic Linux kernel driver for a device and want everyone to use it
- You've fixed a generic kernel bug
- You've fixed a Raspberry Pi-specific kernel bug
- …

In this lab, you learn to make your own Raspberry Pi OS. So, you need to

1. Create a virtual machine
   - Create a VM running with **Ubuntu** in your Host PC.
     We tend to use Ubuntu since Raspberry Pi OS is also a Debian distribution, it means many aspects are similar, such as the command lines.

2. Install the required dependencies and toolchain
   - To build the sources for cross-compilation, you need:
     ```
     sudo apt install git bc bison flex libssl-dev make libc6-dev libncurses5-dev
     ```
   - Install the toolchain for your kernel
     - Install the 32-bit Toolchain for a 32-bit Kernel
       ```
       sudo apt install crossbuild-essential-armhf
       ```
     - Install the 64-bit Toolchain for a 64-bit Kernel (In this course, we use this one.)
       ```
       sudo apt install crossbuild-essential-arm64
       ```

4 Build Kernel

3. Get the kernel sources, run:
```
git clone --depth=1 https://github.com/raspberrypi/linux
```

   Notes: Forking the Linux repository and cloning that on your build system and be Done on the Raspberry Pi or ***on the VM for cross-compiling***, we choose to use the latter one.

   A cross-compiler is configured to build code for a target other than the one running the build process, and using. It is so called cross-compilation. Cross-compilation of the Raspberry Pi kernel is useful for two reasons:
   - it allows a 64-bit kernel to be built using a 32-bit OS, and vice versa, and
   - a traditional PC can cross-compile a Raspberry Pi kernel significantly faster than the Raspberry Pi itself.

   You can then make your changes to the repository, test your changes, and commit them into your fork.

4. Build kernel sources and device tree files (For Raspberry Pi 3, 3+, and 4; 64-bit configs):
```
cd linux
KERNEL=kernel8
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2711_defconfig
```

5. Build with Configs (for 64-bit builds)
```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules dtbs
```

5 Flash Image

6. Install directly onto the SD card
   Having built the kernel, you need to copy it onto your Raspberry Pi and install the modules; this is best done directly using an SD card reader.
   First, use `lsblk` before and after plugging in your SD card to identify it. You should end up with something a lot like this:
```
sdb
   sdb1
   sdb2
```
   `sdb1`: the FAT filesystem (boot) partition
   `sdb2`: the ext4 filesystem (root) partition

7. Mount the above partitions and adjust the partition letter later as necessary.
```
mkdir mnt
mkdir mnt/fat32
mkdir mnt/ext4
sudo mount /dev/sdb1 mnt/fat32
sudo mount /dev/sdb2 mnt/ext4
```

8. Install the kernel modules onto the SD card.
```
sudo env PATH=$PATH make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
INSTALL_MOD_PATH=mnt/ext4 modules_install
```

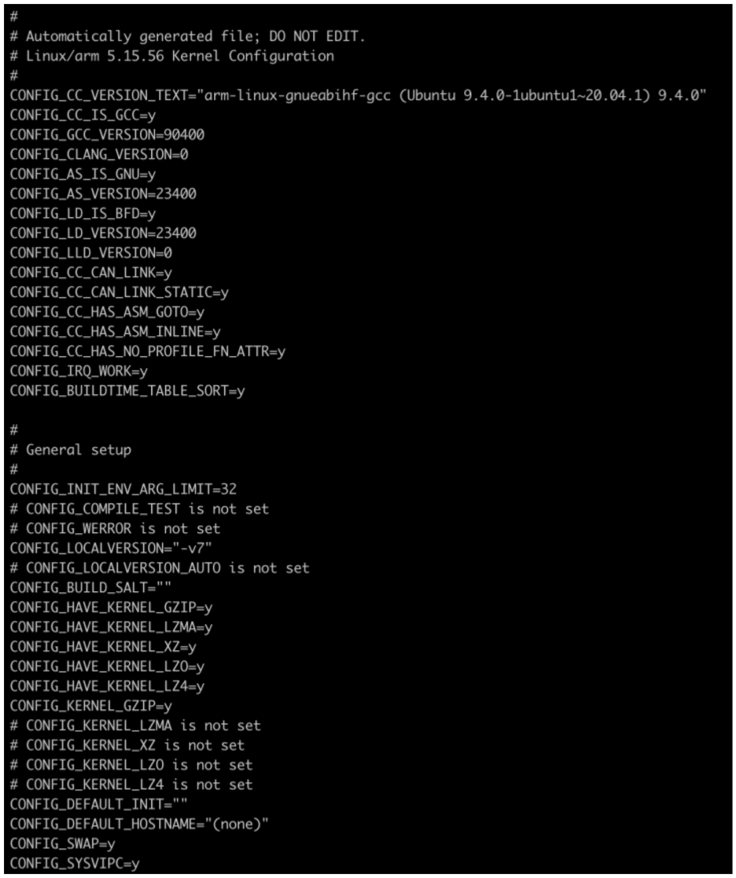
9. Copy the kernel and Device Tree blobs onto the SD card, making sure to back up your old kernel:
```
sudo cp mnt/fat32/$KERNEL.img mnt/fat32/$KERNEL-backup.img
sudo cp arch/arm64/boot/Image mnt/fat32/$KERNEL.img
sudo cp arch/arm64/boot/dts/broadcom/*.dtb mnt/fat32/
sudo cp arch/arm64/boot/dts/overlays/*.dtb* mnt/fat32/overlays/
sudo cp arch/arm64/boot/dts/overlays/README mnt/fat32/overlays/
sudo umount mnt/fat32
sudo umount mnt/ext4
```

10. Another option is to copy the kernel into the same place, but with a different filename - for instance, `kernel-myconfig.img` - rather than overwriting the `kernel.img` file. You can then edit the `config.txt` file to select the kernel that the Raspberry Pi will boot:
```
kernel=kernel-myconfig.img
```
This can keep your custom kernel separate from the original kernel, and allow you to revert to the original kernel in the event that the custom kernel cannot boot.

11. Plug the card into the Raspberry Pi and boot it!

6 Configure your Kernel

Raspberry Pi OS is built based on the Linux kernel. As you may have known, the Linux kernel is highly configurable; users may modify the default configuration as they need. Users can configure the kernel to enable/disable experimental modules (network protocols) or hardware. To configure the Linux kernel, a user can manually modify the kernel configuration file (`.config`.) Note that, `.config` is not existed with the original Linux kernel source. You have to create it.



```
#
# Automatically generated file; DO NOT EDIT.
# Linux/arm 5.15.56 Kernel Configuration
#
CONFIG_CC_VERSION_TEXT="arm-linux-gnueabihf-gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0"
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=90400
CONFIG_CLANG_VERSION=0
CONFIG_AS_IS_GNU=y
CONFIG_AS_VERSION=23400
CONFIG_LD_IS_BFD=y
CONFIG_LD_VERSION=23400
CONFIG_LLD_VERSION=0
CONFIG_CC_CAN_LINK=y
CONFIG_CC_CAN_LINK_STATIC=y
CONFIG_CC_HAS_ASM_GOTO=y
CONFIG_CC_HAS_ASM_INLINE=y
CONFIG_CC_HAS_NO_PROFILE_FN_ATTR=y
CONFIG_IRQ_WORK=y
CONFIG_BUILDTIME_TABLE_SORT=y

#
# General setup
#
CONFIG_INIT_ENV_ARG_LIMIT=32
# CONFIG_COMPILE_TEST is not set
# CONFIG_WERROR is not set
CONFIG_LOCALVERSION="-v7"
# CONFIG_LOCALVERSION_AUTO is not set
CONFIG_BUILD_SALT=""
CONFIG_HAVE_KERNEL_GZIP=y
CONFIG_HAVE_KERNEL_LZMA=y
CONFIG_HAVE_KERNEL_XZ=y
CONFIG_HAVE_KERNEL_LZO=y
CONFIG_HAVE_KERNEL_LZ4=y
CONFIG_KERNEL_GZIP=y
# CONFIG_KERNEL_LZMA is not set
# CONFIG_KERNEL_XZ is not set
# CONFIG_KERNEL_LZO is not set
# CONFIG_KERNEL_LZ4 is not set
CONFIG_DEFAULT_INIT=""
CONFIG_DEFAULT_HOSTNAME="(none)"
CONFIG_SWAP=y
CONFIG_SYSVIPC=y
```

Alternatively, you can also use the `make menuconfig` command. If you clone the Raspberry Pi OS source in `linux/`, then you can find the `Makefile` file in that directory for compiling the kernel. You need to run `make menuconfig` inside `linux/` to configure your kernel manually.
You don't have to configure everything manually. There are `make bcmrpi_defconfig`, `make bcm2709_defconfig`, and `make bcm2711_defconfig` commands for creating default kernel configuration for Pi1, Pi 2 and 3, and Pi 4, respectively.

Since `menuconfig` requires the `ncurses` development headers to compile properly.

12. You have to install `ncurses` by running:
    ```
    sudo apt install libncurses5-dev
    ```

13. Run the `menuconfig` utility as follows:
    ```
    make menuconfig
    make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- menuconfig
    ```
    Use the arrow keys and Enter key to navigate and select the configuration items and values. You can press h on most entries to get help about that specific option or menu. Press Escape key to save and exit the configuration. What you've chosen will be saved in the `.config` file.

7 Miscellaneous

- (optional) Patching the Kernel

Normally, patches are provided as a temporary measure, before the patches are applied to the mainline Linux kernel and then propagated down to the Raspberry Pi OS kernel sources. Sometimes, you may need to apply patches to your kernel source for enabling newer hardware.
It's important to check what version of the kernel you downloaded and apply proper patches. You can see the version of the running kernel with the `uname -r` command. And, in a kernel source directory, running `head Makefile -n 3` will show you the version the sources relate to, such as:

```
VERSION = 3
PATCHLEVEL = 10
SUBLEVEL = 25
```

The example shows the source for a 3.10.25 kernel.

- (Optional) Applying Patches
  Most patches are a single file, and applied with the `patch` utility. For example, let's download and patch our example kernel version with the real-time kernel patches:

  ```
  wget https://www.kernel.org/pub/linux/kernel/projects/rt/3.10/older/patch-3.10.25-rt23.patch.gz
  gunzip patch-3.10.25-rt23.patch.gz
  cat patch-3.10.25-rt23.patch | patch -p1
  ```

  In our example we simply download the file, uncompress it, and then pass it to the `patch` utility using the `cat` tool and a UNIX pipe `|`.
  Some patchsets come with different format, you need to read the instructions provided by the patch distributor to know how to apply them.
- [**Important**] Kernel Headers
  Sometimes, you need the Linux kernel headers to compile a kernel module. Kernel headers provide the function and structure definitions required for the compilation. Without these definitions, you may fail to compile the module codes. If you have cloned the entire kernel from github, the headers are already included in the source tree. However, if not, then it is possible you need to install the kernel headers (only) from the Raspberry Pi OS repo. Try this command for installing the kernel headers (this might take time):

  ```
  sudo apt install raspberrypi-kernel-headers
  ```

  NOTE: When a new kernel is released, you will need the headers of the new version. Since it may take several weeks for the repo to be updated, the best approach is to clone the kernel as described in Step 3.

