Engineering & Computer Science
Syracuse University

CIS 657 – Principles of Operating Systems

Topic: Introduction to Operating Systems

Endadul Hoque

Acknowledgement

- Youjip Won (Hanyang University)
- OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)

What happens when a program runs?

- A running program executes instructions.
  1. The processor **fetches** an instruction from memory.
  2. **Decode**: Figure out which instruction this is
  3. **Execute**: i.e., add two numbers, access memory, check a condition, jump to function, and so forth.
  4. The processor moves on to the **next instruction** and so on.

Von Neumann model of computing

An Operating System (OS)

- Responsible for
  - Making it easy to **run** programs
  - Allowing programs to **share** memory
  - Enabling programs to **interact** with devices

OS is in charge of making sure the system operates **correctly** and **efficiently**.

How does OS ensure this?

VIRTUALIZATION

Virtualization

- The OS takes a physical resource and transforms it into a virtual form of itself.
  - **Physical resource**: Processor, Memory, Disk …
- The virtual form is more general, powerful and easy-to-use.
- Sometimes, we refer to the OS as a **virtual machine**.

System Calls

- System call allows users **to tell the OS what to do**.
  - The OS provides some interface (APIs, standard library).
  - A typical OS exports a few hundred system calls.
    - Run programs
    - Access memory
    - Access devices

The OS as a resource manager

- The OS **manage resources** such as *CPU*, *memory* and *disk*.
- The OS allows
  - Many programs to run → Sharing the CPU
  - Many programs to *concurrently* access their own instructions and data → Sharing memory
  - Many programs to access devices → Sharing disks

Virtualizing the CPU

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: cpu <string>\n");
                exit(1);
        }
        char *str = argv[1];
        while (1) {
                Spin(1); // Repeatedly checks the time and
                         returns once it has run for a second
                printf("%s\n", str);
        }
        return 0;
}
```

Simple Example(cpu.c): Code That Loops and Prints

Virtualizing the CPU

- Execution result 1

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
ˆC
prompt>
```

Run forever; Only by pressing “Control-c” we can halt the program

Virtualizing the CPU

- Execution result 2

```
prompt> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
...
```

Even though we have only **one processor**, all four of programs seem to be running **at the same time**!

Virtualizing the CPU

- The system has a very large number of virtual CPUs.
  - Turning a single CPU into a seemingly infinite number of CPUs.
  - Allowing many programs to seemingly run at once
    → **Virtualizing the CPU**

Virtualizing Memory

- The physical memory is *an array of bytes*.
- A program keeps all of its data structures in memory.
  - **Read memory** (load):
    - Specify an address to be able to access the data
  - **Write memory** (store):
    - Specify the data to be written to the given address

Virtualizing Memory

- A program that Accesses Memory (mem.c)

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));  // a1: allocate some memory
    assert(p != NULL);
    printf("(%d) address of p: %08x\n",
           getpid(), (unsigned) p); // a2: print out the
                                       address of the memory
    *p = 0; // a3: put zero into the first slot of the memory
    while (1) {
      Spin(1);
     *p = *p + 1;
     printf("(%d) p: %d\n", getpid(), *p); // a4
    }
    return 0;
}
```

Virtualizing Memory

- The output of the program `mem.c`

```
prompt> ./mem
(2134) memory address of p: 00200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
^C
```

  - The newly allocated memory is at address `00200000`.
  - It updates the value and prints out the result.

Virtualizing Memory

- Running
  `mem.c`
  multiple times

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) memory address of p: 00200000
(24114) memory address of p: 00200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
...
```

- As if each running program has its **own private memory**.
  – Each running program has allocated memory at the same address.
  – Each seems to be updating the value at `00200000` independently.

Virtualizing Memory

- Each process accesses its own private **virtual address space**.
  - The OS maps address space onto the physical memory.
  - A memory reference within one running program does not affect the address space of other processes.
  - Physical memory is a **shared resource**, managed by the OS.

