

2019년 2학기 운영체제

Task Management

System Software Laboratory

School of Computer and Information Engineering

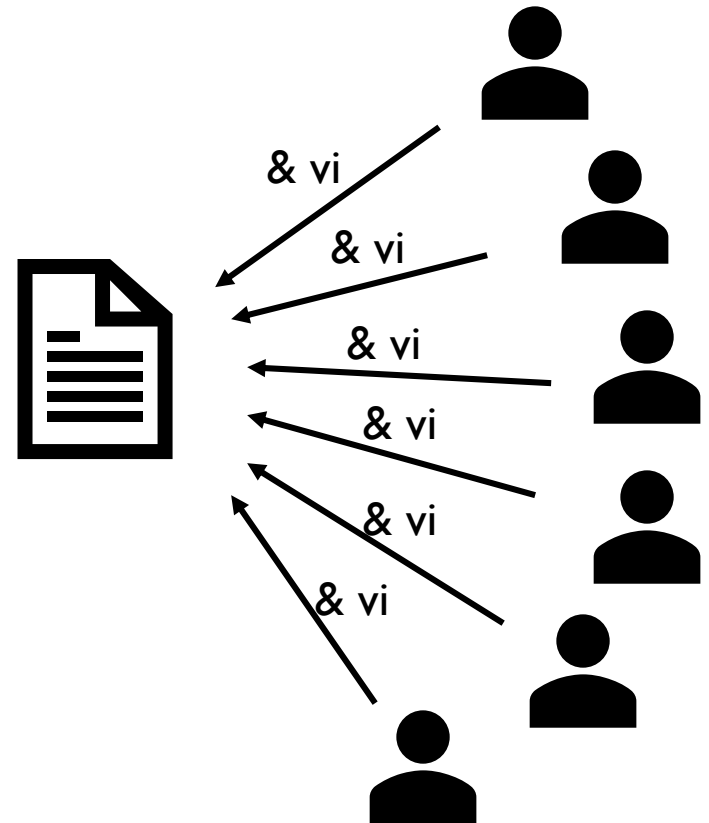
Kwangwoon Univ.

Contents

- Process와 Process Descriptor의 이해
- task_struct 구조체
- Lab. 1
- 프로세스 계보 (family)

프로세스

- 6명의 사용자가 vi 프로그램을 동시에 사용하는 경우
 - 각각 다른 프로세스가 6개 존재
 - 각각 다른 task_struct 구조체가 6개 존재
- 리눅스 코드에서 프로세스는 Task 또는 Thread라 부르기도 한다.



Process와 Process Descriptor의 이해

- **Process Descriptor**
 - 변화하는 process의 모든 정보를 담고 있는 자료구조
 - General Term : **PCB** (Process Control Block)
 - Linux Specific : **task_struct** 라는 자료 구조를 사용
 - 즉, 리눅스에서 Process descriptor(프로세스 기술자)는 task_struct 자료 구조
 - 커널은 각 프로세스가 무엇을 하고 있는지 명확히 알아야 한다.
 - E.g.
 - 프로세스의 ID 및 우선 순위
 - 프로세스가 실행 상태
 - 프로세스가 할당 되어있는 주소 공간
 - 다룰 수 있는 파일
 -

task_struct 구조체 (1/11)

■ 구조체를 통한 정보 관리

- 프로세스가 생성되면 task_struct 구조체를 통해 프로세스의 모든 정보를 저장하고 관리
 - 모든 태스크들에게 하나씩 할당
 - include/linux/sched.h
 - 태스크 ID, 상태 정보, 가족 관계, 명령, 데이터, 시그널, 우선순위, CPU 사용량 및 파일 디스크립터 등 생성된 태스크의 모든 정보를 가짐
- task_struct ***current**
 - 현재 실행되고 있는 태스크를 가리키는 매크로
 - in <include/asm-generic/current.h>
 - get_current()

태스크 식별 정보
상태 정보
스케줄링 정보
태스크 관계 정보
시그널 정보
콘솔 정보
메모리 정보
파일 정보
문맥교환 정보
시간 정보
자원 정보
기타 정보

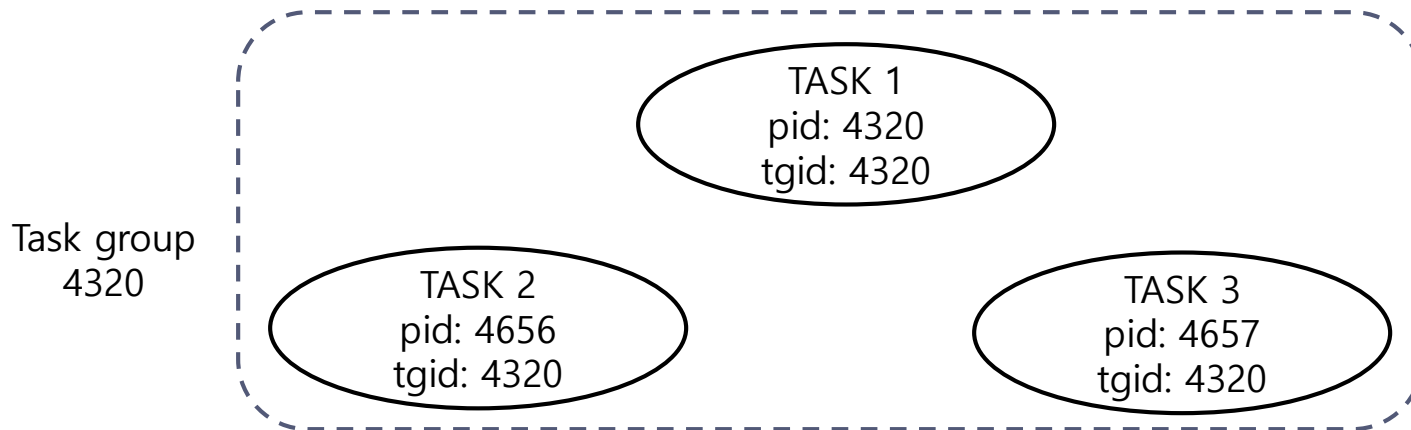
task_struct 구조체

task_struct 구조체 (3/11)