8 Design your Kernel

- Q1: Shrink the size of your kernel image.
- Q2: Benchmark your kernel, revise it, and improve its performance. Rerun the benchmark to prove your performance.
- Q3: Patch your kernel to support real-time tasks.

Lab 3: Driver (Part I) – Using GPIO in Raspberry Pi

(參考資料來源：https://www.raspberrypi.com/documentation/computers/getting-started.html, https://embetronicx.com/tutorials/linux/device-drivers/gpio-driver-basic-using-raspberry-pi/)

1 Objective

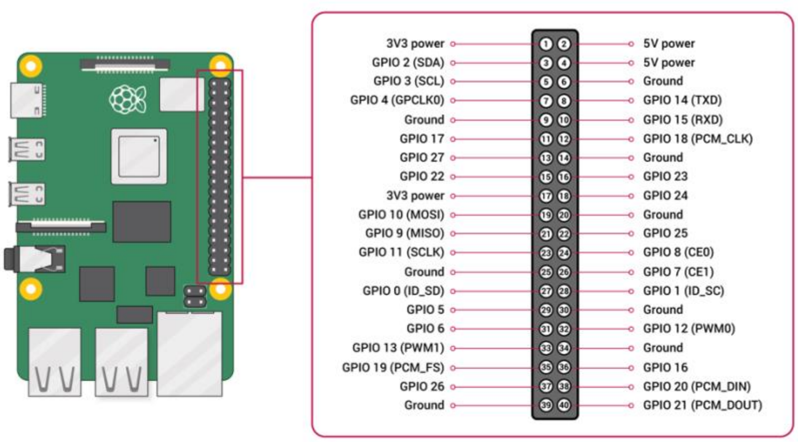
- Be familiar with using GPIO and its driver.

2 Prerequisite

- Read the document regarding to GPIO, etc.

3 GPIO

GPIO stands for General Purpose Input Output. GPIO is one of the most frequent terms which you might have come across with. A GPIO pin can be used to perform digital input or output functions. The GPIO behavior (input or output) is controlled by an application. A powerful feature of the Raspberry Pi is the row of GPIO pins along the top edge of the board, as illustrated in the following figure.



Any of the GPIO pins can be designated in software as an input or output pin and used for a wide range of purposes. You can read more about the GPIO pins of Raspberry Pi on its website.

4 Accessing the GPIO in Linux Kernel

To access the GPIO from the Kernel GPIO subsystem, you have to follow the below steps.

1. Verify whether the GPIO is valid or not.
2. If it is valid, then you can request the GPIO from the Kernel GPIO subsystem.
3. Export the GPIO to sysfs (This is optional).

4. Set the direction of the GPIO
5. Make the GPIO to High/Low if it is set as an output pin.
6. Set the debounce-interval and read the state if it is set as an input pin. You enable IRQ also for edge/level triggered.
7. Then release the GPIO while exiting the driver or once you are done.

The APIs used to do the above steps are given below.

5 GPIO APIs in Linux kernel

Kernel GPIO subsystems provide the APIs to access the GPIOs. You need to include the GPIO header file given below.

```
#include <linux/gpio.h>
```

Next, let's introduce common APIs for GPIO.

- `bool gpio_is_valid(int gpio_number);`
  Before using the GPIO, you must check whether the GPIO is valid. To do so, you have to use the above API, where
  `gpio_number` represents the GPIO that you are planning to use.
  The API returns false if it is not valid otherwise, it returns true. Sometimes this call will return false even if you send a valid number. Because that GPIO pin might be temporarily unused on a given board.

- `int gpio_request(unsigned gpio, const char *label)`
  Once you have validated the GPIO, then you can request the GPIO using the above API, where
  `gpio` is the GPIO pin that you are planning to use;
  `label` is used by the kernel for the GPIO in sysfs.
  You can provide any string that can be seen in `/sys/kernel/debug/gpio`. So, when you execute the command '`cat /sys/kernel/debug/gpio`', you can see the GPIO assigned to the particular GPIO pin. It returns 0 on success and a negative number on failure.
  There are other variants also available. You can use any one of them based on your need.
  - `int gpio_request_one(unsigned gpio, unsigned long flags, const char *label);` – Request one GPIO.
  - `int gpio_request_array(struct gpio *array, size_t num);` – Request multiple GPIOs.

- `int gpio_export(unsigned int gpio, bool direction);`
  For debugging purposes, you can export the GPIO which is allocated using the `gpio_request()` to the sysfs using the above API, where
  `gpio` is the GPIO pin that you want to export.
  `direction` controls whether user space is allowed to change the direction of the GPIO: True – Allow change, False – Disallow change. Returns zero on success, else an error.
  Once you export the GPIO, you can see the GPIO in `/sys/class/gpio/*`. There you can the GPIO's value, etc.

- `void gpio_unexport(unsigned int gpio)`
  If you have exported the GPIO using the gpio_export(), then you can unexport this using the above API, where,
  `gpio` is the GPIO pin that you want to unexport.

- `int  gpio_direction_input(unsigned gpio)`
  The above API is used to set the GPIO as output or input, where
  `gpio` is the GPIO pin that you want to set the direction as input.
  The API returns zero on success, else an error.

- `int  gpio_direction_output(unsigned gpio, int value)`
  This API is used to set the GPIO direction as output, where
  `gpio` is the GPIO pin that you want to set the direction as output.
  `value` represents the value of the GPIO pin once the output direction is effective.
  The API returns zero on success, else an error.

- `void gpio_set_value(unsigned int gpio, int value);`
  Once you set the GPIO direction as an output, then you can use the API to change the GPIO value, where
  `gpio` is the GPIO pin that you want to change the value.
  `value` is thee value to set to the GPIO pin. 0 for Low, 1 for High.

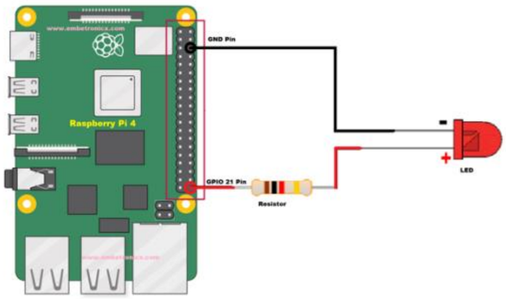
- `int  gpio_get_value(unsigned gpio);`
  You can read the GPIO's value using the API, where
  `gpio` is the GPIO pin that you want to read the value.
  The API returns the GPIO's value, either 0 or 1.

- `void gpio_free(unsigned int gpio);`
  Once you have done with the GPIO, then you need to use this API to release the GPIO which you have allocated previously.
  `gpio` is the GPIO pin that you want to release.
  There are other variants also available. You can use any one of them based on your need.
    - `void gpio_free_array(struct gpio *array, size_t num);` – This API releases multiple GPIOs.

6 Example: Controlling LED via GPIO

This example connects 1 LED in the GPIO 21. The program turns ON the LED by writing "1" to the driver, and turns OFF the LED by writing "0."

- Connection



- Code

```
/***************************************************************************//**
*  \file       led_driver.c
*  \details    Simple GPIO driver explanation
*  \author     EmbeTronicX
*  \Tested with Linux raspberrypi 5.4.51-v7l+
*******************************************************************************/
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/delay.h>
#include <linux/uaccess.h>  //copy_to/from_user()
#include <linux/gpio.h>     //GPIO

//LED is connected to this GPIO
#define GPIO_21 (21)

dev_t dev = 0;
static struct class *dev_class;
static struct cdev etx_cdev;

static int __init etx_driver_init(void);
static void __exit etx_driver_exit(void);


/*************** Driver functions **********************/
static int      etx_open(struct inode *inode, struct file *file);
static int      etx_release(struct inode *inode, struct file *file);
static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len,loff_t * off);
static ssize_t etx_write(struct file *filp,
                const char *buf, size_t len, loff_t * off);
/******************************************************/

//File operation structure
static struct file_operations fops =
{
  .owner          = THIS_MODULE,
  .read           = etx_read,
  .write          = etx_write,
  .open           = etx_open,
  .release        = etx_release,
```

```
};

/*
** This function will be called when we open the Device file
*/
static int etx_open(struct inode *inode, struct file *file)
{
  pr_info("Device File Opened...!!!\n");
  return 0;
}

/*
** This function will be called when we close the Device file
*/
static int etx_release(struct inode *inode, struct file *file)
{
  pr_info("Device File Closed...!!!\n");
  return 0;
}

/*
** This function will be called when we read the Device file
*/
static ssize_t etx_read(struct file *filp,
                char __user *buf, size_t len, loff_t *off)
{
  uint8_t gpio_state = 0;

  //reading GPIO value
  gpio_state = gpio_get_value(GPIO_21);

  //write to user
  len = 1;
  if( copy_to_user(buf, &gpio_state, len) > 0) {
    pr_err("ERROR: Not all the bytes have been copied to user\n");
  }

  pr_info("Read function : GPIO_21 = %d \n", gpio_state);

  return 0;
}

/*
** This function will be called when we write the Device file
*/
static ssize_t etx_write(struct file *filp,
                const char __user *buf, size_t len, loff_t *off)
{
  uint8_t rec_buf[10] = {0};

  if( copy_from_user( rec_buf, buf, len ) > 0) {
    pr_err("ERROR: Not all the bytes have been copied from user\n");
  }

  pr_info("Write Function : GPIO_21 Set = %c\n", rec_buf[0]);

  if (rec_buf[0]=='1') {
    //set the GPIO value to HIGH
    gpio_set_value(GPIO_21, 1);
  } else if (rec_buf[0]=='0') {
    //set the GPIO value to LOW
    gpio_set_value(GPIO_21, 0);
  } else {
```

```
    pr_err("Unknown command : Please provide either 1 or 0 \n");
  }

  return len;
}

/*
** Module Init function
*/
static int __init etx_driver_init(void)
{
  /*Allocating Major number*/
  if((alloc_chrdev_region(&dev, 0, 1, "etx_Dev")) <0){
    pr_err("Cannot allocate major number\n");
    goto r_unreg;
  }
  pr_info("Major = %d Minor = %d \n",MAJOR(dev), MINOR(dev));

  /*Creating cdev structure*/
  cdev_init(&etx_cdev,&fops);

  /*Adding character device to the system*/
  if((cdev_add(&etx_cdev,dev,1)) < 0){
    pr_err("Cannot add the device to the system\n");
    goto r_del;
  }

  /*Creating struct class*/
  if((dev_class = class_create(THIS_MODULE,"etx_class")) == NULL){
    pr_err("Cannot create the struct class\n");
    goto r_class;
  }

  /*Creating device*/
  if((device_create(dev_class,NULL,dev,NULL,"etx_device")) == NULL){
    pr_err( "Cannot create the Device \n");
    goto r_device;
  }

  //Checking the GPIO is valid or not
  if(gpio_is_valid(GPIO_21) == false){
    pr_err("GPIO %d is not valid\n", GPIO_21);
    goto r_device;
  }

  //Requesting the GPIO
  if(gpio_request(GPIO_21,"GPIO_21") < 0){
    pr_err("ERROR: GPIO %d request\n", GPIO_21);
    goto r_gpio;
  }

  //configure the GPIO as output
  gpio_direction_output(GPIO_21, 0);

  /* Using this call the GPIO 21 will be visible in /sys/class/gpio/
  ** Now you can change the gpio values by using below commands also.
  ** echo 1 > /sys/class/gpio/gpio21/value  (turn ON the LED)
  ** echo 0 > /sys/class/gpio/gpio21/value  (turn OFF the LED)
  ** cat /sys/class/gpio/gpio21/value  (read the value LED)
  **
  ** the second argument prevents the direction from being changed.
  */
  gpio_export(GPIO_21, false);
```

```
  pr_info("Device Driver Insert...Done!!!\n");
  return 0;

r_gpio:
  gpio_free(GPIO_21);
r_device:
  device_destroy(dev_class,dev);
r_class:
  class_destroy(dev_class);
r_del:
  cdev_del(&etx_cdev);
r_unreg:
  unregister_chrdev_region(dev,1);

  return -1;
}

/*
** Module exit function
*/
static void __exit etx_driver_exit(void)
{
  gpio_unexport(GPIO_21);
  gpio_free(GPIO_21);
  device_destroy(dev_class,dev);
  class_destroy(dev_class);
  cdev_del(&etx_cdev);
  unregister_chrdev_region(dev, 1);
  pr_info("Device Driver Remove...Done!!\n");
}

module_init(etx_driver_init);
module_exit(etx_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("EmbeTronicX <embetronicx@gmail.com>");
MODULE_DESCRIPTION("A simple device driver - GPIO Driver");
MODULE_VERSION("1.32");
```

