CDA0017: Operating Systems

Donghyun Kang (donghyun@changwon.ac.kr)

NOSLab (https://noslab.github.io)

Changwon National University

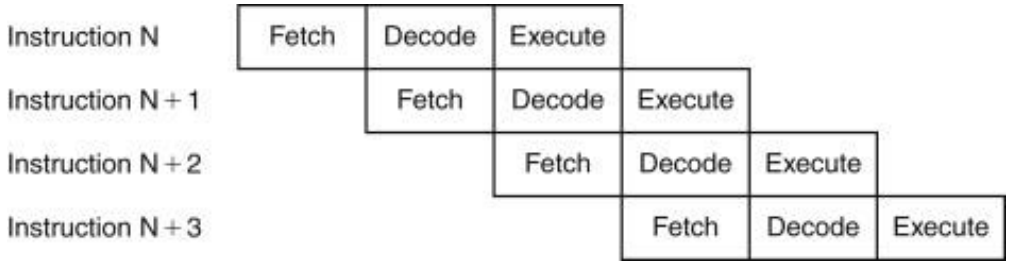
어플리케이션 프로그램은 어떻게 만들어 지는가?

- High-level programming language기반 프로그램 생성
  - C, Java, Python, etc.

- Compile 수행
  - Executable file
    - CPU architecture 기반 명령어 (Instruction)
    - Program data

프로그램이 실행될 때 어떤 일이 일어날까?

- 반입 (Fetch)
  - Memory → CPU register 로 명령어 반입
- 해석 (Decode)
  - CPU register에 저장된 명령어 해석
    - E.g., ADD, AND, BSWAP, etc.
- 실행 (Execute)
  - 해석된 명령어 동작 수행



운영체제란?

- 어플리케이션 프로그램을 실행시키기 위한 소프트웨어
  - CPU, 메모리, 디스크 등의 시스템의 자원 관리

Operating Systems (운영체제)

운영체제의 역사

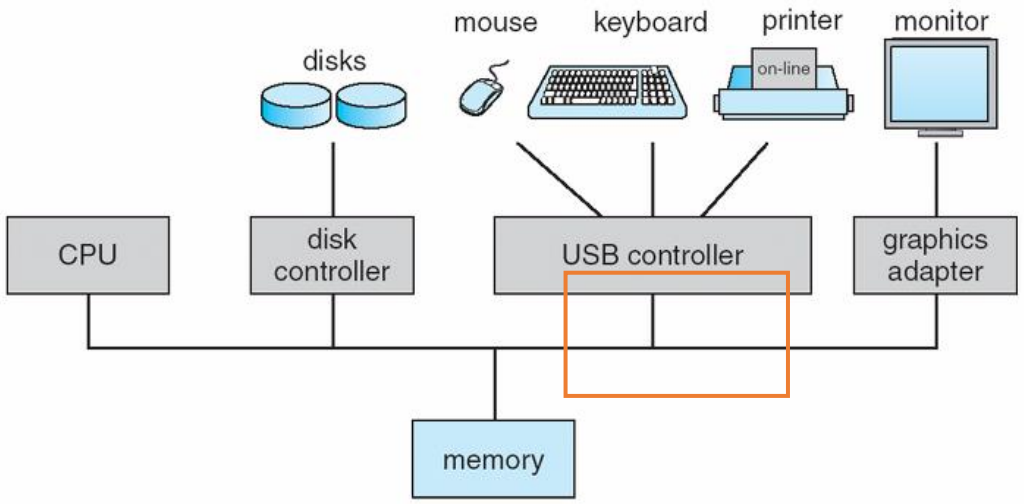
- 초창기 운영체제 : 단순 라이브러리
  - 자주 사용되는 함수들을 모아 놓은 라이브러리
  - 옛날 메인프레임 시스템에서는 컴퓨터 관리자 (사람)이 하나씩 프로그램 수행
    - 일괄 (batch) 처리 기반 사람 스케줄링
- 라이브러리를 넘어서 : 보호
  - 운영체제 코드는 ***장치***를 제어하기 때문에 일반 응용 프로그램 코드와 다르게 취급
  - 장치 제어를 위해 시스템 콜 (system call) 사용
    - 정해진 규칙에 따라 제어 가능한 과정을 거치도록 강제함
  - **사용자 모드**, **커널 모드**로 시스템 보호

