Yufei Lin

Final Project

Jun 7^{th} 2019

ASTRI Project

Plan of Project & Summary from Books

Contents

I. Plan

I.1. Goal

I am going to build a Blockchain Operating System(BOS) that can be migrated to ARM structure in a Arduino Uno machine in order to study applications of BOS in full-filling Internet of Things(IOT).

I.2. Resources(Need Amendment)

Listed below are the resource I may potentially need: Will be specified once start on the project

1. Help in IOT
2. Help in Arduino

I.3. Action Plan

The above schedule is a tentative schedule and my time leaving from Hong Kong is also not determined yet.

Tasks	Time Interval	Notes
Read Books	Jun 6th - 11th	May need more time
Design & Build BOS	Jun 12^{th} - 28^{th}	
Test BOS	Jun 29^{th} - July 3^{rd}	
Intro to Arduino Projects	July 4^{th} - 9^{th}	
Migration of Operating System	July 10^{th} – 11^{th}	
Final test on a model of IOT	July 11^{th} – 21^{th}	
Final adjustment & Write documentations and papers	July 22^{th} – 31^{st}	

II. Summary from *Linux From Scratch*

II.1. How to Build a LFS

II.1.1 What do we need for a LFS

1. An already built Linux distribution, because we need the compiler, linker and shell in it
2. A file system that can compile and install the new LFS system and necessary packages and patches
3. An appropriate working environment

4. Necessary packages needed to create a basic environment suite, also named as a toolchain

II.1.1. Prepare the Host System

II.1.1.1. Create a Partition

Minimal requirement of a partition is 6 GB. From there I see why we need a Plan9 because it could be built smaller.

Quote from the book:"Because there is not always enough Random Access Memory (RAM) available for compilation processes, it is a good idea to use a small disk partition as swap space."
I don't quite understand what does it mean by there may not always be enough RAM for processing. Also, how do we partition a RAM out from an existing space.

II.1.1.2. Create a File System on the Partition

We dominantly use type ext3 and ext4 file system from Linux in LFS. The command that creates a file system is:

$$mkfs - v - text4/dev/ < xxx >$$

II.1.1.2. Setting The $LFS Variable

The variable name LFS should be defined throughout the LFS build process. It should be set to the name of the directory where you will be building your LFS system - we will use $/mnt/lfs$ as an example, but the directory choice is up to you. Choose a directory would involve the following command:

$$\text{export } LFS = /mnt/lfs$$

II.1.1.3. Mounting the New Partition

The mount point of this book is under the directory specified by the LFS variable. The following is the example code:
mkdir -pv $$LFS$$
mount -v -t ext4 $/dev/ < xxx >$ $$LFS$$

II.1.2. Packages and Patches

The following is the things we need to do before actually download packages and patches:

1. Create a source directory with the following command: mkdir -v $LFS/sources

2. Make the directory sticky - means only owner can delete files in the directory, and writable with the following command: chmod -v a+wt $LFS/sources

3. Download patches: wget –input-file=wget-list –continue –directory-prefix=$LFS/sources

II.1.3. Final Preparations

II.1.3.1. Creating the $LFS/tools Directory

We need a separate folder to store all necessary tools that will be installed later for the system. Therefore, we create this tools directory.

II.1.3.2. Adding the LFS User

In this section we are going to add a user account to the system so that we won't risk ourselves damage our system by using root account. Then we use the following code to create a full accessible account:

```
groupadd lfs
useradd -s /bin/bash -g lfs -m -k /dev/null lfs
passwd lfs
chown -v lfs $LFS/tools
chown -v lfs $LFS/sources
su - lfs
```

II.1.3.3. Setting Up the Environment

Set up a good working environment by creating two new start up files for the bash shell. The code will be the same from the book therefore, I am not going to show them here.

II.1.4. Constructing a Temporary System

General Compilation Instructions

In terms of compilation: Use echo $LFS is set properly. Also, double check the patches are installed on the packages before installation. Furthermore, if during installation, errors occur, most of them can be ignored.

II.2. Install and Boot a LFS system

1. Set up a new shell

2. Set up basic configurations

3. Set up kernel and boot loader

4. Reboot the computer

II.2.1. Installing Basic System Software

II.2.1.1. Preparing Virtual Kernel File Systems

1. Creating initial device nodes
2. Mounting and populating /dev
3. Mounting virtual kernel file systems

II.2.1.2. Package Management

Package Management is an often requested addition to the LFS Book. A Package Manager allows tracking the installation of files making it easy to remove and upgrade packages. As well as the binary and library files, a package manager will handle the installation of configuration files.
The following are the necessary techniques needed in package management.

1. Install in separate directories
2. Symlink style package management
3. Timestamp based package management
4. Tracing installation scripts
5. Creating package archives
6. User based management

II.2.1.3. Entering the Chroot Environment

II.2.1.4. Creating Directories

II.2.1.5. Creating Essential Files and Symlinks

II.2.2. System Configuration

The process must mount both virtual and real file systems, initialize devices, activate swap, check file systems for integrity, mount any swap partitions or files, set the system clock, bring up networking, start any daemons required by the system, and accomplish any other custom tasks needed by the user.