- Reference Makefile

```
obj-m += led_driver.o
KDIR = /lib/modules/$(shell uname -r)/build
all:
	make -C $(KDIR)  M=$(shell pwd) modules
clean:
	make -C $(KDIR)  M=$(shell pwd) clean
```

The above box shows the reference makefile. Since the `build` in `lib/modules` links to the directory in Ubuntu, and `/home/usr1/linux` does not exist on RPi (`usr1` is the user ID in the host Ubuntu), so you cannot build the driver on RPi. Instead, you must build the driver on the host and then copy the built driver module to RPi after success.
The compilation may have some errors. Try to figure out how to solve it.

- Test your code
  - Build the driver by using Makefile (`sudo make`)
  - Load the driver using `sudo insmod led_driver.ko`

- use `sudo su` and enter the password if required to get the root permission
- `echo 1 > /dev/etx_device` [This must turn ON the LED].
- `echo 0 > /dev/etx_device` [This must turn OFF the LED].
- `cat /dev/etx_device` [Read the GPIO value].

Lab 4: Driver (Part II)

(參考資料來源：https://www.raspberrypi.com/documentation/computers/getting-started.html, https://kokkonisd.github.io/2020/10/24/linux-drivers-rpi/)

1 Objective

- Be familiar with designing a driver.

2 Prerequisite

- Read Linux Device Driver.

1 Basic Driver

A simple driver can be implemented in couple lines.

```
#include <linux/module.h>


static int __init hello_world_init (void)
{
    pr_info("Hello, World!\n");

    return 0;
}


static void __exit hello_world_exit (void)
{
    pr_info("Goodbye, World!\n");
}


module_init(hello_world_init);
module_exit(hello_world_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Dimitri Kokkonis (kokkonisd@gmail.com)");
MODULE_DESCRIPTION("A simple hello world kernel module");
```

Here's a basic breakdown of this driver:

- line 1: import the linux module header file, because this is a *kernel module* ; any calls to userland functions (such as `printf`) make no sense in the kernel.
- lines 4-9: the initialization entry point. We have the `static` keyword because this function should only be defined and used in this file, and `__init` because it's good

practice (even though this is a dynamic module so the `__init` directive will have no effect). We then just print a “hello world” message and return 0, like a classic `main` “hello world” function would.

- lines 12-15: the de-initialization entry point. Similarly, `static` and `__exit` are used; the latter is needed since dynamic modules can be unloaded/removed (but again it is good practice to always have these directives). Once more, we print a message using kernel functions.
- lines 18-19: assignment of the entry points using kernel macros `module_init` and `module_exit`.
- lines 21-23: various kernel module info (not really necessary in this case but might as well showcase them).

This driver should simply print “Hello, World!” when inserted into the kernel, and “Goodbye, World!” when removed from it.

2 Compiling for X86 Desktop

You can compile the driver source with the Makefile as following.

```
CC=gcc

obj-m:= hello_world.o

all:
	make -C /lib/modules/5.4.0-52-generic/build M=$(PWD) modules
clean:
	make -C /lib/modules/5.4.0-52-generic/build M=$(PWD ) clean
```

You need to compile the driver code against the current kernel in `/lib/modules/<kernel version>/build/`. You can obtain your kernel version by running `uname -r`; in this case, the version returned by `uname -r` is 5.4.0-52-gneric, so the path should be `/lib/modules/5.4.0-52-generic/build`:

Now, you can just build the driver with the `make` utility:

```
$ sudo make all
```

In this example, the source file is `hello_world.c`, and the output file is `hello_world.ko`, where `ko` stands for kernel object.

Now you can insert your module into the kernel:

```
$ sudo insmod hello_world.ko
```

If errors are produced during this step, you probably compiled against the wrong kernel version. If everything goes well, you can verify that your module works by running `dmesg`:

```
$ dmesg
...
[ 7667.013583] Hello, World!
```

To verify that the exit function of the module works as well, you can remove the module by `rmmod` and run `dmesg` again:

```
$ sudo rmmod hello_world.ko
$ dmesg
...
[ 7667.013583] Hello, World!
[ 7672.605206] Goodbye, World!
```

3 Compiling for Raspberry Pi

Get back to the Raspberry Pi OS you've built in Lab2. Modify the `Makefile` for compiling the driver module.

```
obj-m := hello_world.o

all: x86_build #or RPI_build, you can specify only one build at a time

clean: x86_clean RPI_clean

x86_build: hello_world.c
	sudo make -C /lib/modules/5.4.0-52-generic/build M=$(PWD) modules

x86_clean:
	sudo make -C /lib/modules/5.4.0-52-generic/build M=$(PWD) clean

RPI_build: hello_world.c
	sudo make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -C $(PWD)/linux M=$(PWD) modules

RPI_clean:
	sudo make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -C $(PWD)/linux M=$(PWD) clean

.PHONY: all clean x86_build x86_clean RPI_build RPI_clean
```

You should now be able to run `make RPI_build` to cross-compile your driver. If a `hello_world.ko` file is produced without errors, you can transfer it over to the RPI to test it on the your Raspberry Pi (if IP is 192.168.1.2):

```
$ scp hello_world.ko pi@192.168.1.2:~/
```

You can then `ssh` into the RPI and load/unload the module to check if your driver works correctly:

```
$ ssh pi@192.168.1.2
$ sudo insmod hello_world.ko
$ sudo rmmod hello_world.ko
$ dmesg
...
[ 2063.017314] Hello, World!
[ 2070.706195] Goodbye, World!
```

4 Design your Driver

Upon designing your driver, you need to define the reactions of the actions which may be done on your device, such as, read, write, open, we should use the library linux/fs.h, which defines the basic operations use on file. (Everything is seen as a **file** in Linux, including a device.)
The definition of the file operations in the `linux/fs.h` are showed as following.

```
struct file_operations {
        struct module *owner;
```

```
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
        int (*iterate) (struct file *, struct dir_context *);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, loff_t, loff_t, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned
long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
size_t, unsigned int);
        int (*setlease)(struct file *, long, struct file_lock **, void **);
        long (*fallocate)(struct file *file, int mode, loff_t offset,
                          loff_t len);
        void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
        unsigned (*mmap_capabilities)(struct file *);
#endif
        ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
                        loff_t, size_t, unsigned int);
        int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
                        u64);
        ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
                        u64);
};
```

You don't have to define all the operations for your driver. The following example only defines `read()`, `write()` and `open()`.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
MODULE_LICENSE("GPL");

// File Operations
static ssize_t my_read(struct file *fp, char *buf, size_t count, loff_t *fpos) {

       printk("call read\n");

       return count;
}

static ssize_t my_write(struct file *fp,const char *buf, size_t count, loff_t *fpos)
{
```

```
        printk("call write\n");

        return count;
}

static int my_open(struct inode *inode, struct file *fp) {

        printk("call open\n");

        return 0;
}

struct file_operations my_fops = {
        read:  my_read,
        write: my_write,
        open:  my_open
};

#define MAJOR_NUM 244
#define DEVICE_NAME "my_dev"

static int my_init(void) {
        printk("call init\n");
        if(register_chrdev(MAJOR_NUM, DEVICE_NAME, &my_fops) < 0) {
                        printk("Can not get major %d\n", MAJOR_NUM);
                        return (-EBUSY);
        }

        printk("My device is started and the major is %d\n", MAJOR_NUM);
        return 0;
}

static void my_exit(void) {
        unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
        printk("call exit\n");
}


module_init(my_init);
module_exit(my_exit);
```

To declare the relationship between `my_read()` and the `read()`, you need to initialize a structure file operations called `my_fops`. So now, let's register a character device with `my_fops` to the system when the driver is initialized. And don't forget to unregister the character device when the driver is removed.
To let your device driver shown in `/dev`, you should load your module first and get the major number (244, in this example). And then you can create the device node with the major number.

```
$ mknod /dev/mydev c 244 0
```

In the above command, `c` stands for a **c**haracter device driver. 244 is the major number and 0 is the minor number.
Now, you can test your own device driver!

5 Miscellaneous

Some other important functions you may need to know in implementing your driver:

- `copy_from_user(void* to, const void __user * from, unsigned long n)`

This function copies a block of data from user space, where

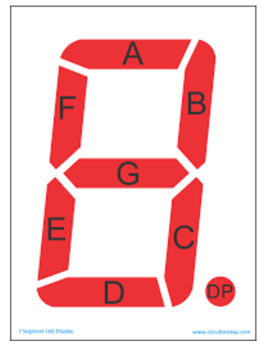
`to` is the destination address in **kernel** space;
`from` is the source address in **user** space;
`n` is the number of bytes to copy.

- `copy_to_user(void __user * to, const void * from, unsigned long n)`

This function copy a block of data into user space, where

`to` is the destination address, in user space.
`From` is the source address, in kernel space.
`n` is the number of bytes to copy.

6 Virtual Device and its Driver



Since we do not have real hardware device, let's create a virtual device in our Raspberry Pi. The virtual device is a 7-Segment Device. Its driver provides a `read()` and a `write()` operations. The `read()` reads the status of the virtual device. The `write()` needs to update the status (segment) of the virtual device. A reader program (reader.cpp, shown as follows) running on the Raspberry Pi connects to a remote server and sends the status read from the virtual device to the server. The status is represented by an array S with the length of eight (or seven), where S[0] represents segment A, S[1] represents segment B, …, and value 0 means OFF and 1 means ON. Once the server receives the array, the server shows the result of the 7-segment (virtual device.)

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <unistd.h>

int main(int argc, char *argv[]) {

       if(argc != 4) {
              printf("Usage: reader <server_ip> <port> <device_path>");
              exit(-1);
       }

       /* Socket client setup*/

       // setup
       int port = (u_short)atoi(argv[2]);
       int connfd = socket(AF_INET, SOCK_STREAM, 0);
       struct sockaddr_in client_sin;
       memset(&client_sin, 0, sizeof(client_sin));
       client_sin.sin_family = AF_INET;
       client_sin.sin_addr.s_addr = inet_addr(argv[1]);
       client_sin.sin_port = htons(port);

       // connect
```

```
	if(connect(connfd, (struct sockaddr *) &client_sin, sizeof(client_sin)) == -
1) {
		printf("Error connect to server\n");
		exit(-1);
	}

	int fd;
	if((fd = open(argv[3], O_RDWR)) < 0) {
		printf("Error open %s\n",argv[3]);
		exit(-1);
	}

	char buf[8] = {0};
	while(1) {
		sleep(1);

		if( read(fd, buf, 7) < 0) {
			printf("Error read %s\n",argv[3]);
			exit(-1);
		}

		// sent msg
		int n;
		if((n = write(connfd, buf, 7)) == -1) {
			printf("Error write to server\n");
			exit(-1);
		}
	}

	close(fd);
	close(connfd);
	return 0;
}
```

Please run the server in your VM according to the instruction provided by TA and complete the implementation of your driver.