운영체제의 역사

- 멀티프로그래밍 시대
  - 미니컴퓨터(minicomputer)시대의 시작
  - 컴퓨터 가격의 하락으로 개발자들의 활동이 활발해짐
  - **멀티프로그래밍**(multiprogramming)기법 사용
    - 운영체제는 여러 작업을 메모리에 탑재하고 작업들을 빠르게 번갈아 가며 실행하여 CPU 사용률을 향상시킴
      - 입출력 요청이 서비스 되고 있는 동안 CPU가 대기하는 것은 CPU 시간 낭비를 초래함

운영체제의 역사

- 현대
  - 개인용 컴퓨터(personal computer) 또는 PC라고 불리는 컴퓨터 시대의 시작
    - 하나 이상의 CPU와 다수의 장치 디바이스로 구성



보조 교재 ★

Unix/Linux 계열 운영체제

Linux distributions

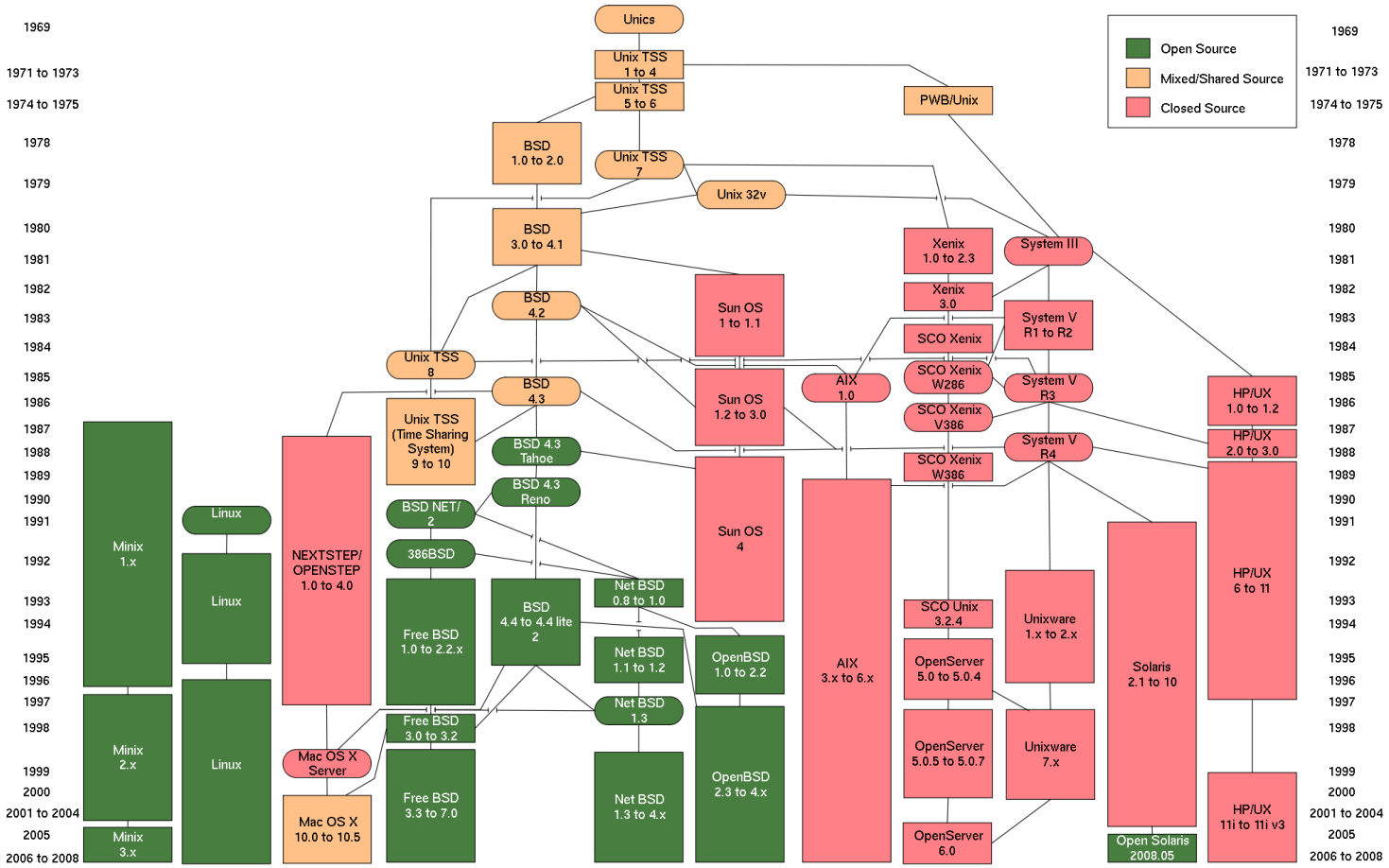
Unix/Linux are eating the world

[Ref: https://en.wikipedia.org/wiki/Usage_share_of_operating_systems]

- 99.8% of smartphones run Unix/Linux
- 99.6% of tablets run Unix/Linux
- 67.8% of public servers run Unix/Linux
- 100% of supercomputers run Unix/Linux
- 81.7% of desktop and laptop computers run Windows
- 95.8% of desktop games run Windows

Version tree of Unix

[Ref: https://en.wikipedia.org/wiki/History_of_Unix]



운영체제의 역할

- CPU 관리
  - 프로그램 실행을 위한 CPU register 초기화 (e.g., program counter)
- Memory 관리
  - 프로그램 code & data 을 Disk에서 memory로 load
- External devices 관리
  - Character / block device의 read & write 명령

가상화 기반 자원 (resource) 관리

CPU 가상화

- 매우 많은 수의 가상 CPU가 존재하는 듯한 환상(illusion)을 제공함
  - 즉, 프로세스 (Process)가 하나의 CPU (Processor)을 개별적으로 소유한다고 착각함