- 태스크 식별 정보에 관련된 변수, 함수들

```
pid_t pid; // Thread(Lightweight Process) ID
pid_t tgid; // Thread Group(Process) ID
struct hlist_node pid_links[PIDTYPE_MAX];
```

- 리눅스 커널에서의 태스크 식별
 - 프로세스와 스레드 모두 task_struct로 관리 (공유, 접근제어의 차이)
 - 시스템에 존재하는 태스크를 구분하기 위해, 각 태스크에 pid를 할당
 - 한 프로세스 내에서 생성된 스레드들은 동일한 tgid를 가지는 그룹을 형성하고, 각각의 스레드들은 유일한 pid를 가짐

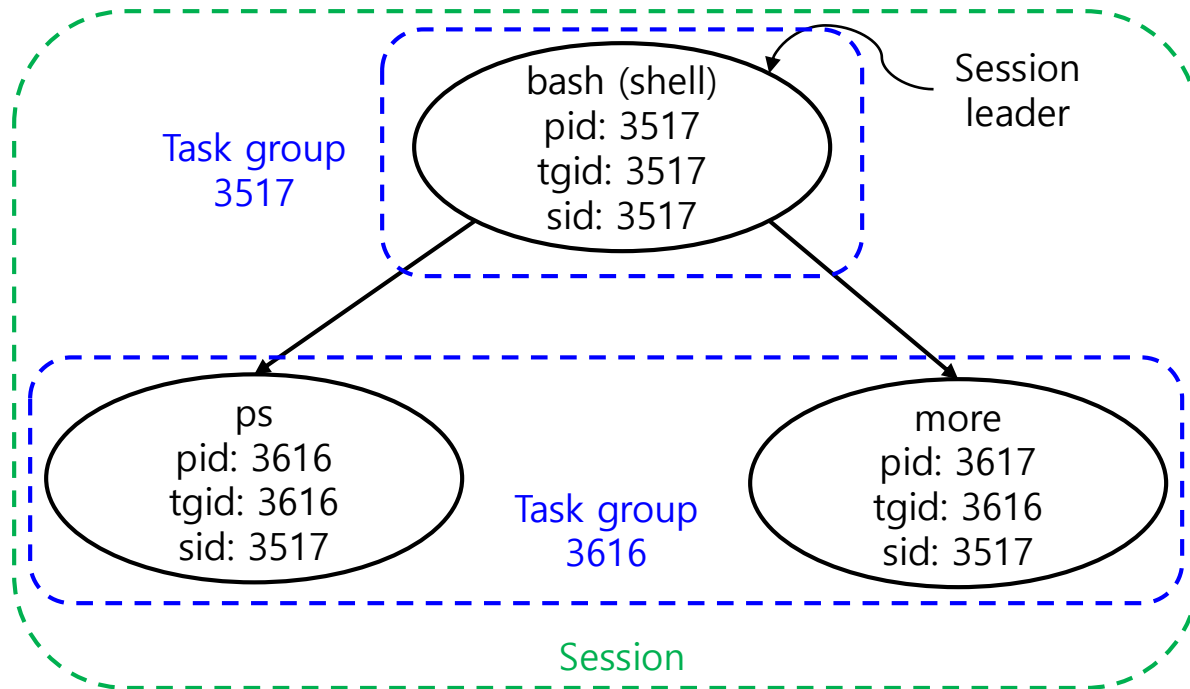


task_struct 구조체 (4/11)

- **태스크 식별 정보에 관련된 변수, 함수들**

- 태스크 그룹과 세션 리더의 예

- 사용자가 콘솔로 로그인하여 shell을 띄웠다고 가정
 - shell도 하나의 태스크이므로 자신의 PID를 가지며, 하나의 태스크 그룹을 형성
 - #ps | more 명령을 실행하면 각 명령에 대해 각각 PID 값을 부여 받고, 이 두 명령은 하나의 태스크 그룹을 형성
 - 두 개의 태스크 그룹은 하나의 세션을 이루며 이때 shell은 세션 리더가 됨