Lab 5: Task

1 Objective

- Be familiar with system calls: `fork()`, `wait()`, `waitpid()`, etc.
- Be familiar with POSIX programming: `pthread_create()`, `pthread_exit()`, etc.

2 Prerequisite

- Read man pages of the above system calls.

3 Process Control

`fork()` creates a child process that differs from the parent process only in its PID and PPID, and in fact that resource utilizations are set to 0. The memory of child process is copied from the parent and a new process structure is assigned by the kernel. The environment, resource limits, controlling terminal, current working directory, root directory, signal masks and other process resources are also duplicated from the parent in the forked child process, while file locks and pending signals are not inherited. The return value of `fork()` discriminates the two processes of execution. A zero is returned by the fork function in the child's process, while the parent process gets the PID (a non-zero integer) of its child.

```
/*
 * process.c
 */

#include <errno.h>      /* Errors */
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <sys/types.h>  /* Primitive System Data Types */
#include <sys/wait.h>   /* Wait for Process Termination */
#include <unistd.h>     /* Symbolic Constants */

pid_t childpid; /* variable to store the child's pid */

void childfunc(void)
{
       int retval;     /* user-provided return code */

       printf("CHILD: I am the child process!\n");
       printf("CHILD: My PID: %d\n", getpid());
       printf("CHILD: My parent's PID is: %d\n", getppid());
       printf("CHILD: Sleeping for 1 second...\n");
       sleep(1); /* sleep for 1 second */

       printf("CHILD: Enter an exit value (0 to 255): ");
       scanf(" %d", &retval);
       printf("CHILD: Goodbye!\n");

       exit(retval); /* child exits with user-provided return code */
```

```
}

void parentfunc(void)
{
	int status;	/* child's exit status */

	printf("PARENT: I am the parent process!\n");
	printf("PARENT: My PID: %d\n", getpid());
	printf("PARENT: My child's PID is %d\n", childpid);
	printf("PARENT: I will now wait for my child to exit.\n");

	/* wait for child to exit, and store its status */
	wait(&status);
	printf("PARENT: Child's exit code is: %d\n", WEXITSTATUS(status));
	printf("PARENT: Goodbye!\n");

	exit(0);  /* parent exits */
}

int main(int argc, char *argv[])
{
	/* now create new process */
	childpid = fork();

	if (childpid >= 0) { /* fork succeeded */
		if (childpid == 0) { /* fork() returns 0 to the child process */
			childfunc();
		} else { /* fork() returns new pid to the parent process */
			parentfunc();
		}
	} else { /* fork returns -1 on failure */
		perror("fork"); /* display error message */
		exit(0);
	}

	return 0;
}
```

You can check the identifiers of the parent and child process by executing ps command in shell. Please refer to its man page for more details.

- `wait()` suspends execution of the current process until one of its children terminates.
- `waitpid()` suspends execution of the current process until a child specified by pid argument has changed state. By default, `waitpid()` waits only for terminated children, but this behavior is modifiable via the options argument. Please refer to its man page.

```
/*
 * waitpid.c -- shows how to get child's exit status
 */

#include <errno.h>	/* Errors */
#include <stdio.h>	/* Input/Output */
#include <stdlib.h>	/* General Utilities */
#include <sys/types.h>	/* Primitive System Data Types */
#include <sys/wait.h>	/* Wait for Process Termination */
#include <time.h>	/* Time functions */
#include <unistd.h>	/* Symbolic Constants */

pid_t childpid; /* variable to store the child's pid */

void childfunc(void)
```

```
{
	int randtime;		/* random sleep time */
	int exitstatus;		/* random exit status */

	printf("CHILD: I am the child process!\n");
	printf("CHILD: My PID: %d\n", getpid());

	/* sleep */
	srand(time(NULL));
	randtime = rand() % 5;
	printf("CHILD: Sleeping for %d second...\n", randtime);
	sleep(randtime);

	/* rand exit status */
	exitstatus = rand() % 2;
	printf("CHILD: Exit status is %d\n", exitstatus);

	printf("CHILD: Goodbye!\n");
	exit(exitstatus); /* child exits with user-provided return code */
}

void parentfunc(void)
{
	int status;	/* child's exit status */
	pid_t pid;

	printf("PARENT: I am the parent process!\n");
	printf("PARENT: My PID: %d\n", getpid());


	printf("PARENT: I will now wait for my child to exit.\n");

	/* wait for child to exit, and store its status */
	do
	{
		pid = waitpid(childpid, &status, WNOHANG);
		printf("PARENT: Waiting child exit ...\n");
		sleep(1);
	}while (pid != childpid);

	if (WIFEXITED(status)) {
		// child process exited normally.
		printf("PARENT: Child's exit code is: %d\n",
				WEXITSTATUS(status));
	}else{
		// Child process exited thus exec failed.
		// LOG failure of exec in child process.
		printf("PARENT: Child process executed but exited failed.\n");
	}

	printf("PARENT: Goodbye!\n");

	exit(0);  /* parent exits */
}

int main(int argc, char *argv[])
{
	/* now create new process */
	childpid = fork();

	if (childpid >= 0) { /* fork succeeded */
		if (childpid == 0) { /* fork() returns 0 to the child process */
			childfunc();
```

```
                } else { /* fork() returns new pid to the parent process */
                        parentfunc();
                }
        } else { /* fork returns -1 on failure */
                perror("fork"); /* display error message */
                exit(0);
        }

        return 0;
}
```

- `nice()` adds `incr` to the nice value for the calling process. (A higher nice value means a low priority.) Only the superuser may specify a negative increment, or priority increase. The range for nice values is described in `getpriority(2)`.

```
#include <uni s td . h>
. . .
int incr = -20;
int ret ;

ret = nice(incr) ;
```

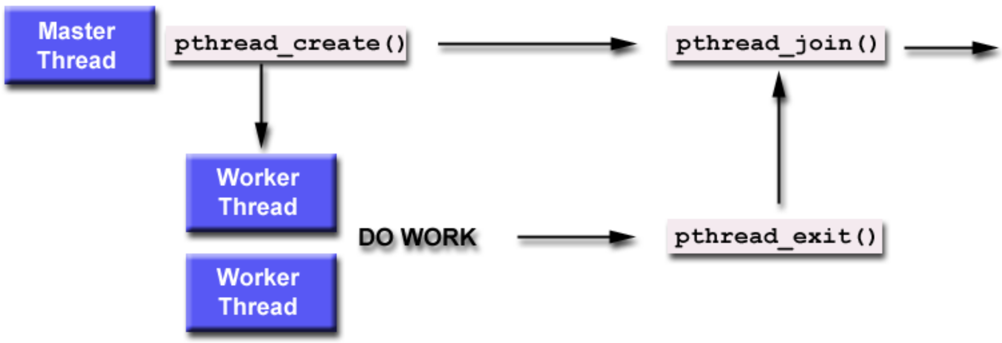
- The `exec()` family of functions initiates a new process image within a program. The initial argument for these functions is the pathname of a file which is to be executed.

```
/*
 * exec.c
 */

#include <unistd.h>
int main (int argc , char *argv[])
{
   execl("/bin/ls", "/bin/ls", "-r", "-t", "-l", (char *) 0);

   return 0 ;
}
```

4 Thread

A standardized programming interface was required to take full advantages provided by threads. For UNIX systems, this interface has been speci ed by the IEEE POSIX 1003.1c standard (1995). Implementations adhering to this standard are referred to as POSIX threads, or pthreads. `pthread_create()` creates a new thread, with attributes specified by `attr`, within a process. Upon successful completion, `pthread_create()` will store the ID of the created thread in the location specified by the thread. For more information, please see its man page.

- `pthread_join()` suspends execution of the calling thread until the target thread terminates.
- `pthread detach()` tells the underlying system that resources allocated to a particular thread can be reclaimed once it terminates. This function should be used when an exit status is not required by other threads.
- `pthread_exit()` terminates the calling thread.



The above APIs are included in the header file, `pthread.h'.

```
/*
 * pthread.c -- shows how to create a thread
 */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 5

void *show(void *threadid)
{
	long tid;

	tid = (long)threadid;
	printf("Hello! I am thread #%ld!\n", tid);

	pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
	pthread_t threads[5];
	int rc;
	int t;

	for(t=0;t<NUM_THREADS;t++){
		printf("In main(): creating thread %d\n", t);
		rc = pthread_create(&threads[t], NULL, show, (void *)t);
		if (rc){
			printf("ERROR; pthread_create() returns %d\n", rc);
			exit(-1);
		}
	}

	printf("Main: program completed. Exiting.\n");
	pthread_exit(NULL);
}
```

```
/*
 * join.c -- shows how to "wait" for thread completions
 */

#include <math.h>
#include <pthread.h>
#include <stdio.h>
```

```
#include <stdlib.h>

#define NUM_THREADS 4

void *BusyWork(void *t)
{
	int i;
	long tid;
	double result=0.0;

	tid = (long)t;
	printf("Thread %ld starting...\n",tid);

	for (i = 0; i < 1000000; i++)
	{
		result += sin(i) * tan(i);
	}

	printf("Thread %ld done. Result = %e\n", tid, result);
	pthread_exit((void*) t);
}

int main (int argc, char *argv[])
{
	pthread_t thread[NUM_THREADS];
	pthread_attr_t attr;
	int rc;
	long t;
	void *status;

	/* Initialize and set thread detached attribute */
	pthread_attr_init(&attr);
	pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

	for (t = 0; t < NUM_THREADS; t++) {
		printf("Main: creating thread %ld\n", t);
		rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
		if (rc) {
			printf("ERROR; pthread_create() returns %d\n", rc);
			exit(-1);
		}
	}

	/* Free attribute and wait for the other threads */
	pthread_attr_destroy(&attr);
	for (t = 0; t < NUM_THREADS; t++) {
		rc = pthread_join(thread[t], &status);
		if (rc) {
			printf("ERROR; pthread_join() returns %d\n", rc);
			exit(-1);
		}
		printf("Main: join with thread %ld (status: %ld)\n",t,(long)status);
	}

	printf("Main: program completed. Exiting.\n");
	pthread_exit(NULL);
}
```

```
/*
 * deteach.c -- shows how to detach a thread
 */

#include <errno.h>
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void *threadfunc(void *parm)
{
	printf("Inside secondary thread\n");
	sleep(3);
	printf("Exit secondary thread\n");
	pthread_exit(NULL);
}

int main(int argc, char **argv)
{
	pthread_t thread;
	int rc = 0;

	printf("Create a thread using attributes that allow detach\n");
	rc = pthread_create(&thread, NULL, threadfunc, NULL);
	if (rc) {
		printf("ERROR; pthread_create() returns %d\n", rc);
		exit(-1);
	}

	printf("Detach the thread after it terminates\n");
	rc = pthread_detach(thread);
	if (rc) {
		printf("ERROR; pthread_detach() returns %d\n", rc);
		exit(-1);
	}

	printf("Detach the thread again\n");
	rc = pthread_detach(thread);
	/* EINVAL: No thread could be found corresponding to that
	 * specified by the given thread ID.
	 */
	if (rc != EINVAL) {
		printf("Got an unexpected result! rc=%d\n", rc);
		exit(1);
	}
	printf("Second detach fails as expected.\n");

	/* sleep() is not a very robust way to wait for the thread */
	sleep(6);
	printf("Main() completed.\n");
	return 0;
}
```

The link to `libpthread.a` library should be specified to the `gcc` compiler when compiling a program with `pthread` functions.

