Lab 1: Getting Started

Yu-Lun Huang

2018-09-14

1 Objective

- Prepare cross development environment for PXA270.
- Be familiar with your PXA270.

2 Prerequisite

- Finish pre-Lab and get familiar with basic Linux and gcc compiler commands.

3 Setting Up

Before we go further, please make sure that you have the Internet connected and connect your PC with development board (PXA270) correctly. Each site has four IPs. The first three IPs are assigned to your Windows (Host OS), Linux (Guest OS on Virtual Machine), and Target (PXA270), accordingly. The fourth IP is reserved. Please configure your systems as following:

3.1 Windows Configuration ($< ip1 >$)

1. Configure IP as mentioned.
2. Configure serial interface:
   - Connect your Windows and PXA270 using RS232.
     (COM/PC $\longleftrightarrow$ RS232 $\longleftrightarrow$ UART/PXA270)
   - Download "Putty" from Internet and launch "Putty".
   - Choose "Connection type" as "Serial".
   - Select the "Connection→Serial" in "Category" and modify the serial control settings as following:
     - Speed (baud rate) = 9600
     - Data bits = 8

- Stop bit = 1
- Parity = None
- Flow Control = None

- Open the connection.

3.2 Linux Configuration for VM and Target ($< ip2 >$ for VM and $< ip3 >$ for PXA)

1. Configure Network:

   - Configure network temporally
     - ifconfig: show/configure a network interface.

       ```
       SHELL> sudo ifconfig eth0 <ip2> broadcast
       192.168.0.255 netmask 255.255.255.0
       ```

     - route: show/manipulate the IP routing table.

       ```
       SHELL> sudo route add default gw 192.168.0.1
       ```

   - Configure network permanently
     - Edit `/etc/network/interfaces`

       ```
       auto eth0
       iface eth0 inet static
            address <ip2>
            netmask 255.255.255.0
            gateway 192.168.0.1
       ```

     - Edit `/etc/resolvconf/resolv.conf.d/base`, and add the following DNS configurations to the file.

       ```
       nameserver 140.113.6.2
       nameserver 140.113.1.1
       ```

     - Use `resolvconf` to generate `/etc/resolv.conf`

       ```
       SHELL> sudo resolvconf -u
       ```

     - Restart network

       ```
       SHELL> sudo /etc/init.d/networking restart
       ```

4 Development Environment

4.1 Setting Up

1. Install package for building kernel and root filesystem:

```
SHELL> sudo apt-get install make patch libncurses32-dev
```

2. Download the following packages to your Guest OS (Linux Host), and put them under your home directory:

   - ToolChain for PXA270: `arm-linux-toolchain-bin-4.0.2.tar.gz` (for building everything).
   - Linux Kernel Source: `mt-linux-2.6.15.3.tar.gz` (for building kernel);
   - Linux Kernel Patch for PXA270: `linux-2.6.15.3-creator-pxa270.patch` (for building kernel);
   - Root Filesystem for PXA270: `rootfs.tar.gz` (for building rootfs)

4.2 Install Toolchain

A toolchain is the set of programming tools that are used to create a product. In the toolchain for PXA270 (`arm-linux-toolchain-bin-4.0.2.tar.gz`, a cross compiler, capable of creating executable code for a platform other than the one on which the compiler is run, is included. To compile a program running on your PXA270, you need to install the toolchain for compilation:

1. Unzip the tarball:

```
SHELL> sudo tar zxvf arm-linux-toolchain-bin-4.0.2.tar.gz
-C /
```

2. Add the `PATH` to `ToolChain` at the end of ∼`/.bashrc`:

```
export PATH=$PATH:/opt/arm-unknown-linux-gnu/bin
```

3. Reload ∼/.bashrc to make the above changes effective.

```
SHELL> source ∼/.bashrc
```

4. Execute Program on PXA270

- Move or copy the executable file to `<export point>`:

```
SHELL> mv hello <absolute path to export point>
```

Or,
```
SHELL> cp hello <absolute path to export point>
```

- Execute program under NFS mount point on PXA270:

```
> cd /mnt
> ./hello
```

5 Configure nfsd (Network Filesystem Daemon) on VM

- Install `nfs` server:

```
SHELL> sudo apt-get install nfs-kernel-server
```

- Create the export point in your home directory for `nfs` server:

```
SHELL> cd ~
SHELL> mkdir <export point>
```

- Configuring the `nfs` server on Linux:
Get the absolute path to `<export point>` by using 'pwd' command. Edit the `/etc/exports` file, it should contain the following item:

```
<absolute path to export point> <
ip3 >(rw,no_root_squash,no_subtree_check)
```

- Starting the server on Linux:

```
SHELL> sudo /etc/init.d/nfs-kernel-server restart
```

6 Configure nfs on PXA

- Mount to the `nfs` server (`nfsd`) on Linux:

```
> portmap&
> mount < ip2 >:<absolute path to export point> /mnt
```

7 Configure tftpd (Trivial File Transfer Protocol server) on VM

The `tftpd` is normally started by the `xinetd` daemon. Before that, please install necessary packages and create a root directory <tftproot> for `tftpd`:

```
SHELL> sudo apt-get install xinetd tftp tftpd
SHELL> cd ~
SHELL> mkdir <tftproot>
SHELL> cd <tftproot>
SHELL> pwd
```

In the above commands, `apt-get` is a command of APT (Advanced Packaging Tool). It helps automate the retrieval, configuration and installation of software packages. Get the absolute path, `<tftproot>`, of the newly created directory by using 'pwd' command. Then, edit `/etc/xinetd.d/tftp` to config the `tftpd` settings:

```
service tftp
{
socket_type = dgram
protocol = udp
wait = yes
user = lab616
server = /usr/sbin/in.tftpd
server_args = -s <tftprootpath>
disable = no
}
```

Then, reload `xinetd`:

```
SHELL> sudo /etc/init.d/xinetd reload
```

8 Build uImage

8.1 Build Kernel

1. Unzip Kernel Source:

   ```
   SHELL> cd ~
   SHELL> tar zxvf mt-linux-2.6.15.3.tar.gz
   ```

   The extraction will create a directory `microtime/` in your home directory, and the following subdirectories:

   - build-linux/ - tools for building kernel image and root filesystem.

- create-pxa270/ - the link to `pro/devkit/lsp/create-pxa270`.
- linux/ - the link to `pro/devkit/lsp/create-pxa270/linux-2.6.15.3`.
- package/ - the link to `pro/host/src/BUILD`.
- pro/ - it contains the linux kernel source and some demo codes.

Then, we need to patch the kernel for PXA270.

```
SHELL> cp linux-2.6.15.3-creator-pxa270.patch
~/microtime/create-pxa270/
SHELL> cd ~/microtime/create-pxa270/
SHELL> patch -p0 < linux-2.6.15.3-creator-pxa270.patch
```

2. Build Kernel:
   Create the configure file, '.config', for PXA270:

```
SHELL> cd ~/microtime/linux
SHELL> make mrproper
SHELL> make creator_pxa270_defconfig
```

   Build Kernel:

```
SHELL> make clean
SHELL> make
```

   It takes couple minutes to complete the compilation. Please be patient.

3. Build uImage for PXA270:

   - Method 1 – Convert compressed kernel image (`zImage`) to U-Boot format (`uImage`):
     After successfully compile the kernel, you will see a `zImage` files in `linux/arch/arm/boot/`. We need to convert the `zImage` to `uImage`. The `zImage` contains a compressed linux kerenl and a self-decompression program at the head of the file. After system start up, the boot-loader moves the `zImage` to SDRAM, the zImage will execute the self-decompression program to extract the linux kernel, and start normal linux kernel boot process.

     ```
     SHELL> cd ~/microtime
     SHELL> ./build-linux/mkimage -A arm -O linux -T
     kernel -C none
     -a 0xa0008000 -e 0xa0008000 -n ‘‘2.6.15.3 kernel for
     Creator-PXA270’’
     -d linux/arch/arm/boot/zImage uImage
     ```

   - Method 2 – Convert uncompressed kernel image (`vmlinux`) to U-Boot format (`uImage`):

After successfully compile the kernel, we can find a uncompressed linux kerenl `vmlinux` in `linux/`. We should compressed the vmlinux, and convert it to U-Boot readable format. After system start up, the bootloader will directly decompressed `zImage` from NAND FLASH to SDRAM, and start normal linux kernel boot process.

```
SHELL> cd ~/microtime
SHELL> arm-unknown-linux-gnu-objcopy -O binary -R
.note -R .comment -S linux/vmlinux linux.bin
SHELL> gzip -9 -c linux.bin > zImage
SHELL> ./build-linux/mkimage -A arm -O linux -T
kernel -C gzip
-a 0xa0008000 -e 0xa0008000 -n ‘‘2.6.15.3 kernel for
Creator-PXA270’’
-d zImage uImage
```

Please check the size of `uImage`. It should be about 1.5MB. You need to repeat the above steps if you get a very small uImage.

8.2 Build Root Filesystem

The root filesystem is the filesystem that is contained on the same partition on which the *root* directory is located, and it is the filesystem on which all the other filesystems are mounted (i.e., logically attached to the system) as the system is booted up (i.e., started up). In this practice, we are going to build a root filesystem in JFFS2 format. JFFS2 (Journalling Flash File System version 2) is a log-structured filesystem for use with flash memory devices.

1. Unzip Root Filesystem Source

   ```
   SHELL> cd ~
   SHELL> sudo tar xzvf rootfs.tar.gz
   SHELL> cd microtime/
   SHELL> sudo chown -R lab616\:lab616 rootfs/
   ```

2. Put permanent items into your rootfs:

   If you want to put file /usr/lab616/tt.bin into /usr) and you are now in the directory of `microtime/rootfs/`:

   ```
   SHELL> cp /usr/lab616/tt.bin usr/
   ```
   //

3. Build root filesystem

   ```
   SHELL> cd ~/microtime
   SHELL> ./build-linux/mkfs.jffs2 -v -e 131072 --pad=0xa00000
   -r rootfs/
   -o rootfs.jffs2
   ```

9 Flash & Boot

In computing, 'booting' is a process that starts operating systems when the user turns on a computer system. A boot sequence is the initial set of operations that the computer performs when power is switched on. The boot loader typically loads the main operating system for the computer. In this practice, we are using 'U-Boot' bootloader to boot up PXA270.

9.1 Flash `uImage`

- Power on PXA270 and press any key before U-Boot autoboots the target.
- Configure U-Boot environment variables for `tftp` download:

```
UBOOT> setenv ipaddr < ip3 >
UBOOT> setenv serverip < ip2 >
UBOOT> setenv bootargs root=/dev/mtdblock3 rw
rootfstype=jffs2 console=ttyS0,9600n8 mem=64M ip=<
ip3 >:< ip2 >:< gw  ip >:255.255.255.0::eth0:off
UBOOT> saveenv
```

  The $< gw\ ip >$ is the gateway IP. It can be skipped if PXA270 and Linux Host are located in the same subnet. You can use `printenv` to check the settings.