CONCURRENCY

The Problem of Concurrency

- Where does it stem from?

- The OS is juggling many things at once, first running one process, then another, and so forth.

- Modern multi-threaded programs also exhibit the concurrency problem.

The Problem of Concurrency

- A multi-threaded program (`thread.c`)

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}
...
```

The Problem of Concurrency

- A multi-threaded program (`thread.c`)

```
int
main(int argc, char *argv[])
{
        if (argc != 2) {
                fprintf(stderr, "usage: threads <value>\n");
                exit(1);
        }
        loops = atoi(argv[1]);
        pthread_t p1, p2;
        printf("Initial value : %d\n", counter);

        Pthread_create(&p1, NULL, worker, NULL);
        Pthread_create(&p2, NULL, worker, NULL);
        Pthread_join(p1, NULL);
        Pthread_join(p2, NULL);
        printf("Final value : %d\n", counter);
        return 0;
}
```

The Problem of Concurrency

- A multi-threaded program (`thread.c`)

```c
int
main(int argc, char *argv[])
        pthread_t p1, p2;
        printf("Initial value : %d\n", counter);

        Pthread_create(&p1, NULL, worker, NULL);
        Pthread_create(&p2, NULL, worker, NULL);
        Pthread_join(p1, NULL);
        Pthread_join(p2, NULL);
        printf("Final value : %d\n", counter);
        return 0;
}
```

The main program creates **two threads**.

- <u>Thread</u>: runs within the same memory space. Each thread starts running in a function called **`worker`**`()`.
- **`worker`**`()`: increments a counter

The Problem of Concurrency

- `loops` determines how many times each of the two workers will **increment the shared counter** in a loop.
  - `loops: 1000.`

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000
```

  - `loops: 100000.`

```
prompt> ./thread 100000
Initial value : 0
Final value : 143012 // huh??
prompt> ./thread 100000
Initial value : 0
Final value : 137298 // what the??
```

The Problem of Concurrency

- **Why is this happening?**

- Increment a shared counter → take three instructions.
  1. Load the value of the counter from memory into register.
  2. Increment it
  3. Store it back into memory

- These three instructions do not execute atomically. → Problem of **concurrency** happen.

PERSISTENCE

Persistence

- Devices such as DRAM store values in a volatile.
- *Hardware* and *software* are needed to store data persistently.
  - **Hardware**: I/O device such as a hard drive, solid-state drives(SSDs)
  - **Software**:
    - File system manages the disk.
    - File system is responsible for storing any files the user creates.

Persistence

- Create a file (`/tmp/file`) that contains the string “hello world”

```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int
main(int argc, char *argv[])
{
        int fd = open("/tmp/file", O_WRONLY | O_CREAT
                        | O_TRUNC, S_IRWXU);
        assert(fd > -1);
        int rc = write(fd, "hello world\n", 13);
        assert(rc == 13);
        close(fd);
        return 0;
}
```

`open`(), **`write`**(), and **`close`**() system calls are routed to the part of OS called the file system, which handles the requests

Persistence

- What OS does in order to write to disk?
  - Figure out **where** on disk this new data will reside
  - **Issue I/O** requests to the underlying storage device

- File system handles system crashes during write.
  - **Journaling** or **copy-on-write**
  - Carefully <u>ordering</u> writes to disk

DESIGN GOALS OF AN OS

Design Goals

- Build up **abstraction**
  - Make the system convenient and easy to use.

- Provide high **performance**
  - Minimize the overhead of the OS.
  - OS must strive to provide virtualization <u>without excessive overhead</u>.

- **Protection** between applications
  - <u>Isolation</u>: Bad behavior of one does not harm other and the OS itself.

Design Goals

- High degree of **reliability**
  - The OS must also run non-stop.

- Other issues
  - Energy-efficiency
  - Security
  - Mobility

Reading Material

- **Chapter 2** of OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin) http://pages.cs.wisc.edu/~remzi/OSTEP/intro.pdf

Questions?