- 프로그램 예제 (단일 어플리케이션 실행)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <assert.h>
5  #include "common.h"
6  int
7  main(int argc, char *argv[])
8  {
9      if (argc != 2) {
10         fprintf(stderr, "usage: cpu <string>\n");
11         exit(1);
12     }
13     char *str = argv[1];
14     while (1) {
15         Spin(1);
16         printf("%s\n", str);
17     }
18     return 0;
19 }
```

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

CPU 가상화

- 매우 많은 수의 가상 CPU가 존재하는 듯한 환상(illusion)을 제공함
  - 즉, 프로세스 (Process)가 하나의 CPU (Processor)을 개별적으로 소유한다고 착각함
- 프로그램 예제 (다수의 어플리케이션 실행)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <assert.h>
5  #include "common.h"
6  int
7  main(int argc, char *argv[])
8  {
9      if (argc != 2) {
10         fprintf(stderr, "usage: cpu <string>\n");
11         exit(1);
12     }
13     char *str = argv[1];
14     while (1) {
15         Spin(1);
16         printf("%s\n", str);
17     }
18     return 0;
19 }
```

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

CPU 가상화

- 운영체제는 어떻게 동시에 다수의 프로그램을 실행 시키는가?
  - Time sharing
- 다수의 프로그램을 동시에 실행시킬 경우, 누가 실행되어야 하는가?
  - CPU scheduling

메모리 가상화

- 물리 메모리는 단순한 바이트의 배열이며, 주소(address)을 통해서 접근 가능함
  - 메모리 데이터 **읽기** (read): 메모리 주소
  - 메모리 데이터 **쓰기** (write): 메모리 주소, 데이터