- Prepare `uImage` and `rootfs.jffs2`:
  Build the `uImage` and `rootfs.jffs2`, then move them to the <tftproot>.
- Download and flash `uImage`:

```
UBOOT> tftp a1100000 uImage
UBOOT> protect off 100000 47ffff
UBOOT> erase 100000 47ffff
UBOOT> cp.b a1100000 100000 200000
```

9.2 Flash `rootfs`

- Download and flash `rootfs.jffs2`:

```
UBOOT> tftp a1480000 rootfs.jffs2
UBOOT> protect off 480000 137ffff
UBOOT> erase 480000 137ffff
UBOOT> cp.b a1480000 480000 a00000
```

9.3 Boot your PXA270

Now, you can either boot your Linux automatically or manually. For manual boot, please press 'Enter' before U-Boot starts the autoboot, then type:

```
UBOOT> bootm 100000
```

Otherwise, to set up a correct autoboot, you need to configure U-Boot environment variables:

```
UBOOT> setenv bootm 100000
UBOOT> setenv bootcmd run linux
UBOOT> saveenv
```

Then, press the 'Reset' bottom on PXA270 to enable the autoboot.

Lab 2-1 Prepare Boot Image of Xilinx ZC702

Wen-Lin Sun

2018-9-21

In this lab, we will prepare the things that we need when booting up Linux on Xilinx ZC702. The files are first-stage boot loader(FSBL), second-stage boot loader(e.g. U-boot), device tree, kernel image and root filesystem. Make sure you have these files after you finish this lab:

- boot.bin (fsbl.elf + u-boot.elf)
- uImage
- uramdisk.image.gz
- devicetree.dtb

And put them into a directory (create it yourself).

1 Environment Setup

Before building all of these files, we should install several tools.

- Install 32-bit libraries:
  Update the database before we install anything.

```
SHELL> sudo apt-get update
SHELL> sudo apt-get install lib32z1 lib32ncurses5 lib32ncurses5-dev lib32stdc++6
libbz2-1.0:i386
```

- Download & Install Xilinx Vivado Design Suite (you may need to register):
  Recommend: Install it under your home directory.
  Official website: https://www.xilinx.com/support/download.html
- Setup Environment Variables:

```
SHELL> export CROSS_COMPILE=arm-linux-gnueabihf-
SHELL> source [xilinx_directory]/Xilinx/SDK/2018.2/settings64.sh
```

- Install Git:

```
SHELL> sudo apt-get install git
```

- Build & Install device tree compiler:

```
SHELL> git clone https://git.kernel.org/pub/scm/utils/dtc/dtc.git
SHELL> cd dtc
SHELL> sudo apt-get install flex bison swig python-dev
SHELL> make
SHELL> export PATH=‘pwd‘:$PATH
```

2 Generate FSBL

You should download the hardware definition file `system.hdf from E3 first.`

- `Create a link gmake to make:`
  `Formally, we should install gmake, but we can also use make instead here.`

```
SHELL> sudo ln -s /usr/bin/make /usr/bin/gmake
SHELL> cd <hdf_dir>
SHELL> hsi
hsi% set hwdsgn [open_hw_design system.hdf]
hsi% generate_app -hw $hwdsgn -os standalone -proc ps7_cortexa9_0 -app zynq_fsbl
-compile -sw fsbl -dir <fsbl_dir>
hsi% exit
```

  `The file fsbl will be placed at <fsbl_dir> as you set in the command. Just rename it to fsbl.elf.`

3 Build U-boot

`In this lab, we use u-boot as our second-stage boot loader, for helping us boot from Linux. Since the tool mkimage we use when building kernel image will also be generated when building u-boot, we should make sure this step is done correctly before we continue.`

- `Build u-boot:`

```
SHELL> git clone https://github.com/Xilinx/u-boot-xlnx.git
SHELL> cd u-boot
SHELL> make zynq_zc702_defconfig
SHELL> make
```

  `The file u-boot will be generated in this directory and the mkimage will be placed under the directory tools. Just rename it to u-boot.elf.`

- `Export mkimage tools to PATH`

```
SHELL> cd tools
SHELL> export PATH=‘pwd‘:$PATH
```

4 Build uImage & Device Tree

`In this section, we will build kernel image and attach it with u-boot recognizable header. Alternatively, we will build the the corresponding device tree.`

- Download & Configure Linux kernel:
  In this lab, we use the version 2016.4.

```
SHELL> git clone https://github.com/Xilinx/linux-xlnx.git
SHELL> git checkout tags/xilinx-v2016.4
SHELL> make ARCH=arm xilinx_zynq_defconfig
SHELL> make ARCH=arm menuconfig
```

- Build kernel image with u-boot header (uImage):

```
SHELL> make ARCH=arm UIMAGE_LOADADDR=0x8000 uImage
```

  The uImage will be placed under arch/arm/boot.

- Build device tree:

```
SHELL> make ARCH=arm zynq-zc702.dtb
```

  The device tree zynq-zc702.dtb will be placed under arch/arm/boot/dts. Just rename it to devicetree.dtb.

5 Modify Root Filesystem

In this lab, we will use BusyBox as our root filesystem.

- Download & Configure BusyBox:
  We use the branch 1_29_stable for example.

```
SHELL> git clone git://git.busybox.net/busybox -b 1_29_stable
SHELL> make ARCH=arm menuconfig
```

  Modify the configuration as you need. Make sure [Setting] -> [Build Options] -> [Build static binary (no shared libs)] is selected. And then, build it.

- Build & Generate Busybox and related files:

```
SHELL> make ARCH=arm
```

  After that, use make install to gather the compiled Busybox and symlink into the directory _install.

```
SHELL> make ARCH=arm install
```

- Modify the filesystem:
  - Manually add missing directories of root filesystem to _install. (/dev, /sys, /proc, /root and /etc/init.d)

```
SHELL> cd _install
SHELL> mkdir -p dev sys proc root etc/init.d
```

– Manually create initial script rcS under etc/init.d and make it executable.

```
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys
/sbin/mdev -s
```

– Manually create inittab under etc.

```
#!/bin/sh

# Init script
::sysinit:/etc/init.d/rcS

# Start shell on the serial ports
::respawn:/sbin/getty -L ttyPS0 115200 vt100

# Execute /sbin/init when restarting the init process
::restart:/sbin/init

# Umount everything before rebooting
::shutdown:/bin/umount -a -r
```

– Set etc/passwd to make root be able to login without password.

```
root::0:0:root:/root:/bin/sh
```

– Create soft link to /init to avoid that kernel cannot found the init.

```
SHELL> ln -s /sbin/init init
```

- After changing all the directory to be owned by root, you can pack the root filesystem with cpio format. And then, compress it with gzip format. The output file will be named ramdisk.cpio.gz and placed under _install.

```
SHELL> cd ..
SHELL> sudo chown -R root:root _install
SHELL> cd _install
SHELL> find .  | cpio -H newc -o | gzip -9 > ../ramdisk.cpio.gz
```

- Transform the ramdisk.cpio.gz into u-boot format with mkimage.

```
SHELL> mkimage -A arm -T ramdisk -C gzip -d ramdisk.cpio.gz uramdisk.image.gz
```

Now you have the compressed root filesystem file uramdisk.image.gz under busybox.

6 Prepare Boot Image

In this section, we will generate the boot image boot.bin. Before doing this step, make sure you have fsbl.elf and u-boot.elf.

- Put them into a directory.
- Enter the directory and create a file boot.bif with following content (sample):

```
1 image : {
2         [bootloader]fsbl.elf
3         u-boot.elf
4 }
```

- Generate the boot image.
  bootgen merges fsbl.edf and u-boot.elf to boot image boot.bin.

```
SHELL> bootgen -image boot.bif -o i boot.bin
```

Lab 2-2 Prepare Boot Medium of Xilinx ZC702

Wen-Lin Sun

2018-9-21

In this lab, we will practice using two kinds of boot media: SD card and QSPI flash to boot up ZC702 with the image that we prepared in previous part.

1 SD Boot

Use SD card as boot medium is the most convenient way to test your boot image.

- Format the sd card with FAT format.
- Put all the files that you have generated into the card.
  - boot.bin
  - uImage
  - uramdisk.image.gz
  - devicetree.dtb
- Check `SW16 setting. (Figure 2)`

2 QSPI Boot

Because there is nothing in QSPI flash now, we cannot boot from it directly now. There are several ways to write something to the QSPI flash, such as JTAG(with Xilinx SDK), Linux driver, u-boot. In this lab, we provide a way which requires the well prepared SD card in last section and write things to flash through u-boot, which is similar to what we have done in previous lab.

- Use SD boot to boot into u-boot.
  After booting into u-boot, you will see the screen shows `Zync> as following.`

```
Zync>
```

- `Load the files from SD card to Ram.`

```
Zync> load mmc 0 0x6000000 ${boot_image}
Zync> load mmc 0 ${kernel_load_address} ${kernel_image}
Zync> load mmc 0 ${devicetree_load_address} ${devicetree_image}
Zync> load mmc 0 ${ramdisk_load_address} ${ramdisk_image}
```

- Init flash device on given SPI bus and chip select.

```
Zync> sf probe 0 20000000 0
```

- Write boot.bin from ram to flash.

```
Zync> sf protect unlock 0 ${boot_size}
Zync> sf erase 0 ${boot_size}
Zync> sf write 0x6000000 0 ${boot_size}
```

- Write uImage from ram to flash.

```
Zync> sf protect unlock 0x100000 ${kernel_size}
Zync> sf erase 0x100000 ${kernel_size}
Zync> sf write ${kernel_load_address} 0x100000 ${kernel_size}
```

- Write devicetree.dtb from ram to flash.

```
Zync> sf protect unlock 0x600000 ${devicetree_size}
Zync> sf erase 0x600000 ${devicetree_size}
Zync> sf write ${devicetree_load_address} 0x600000 ${devicetree_size}
```

- Write uramdisk.image.gz from ram to flash.

```
Zync> sf protect unlock 0x620000 ${ramdisk_size}
Zync> sf erase 0x620000 ${ramdisk_size}
Zync> sf write ${ramdisk_load_address} 0x620000 ${ramdisk_size}
```

- Reboot it with the QSPI boot SW16 setting (Figure 2).