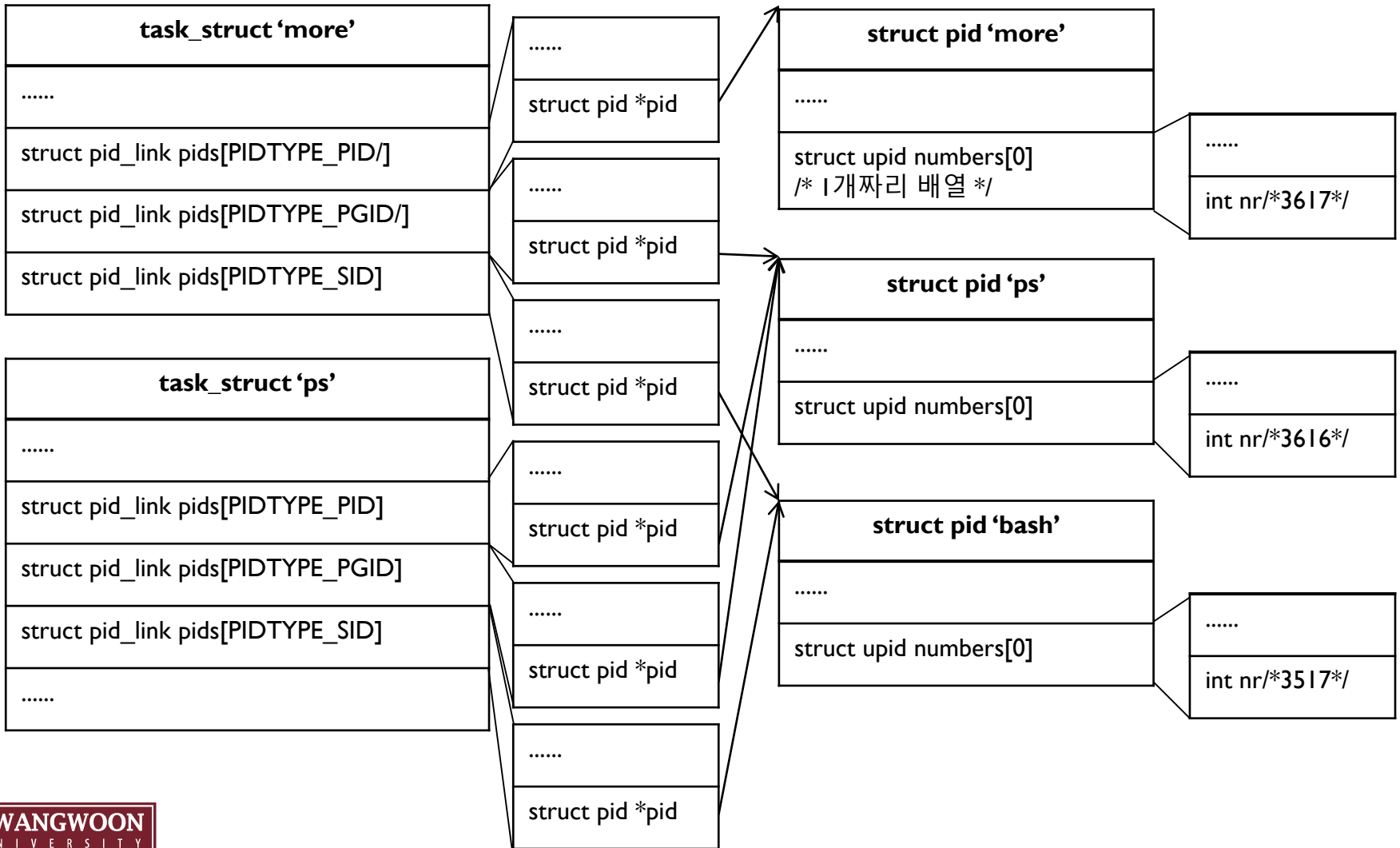


task_struct 구조체 (5/11)

■ task_struct와 pid관계

```
$ ps -jt pts/0 | more
```

PID	PGID	SID	TTY	TIME	CMD
3517	3517	3517	pts/0	00:00:00	bash
3616	3616	3517	pts/0	00:00:00	ps
3617	3616	3517	pts/0	00:00:00	more



task_struct 구조체 (6/11)

- **태스크들에 대한 사용자의 접근 제어에 관련된 변수, 함수, 매크로**
 - 태스크가 생성되면 사용자의 ID와 사용자가 속한 그룹의 ID가 등록됨

```
struct cred *cred;  
gid_t GROUP_AT(struct group_info *gi, int i);
```

- struct cred
 - 권한 관련 변수들이 저장된 구조체.
 - (uid, suid, euid, fsuid, gid sgid, egid, fguid)
- struct group_info
 - 그룹들의 정보를 가진 구조체.

```
for(i=0; i < current->cred->group_info->ngroup; ++i )  
{  
    gid = GROUP_AT(current->cred->group_info, i);  
}
```

task_struct 구조체 (7/11)

task_struct, cred, group_info의 관계

task_struct 'passwd'
.....
struct cred *cred;

cred 'passwd'
.....
uid_t uid, suid, euid, fsuid; /*0, ...*/
gid_t gid, sgid, egid, fsgid; /*1000, ...*/
struct group_info *group_info;

group_info 'passwd'
.....
int ngroups;
gid_t *blocks[0];

\$ sudo ps -o
pid,ppid,uid,suid,euid,fsuid,gid,sgid,egid,fsgid,cmd

```
sslab@sslab:~$ passwd &
[2] 5306
sslab@sslab:~$ Changing password for sslab.
(current) UNIX password: 123
123: command not found

[2]+  Stopped                  passwd

[2]+  Stopped                  passwd
sslab@sslab:~$ sudo ps -o pid,ppid,uid,suid,euid,fsuid,gid,sgid,egid,fsgid,cmd
  PID  PPID   UID   SUID   EUID  FSUID    GID   SGID   EGID  FSGID  CMD
  5294  5196    0     0     0     0   1000   1000   1000   1000  passwd
  5306  5196    0     0     0     0   1000   1000   1000   1000  passwd
  5309  5196    0     0     0     0   1000   1000   1000   1000  sudo ps -o
  5310  5309    0     0     0     0     0     0     0     0    ps -o pid,p
```

suid (saved uid) : 권한 전환을 지원하는데 사용, uid 값을 저장하고 있다

euid (effective uid) : 프로세스가 파일에 대해 가지는 권한
파일에 접근 시 euid를 통해 파일 접근 허용 여부 결정

fsuid (filesystem uid) : 파일시스템 접근 제어 용도로 사용됨

task_struct 구조체 (8/11)