In a X86 desktop, you can natively compile your code with `gcc` by:

```
$gcc -o pthread pthread.c -lpthread
```

In a Raspberry Pi, find the way to cross-compile your code and make it runnable.

Lab 6: Inter-Process Communication

1 Objective

- Be familiar with inter-process communication: semaphore, mutex, etc.

2 Prerequisite

- Read man pages of `semop()`, `semctl()`, `semget()`, `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pipe()`, `shmget()`, `shmctl()`, `shmat()`, `shmdt()`, etc.

3 Semaphore

Unix allocates arrays of semaphores, rather than creating them one at a time. When you create such an array, you must supply the following information:

- An integer called the "key" which acts as the semaphore array's "name" on the system
- The number of semaphores in the array
- Ownership of semaphore array (permission bits/mode)

The system call `semget(2)` is used by a program to ask the operating system if it knows of a semaphore. The program passes the semaphore's key. If the semaphore exists, then the operating system returns an integer which is used by the program as the semaphore's identifier for the duration of that program. (NOTE: the semaphore key is permanent for as long as the semaphore exists. The semaphore ID returned by the operating system is just a temporary "handle" for accessing the semaphore and may be different each time.) `semget(2)` can also be used to create semaphores that don't exist by passing additional options. `semctl(2)` is used to set the value of a semaphore, remove it from the system, etc. Think of it as the way to "manage" the semaphore array. At it's simplest, `semop(2)` is used to implement `P()` (wait) and `V()` (signal). However, `semop(2)` can do multiple semaphore operations with one call. For our programs, we will only be creating arrays with one semaphore on them and doing only one operation at a time. Run the following commands on both your Linux host and target, and observe the status of System V IPC status.

```
$ ipcs
$ ipcs -s
```

4 Example

4.1 Create Semaphore: makesem

Here is a program, `makesem.c`. Compile it and run it with two parameters:

- the first parameter should be a large number;
- the second one is number '1' or '0'.

For example: `makesem 428361733 1`.

Now run `ipcs` again. You should see your semaphore in the list. Note that the key is the number you entered that converted to hex.

```
/* makesem.c
 *
 * This program creates a semaphore. The user should pass
 * a number to be used as the semaphore key and initial
 * value as the only command line arguments. If that
 * identifier is not taken, then a semaphore will be created.
 * If a semaphore is set so that then no semaphore will be
 * created. The semaphore is set so that anyone on the system
 * can use it.
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sem.h>

#define SEM_MODE 0666 /* rw(owner)-rw(group)-rw(other) permission */

int main (int argc, char **argv)
{
	int s;
	long int key;
	int val;

	if (argc != 3)
	{
		fprintf(stderr,
			"%s: specify a key (long) and initial value (int)\n",
			argv[0]);

		exit(1);
	}

	/* get values from command line */
	if (sscanf(argv[1], "%ld", &key) != 1)
	{
		/* convert arg to long integer */
		fprintf(stderr, "%s: argument #1 must be an long integer\n",
				argv[0]);
		exit(1);
	}
	if (sscanf(argv[2], "%d", &val) != 1)
	{
		/* convert arg to long integer */
		fprintf(stderr, "%s: argument #2 must be an integer\n",
				argv[0]);
		exit(1);
	}

	/* semget() takes three parameters */
	/* Options:
	 *     IPC_CREAT - create a semaphore if not exists
	 *     IPC_EXCL  - creation fails if it already exists
	 *     SEM_MODE  - access permission
	 */
	s = semget(
```

```
                    key, /* the unique name of the semaphore on the system */
                    1,   /* we create an array of semaphores, but just need 1.
*/
                    IPC_CREAT | IPC_EXCL | SEM_MODE);

       /* If semget () returns -1 then it failed. However,
        * if it returns any other number >= 0 then that becomes
        * the identifier within the program for accessing the semaphore.
        */
       if (s < 0)
       {
              fprintf(stderr,
                            "%s: creation of semaphore %ld failed: %s\n", argv[0],
                            key, strerror(errno));
              exit(1);
       }
       printf("Semaphore %ld created\n", key);

       /* set semaphore (s[0]) value to initial value (val) */
       if ( semctl(s, 0, SETVAL, val) < 0 )
       {
              fprintf(stderr,
                            "%s: Unable to initialize semaphore: %s\n",
                            argv[0], strerror(errno));
              exit(0);
       }
       printf("Semaphore %ld has been initialized to %d\n", key, val);

       return 0;
}
```

4.2 Remove Semaphore: rmsem

Program `rmsem.c` removes the semaphore identified by its key from the system. Compile and run the program with your key from the last step, then run `ipcs` to see if your semaphore is still listed.

NOTE: the easiest way to create the file is probably to copy `makesem.c` and make modifications.

```
/* rmsem.c
 *
 * This program destroys a semaphore. The user should pass a number
 * to be used as the semaphore key.
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/sem.h>

int main(int argc, char **argv)
{
       int s;
       long int key;

       if (argc != 2)
       {
              fprintf(stderr, "%s: specify a key (long)\n", argv[0]);
              exit(1);
```

```
        }

        /* get values from command line */
        if (sscanf(argv[1], "%ld", &key)!=1)
        {
               /* convert arg to long integer */
               fprintf(stderr,
                      "%s: argument #1 must be an long integer\n", argv[0]);
               exit(1);
        }

        /* find semaphore */
        s = semget(key,1,0);
        if (s < 0)
        {
               fprintf(stderr,     "%s: failed to find semaphore %ld: %s\n",
                      argv[0], key, strerror(errno));
               exit(1);
        }
        printf("Semaphore %ld found\n",key);

        /* remove semaphore */
        if (semctl (s, 0, IPC_RMID, 0) < 0)
        {
               fprintf (stderr, "%s: unable to remove semaphore %ld\n",
                      argv[0], key);
               exit(1);
        }
        printf("Semaphore %ld has been remove\n", key);

        return 0;
}
```

4.3 Use Semaphore: doodle

The program `doodle.c` has definitions for the operations `P()` (wait) and `V()` (signal). The program first "opens" the semaphore with `semget()`, then it waits for the user to type in a small integer number (number of seconds to stay in the critical section). It then performs `P()` on a semaphore. Once inside the critical section, it doodles for the number of seconds you specified, then it leaves performing `V()` on the semaphore.

To demonstration the operation of doodle, please run the following two experiments:

1. Use `makesem` to create a semaphore with initial value 1 and run three doodle programs to acquire the same semaphore simultaneously.
2. Use `makesem` to create a semaphore with initial value 2 and run three doodle programs to acquire the same semaphore simultaneously.

Observe the inter-operations between the programs.

```
/* doodle.c
 *
 * This program shows how P () and V () can be implemented,
 * then uses a semaphore that everyone has access to.
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <sys/sem.h>
#include <unistd.h>

#define DOODLE_SEM_KEY 1122334455

/* P () - returns 0 if OK; -1 if there was a problem */
int P(int s)
{
	struct sembuf sop;  /* the operation parameters */
	sop.sem_num =  0;   /* access the 1st (and only) sem in the array */
	sop.sem_op  = -1;	    /* wait..*/
	sop.sem_flg =  0;	    /* no special options needed */

	if (semop (s, &sop, 1) < 0) {
		fprintf(stderr,"P(): semop failed: %s\n",strerror(errno));
		return -1;
	} else {
		return 0;
	}
}

/* V() - returns 0 if OK; -1 if there was a problem */
int V(int s)
{
	struct sembuf sop;  /* the operation parameters */
	sop.sem_num =  0;   /* the 1st (and only) sem in the array */
	sop.sem_op  =  1;   /* signal */
	sop.sem_flg =  0;   /* no special options needed */

	if (semop(s, &sop, 1) < 0) {
		fprintf(stderr,"V(): semop failed: %s\n",strerror(errno));
		return -1;
	} else {
		return 0;
	}
}

int main ( int argc, char **argv)
{
	int s, secs;
	long int key;

	if  (argc != 2) {
		fprintf(stderr, "%s: specify a key (long)\n", argv[0]);
		exit(1);
	}

	/* get values from command line */
	if (sscanf(argv[1], "%ld", &key)!=1)
	{
		/* convert arg to long integer */
		fprintf(stderr,
			"%s: argument #1 must be an long integer\n", argv[0]);
		exit(1);
	}


	s = semget(key, 1, 0);

	if (s < 0) {
		fprintf (stderr,
			"%s: cannot find semaphore %ld: %s\n",
			argv[0], key, strerror(errno));
```

```
                exit(1);
        }

        while (1) {
                printf ("#secs to doodle in the critical section? (0 to exit):");
                scanf ("%d",&secs);

                if (secs == 0)
                        break;

                printf ("Preparing to enter the critical section..\n");
                P(s);
                printf ("Now in the critical section! Sleep %d secs..\n", secs);

                while (secs) {
                        printf ("%d...doodle...\n",secs--);
                        sleep (1);
                }

                printf ("Leaving the critical section..\n");
                V(s);
        }

        return 0;
}
```

5 Race Condition

Race conditions arise in software when separate processes or threads of execution depend on some shared state. Operations upon shared states are critical sections that must be mutually exclusive in order to avoid harmful collision between processes or threads that share those states. Assume that two threads (T1 and T2) want to increment the value of a global integer by one. Ideally, the following sequence of operations would take place:

- Integer `i = 0`; (memory)
- T1 reads the value of `i` from memory into register1: 0
- T1 increments the value of `i` in register1: (register1 contents) + 1 = 1
- T1 stores the value of register1 in memory: 1
- T2 reads the value of `i` from memory into register2: 1
- T2 increments the value of `i` in register2: (register2 contents) + 1 = 2
- T2 stores the value of register2 in memory: 2
- Integer `i` = 2; (memory)

In the above case, the final value of `i` is 2, as expected. However, if the two threads run simultaneously without locking or synchronization, the outcome of the operation could be wrong. The alternative sequence of operations below demonstrates this scenario:

- Integer `i` = 0; (memory)
- T1 reads the value of `i` from memory into register1: 0
- T2 reads the value of `i` from memory into register2: 0
- T1 increments the value of `i` in register1: (register1 contents) + 1 = 1
- T2 increments the value of `i` in register2: (register2 contents) + 1 = 1
- T1 stores the value of register1 in memory: 1
- T2 stores the value of register2 in memory: 1

- Integer `i` = 1; (memory)