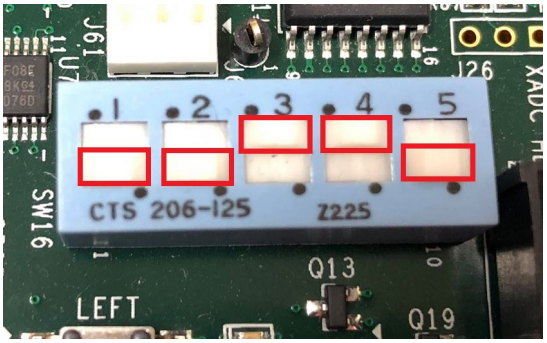


Figure 1: SD boot mode switch setting

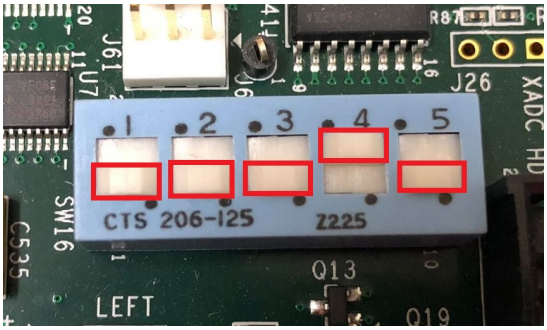


Figure 2: QSPI boot mode switch setting

Lab 3: Task

Yu-Lun Huang

2019-7-16

1 Objective

- Be familiar with system calls: `fork()`, `wait()`, `waitpid()`, `nice()`, `exec()`, 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, umask, 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 the `fork()` function 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.

```
1  /*
2   * process.c
3   */
4
5  #include <errno.h>      /* Errors */
6  #include <stdio.h>      /* Input/Output */
7  #include <stdlib.h>     /* General Utilities */
8  #include <sys/types.h>  /* Primitive System Data Types */
9  #include <sys/wait.h>   /* Wait for Process Termination */
10 #include <unistd.h>     /* Symbolic Constants */
11
12 pid_t childpid; /* variable to store the child's pid */
13
14 void childfunc(void)
15 {
16    int retval;     /* user-provided return code */
17
18    printf("CHILD: I am the child process!\n");
19    printf("CHILD: My PID: %d\n", getpid());
```

```
20      printf("CHILD: My parent's PID is: %d\n", getppid());
21      printf("CHILD: Sleeping for 1 second...\n");
22      sleep(1); /* sleep for 1 second */
23
24      printf("CHILD: Enter an exit value (0 to 255): ");
25      scanf(" %d", &retval);
26      printf("CHILD: Goodbye!\n");
27
28      exit(retval); /* child exits with user-provided return code */
29  }
30
31  void parentfunc(void)
32  {
33      int status;     /* child's exit status */
34
35      printf("PARENT: I am the parent process!\n");
36      printf("PARENT: My PID: %d\n", getpid());
37      printf("PARENT: My child's PID is %d\n", childpid);
38      printf("PARENT: I will now wait for my child to exit.\n");
39
40      /* wait for child to exit, and store its status */
41      wait(&status);
42      printf("PARENT: Child's exit code is: %d\n", WEXITSTATUS(status));
43      printf("PARENT: Goodbye!\n");
44
45      exit(0);  /* parent exits */
46  }
47
48  int main(int argc, char *argv[])
49  {
50      /* now create new process */
51      childpid = fork();
52
53      if (childpid >= 0) { /* fork succeeded */
54          if (childpid == 0) { /* fork() returns 0 to the child process */
55              childfunc();
56          } else { /* fork() returns new pid to the parent process */
57              parentfunc();
58          }
59      } else { /* fork returns -1 on failure */
60          perror("fork"); /* display error message */
61          exit(0);
62      }
63
64      return 0;
65  }
```

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 behaviour is modifiable via the options argument. Please refer to its man page.

```
1  /*
2   * waitpid.c -- shows how to get child's exit status
3   */
4
5  #include <errno.h>      /* Errors */
6  #include <stdio.h>      /* Input/Output */
7  #include <stdlib.h>     /* General Utilities */
8  #include <sys/types.h>  /* Primitive System Data Types */
9  #include <sys/wait.h>   /* Wait for Process Termination */
10 #include <time.h> /* Time functions */
11 #include <unistd.h>     /* Symbolic Constants */
12
13 pid_t childpid; /* variable to store the child's pid */
14
15 void childfunc(void)
16 {
17    int randtime;     /* random sleep time */
18    int exitstatus;       /* random exit status */
19
20    printf("CHILD: I am the child process!\n");
21    printf("CHILD: My PID: %d\n", getpid());
22
23    /* sleep */
24    srand(time(NULL));
25    randtime = rand() % 5;
26    printf("CHILD: Sleeping for %d second...\n", randtime);
27    sleep(randtime);
28
29    /* rand exit status */
30    exitstatus = rand() % 2;
31    printf("CHILD: Exit status is %d\n", exitstatus);
32
33    printf("CHILD: Goodbye!\n");
34    exit(exitstatus); /* child exits with user-provided return code */
35 }
36
37 void parentfunc(void)
38 {
39    int status;     /* child's exit status */
40    pid_t pid;
41
42    printf("PARENT: I am the parent process!\n");
43    printf("PARENT: My PID: %d\n", getpid());
44
45
46    printf("PARENT: I will now wait for my child to exit.\n");
```

```
47
48      /* wait for child to exit, and store its status */
49      do
50      {
51          pid = waitpid(childpid, &status, WNOHANG);
52          printf("PARENT: Waiting child exit ...\n");
53          sleep(1);
54      }while (pid != childpid);
55
56      if (WIFEXITED(status)) {
57          // child process exited normally.
58          printf("PARENT: Child's exit code is: %d\n",
59                  WEXITSTATUS(status));
60      }else{
61          // Child process exited thus exec failed.
62          // LOG failure of exec in child process.
63          printf("PARENT: Child process executed but exited failed.\n");
64      }
65
66      printf("PARENT: Goodbye!\n");
67
68      exit(0);  /* parent exits */
69  }
70
71  int main(int argc, char *argv[])
72  {
73      /* now create new process */
74      childpid = fork();
75
76      if (childpid >= 0) { /* fork succeeded */
77          if (childpid == 0) { /* fork() returns 0 to the child process */
78              childfunc();
79          } else { /* fork() returns new pid to the parent process */
80              parentfunc();
81          }
82      } else { /* fork returns -1 on failure */
83          perror("fork"); /* display error message */
84          exit(0);
85      }
86
87      return 0;
88  }
```

- `nice()` adds *inc* 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).

```c
#include <unistd.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.

```c
/*
 * 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 specified 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 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'.

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

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

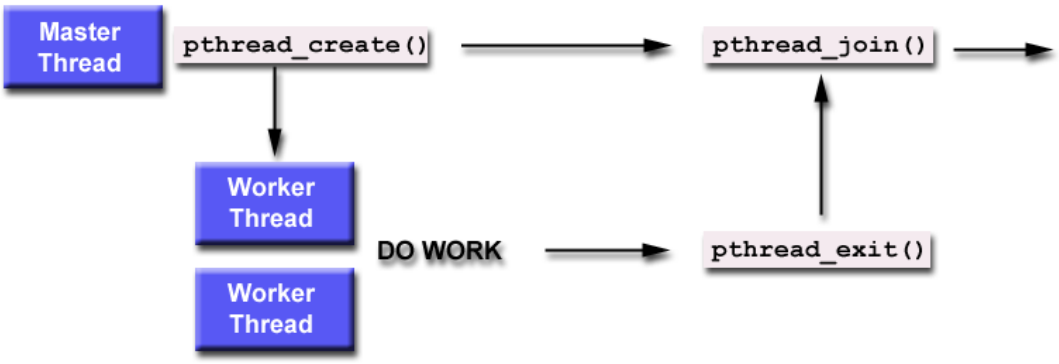


Figure 1: pthread_join()

```
8
9  #define NUM_THREADS  5
10
11 void *show(void *threadid)
12 {
13    long tid;
14
15    tid = (long)threadid;
16    printf("Hello! I am thread #%ld!\n", tid);
17
18    pthread_exit(NULL);
19 }
20
21 int main(int argc, char *argv[])
22 {
23    pthread_t threads[5];
24    int rc;
25    int t;
26
27    for(t=0;t<NUM_THREADS;t++){
28       printf("In main(): creating thread %d\n", t);
29       rc = pthread_create(&threads[t], NULL, show, (void *)t);
30       if (rc){
31          printf("ERROR; return code from pthread_create() is %d\n", rc);
32          exit(-1);
33       }
34    }
35
36    printf("Main: program completed. Exiting.\n");
37    pthread_exit(NULL);
38 }
```

```
1  /*
2   * join.c -- shows how to "wait" for thread completions
3   */
4
```

```
5  #include <math.h>
6  #include <pthread.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 #define NUM_THREADS  4
11
12 void *BusyWork(void *t)
13 {
14    int i;
15    long tid;
16    double result=0.0;
17
18    tid = (long)t;
19    printf("Thread %ld starting...\n",tid);
20
21    for (i = 0; i < 1000000; i++)
22    {
23       result += sin(i) * tan(i);
24    }
25
26    printf("Thread %ld done. Result = %e\n", tid, result);
27    pthread_exit((void*) t);
28 }
29
30 int main (int argc, char *argv[])
31 {
32    pthread_t thread[NUM_THREADS];
33    pthread_attr_t attr;
34    int rc;
35    long t;
36    void *status;
37
38    /* Initialize and set thread detached attribute */
39    pthread_attr_init(&attr);
40    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
41
42    for (t = 0; t < NUM_THREADS; t++) {
43       printf("Main: creating thread %ld\n", t);
44       rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
45       if (rc) {
46          printf("ERROR; return code from pthread_create() is %d\n", rc);
47          exit(-1);
48       }
49    }
50
51    /* Free attribute and wait for the other threads */
52    pthread_attr_destroy(&attr);
53    for (t = 0; t < NUM_THREADS; t++) {
54       rc = pthread_join(thread[t], &status);
55       if (rc) {
```

```
56              printf("ERROR; return code from pthread_join() is %d\n", rc);
57              exit(-1);
58          }
59          printf("Main: join with thread %ld (status: %ld)\n",t,(long)status);
60      }
61
62      printf("Main: program completed. Exiting.\n");
63      pthread_exit(NULL);
64  }
```

```
1   /*
2    * deteach.c -- shows how to detach a thread
3    */
4
5   #include <errno.h>
6   #include <pthread.h>
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <unistd.h>
10
11  void *threadfunc(void *parm)
12  {
13      printf("Inside secondary thread\n");
14      sleep(3);
15      printf("Exit secondary thread\n");
16      pthread_exit(NULL);
17  }
18
19  int main(int argc, char **argv)
20  {
21      pthread_t thread;
22      int rc = 0;
23
24      printf("Create a thread using attributes that allow detach\n");
25      rc = pthread_create(&thread, NULL, threadfunc, NULL);
26      if (rc) {
27          printf("ERROR; return code from pthread_create() is %d\n", rc);
28          exit(-1);
29      }
30
31      printf("Detach the thread after it terminates\n");
32      rc = pthread_detach(thread);
33      if (rc) {
34          printf("ERROR; return code from pthread_detach() is %d\n", rc);
35          exit(-1);
36      }
37
38      printf("Detach the thread again\n");
39      rc = pthread_detach(thread);
40      /* EINVAL: No thread could be found corresponding to that
41       * specified by the given thread ID.
```

```
42      */
43     if (rc != EINVAL) {
44         printf("Got an unexpected result! rc=%d\n", rc);
45         exit(1);
46     }
47     printf("Second detach fails as expected.\n");
48
49     /* sleep() is not a very robust way to wait for the thread */
50     sleep(6);
51     printf("Main() completed.\n");
52     return 0;
53 }
```