- **태스크 상태 정보에 관련된 변수**

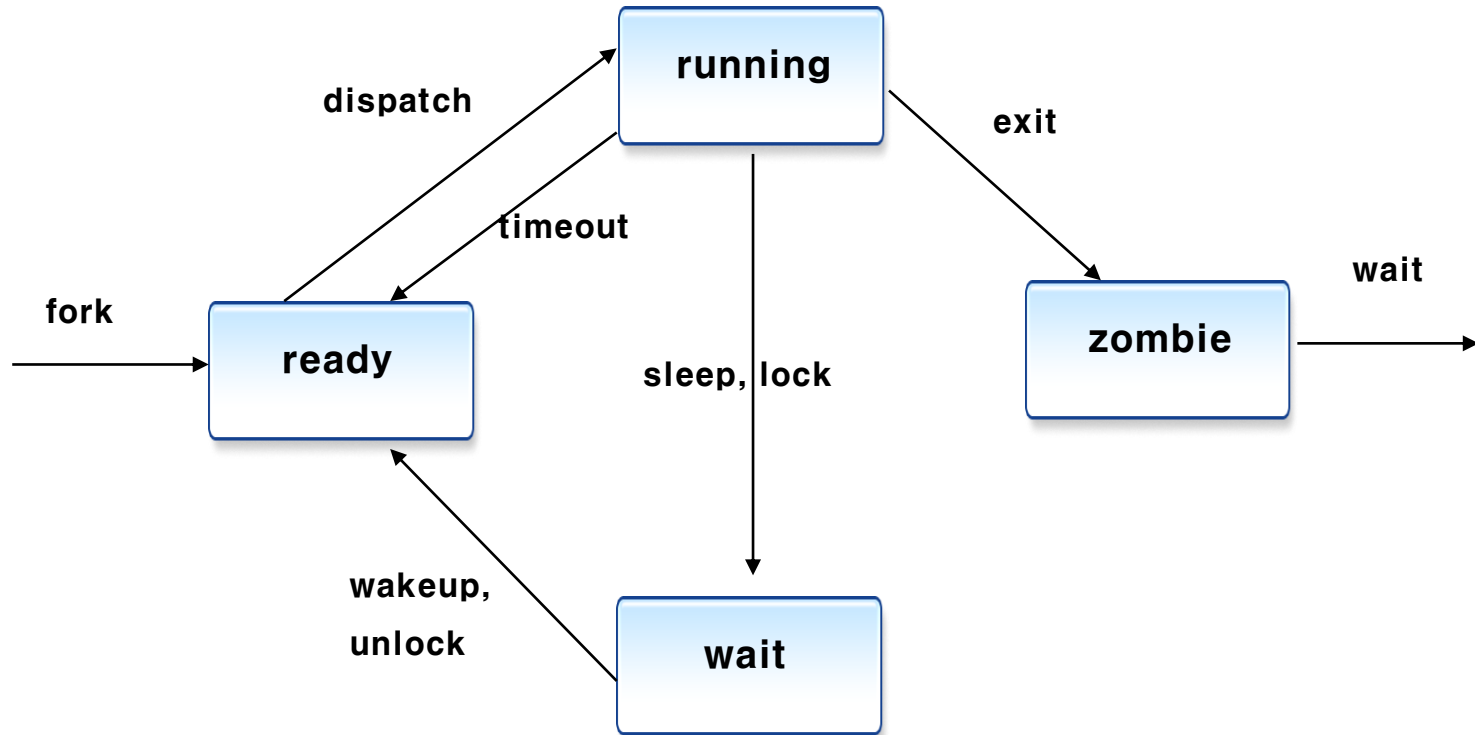
- state 변수

```
#define TASK_RUNNING 0
#define TASK_INTERRUPTIBLE 1
#define TASK_UNINTERRUPTIBLE 2
#define TASK_STOPPED 4
#define EXIT_ZOMBIE 16
```

- TASK_RUNNING
 - 실행 중이거나 준비 상태
- TASK_INTERRUPTIBLE
 - 하드웨어나 시스템 자원을 사용할 수 있을 때까지 기다리고 있는 대기 상태
 - 예) wait for a semaphore
- TASK_UNINTERRUPTIBLE
 - 하드웨어적인 조건을 기다리는 상태, 시그널을 받아도 무시
 - 예) process가 장치 파일을 열 때 해당 장치 드라이버가 자신이 다룰 하드웨어 장치가 있는지 조사할 때, memory swapping
- TASK_STOPPED
 - 수행 중단 상태 (시그널을 받거나 트레이싱 등)
- EXIT_ZOMBIE
 - process 실행은 종료했지만 아직 process의 자원을 반환하지 않은 상태

task_struct 구조체 (9/11)

- 태스크 상태와 전이



task_struct 구조체 (10/11)

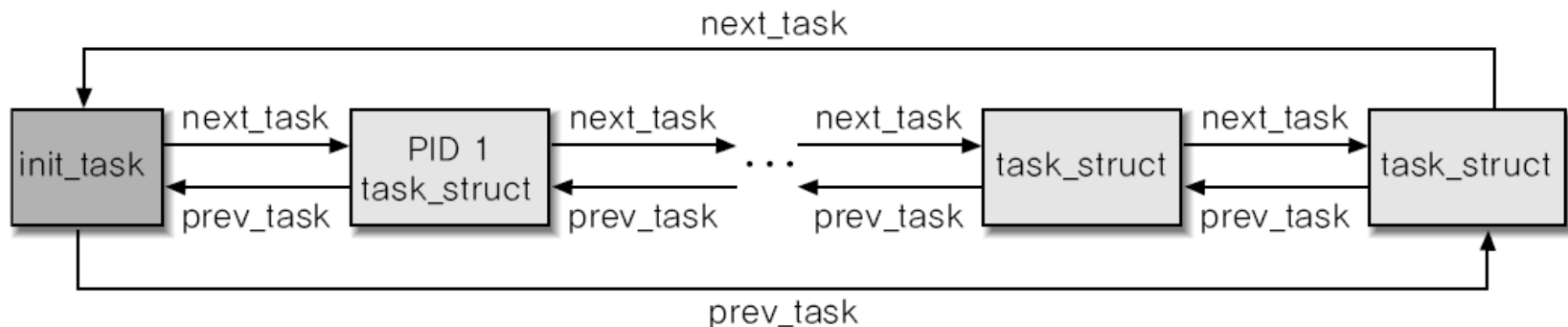
- 태스크 관계와 관련된 변수들

```
struct task_struct *real_parent, *parent;
```

- real_parent : 원래 부모 태스크(실제로 fork한 task)
- parent : 현재 부모 태스크(SIGCHLD signal을 받는 task)

```
struct list_head tasks, children, sibling;
```

- 커널에 존재하는 모든 태스크들은 원형 이중 연결 리스트로 연결



task_struct 구조체 (11/11)

- 그 밖의 변수들
 - 스케줄링 정보
 - 시그널 정보
 - 메모리 정보
 - 파일 정보
 - 문맥 교환 정보
 - 시간 정보
 - 자원 사용 정보 등

Lab. 1 (1/3)

- 커널에 존재하는 모든 태스크들의 pid와 이름을 출력하여 확인하는 프로그램

- 사용할 변수

```
struct task_struct {  
    ...  
    pid_t pid;           // task 식별자  
    ...  
    char comm[TASK_COMM_LEN]; /* executable name excluding path  
                                - access with [gs]et_task_comm (which lock  
                                it with task_lock())  
                                - initialized normally by setup_new_exec */  
    ...  
};
```

