CSE 506: Operating Systems

Project Assignments
Warm-up Project (Part #1 of 3)

- Implement kernel printf()
  - Must support at least %c, %d, %x, %s, %p
  - Should write to the console
    - For fun, you can also support writing to the serial port

- Why?
  - Because every OS needs one, at least for debugging

- What do you get?
  - “hardware”
  - A boot loader
  - A Makefile
Warm-up Project (Part #2 of 3)

• Implement timer ISR
  – Keep track of time since boot
  – Display time since boot in lower-right corner
    • For fun, you can also read RTC to show real-world time

• Why?
  – OS needs to handle interrupts, timer is the easiest one

• What do you get?
  – Your own Part #1
Warm-up Project (Part #3 of 3)

• Implement keyboard ISR
  – React to key presses
  – Display the last pressed glyph next to the clock
    • Don’t forget to handle the Shfit key
    • For fun, include handling for Control characters (show as ^C)

• Why?
  – Output is good, but every OS needs input too

• What do you get?
  – Your own Part #1
  – Code that sets up the GDT for you
Course Project Overview

• This is an OS class
  – You will build an OS - \textit{SBUnix}

• If you missed first class or forgot what was there
  – Revisit Grading Policy in Intro. Lecture

• Milestones
  – Do you need some? If so, send email!

• These slides are “minimum” requirements
  – Can always learn more by doing more
  – Many will try to emulate an existing system
    • It’s OK, but not necessary – be creative!
Points

<table>
<thead>
<tr>
<th>Course Project</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cooperative OS</td>
<td>50</td>
</tr>
<tr>
<td>Preemptive OS</td>
<td>60</td>
</tr>
<tr>
<td>Preemptive OS w/ File System</td>
<td>70</td>
</tr>
<tr>
<td>Preemptive OS w/ File System and Network</td>
<td>80</td>
</tr>
<tr>
<td>Multi-processor OS w/ File System and Network</td>
<td>90</td>
</tr>
<tr>
<td>Multi-processor OS w/ File System and Network and Thread Support</td>
<td>100</td>
</tr>
</tbody>
</table>

- Group of 2 / no experience in systems
  - Go for 60 point project
- Group of 2 + experience, Group of 4 / no experience
  - Go for 80 point project
- 90 point project: Systems PhDs, MS super hackers
- 100 point project: You know who you are
Functional Requirements

- Virtual memory, ring 3 processes
  - malloc(), COW fork() (w/per-user limit)
  - auto-growing stack (w/per-process limit)
  - SEGV handling
- tarfs
  - open, read, close, opendir, readdir, closedir
- stdin, stdout, stderr
  - read() and write()
- Binaries: ls, ps, sleep, sh
- Shell with PATH, “cd”, “ulimit”, and “&” available
  - exec() (ELF or #!), getpid()
API Requirements

• Provide libc with at least the basic implementation of
  – malloc
  – fork, execvpe, wait, waitpid, exit, getpid
  – open, close, read
  – opendir, readdir, closedir
  – sleep
  – printf, scanf
  – w/File System: seek, write, mmap
  – w/Network: socket, bind, connect, listen, accept
  – w/Threads: pthread_create, pthread_join
Handout Directory

bin/
crt/
cse506-pubkey.txt
include/
    sys/
ld/
libc/
LICENSE
linker.script
Makefile
rootfs/
    bin/
    boot/
    etc/
    lib/
sys/
bin/hello/hello.c

#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("Hello World!\n");
    return 0;
}

Linked as: -o hello crt1.o hello.o libc.a
```c
#include <stdlib.h>
void _start(void) {
    int argc = 1;
    char* argv[10];
    char* envp[10];
    int res;
    res = main(argc, argv, envp);
    exit(res);
}
```
libc/

exit.c:
    void exit(int status) {
    }

printf.c:
    int printf(const char *format, ...) {
        return 0;
    }
linker.script and Makefile

ENTRY(boot)
SECTION
{
    physbase = 0x200000;
kernmem = 0xffffffff80000000 + physbase;
    . = kernmem + SIZEOF_HEADERS;
    .text : { *(.text) }
    .rodata : { *(.rodata) }
    .got ALIGN(0x1000): { *(.got) *(.got.plt) }
    .bss ALIGN(0x1000): { *(.bss) *(COMMON) }
    .data : { *(.data) }
}

• Do not edit
• If you *need* to edit
    – Ask on the mailing list, will be changed in handout
rootfs/

bin/
boot/
etc/
lib/

Note: \texttt{bin/} and \texttt{lib/} wiped on "\texttt{make clean}"
libc/ and sys/

• Most of your code will go here
• Create subdirectories as needed
• libc/ will be linked into all bin/ executables
  – Should not be linked against kernel
• sys/ should contain all kernel code
  – Should not rely on libc
  – `start()` should never return
#define INITIAL_STACK_SIZE 4096
char stack[INITIAL_STACK_SIZE];
uint32_t* loader_stack;
extern char kernmem, physbase;

void boot(void) {
    // note: function changes rsp, local stack variables can't be practically used
    register char *temp1, *temp2;
    __asm__ (""
                "movq %rsp, %0;"
                "movq %1, %rsp;"
        :
                '=g'(loader_stack)
        :
                'r'(&stack[INITIAL_STACK_SIZE])
        );
    reload_gdt();
    setup_tss();
    start(
        (char*)(uint64_t)loader_stack[3] + (uint64_t)&kernmem - (uint64_t)&physbase,
        &physbase,
        (void*)(uint64_t)loader_stack[4]
    );
    for(
        temp1 = "!!!!! start() returned !!!!!!", temp2 = (char*)0xb8000;
        *temp1;
        temp1 += 1, temp2 += 2
    ) *temp2 = *temp1;
    while(1);
}
Loader Environment

• 64-bit x86
  – No segmentation, paging enabled

• Boot-time page tables set up by loader (1GB regions)
  – $V:\ 00000000:3ffffff \rightarrow P:\ 00000000:3ffffff$
  – $V:\ fffffffff80000000:ffffffffbfffffff \rightarrow P:\ 00000000:3ffffff$

• $\text{physbase}$ is where kernel starts: $0x200000$

• $\text{physfree}$ is where kernel ends

• $\text{modulep}$ includes e820 info
  – Lists the system physical address ranges
tarfs

• Hack to have some (small) files without disk drivers
  – Most of you will be implementing real disk drivers
    • Later

• Kernel’s ELF headers tell loader what to load where
  – Have section for code (text), rodata, bss, ...
  – Loader faithfully loads these from disk based on headers
  – We add an extra section containing our (small) files
    • It’s actually part of the kernel binary
    • Gets loaded along with the kernel

• Similar (in purpose) to initrd in Linux
  – Provides a filesystem before there is a real filesystem
Accessing tarfs

• Filesystem contents loaded by loader into memory
  – How to find where it is?
    • Starts at _binary_tarfs_start
      – Fake symbol (variable) created using objcopy command
      – Use its address (&) to find start of tarfs
    • Ends at _binary_tarfs_end

• Use #include <sys/tarfs.h>

• Defines a struct for the filesystem (tar) format
  – Contains name and size members
  – If name doesn’t match what you’re looking for
    • Skip size bytes and try again