The final value of `i` is 1 instead of the expected result of 2. This occurs because the increment operations of the second case are not mutually exclusive.
The following example shows a race condition between two processes.
Please create a file `counter.txt` and set initial value 0 to the file by performing “`echo 0 > counter.txt`.” Compile the codes with and without the flag '`-D USE_SEM`' and check the result in `counter.txt`. Explain the result if any difference.

```
/*
 * race.c
 */

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#ifdef USE_SEM
#define SEM_MODE 0666 /* rw(owner)-rw(group)-rw(other) permission */
#define SEM_KEY  1122334455

int sem;

/* P () - returns 0 if OK; -1 if there was a problem */
int P (int s)
{
	struct sembuf sop;  /* the operation parameters */
	sop.sem_num =  0;   /* access the 1st (and only) sem in the array */
	sop.sem_op  = -1;          /* wait..*/
	sop.sem_flg =  0;          /* no special options needed */

	if (semop (s, &sop, 1) < 0) {
		fprintf(stderr,"P(): semop failed: %s\n",strerror(errno));
		return -1;
	} else {
		return 0;
	}
}

/* V() - returns 0 if OK; -1 if there was a problem */
int V(int s)
{
	struct sembuf sop;  /* the operation parameters */
	sop.sem_num =  0;   /* the 1st (and only) sem in the array */
	sop.sem_op  =  1;   /* signal */
	sop.sem_flg =  0;   /* no special options needed */

	if (semop(s, &sop, 1) < 0) {
		fprintf(stderr,"V(): semop failed: %s\n",strerror(errno));
		return -1;
	} else {
		return 0;
	}
}
```

```
#endif

/* increment value saved in file */
void Increment()
{
	int ret;
	int fd;		/* file descriptor */
	int counter;
	char buffer[100];
	int i = 10000;

	while(i)
	{
		/* open file */
		fd = open("./counter.txt", O_RDWR );
		if (fd < 0)
		{
			printf("Open counter.txt error.\n");
			exit(-1);
		}

#ifdef USE_SEM
		/* acquire semaphore */
		P(sem);
#endif

		/**************** Critical Section *****************/
		/* clear */
		memset(buffer, 0, 100);

		/* read raw data from file */
		ret = read(fd, buffer, 100);
		if (ret < 0)
		{
			perror("read counter.txt");
			exit(-1);
		}

		/* transfer string to integer & increment counter */
		counter = atoi(buffer);
		counter++;

		/* write back to counter.txt */
		lseek(fd, 0, SEEK_SET);	/* reposition to the head of file */

		/* clear */
		memset(buffer, 0, 100);
		sprintf(buffer, "%d", counter);
		ret = write(fd, buffer, strlen(buffer));
		if (ret < 0)
		{
			perror("write counter.txt");
			exit(-1);
		}
		/**************** Critical Section ****************/

#ifdef USE_SEM
		/* release semaphore */
		V(sem);
#endif

		/* close file */
		close(fd);
```

```
		i--;
	}
}

int main(int argc, char **argv)
{
	int childpid;
	int status;

#ifdef USE_SEM
	/* create semaphore */
	sem = semget(SEM_KEY, 1, IPC_CREAT | IPC_EXCL | SEM_MODE);
	if (sem < 0)
	{
		fprintf(stderr, "Sem %ld creation failed: %s\n", SEM_KEY,
			strerror(errno));
		exit(-1);
	}

	/* initial semaphore value to 1 (binary semaphore) */
	if ( semctl(sem, 0, SETVAL, 1) < 0 )
	{
		fprintf(stderr, "Unable to initialize Sem: %s\n", strerror(errno));
		exit(0);
	}

	printf("Semaphore %ld has been created & initialized to 1\n", SEM_KEY);
#endif

	/* fork process */
	if ((childpid = fork()) > 0)		/* parent */
	{
		Increment();
		waitpid(childpid, &status, 0);
	}
	else if (childpid == 0)			/* child */
	{
		Increment();
		exit(0);
	}
	else					/* error */
	{
		perror("fork");
		exit(-1);
	}

#ifdef USE_SEM
	/* remove semaphore */
	if (semctl (sem, 0, IPC_RMID, 0) < 0)
	{
		fprintf (stderr, "%s: unable to remove sem %ld\n", argv[0],
SEM_KEY);
		exit(1);
	}
	printf("Semaphore %ld has been remove\n", SEM_KEY);
#endif
	return 0;
}
```

6 Mutex

A mutex is abbreviated from "MUTual EXclusion", and is useful for protecting shared data structures from concurrent modifications, and implementing critical sections and monitors. A mutex has two possible states: unlocked (not owned by any thread), and locked (owned by one thread). A mutex can never be owned by two different threads simultaneously. A thread attempting to lock a mutex that is already locked by another thread is suspended until the owning thread unlocks the mutex first. Some POSIX library APIs for mutex are:

- `pthread_mutex_init()` initializes the mutex referenced by mutex with attributes specified by `attr`. Upon successful initialization, the state of the mutex becomes initialized and unlocked. For more information, please see its man page.
- `Pthread_mutex_lock()` locks the mutex object referenced by the mutex. If the mutex is already locked, the calling thread blocks until the mutex becomes available. This operation returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.
- `Pthread_mutex_unlock()` unlocks the mutex object referenced by the mutex.
- `Pthread_mutex_destroy()` destroys the mutex object referenced by mutex; the mutex object becomes, ineffective, uninitialized.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define checkResults(string, val) {                              \
       if (val) {                                                \
              printf("Failed with %d at %s", val, string);   \
              exit(1);                                           \
       }                                                         \
}

#define                  NUMTHREADS   3

pthread_mutex_t          mutex = PTHREAD_MUTEX_INITIALIZER;
int                      sharedData  = 0;
int                      sharedData2 = 0;

void *theThread(void *parm)
{
       int   rc;

       printf("\tThread %lu: Entered\n", (unsigned long) pthread_self());

       /* lock mutex */
       rc = pthread_mutex_lock(&mutex);
       checkResults("pthread_mutex_lock()\n", rc);

       /**************** Critical Section *****************/
       printf("\tThread %lu: Start critical section, holding lock\n",
                     (unsigned long) pthread_self());

       /* Access to shared data goes here */
       ++sharedData;
       --sharedData2;

       printf("\tsharedData = %d, sharedData2 = %d\n",
```

```
               sharedData, sharedData2);

        printf("\tThread %lu: End critical section, release lock\n",
               (unsigned long) pthread_self());
        /**************** Critical Section *****************/

        /* unlock mutex */
        rc = pthread_mutex_unlock(&mutex);
        checkResults("pthread_mutex_unlock()\n", rc);

        return NULL;
}

int main(int argc, char **argv)
{
        pthread_t              thread[NUMTHREADS];
        int                    rc = 0;
        int                    i;

        /* lock mutex */
        printf("Main thread hold mutex to prevent access to shared data\n");
        rc = pthread_mutex_lock(&mutex);
        checkResults("pthread_mutex_lock()\n", rc);

        /* create thread */
        printf("Main thread create/start threads\n");
        for (i = 0; i < NUMTHREADS; ++i) {
               rc = pthread_create(&thread[i], NULL, theThread, NULL);
               checkResults("pthread_create()\n", rc);
        }

        /* wait for thread creation complete */
        printf("Main thread wait a bit until 'done' with the shared data\n");
        sleep(3);

        /* unlock mutex */
        printf("Main thread unlock shared data\n");
        rc = pthread_mutex_unlock(&mutex);
        checkResults("pthread_mutex_lock()\n",rc);

        /* wait thread complete */
        printf("Main thread Wait for threads to complete, "
                      "and release their resources\n");
        for (i=0; i <NUMTHREADS; ++i) {
               rc = pthread_join(thread[i], NULL);
               checkResults("pthread_join()\n", rc);
        }

        /* destroy mutex */
        printf("Main thread clean up mutex\n");
        rc = pthread_mutex_destroy(&mutex);

        printf("Main thread completed\n");

        return 0;
}
```

7 Pipe

A pipe creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. `filedes[0]` is for reading, `filedes[1]` is for writing. An unnamed pipe can be viewed and accessed by processes with parent-child relationship. To share a pipe among all processes, you need to create a named pipe.

The following program creates a pipe, and then `fork(2)` to create a child process. After the `fork(2)`, each process closes the descriptors that it doesn't need for the pipe (see `pipe(7)`). In the following sample code, the child process reads the file specified in `argv[1]`, and writes the file content to the parent process through the pipe. The parent process reads the data from the pipe and echoes the data on the screen.

```
/* pipe.c
 *
 * child process read the content of file
 * and write the content to parent process through pipe
 */

#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int pfd[2];   /* pfd[0] is read end, pfd[1] is write end */

void ChildProcess(char *path)
{
	int fd;
	int ret;
	char buffer[100];

	/* close unused read end */
	close(pfd[0]);

	/* open file */
	fd = open(path, O_RDONLY);
	if (fd < 0) {
		printf("Open %s failed.\n", path);
		exit(EXIT_FAILURE);
	}

	/* read file and write content to pipe */
	while (1) {
		/* read raw data from file */
		ret = read(fd, buffer, 100);

		if (ret < 0) {		/* error */
			perror("read()");
			exit(EXIT_FAILURE);
		}
		else if (ret == 0) {	/* reach EOF */
			close(fd);		/* close file */
			close(pfd[1]);	/* close write end, reader see EOF */
			exit(EXIT_SUCCESS);
		}
		else {			/* write content to pipe */
			write(pfd[1], buffer, ret);
		}
	}
}
```

```
void ParentProcess()
{
	int ret;
	char buffer[100];

	/* close unused write end */
	close(pfd[1]);

	/* read data from pipe until reach EOF */
	while(1) {
		ret = read(pfd[0], buffer, 100);

		if (ret > 0) {			/* print data to screen */
			printf("%.*s", ret, buffer);
		}
		else if (ret == 0) {		/* reach EOF */
			close(pfd[0]);		/* close read end */
			wait(NULL);
			exit(EXIT_SUCCESS);
		}
		else {
			perror("pipe read()");
			exit(EXIT_FAILURE);
		}
	}
}

int main(int argc, char *argv[])
{
	pid_t cpid;

	if  (argc != 2) {
		fprintf(stderr, "%s: specify a file\n", argv[0]);
		exit(1);
	}

	/* create pipe */
	if (pipe(pfd) == -1) {
		perror("pipe");
		exit(EXIT_FAILURE);
	}

	/* fork child process */
	cpid = fork();
	if (cpid == -1) {	/* error */
		perror("fork");
		exit(EXIT_FAILURE);
	}

	if (cpid == 0)
		ChildProcess(argv[1]);
	else
		ParentProcess();

	return 0;
}
```

Monitoring multiple file descriptors with polling strategy may cause busy waiting, which lowers down the system performance. To effectively monitor multiple descriptors, you can use the system call `select()` to perform block waiting, rather than busy waiting. The calling process specifies the interesting file descriptors to `select()` and performs blocking waiting. When one or more descriptors become "ready" (ready for read or ready for write), `select()` informs the calling process to awake from blocking state. Then, a user can check each file descriptor to perform read/write operation. In following program, two child processes sleep a random time, then send a message to the parent process. The parent process uses `select()` to perform blocking waiting, until one of the pipes is ready to read.