- 사용할 매크로

- include/linux/sched/signal.h

```
#define next_task(p) \  
    list_entry_rcu((p)->tasks.next, struct task_struct, tasks)
```

Lab. 1 (2/3)

- 커널에 존재하는 모든 태스크들의 pid와 이름을 출력하여 확인하는 프로그램

```
displaytask.c + (~/.test_task) - VIM
1 #include <linux/module.h>
2 #include <linux/sched.h>
3 #include <linux/init_task.h>
4
5 int displaytask_init(void)
6 {
7     struct task_struct *findtask = &init_task;
8
9     do
10    {
11        printk("%s[%d] -> ", findtask->comm, findtask->pid);
12        findtask = next_task(findtask);
13    }
14    while( (findtask->pid != init_task.pid));
15    printk("%s[%d]\n", findtask->comm, findtask->pid);
16    return 0;
17 }
18
19 void displaytask_exit(void)
20 {
21 }
22
23 module_init(displaytask_init);
24 module_exit(displaytask_exit);
25 MODULE_LICENSE("GPL");
```


Lab. 1 (3/3)

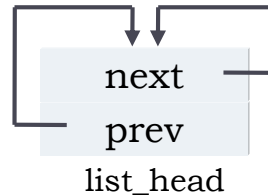
- 커널에 존재하는 모든 태스크들의 pid와 이름을 출력하여 확인하는 프로그램 (cont'd)
 - 결과 화면

```
root@ubuntu:/home/sslab/test_task# dmesg
[ 3783.031518] init swapper/0[0]
[ 3783.031518] swapper/0[0] ->
[ 3783.031519] systemd[1] ->
[ 3783.031519] kthreadd[2] ->
[ 3783.031520] rcu_gp[3] ->
[ 3783.031520] rcu_par_gp[4] ->
[ 3783.031521] kworker/0:0H[6] ->
[ 3783.031521] mm_percpu_wq[8] ->
[ 3783.031522] ksoftirqd/0[9] ->
[ 3783.031522] rcu_sched[10] ->
[ 3783.031523] rcu_bh[11] ->
[ 3783.031523] migration/0[12] ->
[ 3783.031524] cpuhp/0[14] ->
[ 3783.031524] cpuhp/1[15] ->
[ 3783.031525] migration/1[16] ->
[ 3783.031525] ksoftirqd/1[17] ->
[ 3783.031526] kworker/1:0H[19] ->
[ 3783.031526] cpuhp/2[20] ->
[ 3783.031526] migration/2[21] ->
[ 3783.031527] ksoftirqd/2[22] ->
[ 3783.031527] kworker/2:0H[24] ->
```

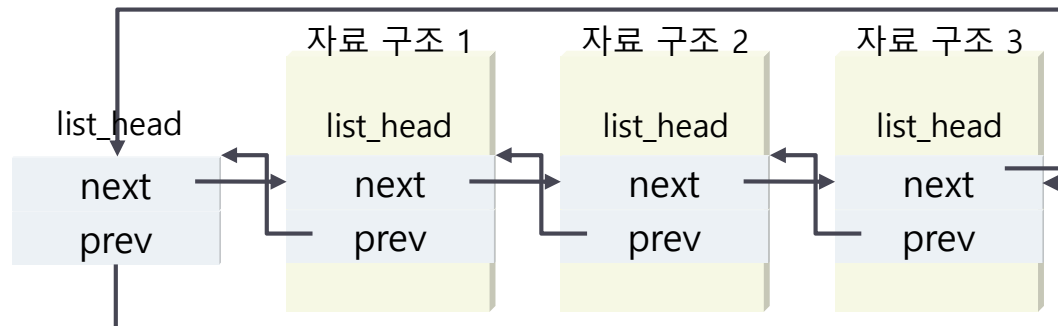
프로세스 계보 (family) (1/5)

- 이중 연결 리스트 (doubly linked list)

```
struct list_head {  
    struct list_head *next, *prev;  
};
```



(a) 비어있는 이중 연결 리스트



(b) 몇 개의 자료 구조를 가진 이중 연결 리스트

프로세스 계보 (family) (2/5)

- 리스트를 처리하는 매크로 및 함수

- `list_add(n, p)`
 - `p` 바로 다음에 `n`을 삽입.
- `list_add_tail(n, p)`
 - `p` 바로 앞에 `n`을 삽입.
- `list_del(p)`
 - `p`를 삭제.
- `list_empty(p)`
 - 헤드가 `p`인 리스트가 비어있는지를 검사.
- `list_entry(p, t, m)`
 - 이름이 `m`이고 주소가 `p`인 `list_head` 필드를 포함한 `t`타입 자료구조의 주소를 반환.
- `list_for_each(p, h)`
 - 헤드의 주소가 `h`로 지정된 모든 원소를 순환.
 - 각 원소의 `list_head` 자료 구조의 주소가 `p`에 반환

프로세스 계보 (family) (3/5)

- 부모 프로세스의 디스크립터를 얻으려면

- 부모 프로세스의 task_struct에 대한 포인터 → **parent**

```
struct task_struct *my_parent = current->parent;
```

- 자식 프로세스들의 디스크립터를 얻으려면

- 자식 프로세스의 task_struct에 대한 포인터 → **children**

```
struct task_struct *task;  
struct list_head *list;
```

```
list_for_each(list, &current->children) {  
    task = list_entry(list, struct task_struct, sibling);  
    /* task now points to one of current's children */  
}
```



```
list_for_each_entry(task, &current->children, list) {  
    /* task now points to one of current's children */  
}
```

프로세스 계보 (family) (4/5)

- 조상 프로세스를 끝까지 따라가려면
 - init 프로세스의 task_struct에 대한 포인터

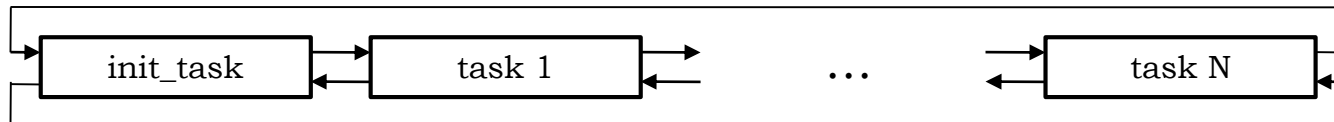
```
struct task_struct *task;  
for (task = current; task != &init_task; task = task->parent)  
    ;  
/* task now points to init */
```