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

In host machine, you can natively compile your code with `gcc` by:

```
SHELL> gcc -o pthread pthread.c -lpthread
```

If you want to cross compile the code for your target, please compile it with:

```
SHELL> arm-unknown-linux-gnu-gcc -o pthread pthread.c
-L /opt/arm-unknown-linux-gnu/arm-unknown-linux-gnu/lib/
-I /opt/arm-unknown-linux-gnu/arm-unknown-linux-gnu/include/
-lpthread
```

In case you need to compile similar codes on your host, you need to use

```
HOST SH> gcc -o pthread pthread.c -lpthread
```

Lab 4: Inter-Process Communication (Part I)

Yu-Lun Huang

2019-07-16

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()`, 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
- 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 SEMMODE 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;
}
```

- 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 the 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>
```

```c
#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;
}
```

- 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 three `doodle` 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;
}
```

- 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>
```

```c
#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, "Creation of semaphore %ld 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 semaphore: %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 semaphore %ld\n",
              argv[0], SEM_KEY);
        exit(1);
    }
    printf("Semaphore %ld has been remove\n", SEM_KEY);
#endif

    return 0;
}
```

4 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;
}
```

Lab 5: Inter-Process Communication (Part II)

Yu-Lun Huang

2019-07-16

1 Objective

- Be familiar with inter-process communication: pipe, shared memory, etc.

2 Prerequisite

- Read man pages of `pipe()`, `shmget()`, `shmctl()`, `shmat()`, `shmdt()`, etc.

3 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.

```
1  /* pipe.c
2   *
3   * child process read the content of file
4   * and write the content to parent process through pipe
5   */
6
7  #include <fcntl.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <string.h>
11 #include <sys/stat.h>
12 #include <sys/types.h>
13 #include <sys/wait.h>
14 #include <unistd.h>
15
16 int pfd[2]; /* pfd[0] is read end, pfd[1] is write end */
17
18 void ChildProcess(char *path)
19 {
20     int fd;
```

```
21      int ret;
22      char buffer[100];
23
24      /* close unused read end */
25      close(pfd[0]);
26
27      /* open file */
28      fd = open(path, O_RDONLY);
29      if (fd < 0) {
30          printf("Open %s failed.\n", path);
31          exit(EXIT_FAILURE);
32      }
33
34      /* read file and write content to pipe */
35      while (1) {
36          /* read raw data from file */
37          ret = read(fd, buffer, 100);
38
39          if (ret < 0) {     /* error */
40              perror("read()");
41              exit(EXIT_FAILURE);
42          }
43          else if (ret == 0) { /* reach EOF */
44              close(fd);  /* close file */
45              close(pfd[1]); /* close write end, reader see EOF */
46              exit(EXIT_SUCCESS);
47          }
48          else {          /* write content to pipe */
49              write(pfd[1], buffer, ret);
50          }
51      }
52  }
53
54  void ParentProcess()
55  {
56      int ret;
57      char buffer[100];
58
59      /* close unused write end */
60      close(pfd[1]);
61
62      /* read data from pipe until reach EOF */
63      while(1) {
64          ret = read(pfd[0], buffer, 100);
65
66          if (ret > 0) { /* print data to screen */
67              printf("%.*s", ret, buffer);
68          }
69          else if (ret == 0) { /* reach EOF */
70              close(pfd[0]); /* close read end */
71              wait(NULL);
```

```
72              exit(EXIT_SUCCESS);
73          }
74          else {
75              perror("pipe read()");
76              exit(EXIT_FAILURE);
77          }
78      }
79  }
80
81  int main(int argc, char *argv[])
82  {
83      pid_t cpid;
84
85      if (argc != 2) {
86          fprintf(stderr, "%s: specify a file\n", argv[0]);
87          exit(1);
88      }
89
90      /* create pipe */
91      if (pipe(pfd) == -1) {
92          perror("pipe");
93          exit(EXIT_FAILURE);
94      }
95
96      /* fork child process */
97      cpid = fork();
98      if (cpid == -1) { /* error */
99          perror("fork");
100         exit(EXIT_FAILURE);
101     }
102
103     if (cpid == 0)
104         ChildProcess(argv[1]);
105     else
106         ParentProcess();
107
108     return 0;
109 }
```

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.

```
1  /*
2   * select.c
3   */
4
```

```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);

```

```
56      /* create 2 child processes and set corresponding pipe & sleep time */
57      cpid1 = fork();
58      if (cpid1 == 0)
59          ChildProcess(pfd1, random() % 5);
60
61      cpid2 = fork();
62      if (cpid2 == 0)
63          ChildProcess(pfd2, random() % 4);
64
65      /* close unused write end */
66      close(pfd1[1]);
67      close(pfd2[1]);
68
69      /* set pfd1[0] & pfd2[0] to watch list */
70      FD_ZERO(&rfds);
71      FD_ZERO(&arfds);
72      FD_SET(pfd1[0], &arfds);
73      FD_SET(pfd2[0], &arfds);
74      max_fd = max(pfd1[0], pfd2[0]) + 1;
75
76      /* Wait up to five seconds. */
77      tv.tv_sec = 5;
78      tv.tv_usec = 0;
79
80      while(1)
81      {
82          /* config fd_set for select */
83          memcpy(&rfds, &arfds, sizeof(rfds));
84
85          /* wait until any fd response */
86          retval = select(max_fd, &rfds, NULL, NULL, &tv);
87
88          if (retval == -1) {  /* error */
89              perror("select()");
90              exit(EXIT_FAILURE);
91          }
92          else if (retval) {   /* # of fd got respone */
93              printf("Data is available now.\n");
94          }
95          else {   /* no fd response before timer expired */
96              printf("No data within five seconds.\n");
97              break;
98          }
99
100         /* check if any response */
101         for (fd_index = 0; fd_index < max_fd; fd_index++)
102         {
103             if (!FD_ISSET(fd_index, &rfds))
104                 continue;   /* no response */
105
106             retval = read(fd_index, buffer, 100);
```

```
107
108             if (retval > 0)      /* read data from pipe */
109                 printf("%.*s", retval, buffer);
110             else if (retval < 0) /* error */
111                 perror("pipe read()");
112             else {          /* write fd closed */
113                 /* close read fd */
114                 close(fd_index);
115                 /* remove fd from watch list */
116                 FD_CLR(fd_index, &arfds);
117             }
118         }
119     }
120
121     return 0;
122 }
```

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:

```
1  /*
2   * shm_server.c -- creates the string and shared memory.
3   */
4
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <sys/ipc.h>
8  #include <sys/shm.h>
9  #include <sys/types.h>
10 #include <unistd.h>
11
12 #define SHMSZ     27
13
14 int main(int argc, char *argv[])
15 {
16     char c;
17     int shmid;
18     key_t key;
19     char *shm, *s;
20     int retval;
21
22     /* We'll name our shared memory segment "5678" */
23     key = 5678;
24
```

```
25      /* Create the segment */
26      if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
27          perror("shmget");
28          exit(1);
29      }
30
31      /* Now we attach the segment to our data space */
32      if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
33          perror("shmat");
34          exit(1);
35      }
36      printf("Server create and attach the share memory.\n");
37
38      /* Now put some things into the memory for the other process to read */
39      s = shm;
40
41      printf("Server write a ~ z to share memory.\n");
42      for (c = 'a'; c <= 'z'; c++)
43          *s++ = c;
44      *s = '\0';
45
46      /*
47       * Finally, we wait until the other process changes the first
48       * character of our memory to '*', indicating that it has read
49       * what we put there.
50       */
51      printf("Waiting other process read the share memory ...\n");
52      while (*shm != '*')
53          sleep(1);
54      printf("Server read * from the share memory.\n");
55
56      /* Detach the share memory segment */
57      shmdt(shm);
58
59      /* Destroy the share memory segment */
60      printf("Server destroy the share memory.\n");
61      retval = shmctl(shmid, IPC_RMID, NULL);
62      if (retval < 0)
63      {
64          fprintf(stderr, "Server remove share memory failed\n");
65          exit(1);
66      }
67
68      return 0;
69  }
```

```
1   /*
2    * shm_client.c -- attaches itself to the created shared memory
3    *                 and uses the string (printf).
4    */
5
```

```c
#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 6-1: I/O Control for PXA270

Yu-Lun Huang

2019-07-12

There is an 8-bit LED lamps on the motherboard of PXA270, numbered frome D9(1) to D16(8) We use the `creator-pxa270-lcd.ko` module to control the LED lamps, it is also used to drive the LCD, 7-segement LED, KeyPAD, and DIP Switch.

1 Compile Modules

Rebuild your kernel and rootfile system to support `creator-pxa270-lcd.ko` module:

- Get the source code of `creator-pxa270-lcd.ko`:
  Download `Creator_PXA270_LCD_Device_Driver.src.tar.gz` from E3 website and decompressed it to your kernel source.

```
SHELL> cd ~
SHELL> tar xzvf Creator_PXA270_LCD_Device_Driver.src.tar.gz
```

- Configure kernel source:

```
SHELL> cd ~/microtime/linux
SHELL> make mrproper
SHELL> make menuconfig
```

  - In the window of "Linux Kernel Configuration", select "Load an Alternate Configuration from File" and load the configuration file `arch/arm/configs/creator_pxa270_defconfig`.
  - Select "Device Drivers" → "Character devices" and mark "Creator-pxa270 LCD" as `[M]`.
  - Save and exit kernel configuration.

- Make Image:
  Compile Linux kernel and `creator-pxa270-lcd.ko` module.

```
SHELL> make clean
SHELL> make
```

  The `creator-pxa270-lcd.ko` module will be placed at `microtime/linux/drivers/char/`.