```
/*
 * select.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

#define max(a, b)   ((a > b) ? a : b)

void ChildProcess(int *pfd, int sec)
{
	char buffer[100];

	/* close unused read end */
	close(pfd[0]);

	/* sleep a random time to wait parent process enter select() */
	printf("Child process (%d) wait %d secs\n", getpid(), sec);
	sleep(sec);

	/* write message to parent process */
	memset(buffer, 0, 100);
	sprintf(buffer, "Child process (%d) sent message to parent process\n",
		getpid());
	write(pfd[1], buffer, strlen(buffer));

	/* close write end */
	close(pfd[1]);

	exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[])
{
	int pfd1[2], pfd2[2];      /* pipe's fd */
	int cpid1, cpid2;   /* child process id */
	fd_set rfds, arfds;
	int max_fd;
	struct timeval tv;
	int retval;
	int fd_index;
	char buffer[100];

	/* random seed */
	srand(time(NULL));

	/* create pipe */
	pipe(pfd1);
	pipe(pfd2);
```

```
	/* create 2 child processes and set corresponding pipe & sleep time */
	cpid1 = fork();
	if (cpid1 == 0)
		ChildProcess(pfd1, random() % 5);

	cpid2 = fork();
	if (cpid2 == 0)
		ChildProcess(pfd2, random() % 4);

	/* close unused write end */
	close(pfd1[1]);
	close(pfd2[1]);

	/* set pfd1[0] & pfd2[0] to watch list */
	FD_ZERO(&rfds);
	FD_ZERO(&arfds);
	FD_SET(pfd1[0], &arfds);
	FD_SET(pfd2[0], &arfds);
	max_fd = max(pfd1[0], pfd2[0]) + 1;

	/* Wait up to five seconds. */
	tv.tv_sec = 5;
	tv.tv_usec = 0;

	while(1)
	{
		/* config fd_set for select */
		memcpy(&rfds, &arfds, sizeof(rfds));

		/* wait until any fd response */
		retval = select(max_fd, &rfds, NULL, NULL, &tv);

		if (retval == -1) { /* error */
			perror("select()");
			exit(EXIT_FAILURE);
		}
		else if (retval) {  /* # of fd got respone */
			printf("Data is available now.\n");
		}
		else { /* no fd response before timer expired */
			printf("No data within five seconds.\n");
			break;
		}

		/* check if any response */
		for (fd_index = 0; fd_index < max_fd; fd_index++)
		{
			if (!FD_ISSET(fd_index, &rfds))
				continue;    /* no response */

			retval = read(fd_index, buffer, 100);

			if (retval > 0)             /* read data from pipe */
				printf("%.*s", retval, buffer);
			else if (retval < 0)        /* error */
				perror("pipe read()");
			else {              /* write fd closed */
				/* close read fd */
				close(fd_index);
				/* remove fd from watch list */
				FD_CLR(fd_index, &arfds);
			}
		}
```

```
        }
        return 0;
}
```

4 Shared Memory

Shared memory is a memory space that may be simultaneously accessed by multiple processes with an intent to provide inter-process communication among them or avoid redundant copies. Unlike unnamed pipes, only exist among processes with parent-child relationship, every process can access the shared memory space with the share memory key specified.

The followings are two processes communicating via shared memory: `shm_server.c` and `shm_client.c`. The two programs here illustrate the passing of a simple piece of memory (a string) between the processes if running simultaneously:

```
/*
 * shm_server.c -- creates the string and shared memory.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>

#define SHMSZ     27

int main(int argc, char *argv[])
{
        char c;
        int shmid;
        key_t key;
        char *shm, *s;
        int retval;

        /* We'll name our shared memory segment "5678" */
        key = 5678;

        /* Create the segment */
        if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
                perror("shmget");
                exit(1);
        }

        /* Now we attach the segment to our data space */
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                perror("shmat");
                exit(1);
        }
        printf("Server create and attach the share memory.\n");

        /* Now put some things into the memory for the other process to read */
        s = shm;

        printf("Server write a ~ z to share memory.\n");
        for (c = 'a'; c <= 'z'; c++)
                *s++ = c;
        *s = '\0';

        /*
```

```
         * Finally, we wait until the other process changes the first
         * character of our memory to '*', indicating that it has read
         * what we put there.
         */
        printf("Waiting other process read the share memory ...\n");
        while (*shm != '*')
                sleep(1);
        printf("Server read * from the share memory.\n");

        /* Detach the share memory segment */
        shmdt(shm);

        /* Destroy the share memory segment */
        printf("Server destroy the share memory.\n");
        retval = shmctl(shmid, IPC_RMID, NULL);
        if (retval < 0)
        {
                fprintf(stderr, "Server remove share memory failed\n");
                exit(1);
        }

        return 0;
}
```

```
/*
 * shm_client.c -- attaches itself to the created shared memory
 *                 and uses the string (printf).
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>

#define SHMSZ     27

int main(int argc, char *argv[])
{
        int shmid;
        key_t key;
        char *shm, *s;

        /* We need to get the segment named "5678", created by the server */
        key = 5678;

        /* Locate the segment */
        if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
                perror("shmget");
                exit(1);
        }

        /* Now we attach the segment to our data space */
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                perror("shmat");
                exit(1);
        }
        printf("Client attach the share memory created by server.\n");

        /* Now read what the server put in the memory */
        printf("Client read characters from share memory ...\n");
        for (s = shm; *s != '\0'; s++)
                putchar(*s);
```

```
        putchar('\n');

        /*
         * Finally, change the first character of the segment to '*',
         * indicating we have read the segment.
         */
        printf("Client write * to the share memory.\n");
        *shm = '*';

        /* Detach the share memory segment */
        printf("Client detach the share memory.\n");
        shmdt(shm);

        return 0;
}
```

Lab 7: Signal & Timer

1 Objective

- Be familiar with signal, timer and process reaper.

2 Prerequisite

- Read man pages of Read man pages of `kill()`, `sigaction()`, `setitimer()`, etc.

3 Signal

Signals notify tasks of events that occurred during the execution of other tasks or Interrupt Service Routines (ISRs) . It diverts the notified task from its normal execution path and triggers the associated signal handler. Signal handler is a function run in "asynchronous mode", which gets called when the task receives that signal, no matter which code the task is executed on. This just like the relationship between hardware interrupts and ISRs, which are also asynchronous to the execution of OS and do not occur at any predetermined point of time.

The difference between a signal and a normal interrupt is that signals are software interrupts, which are generated by other task or OS to trap normal execution of task. Another difference is that task can only receive a signal when it is running in user mode. If the receiving process is running in kernel mode, the execution of the signal will start only after the process returns to user mode. When the signal handler completes, the normal execution resumes. The task continues execution from wherever it happened to be before the signal was received.

Signals have been used for approximately 30 years without any major modifications. Although we now have advanced synchronization tools and many IPC mechanisms, signals play a vital role in Linux for handling exceptions and interrupts. For example, when child process exits its execution, it sends a `SIGCHLD` signal to parent process and becomes a zombie process. When the parent process catches this signal, it performs `wait()` or `waitpid()` to reclaim the resources used by child process in the signal handler. This design prevents the parent process from blocked in `wait()` or `waitpid()` function calls.

3.1 Sending Signals

Linux supports various types of signal, you should specify the desired type when sending signal to other tasks. Each signal in Linux is identified by an integer value, which is called signal number or vector number. The first 31 signals are standard signals, some of which date back to 1970s UNIX from Bell Labs. The POSIX (Portable Operating Systems and Interface for UNIX) standard introduced a new class of signals designated as real-time signals, with numbers ranging from 32 to 63. Usually, all signal numbers as well as the associated symbolic name are de ned in `/usr/include/signal.h`, or you can use the command '`kill -l`' to see a list of signals supported by your Linux system.

Signals can be generated from within the shell using the kill command. (Man the command "kill" to obtain detailed information.) The name of kill may seem strange, but actually most signals serve the purpose of terminating processes, so this is not really that unusual.

For example, the following command sends the SIGUSR1 signal to process 3423:

```
$ kill -USR1 3423
```

Another method is to use the kill system call within a C program to send a signal to a process. This call takes a process ID and a signal number as parameters.

```
int kill(pid t pid, int sig no);
```

One common use of this mechanism is to end another process by sending it a SIGTERM or SIGKILL signal. Another common use is to send a command to a running program. Two "userdefined" signals are reserved for this purpose: SIGUSR1 and SIGUSR2. The SIGHUP signal is sometimes used for this purpose as well, commonly to wake up an idling program or cause a program to reread its configuration files.

3.2 Catching Signals

When a process receives a signal, it may do one of several things, depending on the signal's disposition. For each signal, there is a default disposition, which determines what happens to the task if the program does not specify any signal handler. If a signal handler is used, the currently executing task is paused, the signal handler is executed; and when the signal handler returns, the task resumes.

For most signal types, a program can specify some other behavior - either to ignore the signal or to call a special signal handler function to respond to the signal. We can use sigaction system call to register the signal handler or set a signal disposition. The syntax of sigaction is:

```
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact)
```

The first argument, signum, is a specified signal. The next two parameters are pointers to sigaction structures; the second argument, is used to set the new disposition of the signal signum; and the third argument is used to store the previous disposition, usually NULL. The sigaction structure is defined as:

```
struct sigaction {
   void (*sa_handler)(int);
   void (*sa_sigaction)(int , siginfo_t , void *);
   sigset_t sa_mask;
   int sa_flags;
}
```

The members of the sigaction structure are described as follows.

- sa_hander
  sa_hander is a pointer pointed to a user-defined signal handler or default signal handler:
  - SIG_DFL, which specifies the default disposition for the signal.
  - SIG_IGN, which specifies that the signal should be ignored.
  - A pointer to a signal-handler function. This function receives the signal number as its only argument.

- sa_mask
  sa_mask gives a mask of signals which should be blocked during execution of the signal handler. The sigset_t structure consists an array of unsigned long, each bit inside controls the block state of a particular signal type. In the first unsigned long in sigset_t, the least significant bit (0) is unused since no signal has the number 0, the other 31 bits represents the 31 standard UNIX signals The bits in the second unsigned long are the real-time signal numbers from 32 to 64.
- sa_flags
  sa_flags specifies the action of signal. Sets of flags are available for controlling the signal in a different manner. More than one flag can be used by OR operation:
  - SA_NOCLDWAIT:
    If signum is SIGCHLD, do not transform child processes into zombies when they terminate.
  - SA_RESETHAND:
    SA_RESETHAND restores the default action of the signal after the user-defined signal handler has been executed.
  - SA_NODEFER:
    The signal which triggered the handler will be blocked. Set the SA_NODEFER allows signal can be received from within its own signal handler.
  - SA_SIGINFO:
    When SA_SIGINFO is set, the sa_sigaction should be used, instead of specifying the signal handler in sa_handler.

For more flags information, please refer the sigaction manpages.

- sa_sigaction:
  If the SA_SIGINFO flag is set in sa_flags, sa_sigaction should be used. sa_sigaction is a pointer to a function that takes three arguments, for example:
  ```
  void my handler (int signo, siginfo_t *info, void *context);
  ```
  Here, signo is the signal number, and info is a pointer to the structure of type siginfo_t, which specifies the signal-related information; and context is a pointer to an object of type ucontext_t, which refers to the receiving process context that was interrupted with the delivered signal. For the detail structure of siginfo_t and ucontext_t, please refer the manpage of sigaction and getcontext.

The following example uses SA_SIGINFO and sa_sigaction to extract information from a signal. Use the kill command to send a SIGUSR1 signal to a program.

