CS356 Operating System Projects Spring 2017

Project 1: Android Process Tree

Objectives:

- Install and use Android Virtual Devices.
- Install NDK, cross compile the program and run it on AVD.
- Effectively use Linux system calls for process control and management.
- Familiarize task_struct
- Concurrent execution of processes.

Make sure your system is 64-bits Linux system.

Problem Statement:

1. Install Android Virtual Device (AVD), and create a new AVD.

   The links of necessary files are given in [1].

   The location of JDK and SDK is up to you. Finally, create the AVD named as "OsPrj-StudentID", make target as "Android 6.0-API Level 23". If your system is 64-bit Linux, [2] should to be considered.

2. Install Android NDK, run HelloWorld in your AVD.

   The link of NDK is given in [1]. You should download it, and extract it to a proper location.

   Add the location of NDK to Environment Variables [4] so that we can use "ndk-build" in other directory.

Make a directory for HelloWorld project and write a “HelloWorld!” program. The files structure should be:

```
-Helloworld
    -JNI
        -HelloWorld.c
        -HelloWorld.h
        -Android.mk
```

The content of Android.mk is like this:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_SRC_FILES := hello.c

LOCAL_MODULE := helloARM

LOCAL_CFLAGS += -pie -fPIE

LOCAL_LDFLAGS += -pie -fPIE

LOCAL_FORCE_STATIC_EXECUTABLE := true

include $(BUILD_EXECUTABLE)
```

More information about Android.mk is in [3].

If you do not familiar with the Android.mk, please do not change everything but project name in the Android.mk

Finally, use adb to debug the HelloWorld on AVD.

3. Write a new system call in Linux

In android system, we can use *ps* to see the information of all process, but we cannot use *pstree* to see the relationship of those process intuitively like what we can do in Linux. So we need a new system call. The system call you write should take two arguments and return the process tree information in a depth-first-search (DFS) order. The prototype for your system call will be:

```
int ptree(struct prinfo *buf, int *nr);
```

You should define struct prinfo as:

```
struct prinfo {
        pid_t parent_pid;       /* process id of parent, set 0 if it has no parent*/
        pid_t pid;              /* process id */
        pid_t first_child_pid;  /* pid of youngest child, set 0 if it has no child */
        pid_t next_sibling_pid;  /* pid of older sibling, set 0 if it has no sibling*/
        long state;             /* current state of process */
        long uid;           /* user id of process owner */
        char comm[64];          /* name of program executed */
    };
```

The argument *buf* points to a buffer for the process data, and *nr* points to the size of this buffer (number of entries). The system call copies as many entries of the process tree data to the buffer as possible, and stores the number of entries actually copied in *nr*.

The original Android kernel does not support module. Therefore, you should use the kernel we supported online [6]. You can learn how to start AVD with a new kernel in [7].

Your system call should return the total number of entries on success (this may be bigger than the actual number of entries copied).

4. **Test your new system call**

   Write a simple C program which calls *ptree*. Your program should print the entire process tree (in DFS order) using tabs to indent children with respect to their parents. The output format should be:

```
    printf(/* correct number of \t */);
```

```
printf("%s,%d,%ld,%d,%d,%d,%d\n", p.comm, p.pid, p.state,     p.parent_pid,
                    p.first_child_pid, p.next_sibling_pid, p.uid);
```

5. Test *ptree*

   Generate a new process, output (“*StudentID*Parent is %d”, pid).

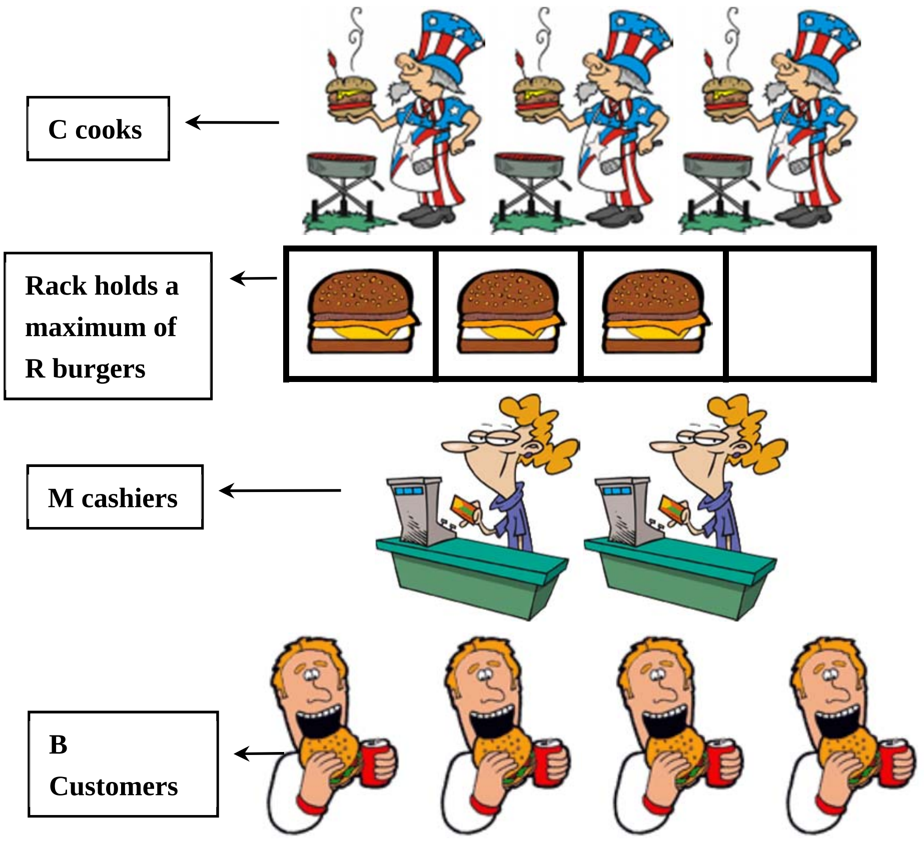
   Then generates its child process, output (“*StudentID*Child is %d”, pid).

   Use *execl* to execute *ptree* in the child process，show the relationship between above two process.

6. **Burger Buddies Problem**: Design, implement and test a solution for the IPC problem specified below. Suppose we have the following scenario:

   - Cooks, Cashiers, and Customers are each modeled as a thread.
   - Cashiers sleep until a customer is present.
   - A Customer approaching a cashier can start the order process.
   - A Customer cannot order until the cashier is ready.
   - Once the order is placed, a cashier has to get a burger from the rack.
   - If a burger is not available, a cashier must wait until one is made.
   - The cook will always make burgers and place them on the rack.
   - The cook will wait if the rack is full.
   - There are NO synchronization constraints for a cashier presenting food to the customer.

   Implement a (concurrent multi-threaded) solution to solve the problem and test it thoroughly. Show output runs that illustrate the various possibilities of the set up.



Implementation Details:

In general, the execution of any of the programs above will is carried out by specifying the executable program name followed by the command line arguments.

1. When using system or library calls you have to make sure that your program will exit gracefully when the requested call cannot be carried out.
2. One of the dangers of learning about forking processes is leaving unwanted processes active and wasting system time. Make sure each process terminates cleanly when processing is completed. Parent process should wait until the child processes complete, print a message and then quit.

3. Your program should be robust. If any of the calls fail, it should print error message and exit with appropriate error code. Always check for failure when invoking a system or library call.

Material to be submitted:

1. Compress the source code of the programs into **Prj1+StudentID.tar** file. It contains all *.c, *.h and Android.mk files. Use meaningful names for the file so that the contents of the file are obvious. Enclose a README file that lists the files you have submitted along with a one sentence explanation. Call it **Prj1README**.
2. Only internal documentation is needed. Please state clearly the purpose of each program at the start of the program. Add comments to explain your program. (-5 points, if insufficient.)
3. Test runs: It is very important that you show that your program works for all possible inputs. Submit online a single typescript file clearly showing the working of all the programs for correct input as well as graceful exit on error input.
4. Send your **Prj1+StudentID.tar** file to cs356.sjtu@gmail.com.
5. Due date: **Apr. 13, 2017**, submit on-line **before midnight**.

Appendix

[1] Official Download

JDK: www.oracle.com/technetwork/java/javase/downloads/index.html

SDK: http://developer.android.com/sdk/index.html#Other

NDK: http://developer.android.com/ndk/downloads/index.html#download

The complete version (no need to download anything from Google) we supplied:

SDK:http://www.cs.sjtu.edu.cn/~fwu/teaching/res/android-sdk-linux.tar.gz

NDK:http://www.cs.sjtu.edu.cn/~fwu/teaching/res/android-ndk-r11-linux-x86_64.zip

[2] If your Linux is 64-bits, execute the following command before setting up your AVD

```
sudo apt-get install libc6:i386 libgcc1:i386 gcc-4.6-base:i386 libstdc++5:i386 libstdc++6:i386
```

[3] http://developer.android.com/ndk/guides/android_mk.html

[4] How to add location to Environment Variables.

Add following sentence in ~/.bashrc(for common user) or /etc/profile(for root user):

```
export PATH=#l absolute path you want#:$PATH
```

Then type source ~/.bashrc or source /etc/profile in terminal.

DO NOT change any other value in Environment Variables

[5] Some useful adb command:

To check the AVD status:

```
adb devices
```