- Make new root filesystem:
  Copy `creator-pxa270-lcd.ko` module into root filesystem.

```
SHELL> cp ~/microtime/linux/drivers/char/creator-pxa270-lcd.ko
~/microtime/rootfs/lib/modules/2.6.15.3/kernel/drivers/char/
```

  Then, rebuild and flash root filesystem.

2 Load Modules

Type the following command to load the `creator-pxa270-lcd.ko` on PXA270.

```
> insmod lib/modules/2.6.15.3/kernel/drivers/char/creator-pxa270-lcd.ko
```

3 PXA I/O: Control LED

- LED programming guide

Header file:

```
#include "asm-arm/arch-pxa/lib/creator_pxa270_lcd.h"
```

Commands:

```
LED_IOCTL_SET         // set the specified LED (D9 - D16)
LED_IOCTL_CLEAR       // clear the specified LED (D9 - D16)
```

Values:

```
LED_ALL_ON          0xFF
LED_ALL_OFF         0x00
LED_D9_INDEX        1
LED_D10_INDEX       2
LED_D11_INDEX       3
LED_D12_INDEX       4
LED_D13_INDEX       5
LED_D14_INDEX       6
LED_D15_INDEX       7
LED_D16_INDEX       8
```

Sample code:

```
/*
 * led.c -- the sample code for controlling LEDs on Creator.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/fcntl.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include "asm-arm/arch-pxa/lib/creator_pxa270_lcd.h"

int main(int argc, char *argv[])
{
    int fd;          /* file descriptor for /dev/lcd */
    int retval;

    unsigned short data;

```

```
19      /* Open device /dev/lcd */
20      if((fd = open("/dev/lcd", O_RDWR)) < 0)
21      {
22          printf("Open /dev/lcd faild.\n");
23          exit(-1);
24      }
25
26      /* Turn on all LED lamps */
27      data = LED_ALL_ON;
28      ioctl(fd, LED_IOCTL_SET, &data);
29      printf("Turn on all LED lamps\n");
30      sleep(3);
31
32      /* Turn off all LED lamps */
33      data = LED_ALL_OFF;
34      ioctl(fd, LED_IOCTL_SET, &data);
35      printf("Turn off all LED lamps\n");
36      sleep(3);
37
38      /* Turn on D9 */
39      data = LED_D9_INDEX;
40      ioctl(fd, LED_IOCTL_BIT_SET, &data);
41      printf("Turn on D9 \n");
42      sleep(3);
43
44      /* Turn off D9 */
45      data = LED_D9_INDEX;
46      ioctl(fd, LED_IOCTL_BIT_CLEAR, &data);
47      printf("Turn off D9 \n");
48      sleep(3);
49
50      /* Close fd */
51      close(fd);
52
53      return 0;
54  }
```

Add the header search path when compile `led.c`.

```
SHELL> arm-unknown-linux-gnu-gcc -o led led.c
-L /opt/arm-unknown-linux-gnu/arm-unknown-linux-gnu/lib/
-I /opt/arm-unknown-linux-gnu/arm-unknown-linux-gnu/include/
-I /home/lab616/microtime/linux/include/
```

4 PXA I/O: 7-Segment

- Header files: asm-arm/arch-pxa/lib/creator_pxa270_lcd.h
- Function: ioctl(fd, command, data)
  - Command:

_7SEG_IOCTL_ON: turn on 7 segment LED (no data is needed)
_7SEG_IOCTL_OFF: turn off 7 segment LED (no data is needed)
_7SEG_IOCTL_SET: set 7 segment LED (_7seg_info_t)

- Data:

```
typedef struct _7Seg_Info{
   unsigned char Mode;  // _7SEG_MODE_PATTERN or _7SEG_MODE_HEX_VALUE
   unsigned char Which; // D5 ~ D8
   unsigned long Value; // pattern or hex
} _7seg_info_t;
```

- The setting of _7Seg_Info:
  - flags used in Mode:

```
#define _7SEG_MODE_PATTERN        0
#define _7SEG_MODE_HEX_VALUE      1
```

  - flags used in Which:

```
#define _7SEG_D5_INDEX  8  // Segment D5 (1)
#define _7SEG_D6_INDEX  4  // Segment D6 (2)
#define _7SEG_D7_INDEX  2  // Segment D7 (3)
#define _7SEG_D8_INDEX  1  // Segment D8 (4)
#define _7SEG_ALL
       (_7SEG_D5_INDEX|_7SEG_D6_INDEX|_7SEG_D7_INDEX|_7SEG_D8_INDEX)
```

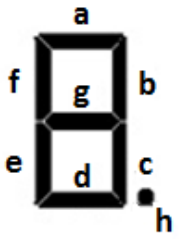
The following is the sample code of 7 segment display control.

```
_7seg_info_t    data;
int             fd, ret, i;

if ((fd = open("/dev/lcd", O_RDWR)) < 0)  return (-1);

ioctl(fd, _7SEG_IOCTL_ON, NULL);
data.Mode = _7SEG_MODE_HEX_VALUE;
data.Which = _7SEG_ALL;
data.Value = 0x2004;
ioctl(fd, _7SEG_IOCTL_SET, &data);
sleep (3);
data.Mode = _7SEG_MODE_PATTERN;
data.Which = _7SEG_D5_INDEX | _7SEG_D8_INDEX;
data.Value = 0x6d7f; /* change to 5008 */
ioctl(fd, _7SEG_IOCTL_SET, &data);
ioctl(fd, _7SEG_IOCTL_OFF, NULL);
close(fd);
```

5 PXA I/O: Keypad

- Keypad I/O

The bit value 1: on
The bit value 0: off



8 bits to represents each Segment

h	g	f	e	d	c	b	a

Figure 1: The layout of 7 segment display

- Header files: asm-arm/arch-pxa/lib/creator_pxa270_lcd.h
- Function: ioctl(fd, command, data)
  - Command:
    KEY_IOCTL_GET_CHAR: unsigned short, get its ASCII value.
    KEY_IOCTL_WAIT_CHAR: wait until get a character.
    KEY_IOCTL_CHECK_EMPTY
    KEY_IOCTL_CLEAR
    KEY_IOCTL_CANCEL_WAIT_CHAR

  - Definition

```
1  #define VK_S2   1      /* ASCII = '1' */
2  #define VK_S3   2      /* ASCII = '2' */
3  #define VK_S4   3      /* ASCII = '3' */
4  #define VK_S5   10     /* ASCII = 'A' */
5  #define VK_S6   4
6  #define VK_S7   5
7  #define VK_S8   6
8  #define VK_S9   11
9  #define VK_S10  7
10 #define VK_S11  8
11 #define VK_S12  9
12 #define VK_S13  12
13 #define VK_S14  14     /* ASCII = '*' */
14 #define VK_S15  0
15 #define VK_S16  15     /* ASCII = '#' */
16 #define VK_S17  13
```

  - Sample

```
1  unsigned short     key;
2  int                fd, ret;
```

```
3
4  if ((fd = open("/dev/lcd", O_RDWR)) < 0)  return (-1);
5
6  ioctl(fd, KEY_IOCTL_CLEAR, key);
7  while(1) {
8    ret = ioctl(fd, KEY_IOCTL_CHECK_EMTPY, &key)
9    if (ret < 0) {
10     sleep(1);
11     continue;
12   }
13   ret = ioctl(fd, KEY_IOCTL_GET_CHAR, &key)
14   if (key & 0xff) == '#') break;    /* terminate */
15 }
16 close(fd);
```

6 PXA I/O: LCD Control

- Header files: asm-arm/arch-pxa/lib/creator_pxa270_lcd.h
- Function: ioctl(fd, command, data)
  - Device Name: `/dev/lcd`
  - Data Structure

```
1  /* Data structure for writing char to LCD screen */
2  typedef struct lcd_write_info {
3     unsigned char Msg[512];       /* the array for saving input */
4     unsigned short Count;         /* the number of input char */
5     int CursorX, CursorY;         /* X, Y axis of cursor */
6  } lcd_write_info_t;
7
8  /* Data structure for writing a picture to LCD screen */
9  typedef struct lcd_full_image_info {
10    unsigned short data[0x800];   /* the array for saving picture */
11 } lcd_full_image_info_t;
```

  - Command

```
1  /* Clear LCD data and move cursor back to the Upper-left corner */
2  #define LCD_IOCTL_CLEAR    LCD_IO ( 0x0 )
3
4  /* Write char to LCD */
5  #define LCD_IOCTL_WRITE    LCD_IOW( 0x01, lcd_write_info_t )
6
7  /* Turn On or Off cursor */
8  #define LCD_IOCTL_CUR_ON   LCD_IO( 0x02 )
9  #define LCD_IOCTL_CUR_OFF  LCD_IO( 0x03 )
10
11 /* Get and Set the position (X, Y) of cursor */
12 #define LCD_IOCTL_CUR_GET  LCD_IOR( 0x04, lcd_write_info_t )
13 #define LCD_IOCTL_CUR_SET  LCD_IOW( 0x05, lcd_write_info_t )
```

```
14
15 /* Write a picture to LCD */
16 #define LCD_IOCTL_DRAW_FULL_IMAGE LCD_IOW(0x06, lcd_full_image_info_t)
```

The following is the sample code of LCD control.

```
1  /*
2   * lcd.c -- The sample code to print "Hello World" on LCD screen.
3   */
4
5  #include <stdio.h>
6  #include <sys/fcntl.h>
7  #include <sys/ioctl.h>
8  #include <unistd.h>
9  #include "asm-arm/arch-pxa/lib/creator_pxa270_lcd.h"
10
11 int main()
12 {
13    int fd;
14    lcd_write_info_t display;  /* struct for saving LCD data */
15
16    /* Open device /dev/lcd */
17    if ((fd = open("/dev/lcd",O_RDWR) < 0))
18    {
19       printf("open /dev/lcd error\n");
20       return (-1);
21    }
22
23    /* Clear LCD */
24    ioctl(fd,LCD_IOCTL_CLEAR,NULL);
25
26    /* Save output string to display data structure */
27    display.Count = sprintf((char *) display.Msg, "Hello World\n");
28    /* Print out "Hello World" to LCD */
29    ioctl(fd, LCD_IOCTL_WRITE, &display);
30
31
32    /* Get the cursor position */
33    ioctl(fd, LCD_IOCTL_CUR_GET, &display);
34    printf("The cursor position is at (x,y) = (%d,%d)\n",
35          display.CursorX, display.CursorY);
36
37    close(fd);
38    return 0;
39 }
```

7 PXA I/O: Audio Control

- Header files: asm-arm/arch-pxa/lib/creator_pxa270_lcd.h
- Function: ioctl(fd, command);