II.2.3. Making the LFS System Bootable

1. Create the /etc/fstab file
2. Use Linux 4.20.12
3. Use GRUB to set up the boot process

III. Summary from *Plan9-intro*

III.1. Operating System Concept

An operating system is also known as a kernel, a virtual machine, or a resource manager. It must be used as a way that can be installed on any possible machine and is able to be compatible for all versions of hardware.
In an operating system, we must be able to running and editing commands, using files, manipulating directories and controlling permissions.

III.2. Programs and Processes

III.2.1. Processes

A running program is called a process. On the other hand, a program is just a bunch of data that is not alive, meaning no interaction between users and programs.
A process is not called, but executed. It also does not have a return state in Plan 9, but would terminate when it has to or misbehaved.
The following picture shows the entire procedure for running a process:

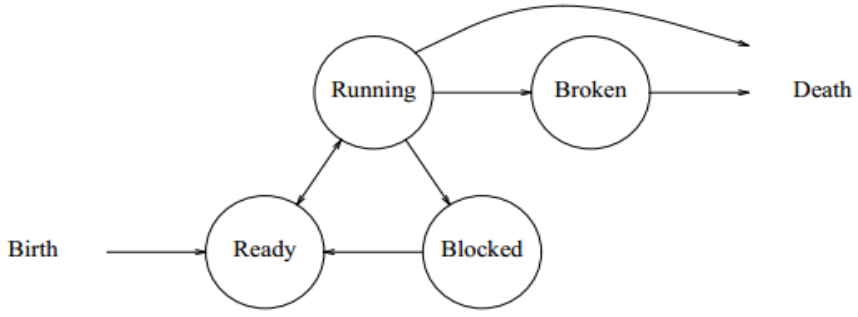

Figure 1: Process states and transitions between them

III.2.1.1. Process names and states

1. Each process is given a unique number by the system when it is created, which is named as process id or pid.
2. Process state indicates what the process is doing.

3. Each process is given a small amount of processor time, named quantum, and later, the system decides to jump to another one.

4. The state for a process is called its context.

5. A transfer of control from one process to another, by saving the state for the old process and reloading the state for the new one, is called a context switch.

6. The part of the kernel deciding which process runs each time is called the scheduler.

7. The decisions made by the scheduler to multiplex the processor among processes are collectively known as scheduling.

III.2.2. Loaded Prgrams

When a program in source form is compiled and linked, a binary file is generated. This file keeps all the information needed to execute the program. The system would load a program by reading the information kept in binary with a loader. A loader is specified as follows:

- The header in the binary file reports the memory size required for the program text, and the file keeps the memory image of that text. Therefore, the system can just copy all this into memory. For a given system and architecture, there is a convention regarding which addresses the program must use. Therefore, the system knows where to load the program.

- The header in the binary reports the memory size required for initialized variables (globals) and the file contains a memory image for them. Thus, the system can copy those bytes to memory. Note that the system has no idea regarding where does one variable start or how big it is. The system only knows how many bytes it has to copy to memory, and at which address should they be copied.

- For uninitialized global variables, the binary header reports their total size. The system allocates that amount of memory for the program. That is all it has to do. As a courtesy, Plan 9 guarantees that such memory is initialized with all bytes being zero. This means that all your global variables are initialized to null values by default. That is a good thing, because most programs will misbehave if variables are not properly initialized, and null values for variables seem to be a nice initial value by default.

III.2.3. Environment

Environment variable is a way to supply arguments to a process. All environment variables in a process are defined as a set of *name* = *value* strings. Usually, all processes running in

the same window share the environment variables.

III.2.4. Everything is a File

For most abstractions provided by Plan 9, to let you use your hardware, a file interface is provided. This means that the system lies to you, and makes you believe that many things, that of course are not, are files. The point is that they appear to be files, so that you can use them as if that was really the case.

III.3. Files

The abstractions provided by Plan 9 can be used through a file interface. If you know how to use the file interface, you also know how to use the interface for most of the abstractions that Plan 9 provides.

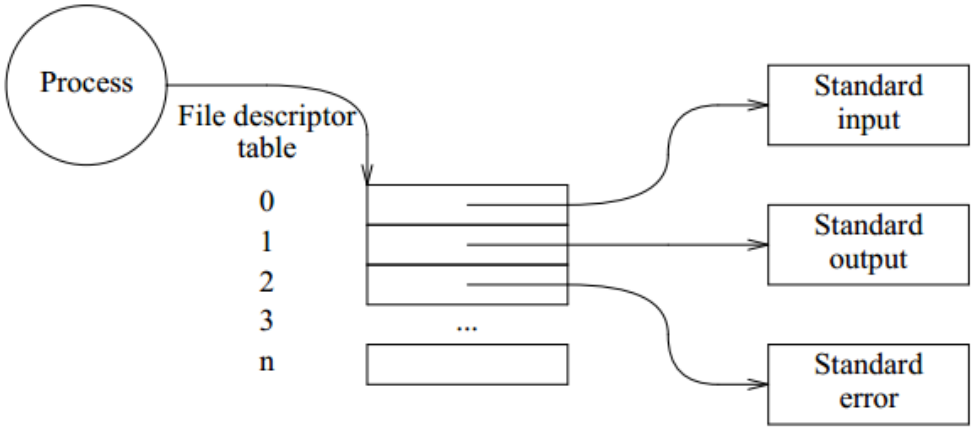

Figure 2: File Descriptor Process Files

III.4. Other Important Components of Plan

All the followings are important components but not necessary ones in an operating system. When I build my BOS, I will consider which ones are important and which ones are not in order to make my OS more flexible and optimal.