- 전체 리스트에서 다음 또는 이전 프로세스를 얻으려면
 - next_task(task);
 - prev_task(task);

프로세스 계보 (family) (5/5)

- 시스템 내 모든 프로세스를 출력하려면
 - for_each_process(task)
 - 리스트에 있는 모든 프로세스를 탐색

```
struct task_struct *task;  
  
for_each_process(task) {  
    /* 각 태스크의 이름과 PID 출력 */  
    printk("%s[%d]\n", task->comm, task->pid);  
}
```



```
#define for_each_process(p) \\\n    for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

2019년 2학기 운영체제

CPU Scheduling

System Software Laboratory

School of Computer and Information Engineering

Kwangwoon Univ.

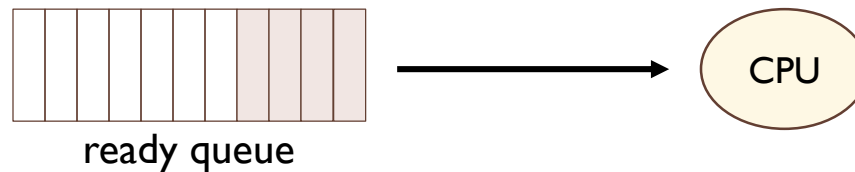
Contents

- Scheduling
- Linux Scheduling Policy
- O(1) Scheduler
- CFS (Completely Fair Scheduler)
- CFS Scheduler

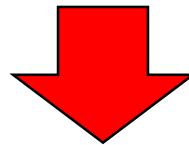
Scheduling

- **Scheduler**

- 한정된 자원을 다수의 client가 사용하려할 때
 - Ready queue에 client들이 대기



- 성능 향상을 위해 ready queue 내의 client들의 순서 조정 필요
 - Throughput, response time 등 ..



Scheduling

Scheduling

- **Scheduler**
 - OS에서의 Scheduling
 - CPU Scheduling
 - Process(task and thread)
 - Storage Scheduling
 - I/O requests
 - Network Scheduling
 - Packets

Linux Scheduling Policy (1/6)

- **Policy**

- 스케줄러가 무엇을 언제 실행할 것인지를 정하는 동작.
- 프로세서 시간의 사용을 최적화하는 책임이 있다.

- **목적**

- 프로세스 응답 시간을 빠르게 하는 것
- 시스템 사용률을 최대화 하는 것

- **프로세스**

- 프로세서 중심 프로세스 (CPU-bound task)
- 입출력 중심 프로세스 (I/O bound task)

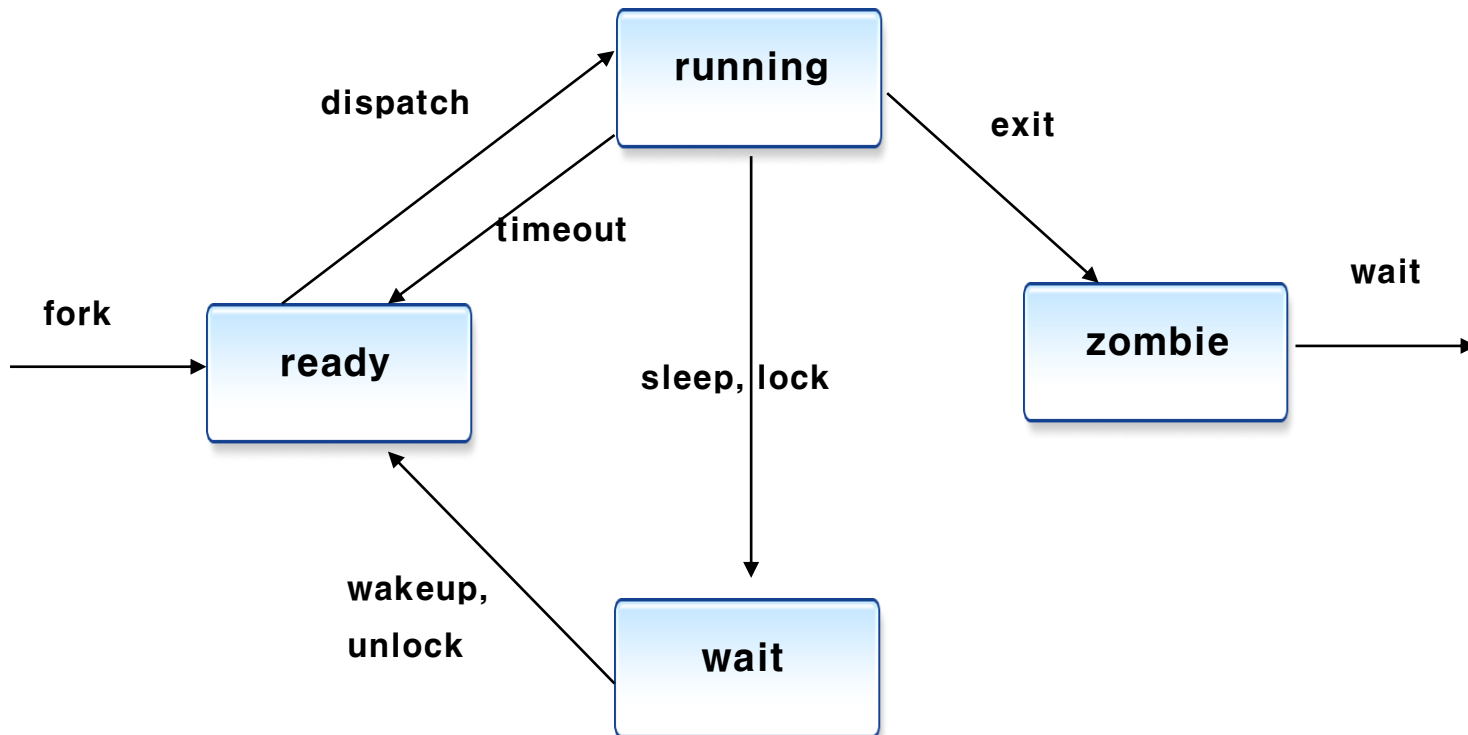
Linux Scheduling Policy (2/6)

- **CPU-bound task vs I/O bound task**
 - CPU의 사용 시간을 고려하여 분류
 - CPU-bound
 - 대부분의 시간을 code를 실행하는 데 사용
 - 입출력 요청으로 중단되는 경우가 드물어 선점될 때까지 계속 실행된다.
ex) compiler, video encoder
 - I/O-bound
 - 대부분의 시간을 I/O 요청을 하고 기다리는 데 사용
 - 실제 실행시간은 아주 짧다.
ex) text editor
- **Linux에서는 I/O-bound task를 우선적으로 처리**

Linux Scheduling Policy (3/6)

▪ Preemptive Scheduling

- 실행 중인 task가 CPU를 뺏길 수 있음
 - time slice의 만료
 - 더 높은 priority를 가지는 task의 실행



Linux Scheduling Policy (4/6)

- **Time Slice**

- 작업을 얼마나 더 실행할 수 있는지 나타내는 값
 - Time slice만큼 CPU를 사용 가능
 - **선점 가능**
 - Time slice를 너무 **길게** 잡으면
 - 시스템의 대화형 성능 저하
 - Time slice를 너무 **짧게** 잡으면
 - time slice를 소진한 프로세스를 다음 프로세스로 전환하는 데
빈번하게 시스템 시간 사용하게 됨.
- context switching
- Task의 priority에 따라 time slice를 배분
 - 높은 priority를 가진 task는 더 많은 time slice를 받는다

Linux Scheduling Policy (5/6)

- **Range of Priority**
 - 2가지의 단위로 표현 가능
 - Real time priority, Nice (실시간 우선순위)
- **Real time priority (특수 task)**
 - 0 ~ 99
 - 모든 실시간 task는 일반 task보다 더 높은 priority를 가진다
- **nice 값 (일반 task)**
 - -20(highest) ~ +19(lowest)
 - default value : 0
 - priority로 변환 시, 100 ~ 139의 값을 갖는다.
- 높은 priority를 가지는 task가 먼저 수행
- Linux에서는 task 실행 중 priority값이 동적으로 변경됨
 - I/O-bound task가 높은 priority를 부여 받음

Linux Scheduling Policy (6/6)

- **Context Switch**

- 하나의 task context(문맥)에서 다른 task의 context로 교환
- Context
 - Address space, Stack pointer, CPU register