- Device Name: `/dev/lcd`
- Command

```
IOCTL_RECORD_START
IOCTL_RECORD_STOP
IOCTL_PLAY_START
IOCTL_PLAY_STOP
```

The following is the sample code of Audio control.

```
#include <sys/ioctl.h>
#include <stdio.h>
#include <stdlib.h>
#include "creator_s3c4510_codec.h"//must write the absolutely path

#define RECORDING_SIZE 8*8000  /* no of bytes for 4 seconds at 8000 per second */
#define SIZE_16 4*8000   /* no of int for 4 seconds */
main() {
int codec_fd;
int TotalReadSize, nRead, count, i;
char AudioBuffer[RECORDING_SIZE];
FILE *audio_fd;

codec_fd = open( "/dev/codec" , O_RDWR);  /* Open the codec device driver */
if  (codec_fd < 0) {
    printf ( "Open /dev/codec  error\n"    );
    return (-1);
}

if( ioctl (codec_fd, IOCTL_RECORD_START) < 0) {  /*Start recording */
    printf ( "Audio recording start error\n"    );
    close(codec_fd);
    return (-1);
}
printf("Say something to the microphone\n");
sleep(4);  /* record 4 seconds of data */

if( ioctl (codec_fd, IOCTL_RECORD_STOP) < 0) {  /* Stop recording */
    printf ("Audio recording stop error\n");
    close(codec_fd);
    return (-1);
}
printf("Recording stopped\n");

sleep(3);  /* sleep for 3 seconds */

if( ioctl (codec_fd, IOCTL_PLAY_START) < 0) {  /*Start playback */
    printf ( "Audio playback start error\n");
    close(codec_fd);
    return (-1);
}
printf("Start playing recorded data repeatedly\n");
```

```
43 sleep(12);  /* Sleep for 12 seconds to allow repeating twice */
44
45 if( ioctl (codec_fd, IOCTL_PLAY_STOP) < 0) {  /*Stop playback */
46     printf ( "Audio playback stop error\n");
47     close(codec_fd);
48     return (-1);
49 }
50 printf("Playback stopped\n");
51
52 /* Begin reading the data*/
53 TotalReadSize = 0;
54 count =RECORDING_SIZE;
55 do {
56    if (count + TotalReadSize > RECORDING_SIZE)
57         count = RECORDING_SIZE- TotalReadSize ;
58
59    nRead = read(codec_fd, AudioBuffer+TotalReadSize, count);
60    if (nRead > 0 )
61 TotalReadSize += nRead;
62    else if (nRead == 0)  /* EOF */
63         break;
64   else {
65         printf("Reading audio data failed!\n");
66         close(codec_fd);
67         exit(1);
68         }
69 } while (TotalReadSize < RECORDING_SIZE);
70
71 /* write the audio data to a file */
72 audio_fd = fopen("myaudio.txt", "w");
73 for( i = 0; i < SIZE_16; i = i+2) {
74 fprintf(audio_fd,"%d\n", (int) AudioBuffer[i]);  /* convert into 16-bit 2's complement */
75 }
76
77 fclose(audio_fd);
78 close(codec_fd);
79 }
```

Lab 6-2: Socket Programming

Yu-Lun Huang

2019-07-16

1 Objective

- Understand basic socket programming and client-server paradigms.

2 Prerequisite

- Read man pages of `socket()`, `read()`, `write()`, etc.

3 Client-Server Model

The client–server programming model partitions a task between the service providers (servers) and service requesters (clients). Generally, clients initiate communication sessions by sending requests to a server which await incoming requests.

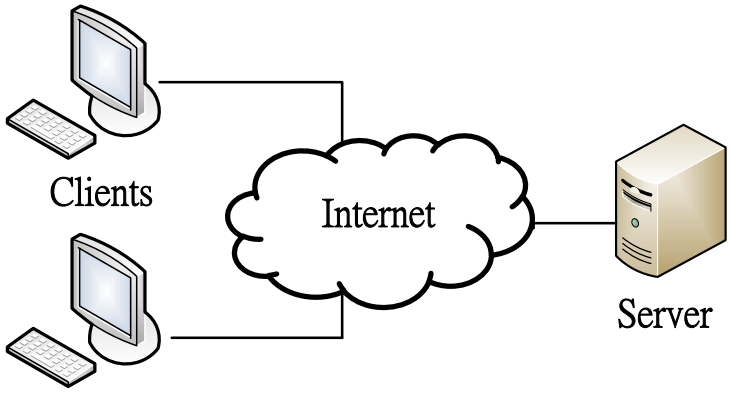


Figure 1: Client-Server Model

If clients and servers communicate over the network, then socket-based programming skills may be used to exchange messages in between. Figure 1 shows a schematic client-server interaction model.

4 Socket Fundamentals

Socket is an interface between an application process and transport layer. With the socket interface, the application process can send and receive messages to or from another application process running on a remote host. Similar to accessing an I/O device, a socket is accessible via a descriptor, as illustrated in Figure 2.

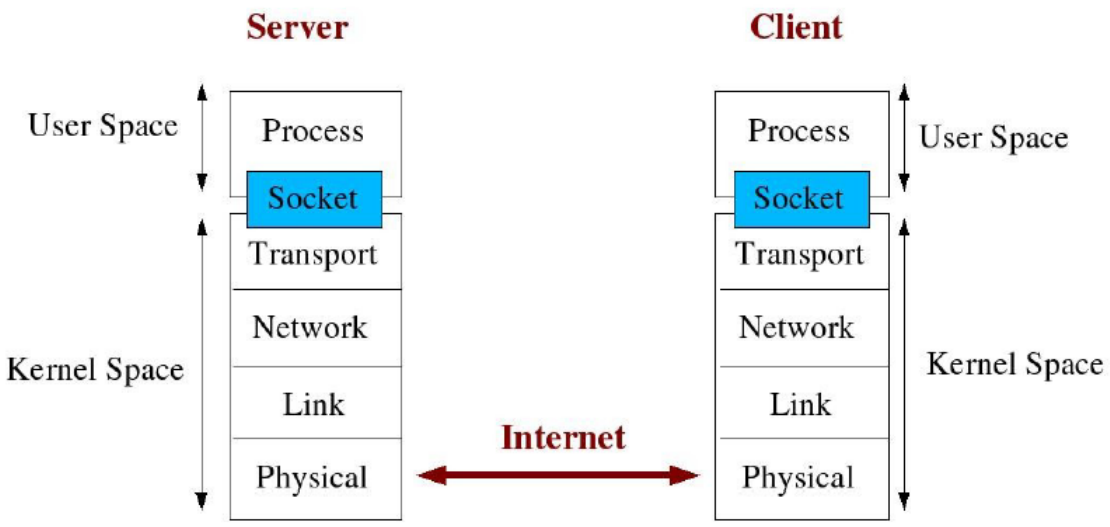


Figure 2: Socket Description.

There are several types of sockets, including internet socket, unix socket, X.25 socket, etc. Internet sockets can be further classified as STREAM socket and DATAGRAM socket, whcih is used for TCP protocol and UDP protocol. The STREAM socket, relying on TCP to provide reliable two-way communication, is used for connection-oriented protocols, as shown in Figure 3.

There are several important system calls for creating sockets for client/server programs:

- `socket()` creates an endpoint for communication and returns a descriptor.
- `connect()` initiates a connection on a socket. Normally used in a client program for sending requests.
- `bind()` binds a name to a socket. Normally, `listen()` is invoked after binding to socket. Used in a server program.
- `listen()` waits for connections on a socket. Used in a server program to wait for incoming requests.
- `accept()` accepts a connection on a socket. Used in a server program.

In a server program, you generally create a socket (using `socket()`, bind to it (`bind()` and listen `listen()` for incoming requests. Once a request is initiated by a client, `accept()` is invoked to accept the connection.

To simplify the sample programs, the common APIs for creating different types of sockets are listed in 'sockop.c':

- `passivesock()` creates a socket and binds to the corresponding socket descriptor to passively accept the incoming connection requests.
- `connectsock()` creates a socket and initiates a connection to the remote host (specified in **struct sockadd_in sin**).

Note that, we also create a new header file (`sockop.h` to include all header files required for socket programming. In `sockop.h`, `sys/socket.h` is necessary for those socket related APIs. In addition, the function prototypes of `passivesock()` and `connectsock()` are also defined in `sockop.h`.

```
1 /*
2  * sockop.h
3  */
```