- 메모리는 프로그램이 실행되는 동안 항상 접근
  - Load, Store 등의 명령어 사용

메모리 가상화

- 프로세스는 자신만의 가상 주소 공간(virtual address space)을 가지고 있는 듯한 환상(illusion)을 제공함
  - 즉, 하나의 프로그램이 수행하는 각종 메모리 연산은 다른 프로그램의 주소 공간에 영향을 주지 않음

메모리 가상화

- 프로그램 예제 (단일 어플리케이션 실행)

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int
6  main(int argc, char *argv[])
7  {
8      int *p = malloc(sizeof(int)); // a1
9      assert(p != NULL);
10     printf("(%d) memory address of p: %08x\n",
11         getpid(), (unsigned) p); // a2
12     *p = 0; // a3
13     while (1) {
14         Spin(1);
15         *p = *p + 1;
16         printf("(%d) p: %d\n", getpid(), *p); // a4
17     }
18     return 0;
19 }
```



```
prompt> ./mem
(2134) memory address of p: 00200000
```

메모리 가상화

- 프로그램 예제 (단일 어플리케이션 여러 번 실행)

```
1  #include <unistd.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "common.h"
5  int
6  main(int argc, char *argv[])
7  {
8      int *p = malloc(sizeof(int)); // a1
9      assert(p != NULL);
10     printf("(%d) memory address of p: %08x\n",
11         getpid(), (unsigned) p); // a2
12     *p = 0; // a3
13     while (1) {
14         Spin(1);
15         *p = *p + 1;
16         printf("(%d) p: %d\n", getpid(), *p); // a4
17     }
18     return 0;
19 }
```



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

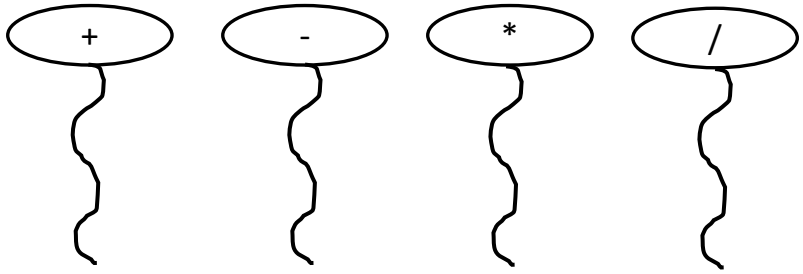
메모리 가상화

- 가상 주소 공간을 물리 주소 공간으로 어떻게 연결하는가?
  - Paging
- 물리 주소 공간의 데이터는 어떻게 영속성을 보장하는가?
  - File system

병행성

- 병행성은 프로그램이 한번에 많은 일을 수행할 수 있도록 함

Input A: 10
Input B: 10



Output: **20** Output: **0** Output: **100** Output: **1**

Single process ??

Multi processes ??

병행성

- 예제

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4  volatile int counter = 0;
5  int loops;
6  void *worker(void *arg) {
7      int i;
8      for (i = 0; i < loops; i++) {
9          counter++;
10     }
11     return NULL;
12 }
13
14 int
15 main(int argc, char *argv[])
16 {
17     if (argc != 2) {
18         fprintf(stderr, "usage: threads <value>\n");
19         exit(1);
20     }
21     loops = atoi(argv[1]);
22     pthread_t p1, p2;
23     printf("Initial value : %d\n", counter);
24
25     Pthread_create(&p1, NULL, worker, NULL);
26     Pthread_create(&p2, NULL, worker, NULL);
27     Pthread_join(p1, NULL);
28     Pthread_join(p2, NULL);
29     printf("Final value : %d\n", counter);
30     return 0;
31 }
```

