Operating System: Introduction

Sang Ho Choi (shchoi@kw.ac.kr)
School of Computer & Information Engineering
KwangWoon University

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

PC (program counter)

Operating System (OS)

OS의 역할

- 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

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

한정된 자원을 가상화된 형태로 만듦으로서
손쉽게 사용하도록 만드는것.

System call

- System call allows user **to tell the OS what to do** OS가 어떤 일을 하도록 만드는 함수
  - 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 is a resource manager

- The OS **manage resources** such as *CPU, memory* and *disk*   OS를 리소스 매니저라고 표현하기도 함.
- 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

다수의 프로그램이 CPU, memory, disk를 어떻게
나눌지에 대한 알고리즘이 포함

Virtualizing the CPU

- The system has a very large number of virtual CPUs
  - Turning a single CPU into a <u>seemingly infinite number</u> of CPUs
  - Allowing many programs to <u>seemingly run at once</u>

    → **Virtualizing the CPU**

ex) CPU는 4개일때 다수의 프로그램이 이걸 다 써야하니까

1:1 매칭이 되어있는 것처럼 보이게 함 (가상화)

Virtualizing the CPU (Cont.)

```c
#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 (Cont.)

- Execution result 1

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

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

Virtualizing the CPU (Cont.)

- 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
C
B
D
...
```

동시에 실행되는 것처럼 보이는 효과.

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

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 (Cont.)

- A program that Accesses Memory (`mem.c`)

```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 memmory
        *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 (Cont.)

- The output of the program `mem.c`

```
prompt> ./mem
(2134) 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 (Cont.)

- 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
...
```

  – It is as if each running program has its **own private memory**
    - ➢ Each running program has allocated memory at <u>the same address</u>
    - ➢ Each seems to be updating the value at `00200000` independently

Virtualizing Memory (Cont.)

- 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 프로그램간 주소공간에 영향을 X.
  - Physical memory is a shared resource, managed by the OS

The problem of Concurrency

- 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

Concurrency Example

- 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;
}
...
```

Concurrency Example (Cont.)

```
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 main program creates **two threads**
  - Thread: a function running within the same memory space. Each thread start running in a routine called `worker()`
  - `worker()`: increments a counter

Concurrency Example (Cont.)

- `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

200000이 안나왔다.

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

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

원자적으로

세개의 명령어가 모두 실행되기 전에 다른 프로그램의 명령어도 실행됨

race condition

Persistence

휘발성

- 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 (Cont.)

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

```c
#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.

데이터가 소멸되지 않고 지속적으로 존재할 수 있게끔 하는 OS의 tool이 존재한다!

Persistence (Cont.)

- 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** — 디스크에 저장할때 사용하는 기법.
  - Carefully <u>ordering</u> writes to disk

Design Goals 운영체제 설계 목표.

- Build up **abstraction** 사용하기 쉬워야 함.
  - Make the system convenient and easy to use
- Provide high **performance** 성능이 높아야 함.
  - Minimize the overhead of the OS
  - OS must strive to provide virtualization without excessive overhead
- **Protection** between applications 프로그램간 간섭 X
  - Isolation: Bad behavior of one does not harm other and the OS itself

Design Goals (Cont.)

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

신뢰성 ↑

계속작동되야함

- Other issues
  - Energy-efficiency
  - Security
  - Mobility

부가적인 추가요소.

중요적인 부분은

OS의 virtualization, concurrently, persistence.