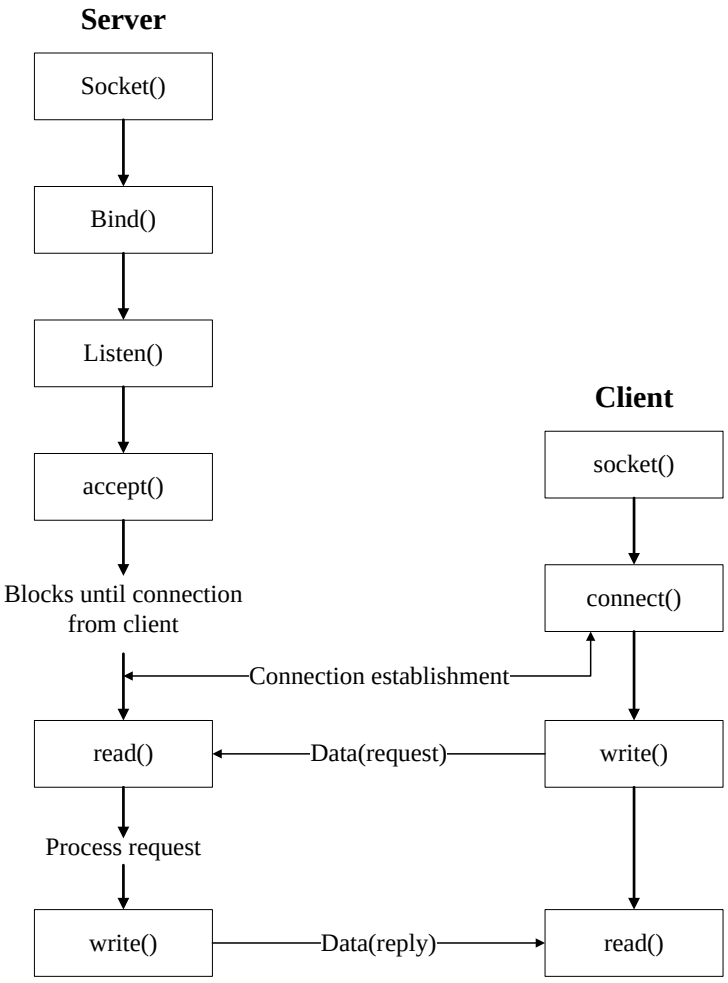


Figure 3: The design flow of Connection-Oriented Protocol.

```
4
5  #ifndef  _SOCKOP_H_
6  #define  _SOCKOP_H_
7
8  #include <arpa/inet.h>
9  #include <errno.h>
10 #include <netdb.h>
11 #include <netinet/in.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <string.h>
15 #include <sys/socket.h>
16 #include <sys/types.h>
17 #include <sys/wait.h>
18
```

```
19 #define errexit(format,arg...) exit(printf(format,##arg))
20
21 /* Create server */
22 int passivesock(const char *service, const char *transport, int qlen);
23
24 /* Connect to server */
25 int connectsock(const char *host, const char *service, const char *transport);
26
27 #endif   /* _SOCKOP_H_ */
```

```
1  /*
2   * sockop.c
3   */
4
5  #include "sockop.h"
6
7  /*
8   * passivesock - allocate & bind a server socket using TCP or UDP
9   *
10  * Arguments:
11  *    service   - service associated with the desired port
12  *    transport - transport protocol to use ("tcp" or "udp")
13  *    qlen      - maximum server request queue length
14  */
15 int passivesock(const char *service, const char *transport, int qlen)
16 {
17    struct servent *pse; /* pointer to service information entry */
18    struct sockaddr_in sin; /* an Internet endpoint address */
19    int s, type;      /* socket descriptor and socket type */
20
21    memset(&sin, 0, sizeof(sin));
22    sin.sin_family = AF_INET;
23    sin.sin_addr.s_addr = INADDR_ANY;
24
25    /* Map service name to port number */
26    if ((pse = getservbyname(service, transport)))
27       sin.sin_port = htons(ntohs((unsigned short)pse->s_port));
28    else if ((sin.sin_port = htons((unsigned short)atoi(service))) == 0)
29       errexit("Can't find \"%s\" service entry\n", service);
30
31    /* Use protocol to choose a socket type */
32    if (strcmp(transport, "udp") == 0)
33       type = SOCK_DGRAM;
34    else
35       type = SOCK_STREAM;
36
37    /* Allocate a socket */
38    s = socket(PF_INET, type, 0);
39    if (s < 0)
40       errexit("Can't create socket: %s\n", strerror(errno));
41
```

```
42      /* Bind the socket */
43      if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
44          errexit("Can't bind to port %s: %s\n",
45                  service, strerror(errno));
46
47      /* Set the maximum number of waiting connection */
48      if (type == SOCK_STREAM && listen(s, qlen) < 0)
49          errexit("Can't listen on port %s: %s\n",
50                  service, strerror(errno));
51
52      return s;
53  }
54
55  /*
56   * connectsock - allocate & connect a socket using TCP or UDP
57   *
58   * Arguments:
59   *    host      - name of host to which connection is desired
60   *    service   - service associated with the desired port
61   *    transport - name of transport protocol to use ("tcp" or "udp")
62   */
63  int connectsock(const char *host, const char *service, const char *transport)
64  {
65      struct hostent *phe; /* pointer to host information entry */
66      struct servent *pse; /* pointer to service information entry */
67      struct sockaddr_in sin; /* an Internet endpoint address */
68      int s, type;      /* socket descriptor and socket type */
69
70      memset(&sin, 0, sizeof(sin));
71      sin.sin_family = AF_INET;
72
73      /* Map service name to port number */
74      if ((pse = getservbyname(service, transport)))
75          sin.sin_port = pse->s_port;
76      else if ((sin.sin_port = htons((unsigned short)atoi(service))) == 0)
77          errexit("Can't get \"%s\" service entry\n", service);
78
79      /* Map host name to IP address, allowing for dotted decimal */
80      if ((phe = gethostbyname(host)))
81          memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
82      else if ((sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE)
83          errexit("Can't get \"%s\" host entry\n", host);
84
85      /* Use protocol to choose a socket type */
86      if (strcmp(transport, "udp") == 0)
87          type = SOCK_DGRAM;
88      else
89          type = SOCK_STREAM;
90
91      /* Allocate a socket */
92      s = socket(PF_INET, type, 0);
```

```
93      if (s < 0)
94          errexit("Can't create socket: %s\n", strerror(errno));
95
96      /* Connect the socket */
97      if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
98          errexit("Can't connect to %s.%s: %s\n", host,
99                  service, strerror(errno));
100
101     return s;
102 }
```

The following program shows the client side sample program (echoc.c).

```
1   /*
2    * echoc.c
3    */
4
5   #include <stdio.h>
6   #include <stdlib.h>
7   #include <string.h>
8   #include <unistd.h>
9
10  #include "sockop.h"
11
12  #define BUFSIZE    1024
13
14  int main(int argc, char *argv[])
15  {
16      int connfd; /* socket descriptor */
17      int n;
18      char buf[BUFSIZE];
19
20      if (argc != 4)
21          errexit("Usage: %s host_address host_port message\n", argv[0]);
22
23      /* create socket and connect to server */
24      connfd = connectsock(argv[1], argv[2], "tcp");
25
26      /* write message to server */
27      if ((n = write(connfd, argv[3], strlen(argv[3]))) == -1)
28          errexit("Error: write()\n");
29
30      /* read message from the server and print */
31      memset(buf, 0, BUFSIZE);
32      if ((n = read(connfd, buf, BUFSIZE)) == -1)
33          errexit("Error: read()\n");
34      printf("%s\n", buf);
35
36      /* close client socket */
37      close(connfd);
38
39      return 0;
```

```
}
```

The server side sample program (echod.c)

```
/*
 * echod.c
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "sockop.h"

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
   int sockfd, connfd;  /* socket descriptor */
   struct sockaddr_in addr_cln;
   socklen_t sLen = sizeof(addr_cln);
   int n;
   char snd[BUFSIZE], rcv[BUFSIZE];

   if (argc != 2)
      errexit("Usage: %s port\n", argv[0]);

   /* create socket and bind socket to port */
   sockfd = passivesock(argv[1], "tcp", 10);

   while(1)
   {
      /* waiting for connection */
      connfd = accept(sockfd, (struct sockaddr *) &addr_cln, &sLen);
      if (connfd == -1)
         errexit("Error: accept()\n");

      /* read message from client */
      if ((n = read(connfd, rcv, BUFSIZE)) == -1)
         errexit("Error: read()\n");

      /* write message back to the client */
      n = sprintf(snd, "Server: %.*s", n, rcv);
      if ((n = write(connfd, snd, n)) == -1)
         errexit("Error: write()\n");

      /* close client connection */
      close(connfd);
   }

   /* close server socket */
```

```
49      close(sockfd);
50
51      return 0;
52  }
```

The above 'echod.c' is the simplest server code that accepts one client request at a time. To concurrently accept multiple client requests, you can either use the '`select()`' system call or create multiple threads/processes to handle the incoming requests.

5 Reference Books

You can obtain more information about socket programming from the following books:

1. Douglas E. Comer, "Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture (4th Edition)," Prentice Hall, 2000.

2. Douglas E. Comer and David L. Stevens, "Internetworking with TCP/IP Vol. II: ANSI C Version: Design, Implementation, and Internals (3rd Edition)," Prentice Hall, 1998.

3. Douglas E. Comer, David L. Stevens and Michael Evangelista, "Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version," Prentice Hall, 2000.

Lab 7 Linux Driver

Wen-Lin Sun

2018-10-19

In last lab, we used PXA270's LCD driver module to control the I/O on it. This time, similarly, we will use a driver to control the I/O, but the driver and the device are created by yourself on ZC702. There are several kinds of devices in Linux, such as block devices, character devices, pipe devices and socket devices. In this lab, we will create our own character devices step by step.

1 Environment Setup

Before we start to build our own devices, make sure you have done the environment setup in Lab2. The way we build up driver modules is just the same as the way we build up the boot image. So now, go to the directory where you build up the boot image (maybe `linux-xlnx`), create a directory to place your driver sources, and check the environment variables (PATH and CROSS_COMPILE) before you start this lab.

2 A Basic Driver

A simple driver can be implemented in two lines.

```
1 #include <linux/module.h>
2 MODULE_LICENSE("GPL");
```

And you can compile the source with the Makefile as following.

```
1 CC=arm-linux-gnueabihf-gcc
2
3 obj-m := mydev.o
4
5 all:
6         make -C ../ M=$(PWD) modules
7 clean:
8         make -C ../ M=$(PWD) clean
```

Now, you can just build the driver with the make tool.

```
SHELL> make ARCH=arm
```

In this example, the source file is mydev.c, and the output file is mydev.ko.

3 Initial/Remove Module

We have created a driver module without any functionality so far. In this section, we are going to add two functions to define the actions that the driver will do on its initialization

and remove.

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

static int my_init(void) {
   printk("call␣init\n");

   return 0;
}

static void my_exit(void) {
   printk("call␣exit\n");
}

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

my_init is the function which will be called when we use insmod to load the driver. Relatively, my_exit is called when we use rmmod to unload the driver. To declare the relationship between the function and the action, the functions module_init and module_exit should be used. So now, after compiling the source, you can test the driver on your board. Try to load and unload it, and check if the behavior is just the same as you defined. By the way, the function printk is the way we use to print out the message in kernel space, instead of printf.

4 Define the Operations

To define the reactions of the actions which may be done on our 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 file in Linux, including the devices.) 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 *);
```

```
16          int (*fsync) (struct file *, loff_t, loff_t, int datasync);
17          int (*aio_fsync) (struct kiocb *, int datasync);
18          int (*fasync) (int, struct file *, int);
19          int (*lock) (struct file *, int, struct file_lock *);
20          ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
                loff_t *, int);
21          unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                unsigned long, unsigned long, unsigned long);
22          int (*check_flags)(int);
23          int (*flock) (struct file *, int, struct file_lock *);
24          ssize_t (*splice_write)(struct pipe_inode_info *, struct file *,
                loff_t *, size_t, unsigned int);
25          ssize_t (*splice_read)(struct file *, loff_t *, struct
                pipe_inode_info *, size_t, unsigned int);
26          int (*setlease)(struct file *, long, struct file_lock **, void **)
                ;
27          long (*fallocate)(struct file *file, int mode, loff_t offset,
28                            loff_t len);
29          void (*show_fdinfo)(struct seq_file *m, struct file *f);
30  #ifndef CONFIG_MMU
31          unsigned (*mmap_capabilities)(struct file *);
32  #endif
33          ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
34                          loff_t, size_t, unsigned int);
35          int (*clone_file_range)(struct file *, loff_t, struct file *,
                loff_t,
36                          u64);
37          ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file
                *,
38                          u64);
39  };
```

We don't have to define all the operations. In this example, we define read, write and open.