```
static inline struct task_struct *context_switch(struct rq *rq, struct task_struct *prev, struct task_struct *next)
{
    struct mm_struct *mm = next->mm;
    struct mm_struct *oldmm = prev->active_mm;
    ...
    switch_mm(oldmm, mm, next);
    ...
    switch_to(prev, next, prev);
}
```

- switch_mm(), switch_to()
 - Task의 address space, stack pointer, register를 switch할 task의 값들과 교환.

O(1) Scheduler (1/5)

- **O(1) Scheduler**

- Linux kernel 2.6.22까지 사용하는 CPU Scheduler
 - time slice를 가지면서 가장 높은 priority를 가지는 task를 먼저 수행
 - Time slice 계산 과정에 상수시간 알고리즘 적용
 - 프로세서마다 별도의 실행대기 큐를 만듦.
- 시간 복잡도
 - $O(1)$: 여러 task가 ready queue에 존재하더라도 scheduling 시 발생하는 overhead가 $O(1)$
- 특징
 - SMP(Symmetric Multi Processor) 지원
 - Response time의 감소
 - Fairness 제공
- CPU 당 하나의 Run queue를 가짐
 - 실행 가능(TASK_RUNNING)한 task들의 목록
 - task는 하나의 run queue에만 존재

O(1) Scheduler (2/5)

■ O(1) Scheduler

■ Priority Arrays

- 하나의 Run queue는 두 개의 priority array를 가짐
 - Active : time slice가 남은 task들의 목록
 - Expired : time slice가 만료된 task들의 목록

```
struct runqueue{
    spinlock_t lock;                /* spin lock that protects this runqueue */
    unsigned long nr_running;        /* number of runnable tasks */
    unsigned long nr_switches;       /* context switch count */
    unsigned long expried_timestamp; /* time of last array swap */
    unsigned long nr_uninterruptible; /* uninterruptible tick */
    unsigned long long timestamp_last_tick; /* last scheduler tick */
    struct task_struct *curr;         /* currently running task */
    struct task_struct *idle;         /* this processor's idle task */
    struct mm_struct *prev_mm;        /* mm_struct of last ran task */
    struct prio_array *active;        /* active priority array */
    struct prio_array *expired;       /* the expired priority array */
    struct prio_array arrays[2];      /* the actual priority arrays */
    struct task_struct *migration_thread; /* migration thread */
    struct list_head migration_queue; /* migration queue */
    atimic_t nr_iowait;              /* number of tasks waiting on I/O */
};
```

- 각 priority array는 하나의 queue를 가짐
- Priority bitmap
 - 가장 높은 priority를 가진 task를 빠르게 검색

```
struct prio_array{
    int nr_active;                  /* number of tasks in the queues */
    unsigned long bitmap[BITMAP_SIZE]; /* priority bitmap */
    struct list_head queue[MAX_PRIO]; /* priority queues */
};
```

O(1) Scheduler (3/5)

- **Time Slice 계산**

- Priority array를 이용
- task의 time slice 만료 시
 - task의 priority에 따라 동적으로 time slice 계산
 - `task_timeslice(struct task_struct *p)`

