01. introduction, 02. Processes

▼ OS

▼ program이 실행될 때 어떤 일이 발생?

- 동시에 많은 프로그램들이 실행됨
- 프로그램들이 메모리 공유하는 것을 허용함
- device와 상호작용도 가능
- etc..

⇒ OS : 시스템이 정확하고 효율적으로 수행하도록 만들어줌

▼ OS

1. virtual machine
    - 물리적인 자원을 더 general, powerful, and 사용하기 편리한 virtual form으로 바꿔줌
        - processor, memory or disk
2. standard library
    - system call과 같이 app에 적용 가능한 inteface 제공
        - 프로그램 실행, 메모리와 기기 접근 등
3. resource manager
    - 많은 프로그램이 cpu를 공유하면서 실행되도록 허용
        - cpu 공유
    - 많은 프로그램이 자신의 명령어와 데이터에 동시에 접근할 수 있도록 허용
        - 메모리 공유
    - 많은 프로그램이 device에 접근 할 수 있도록 허용
        - disk 공유

⇒ virualization, concurrency, persistence 나누어서 수업!

CPU ⇒ Process ⇒ ch2 ∪ ch6

memory ⇒ ch7 ∪ ch13

▼ Process

▼ Program vs Process

- **program** : disk에 exe file로 저장된 instruction과 static data
  - exe file : static data 존재 ⇒ program 내에서 변경/읽기 가능
    ↳ 항상 사용되더라도 메모리 영역, 값 유지 (↔ local variable : 사용될 때만 메모리 영역 할당)
- **process** : a running program (실행 중인 program)
  - **memory** : instructions(cpu가 이해할 수 있는 코드 in memory) and data
  - **registers** : pc, stack pointer, etc. ⇒ 값이 계속 변경되며 state 정보 담고 있음
    ↳ 임시로 필요한 memory, 위치
    - **pc** : cpu가 실행해야 하는 instruction 위치 가리킴
  - others : open되어 있는 file 정보 등
    ↳ 나중에 꼭 close해야 함.
- **APIs**
  - create, destroy, wait ...

▼ fork() system call

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]){
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork(); ↳ process 생성
  if (rc < 0) {
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    printf("I am child (pid:%d)\n", (int) getpid());
  } else {
    printf("I am parent of %d (pid:%d)\n", rc, (int) getpid());
  }
  return 0;
}
```

에러 / child / parent

⇒ 실행할 때마다 프로세스의 실행 순서가 다름 ⇒ parent, child 무엇 먼저?

운영체제의 policy가 다 다르기 때문에 process의 실행 순서가 달라짐 → wait()

▼ wait() system call

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[]){
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    printf("I am child (pid:%d)\n", (int) getpid());
  } else {
    int wc = wait(NULL); (cf. waitpid : 특정 process 명시하여 wait 가능)
    printf("I am parent of %d (wc:%d) (pid:%d)\n",
                                          rc, wc, (int) getpid());
  }
  return 0;
}
```

child process가 종료되어야 출력 가능(실행 순서 유지)

but, parent와 child 중에 누구에게 cpu 배정인지 정해진 건 없음, ☆

그저 wait에 의한 양보만 이루어짐 ⇒ exec() 로 해결

* fork()

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

] 순서 다름

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

] 순서 다름

* fork() + wait()

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

child process가 종료되어야 출력 가능 ⇒ 실행순서유지.