```
/*
 * sig_catch.c
 */

#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

void handler (int signo, siginfo_t *info, void *context)
{
       /* show the process ID sent signal */
```

```
	printf ("Process (%d) sent SIGUSR1.n", info->si_pid);
}

int main (int argc, char *argv[])
{
	struct sigaction my_action;

	/* register handler to SIGUSR1 */
	memset(&my_action, 0, sizeof (struct sigaction));
	my_action.sa_flags = SA_SIGINFO;
	my_action.sa_sigaction = handler;

	sigaction(SIGUSR1, &my_action, NULL);

	printf("Process (%d) is catching SIGUSR1 ...\n", getpid());
	sleep(10);
	printf("Done.\n");

	return 0;
}
```

Remember that some OSs implement `sa_handler` and `sa_sigaction` as a union, do not assign values to these two arguments at same time.

3.3 Atomic Access in Signal Handler

Assigning a value to a global variable can be dangerous because the assignment may actually be carried out in two or more machine instructions, and a second signal may occur between them, leaving the variable in a corrupted state. If you use a global variable to flag a signal from a signal handler function, it should be of the special type `sig_atomic_t`. In Linux, `sig_atomic_t` is an ordinary integer data type, but which one it is, and how many bits it contains, may vary from machine to machine. In practice, you can also use `sig_atomic_t` as a pointer. If you want to write a program which is portable to any standard UNIX system, though, use `sig_atomic_t` for these global variables. Reading and writing `sig_atomic_t` is guaranteed to happen in a single instruction, so there's no way for a handler to run "in the middle" of an access.

The following program listing uses a signal-handler function to count the number of times that the program receives `SIGUSR1`, one of the signals reserved for application use.

```
/*
 * sig_count.c
 */

#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
{
	++sigusr1_count;	/* add one, protected atomic action */
}
```

```
int main ()
{
	struct sigaction sa;
	struct timespec req;
	int retval;

	/* set the sleep time to 10 sec */
	memset(&req, 0, sizeof(struct timespec));
	req.tv_sec = 10;
	req.tv_nsec = 0;

	/* register handler to SIGUSR1 */
	memset(&sa, 0, sizeof (sa));
	sa.sa_handler = handler;

	sigaction (SIGUSR1, &sa, NULL);

	printf("Process (%d) is catching SIGUSR1 ...\n", getpid());

	/* sleep 10 sec */
	do{
		retval = nanosleep(&req, &req);
	} while(retval);

	printf ("SIGUSR1 was raised %d times\n", sigusr1_count);

	return 0;
}
```

4 Timer

The timer schedules an event according to a predefined time value in the future. When the timer expired, it deliveries a signal to the calling process. Timers are used everywhere in Unix-like systems, from basic delay function implementation to network transmission and performance monitoring. Linux provides two basic POSIX standard timer functions, `alarm` and `setitimer`. The `alarm` system call arranges an alarm clock for delivering a `SIGALRM` signal after the specified time elapsed. The `setitimer` system call is a generalization of the alarm call, it provides three different types of timer for counting the elapsed time. The syntax of `setitimer` is shown as follows.

```
int setitimer(int which, const struct itimerval *new, struct itimerval *old);
```

The first argument which is the timer code, specifying which timer to set.

- If the timer code is `ITIMER_REAL`, the process is sent a `SIGALRM` signal after the specified wall-clock time has elapsed.
- If the timer code is `ITIMER_VIRTUAL`, the process is sent a `SIGVTALRM` signal after the process has executed for the specified time. Time in which the process is not executing (that is, when the kernel or another process is running) is not counted.
- If the timer code is `ITIMER_PROF`, the process is sent a `SIGPROF` signal when the specified time has elapsed either during the process's own execution or the execution of a system call on behalf of the process.

The second argument is a pointer to a `itimerval` structure specifying the new settings for that timer. The third argument, if not null, is a pointer to another `itimerval` structure which receives the old timer settings. The struct `itimerval` variable has two fields:

- `it_value` is a struct `timeval` field that contains the time until the timer next expires and a signal is sent. If this is 0, the timer is disabled.
- `it_interval` is another struct `timeval` field containing the value to which the timer will be reset after it expires. If this is 0, the timer will be disabled after it expires. If this is nonzero, the timer is set to expire repeatedly after this interval.

And the struct `timeval` has the following two members:

- `tv_sec` represents the number of whole seconds of elapsed time.
- `tv_usec` is the rest of the elapsed time (a fraction of a second), represented as the number of microseconds. It is always less than one million.

The following program illustrates the use of `setitimer` to track the execution time of a program. A timer is configured to be expired every 250 ms and sends a `SIGVTALRM` signal.

```
/*
 * timer.c
 */

#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

void timer_handler (int signum)
{
	static int count = 0;

	printf ("timer expired %d times\n", ++count);
}

int main (int argc, char **argv)
{
	struct sigaction sa;
	struct itimerval timer;

	/* Install timer_handler as the signal handler for SIGVTALRM */
	memset (&sa, 0, sizeof (sa));
	sa.sa_handler = &timer_handler;
	sigaction (SIGVTALRM, &sa, NULL);

	/* Configure the timer to expire after 250 msec */
	timer.it_value.tv_sec = 0;
	timer.it_value.tv_usec = 250000;

	/* Reset the timer back to 250 msec after expired */
	timer.it_interval.tv_sec = 0;
	timer.it_interval.tv_usec = 250000;

	/* Start a virtual timer */
	setitimer (ITIMER_VIRTUAL, &timer, NULL);

	/* Do busy work */
	while (1);
```

```
	return 0;
}
```

The following program demonstrates the di erence among three di erent timers. Write a simple “`while(1)` loop” program first and execute several instances of this program as the number of cores in your machine. For example, if you have a dual core machine, then run two “`while(1)` loop” programs simultaneously. After occupy all computing resources, execute the `timer_diff.c` program and observe the results of three counters.

```
/*
 * timer_diff.c
 */

#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

/* counter */
int SIGALRM_count = 0;
int SIGVTALRM_count = 0;
int SIGPROF_count = 0;

/* handler of SIGALRM */
void SIGALRM_handler (int signum)
{
	SIGALRM_count++;
}

/* handler of SIGVTALRM */
void SIGVTALRM_handler (int signum)
{
	SIGVTALRM_count++;
}

/* handler of SIGPROF */
void SIGPROF_handler (int signum)
{
	SIGPROF_count++;
}

void IO_WORKS();

int main (int argc, char **argv)
{
	struct sigaction SA_SIGALRM, SA_SIGVTALRM, SA_SIGPROF;
	struct itimerval timer;

	/* Install SIGALRM_handler as the signal handler for SIGALRM */
	memset (&SA_SIGALRM, 0, sizeof (SA_SIGALRM));
	SA_SIGALRM.sa_handler = &SIGALRM_handler;
	sigaction (SIGALRM, &SA_SIGALRM, NULL);

	/* Install SIGVTALRM_handler as the signal handler for SIGVTALRM */
	memset (&SA_SIGVTALRM, 0, sizeof (SA_SIGVTALRM));
	SA_SIGVTALRM.sa_handler = &SIGVTALRM_handler;
```

```
        sigaction (SIGVTALRM, &SA_SIGVTALRM, NULL);

        /* Install SIGPROF_handler as the signal handler for SIGPROF */
        memset (&SA_SIGPROF, 0, sizeof (SA_SIGPROF));
        SA_SIGPROF.sa_handler = &SIGPROF_handler;
        sigaction (SIGPROF, &SA_SIGPROF, NULL);

        /* Configure the timer to expire after 100 msec */
        timer.it_value.tv_sec = 0;
        timer.it_value.tv_usec = 100000;

        /* Reset the timer back to 100 msec after expired */
        timer.it_interval.tv_sec = 0;
        timer.it_interval.tv_usec = 100000;

        /* Start timer */
        setitimer(ITIMER_REAL, &timer, NULL);
        setitimer(ITIMER_VIRTUAL, &timer, NULL);
        setitimer(ITIMER_PROF, &timer, NULL);

        /* Do some I/O operations */
        IO_WORKS();

        printf("SIGALRM_count   = %d\n", SIGALRM_count);
        printf("SIGVTALRM_count = %d\n", SIGVTALRM_count);
        printf("SIGPROF_count   = %d\n", SIGPROF_count);

        return 0;
}

void IO_WORKS()
{
        int fd, ret;
        char buffer[100];
        int i;

        /* Open/Read/Close file 300000 times */
        for (i = 0; i < 300000; i++) {
                if ((fd = open("/etc/init.d/networking", O_RDONLY)) < 0) {
                        perror("Open /etc/init.d/networking");
                        exit(EXIT_FAILURE);
                }

                do {
                        ret = read(fd, buffer, 100);
                }while(ret);

                close(fd);
        }
}
```

5 Process Reaper

When a child process completes its execution, rather than reclaims all memory resources associated with it, the OS puts this process into a ***zombie*** state and allows the parent process read the exit status of child process. In the previous, we let the parent process executes the `wait()` and `waitpid()` system call to "reap" (remove) the zombie process. But executing `wait()` and `waitpid()` blocks the normal execution of parent process. Even configures the waitpid() with the WNOHANG option, the parent should perform periodic checking on status of child processes.

One solution is that we can allow parent process explicitly ignore SIGCHLD by setting its handler to SIG_IGN (rather than simply ignoring the signal by default) or has the SA_NOCLDWAIT flag set, all child exit status information will be discarded and no zombie processes will be left. But sometimes, we need to perform specific operation when child process end its execution, e.g. check the exit status of child processes. In such case, we register the signal handler with SIGCHLD to reap the child process manually. This signal handler is also known as the process reaper.

The following program illustrates the basic operation of process reaper. This program uses another system call signal() to register the reaper function for SIGCHLD. (Please refer the manpage to see the difference between signal() and sigaction().) When the parent process traps into the reaper function, it performs a nonblocking waitpid() call to reclaim zombie processes. The reaper function checks the return value of waitpid() to determine any zombie process to reap. If the return value is equal to zero, which indicates no zombie exists, we exit the handler to resume normal execution.

```
/*
 * reaper.c
 *     Demonstrate the work of process reaper
 */

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>

#define FORKCHILD 5

volatile sig_atomic_t reaper_count = 0;

/* signal handler for SIGCHLD */
void Reaper(int sig)
{
	pid_t pid;
	int status;

	while((pid = waitpid(-1, &status, WNOHANG)) > 0)
	{
		printf("Child %d is terminated.\n", pid);
		reaper_count++;
	}
}

void ChildProcess()
{
	int rand;

	/* rand a sleep time */
	srand(time(NULL));
	rand = random() % 10 + 2;

	printf("Child %d sleep %d sec.\n", getpid(), rand);
	sleep(rand);
```

```
        printf("Child %d exit.\n", getpid());

        exit(0);
}

int main(int argc, char *argv[])
{
        int cpid;
        int i;

        /* regist signal handler */
        signal(SIGCHLD, Reaper);


        /* fork child processes */
        for (i = 0; i < FORKCHILD; i++)
        {
                if ( (cpid = fork()) > 0)  /* parent */
                        printf("Parent fork child process %d.\n", cpid);
                else   /* child */
                        ChildProcess();

                sleep(1);
        }

        /* wait all child exit */
        while(reaper_count != FORKCHILD)
                sleep(1);

        return 0;
}
```