- Parent and Child of a program
- Communicating Processes
- Network Communications
- Resource, Files, and Names
- Using the Shell

- Concurrent Programming
- Threads and Channels
- User Input/Output
- Build a File Server
- Security
- Tools include: Regular expression, Sorting and searching, Searching for changes, AWK, Processing data, File systems

IV. Summary from *Plan9 Kernel Source-Notes*

Most chapters are introducing similar concepts from the previous book. Therefore, I am not going to put them down in this summary.

IV.I. Memory Management

Plan 9 uses paged virtual memory. Usually users would see a segmentation of memory like the following:

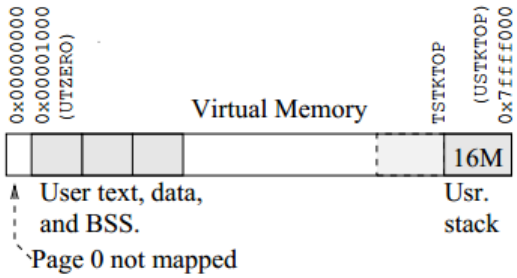

Figure 3: User View of a Memory

IV.I.1. Processes and Segments

IV.I.1.1. Create New Segments

In order for a program to run well, we need to create new segment for it to run on. The procedure creates a segment of a given type, base and length. It aborts if the size is beyond the maximum size allowed for a segmentsize is in pages, as segments must contain an integral number of virtual memory pages because the paging hardware is used to implement them.

Similarly, we need to create new text segments for texts such that it will be easier for users to access texts.

IV.I.2. Page allocation and paging

IV.I.2.1. Allocation and caching

Segments are filled up with pages on demand. The following graph indicates how page frames allocated when segments reclaim more memory.

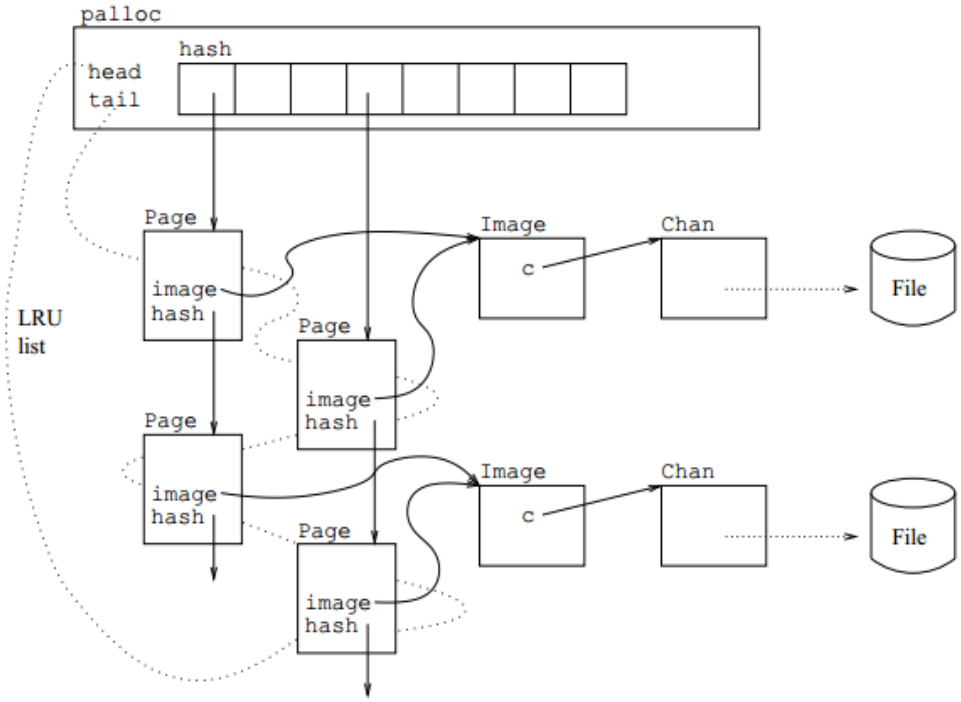

Figure 4: User View of a Memory

IV.I.2.2. Terminating Segments

Segments are usually released by putseg, however, there is another routine (the one called by killbig, which you already saw) that is called to release memory held by a segment. It is also used by a couple other routines besides killbig.

V. Additional Websites and Information

1. An interesting website for learning Plan9 working with Golang:https://www.youtube.com/watch?v=dPdXxex1v_4