▼ exec() system call

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
int main(int argc, char *argv[]){
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    printf("I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    //strdup :malloc을 호출하여 string의 memory 공간 예약(null 포함)
    myargs[0] = strdup("wc");
    myargs[1] = strdup("p3.c");
    myargs[2] = NULL;
    ……
    execvp(myargs[0], myargs);
    printf("this shouldn't print out");
  } else {
    int wc = wait(NULL);
    printf("I am parent of %d (wc:%d) (pid:%d)\n",
                                          rc, wc, (int) getpid());
  }
  return 0;
}
```

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
 29 107 1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

instruction, exe file 실행 → child 생성 후 그 내부에서 사용

but, cpu core 개수 한정적 → 모든 process를 여러 개의 cpu가 있는 것처럼?

⇒ virtualization of the cpu : time sharing

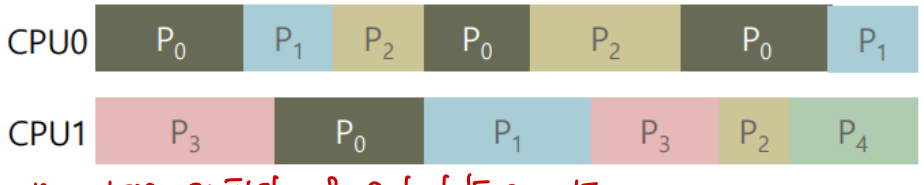
▼ Time Sharing

여러 process들이 cpu 자원을 적절한 시간 동안 가지고 있도록 시간 배분

하나의(고작 몇 개)의 cpu로 엄청나게 많은 가상 cpu를 가지고 있는 것 같은 현상!

⇒ user : 많은 concurrent process running 가능

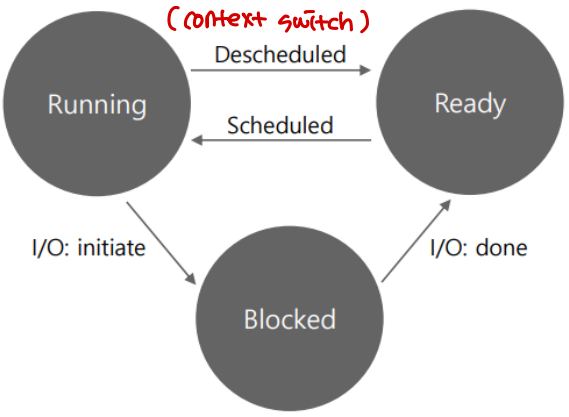
⇒ potential cost : CPU가 반드시 공유한다면 더 느려질 것임 => 어떻게 해결?



*context switch & scheduling policy

▼ Process state

생성 ~ 종료까지 아래의 상태를 반복



- **Running** : cpu 자원 할당 받아서 실행되고 있는 상태
- **Ready** : cpu 자원 받으면 실행될 수 있는 상태
- **Blocked(Waiting)** : process가 다른 event가 도달할 때까지 실행되지 않는 것

ex)
- I/O → cpu에 비해 느림 ⇒ I/O 작업에 도달하지 못 해서 OS가 blocked
- wait system call ⇒ child process를 blocked state로 옮김

Time	P_0	P_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	P_0 initiates I/O
4	Blocked	Running	P_0 is blocked, so P_1 runs
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	P_1 now done
9	Running	-	
10	Running	-	P_0 now done

OS 내부에 있는 코드를 CPU가 실행하긴 해야함
→ I/O 바로처리불가
→ 커널계속사용하도록 OS가 판단한경우(상황마다 다름)
I/O 종료 ⇒ CPU 자원 할당되면 실행 가능
CPU 자원 P1에 할당

CPU 여러개 하나만 있다고 가정한 example

CPU0	0	0	0	1	1	1	1	1	0	0

P_0 Running Blocked Ready Running

P_1 Ready Running ⇒ 상황끝

▼ Data Structures (PCB, process control blocks)

OS : process가 생성될 때마다 관리하는 자료 구조 생성 ⇒ PCB ★

- **Linux kernel** : /include/linux/sched.h

+thread : kernel 진입 ⇒ user mode의 process에 의해 사용되는 모든 register의 content가 kernel mode stack에 저장됨.

각 process : user/kernel mode 따로 존재함 ← 일시적인 정보 위치 저장

PCB 예시 ⇒ 구조체로 구현

```
struct task_struct {
    volatile long state; /* TASK_RUNNING, TASK_INTERRUPTIBLE …*/
    void *stack; /* Pointer to the kernel-mode stack */
    …
    unsigned int cpu;
    …
    struct mm_struct *mm;
    …
    struct task_struct *parent;
    struct list_head children;
    …
    struct files_struct *files;
}
```

running, ready, blocked 상태 알려줌

⇒ Linux : ready 구분은 X

running

blocked 중 하나 (중복 가능)

cpu : cpu core 위치

mm : memory virtualization 구조체

parent의 PCB

parent : process 관계 정보

children : Linked 리스트에 자식들 관리

files : process가 open한 file 정보 관리 (⇒ OS가 자동으로 close 가능한 이유)

▼ Scheduling Queues

OS의 판단에 따라 process의 상태 변경

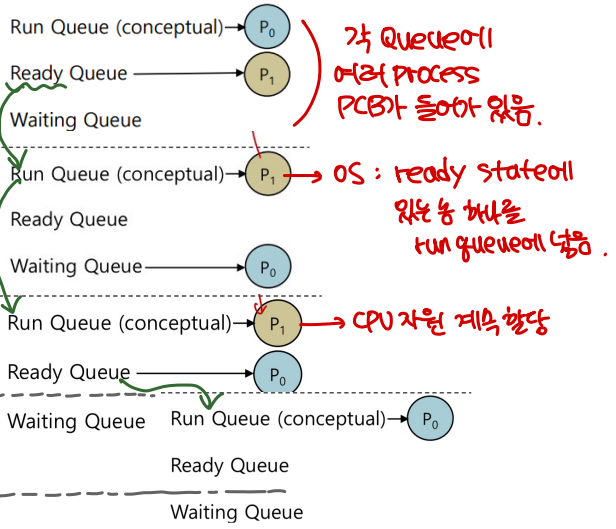
1. 각 상태에 있는 process 찾아야 함(찾는 overhead가 커질 수 있음)
2. I/O event에 해당하는 process 찾아서 ready state로 변경해야 함.

   → cpu마다 상태별로 process 분류하여 queue에 집어 넣음

   (OS마다 자료구조 달라짐)

- Run queue
- Ready queue
- Waiting queue

Time	P_0	P_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	P_0 initiates I/O
4	Blocked	Running	P_0 is blocked, so P_1 runs
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	P_1 now done
9	Running	-	
10	Running	-	P_0 now done



실행됨