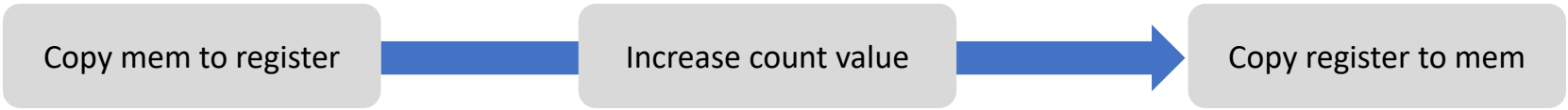
```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value   : 2000
```

```
prompt> ./thread 100000
Initial value : 0
Final value   : 143012 // 어?
prompt> ./thread 100000
Initial value : 0
Final value   : 137298 // 뭐라고?
```

병행성

- 프로그램은 원자적 (atomic)하게 수행되지 않음

- 예제
  - count 변수의 값을 1씩 증가



올바르게 동작하는 병행 프로그램은 어떻게 작성해야 하는가?

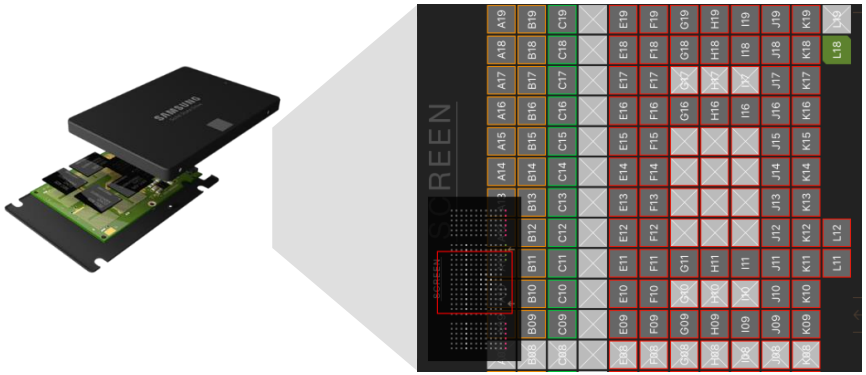
영속성

- 컴퓨터 시스템의 메인 메모리 (DRAM)은 휘발성 메모리임
  - 메인 메모리의 데이터는 쉽게 손실될 수 있음
  - 특히, 시스템 고장 또는 전원 공급의 차단은 메모리의 모든 데이터를 잃어버림
- 영속성이란?
  - 데이터를 영구적으로 안전하게 저장하는 특성을 말함

데이터를 영속적으로 저장하는 방법은 무엇인가?

파일 시스템

- 파일 시스템 (file system)은 운영체제에서 I/O 장치를 관리하기 위한 소프트웨어
  - 파일이 어디에 저장되는지 결정 및 관리
  - **파일 시스템은 CPU, memory 처럼 가상화 기술을 사용하지 않음**



파일 시스템

- 예제

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <assert.h>
4  #include <fcntl.h>
5  #include <sys/types.h>
6  int
7  main(int argc, char *argv[])
8  {
9      int fd = open("/tmp/file", O_WRONLY | O_CREAT | O_TRUNC, S_IRWXU);
10     assert(fd > -1);
11     int rc = write(fd, "hello world\n", 13);
12     assert(rc == 13);
13     close(fd);
14     return 0;
15 }
```

System call

External devices

- 운영체제는 hardware에 접근하기 위한 코드 (즉, 장치 드라이버)을 포함하고 있음
  - 표준 라이브러리(standard library)
    - Disk, network card, keyboard, etc.

- 장치 드라이버 (device driver)의 역할
  - 디바이스에 명령어 전달
  - 초기 interrupt event 처리

운영체제의 설계 목표

- 가상화
  - CPU, 메모리, 디스크와 같은 물리 자원을 가상화 (virtualize)
- 성능
  - 오버헤드 최소화
- 보호
  - 응용 프로그램 간의 보호 (즉, isolation)
- 신뢰성 보장
- 에너지 효율성
- 이동성

Q&A