```
1   #include <linux/init.h>
2   #include <linux/module.h>
3   #include <linux/kernel.h>
4   #include <linux/fs.h>
5   MODULE_LICENSE("GPL");
6
7   // File Operations
8   static ssize_t my_read(struct file *fp, char *buf, size_t count, loff_t *
        fpos) {
9
10      printk("call␣read\n");
11
12      return count;
13  }
14
15  static ssize_t my_write(struct file *fp,const char *buf, size_t count,
        loff_t *fpos) {
16
```

```
17    printk("call␣write\n");
18
19    return count;
20 }
21
22 static int my_open(struct inode *inode, struct file *fp) {
23
24    printk("call␣open\n");
25
26    return 0;
27 }
28
29 struct file_operations my_fops = {
30    read: my_read,
31    write:   my_write,
32    open: my_open
33 };
34
35 #define MAJOR_NUM 244
36 #define DEVICE_NAME "my_dev"
37
38 static int my_init(void) {
39    printk("call␣init\n");
40    if(register_chrdev(MAJOR_NUM, DEVICE_NAME, &my_fops) < 0) {
41          printk("Can␣not␣get␣major␣%d\n", MAJOR_NUM);
42          return (-EBUSY);
43    }
44
45    printk("My␣device␣is␣started␣and␣the␣major␣is␣%d\n", MAJOR_NUM);
46    return 0;
47 }
48
49 static void my_exit(void) {
50    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
51    printk("call␣exit\n");
52 }
53
54
55 module_init(my_init);
56 module_exit(my_exit);
```

To declare the relationship between, for example, my_read and the read operation, we initialize a structure file_operations called my_fops. So now, we can 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 see your device under /dev, use should load your module first and get the major number. And then you can create the device node with the major number (in this example is 244).

```
SHELL> mknod /dev/mydev c 244 0
```

The c in the command means character device.

So now, you can design and test your own device and its driver!

Lab 8: Signal and Timer

Yu-Lun Huang

2011-07-20

1 Objective

- Be familiar with signal, timer and process reaper.

2 Prerequisite

- 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. (We detail this implementation in later section.)

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 defined 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.

```
SHELL> 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 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 catch program.

```
1  /*
2   * sig_catch.c
3   */
4
5  #include <signal.h>
6  #include <stdio.h>
7  #include <string.h>
8  #include <sys/types.h>
9  #include <unistd.h>
10
11 void handler (int signo, siginfo_t *info, void *context)
12 {
13    /* show the process ID sent signal */
```

```
14      printf ("Process (%d) sent SIGUSR1.n", info->si_pid);
15  }
16
17  int main (int argc, char *argv[])
18  {
19      struct sigaction my_action;
20
21      /* register handler to SIGUSR1 */
22      memset(&my_action, 0, sizeof (struct sigaction));
23      my_action.sa_flags = SA_SIGINFO;
24      my_action.sa_sigaction = handler;
25
26      sigaction(SIGUSR1, &my_action, NULL);
27
28      printf("Process (%d) is catching SIGUSR1 ...\n", getpid());
29      sleep(10);
30      printf("Done.\n");
31
32      return 0;
33  }
```

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.

```
1   /*
2    * sig_count.c
3    */
4
5   #include <signal.h>
6   #include <stdio.h>
7   #include <string.h>
8   #include <sys/types.h>
9   #include <time.h>
10  #include <unistd.h>
11
12  sig_atomic_t sigusr1_count = 0;
13
14  void handler (int signal_number)
```

```
15 {
16    ++sigusr1_count;  /* add one, protected atomic action */
17 }
18
19 int main ()
20 {
21    struct sigaction sa;
22    struct timespec req;
23    int retval;
24
25    /* set the sleep time to 10 sec */
26    memset(&req, 0, sizeof(struct timespec));
27    req.tv_sec = 10;
28    req.tv_nsec = 0;
29
30    /* register handler to SIGUSR1 */
31    memset(&sa, 0, sizeof (sa));
32    sa.sa_handler = handler;
33
34    sigaction (SIGUSR1, &sa, NULL);
35
36    printf("Process (%d) is catching SIGUSR1 ...\n", getpid());
37
38    /* sleep 10 sec */
39    do{
40       retval = nanosleep(&req, &req);
41    } while(retval);
42
43    printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
44
45    return 0;
46 }
```

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 following:

```
int setitimer(int which, const struct itimerval *new_val, struct itimerval *old_val);
```

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 milliseconds and send a `SIGVTALRM` signal.

```
1  /*
2   * timer.c
3   */
4
5  #include <signal.h>
6  #include <stdio.h>
7  #include <string.h>
8  #include <sys/time.h>
9
10 void timer_handler (int signum)
11 {
12     static int count = 0;
13
14     printf ("timer expired %d times\n", ++count);
15 }
16
17 int main (int argc, char **argv)
18 {
19     struct sigaction sa;
20     struct itimerval timer;
21
22     /* Install timer_handler as the signal handler for SIGVTALRM */
23     memset (&sa, 0, sizeof (sa));
24     sa.sa_handler = &timer_handler;
25     sigaction (SIGVTALRM, &sa, NULL);
26
27     /* Configure the timer to expire after 250 msec */
```

```
28      timer.it_value.tv_sec = 0;
29      timer.it_value.tv_usec = 250000;
30
31      /* Reset the timer back to 250 msec after expired */
32      timer.it_interval.tv_sec = 0;
33      timer.it_interval.tv_usec = 250000;
34
35      /* Start a virtual timer */
36      setitimer (ITIMER_VIRTUAL, &timer, NULL);
37
38      /* Do busy work */
39      while (1);
40
41      return 0;
42  }
```

The following program demonstrates the difference among three different 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.

```
1   /*
2    * timer_diff.c
3    */
4
5   #include <fcntl.h>
6   #include <signal.h>
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <string.h>
10  #include <sys/stat.h>
11  #include <sys/time.h>
12  #include <sys/types.h>
13  #include <unistd.h>
14
15  /* counter */
16  int SIGALRM_count = 0;
17  int SIGVTALRM_count = 0;
18  int SIGPROF_count = 0;
19
20  /* handler of SIGALRM */
21  void SIGALRM_handler (int signum)
22  {
23      SIGALRM_count++;
24  }
25
26  /* handler of SIGVTALRM */
27  void SIGVTALRM_handler (int signum)
28  {
29      SIGVTALRM_count++;
30  }
31
```

```
32 /* handler of SIGPROF */
33 void SIGPROF_handler (int signum)
34 {
35    SIGPROF_count++;
36 }
37
38 void IO_WORKS();
39
40 int main (int argc, char **argv)
41 {
42    struct sigaction SA_SIGALRM, SA_SIGVTALRM, SA_SIGPROF;
43    struct itimerval timer;
44
45    /* Install SIGALRM_handler as the signal handler for SIGALRM */
46    memset (&SA_SIGALRM, 0, sizeof (SA_SIGALRM));
47    SA_SIGALRM.sa_handler = &SIGALRM_handler;
48    sigaction (SIGALRM, &SA_SIGALRM, NULL);
49
50    /* Install SIGVTALRM_handler as the signal handler for SIGVTALRM */
51    memset (&SA_SIGVTALRM, 0, sizeof (SA_SIGVTALRM));
52    SA_SIGVTALRM.sa_handler = &SIGVTALRM_handler;
53    sigaction (SIGVTALRM, &SA_SIGVTALRM, NULL);
54
55    /* Install SIGPROF_handler as the signal handler for SIGPROF */
56    memset (&SA_SIGPROF, 0, sizeof (SA_SIGPROF));
57    SA_SIGPROF.sa_handler = &SIGPROF_handler;
58    sigaction (SIGPROF, &SA_SIGPROF, NULL);
59
60    /* Configure the timer to expire after 100 msec */
61    timer.it_value.tv_sec = 0;
62    timer.it_value.tv_usec = 100000;
63
64    /* Reset the timer back to 100 msec after expired */
65    timer.it_interval.tv_sec = 0;
66    timer.it_interval.tv_usec = 100000;
67
68    /* Start timer */
69    setitimer(ITIMER_REAL, &timer, NULL);
70    setitimer(ITIMER_VIRTUAL, &timer, NULL);
71    setitimer(ITIMER_PROF, &timer, NULL);
72
73    /* Do some I/O operations */
74    IO_WORKS();
75
76    printf("SIGALRM_count___=_%d\n", SIGALRM_count);
77    printf("SIGVTALRM_count_=_%d\n", SIGVTALRM_count);
78    printf("SIGPROF_count___=_%d\n", SIGPROF_count);
79
80    return 0;
81 }
82
```

```
83  void IO_WORKS()
84  {
85      int fd, ret;
86      char buffer[100];
87      int i;
88
89      /* Open/Read/Close file 300000 times */
90      for (i = 0; i < 300000; i++) {
91          if ((fd = open("/etc/init.d/networking", O_RDONLY)) < 0) {
92              perror("Open /etc/init.d/networking");
93              exit(EXIT_FAILURE);
94          }
95
96          do {
97              ret = read(fd, buffer, 100);
98          }while(ret);
99
100         close(fd);
101     }
102 }
```

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 Lab 3, 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 `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 `siganl` to register the `reaper` function for `SIGCHLD`. (Please refer the manpage to see the difference between `siganl` 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.

```
1  /*
2   * reaper.c
3   * Demonstrate the work of process reaper
4   */
5
6  #include <signal.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <sys/types.h>
```

```
10 #include <sys/wait.h>
11 #include <time.h>
12 #include <unistd.h>
13
14 #define FORKCHILD 5
15
16 volatile sig_atomic_t reaper_count = 0;
17
18 /* signal handler for SIGCHLD */
19 void Reaper(int sig)
20 {
21     pid_t pid;
22     int status;
23
24     while((pid = waitpid(-1, &status, WNOHANG)) > 0)
25     {
26         printf("Child %d is terminated.\n", pid);
27         reaper_count++;
28     }
29 }
30
31 void ChildProcess()
32 {
33     int rand;
34
35     /* rand a sleep time */
36     srand(time(NULL));
37     rand = random() % 10 + 2;
38
39     printf("Child %d sleep %d sec.\n", getpid(), rand);
40     sleep(rand);
41     printf("Child %d exit.\n", getpid());
42
43     exit(0);
44 }
45
46 int main(int argc, char *argv[])
47 {
48     int cpid;
49     int i;
50
51     /* regist signal handler */
52     signal(SIGCHLD, Reaper);
53
54
55     /* fork child processes */
56     for (i = 0; i < FORKCHILD; i++)
57     {
58         if ( (cpid = fork()) > 0)  /* parent */
59             printf("Parent fork child process %d.\n", cpid);
60         else  /* child */
```

```
61              ChildProcess();
62
63          sleep(1);
64      }
65
66      /* wait all child exit */
67      while(reaper_count != FORKCHILD)
68          sleep(1);
69
70      return 0;
71  }
```