```
static inline unsigned int task_timeslice(struct task_struct *p)
{
    return static_prio_timeslice(p->static_prio);
}
```

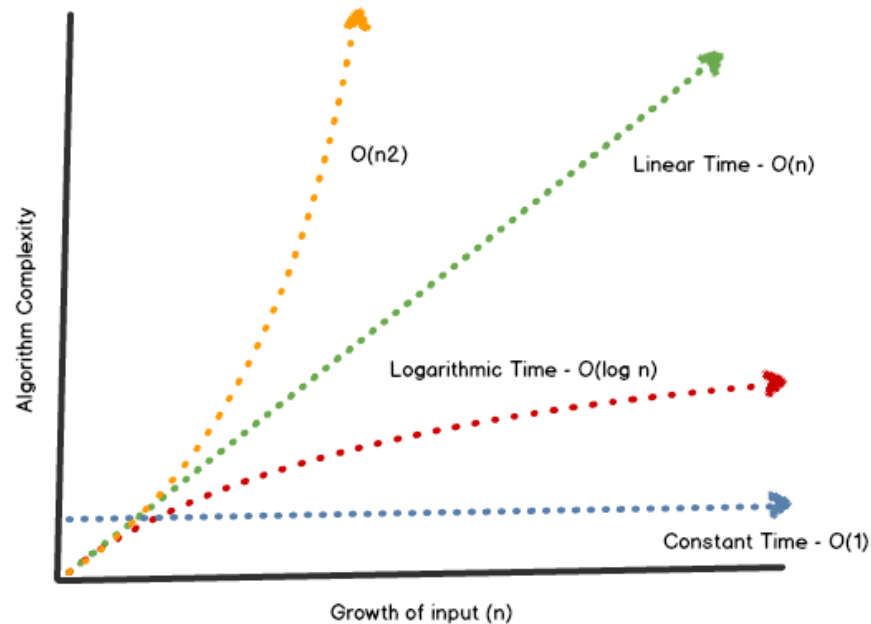
- **O(1) scheduler의 문제점**

- 시스템 내 우선순위가 가장 높은 태스크를 항상 먼저 스케줄링
- 동작하는 태스크의 time slice가 만료되면, 해당 태스크의 time slice를 static하게 재계산한 후 expired list로 옮김
 - Task의 수, CPU의 상황 등을 고려하지 않고 절대적인 time slice값을 계산

O(1) Scheduler (5/5)

- **O(1) scheduler**

- 장점 : 대화형(interactive) 프로세스가 없는 큰 서버 작업에는 이상적
 - 단점 : 대화형 프로세스의 역할이 중요한 시스템에는 부적합
-
- 2.6.23 커널 버전부터 CFS scheduler로 대체



CFS (Completely Fair Scheduler) (1/8)

- **CFS**

- Linux kernel 2.6.23부터 사용되는 CPU scheduler
- O(1) scheduler의 문제점 해결
 - 빈번한 context switching의 발생 가능성
 - 적은 time slice를 갖는 task들이 많을 경우
 - context switching으로 인하 overhead 증가
 - Throughput 및 response time에 악영향
 - Priority에 따른 time slice 배분으로 인한 Unfair
 - expired array에 있는 task의 실행이 미뤄질 수 있음
- 우선 순위(priority)에 대한 가중치(weight)에 따라서 프로세스에 time slice 할당

CFS (Completely Fair Scheduler) (2/8)

■ CFS

- 가장 실행이 덜 된 프로세스를 다음에 실행할 프로세스로 선택
- Time slice 계산
 - 프로세스 별로($1/n$ 로) time slice 할당 X
 - nice 값 자체로 time slice 계산 X
 - 프로세스에 할당할 프로세서 시간 비율의 **weight(가중치)**에 기반한 time slice 계산
 - weight는 priority에 기반
 - nice값이 높을 수록(우선순위가 낮을수록) 낮은 비율의 가중치 할당
 - nice값이 낮을 수록(우선순위가 높을수록) 높은 비율의 가중치 할당
 - 프로세스 실행 시간 = 자신의 가중치 / 전체 프로세스 가중치 총합 x (기본 값)
- priority가 낮더라도 좀 더 fair하게 time slice를 할당 받을 수 있음
- 각 프로세스에 할당하는 time slice의 최소 한계치(최소 세밀도)가 있다.
 - 기본값은 1ms
 - 최소 실행시간을 보장해 context switching 때문에 실행시간이 잠식되는 것을 방지.

CFS (Completely Fair Scheduler) (3/8)

- CFS

- 각 프로세스 별로 공정하게 할당된 몫만큼만 프로세서를 사용해야 한다.
 - 프로세스의 실행시간을 기록해두어야 함.
 - <linux/sched.h>에 정의된 struct sched_entity 를 이용해 정보 저장.

```
struct sched_entity {  
    /* For load-balancing: */  
    struct load_weight      load;  
    unsigned long           runnable_weight;  
    struct rb_node          run_node;  
    struct list_head        group_node;  
    unsigned int            on_rq;  
  
    u64                     exec_start;  
    u64                     sum_exec_runtime;  
    u64                     vruntime;  
    u64                     prev_sum_exec_runtime;  
  
    u64                     nr_migrations;  
  
    struct sched_statistics  statistics;  
};
```

CFS (Completely Fair Scheduler) (4/8)

- **Scheduler entity structure**

- CFS는 관리중인 모든 프로세스에 대해 sched_entity라는 구조체 유지.
 - task_struct->sched_entity

```
struct task_struct {  
    ...  
    int prio, static_prio, normal_prio;  
    unsigned int rt_priority;  
    const struct sched_class *sched_class;  
    struct sched_entity se;  
    struct sched_rt_entity rt;  
    ...  
};
```

- 이 구조체는 CFS 스케줄링 작업을 달성하기 위해 충분한 정보를 포함.
 - task_struct->sched_entity.vruntime

```
struct sched_entity {  
    /* For load-balancing: */  
    struct load_weight          load;  
    unsigned long               runnable_weight;  
    struct rb_node              run_node;  
    struct list_head            group_node;  
    unsigned int                on_rq;  
  
    u64                          exec_start;  
    u64                          sum_exec_runtime;  
    u64                          vruntime;  
    u64                          prev_sum_exec_runtime;  
  
    u64                          nr_migrations;  
  
    struct sched_statistics      statistics;  
};
```


CFS (Completely Fair Scheduler) (5/8)

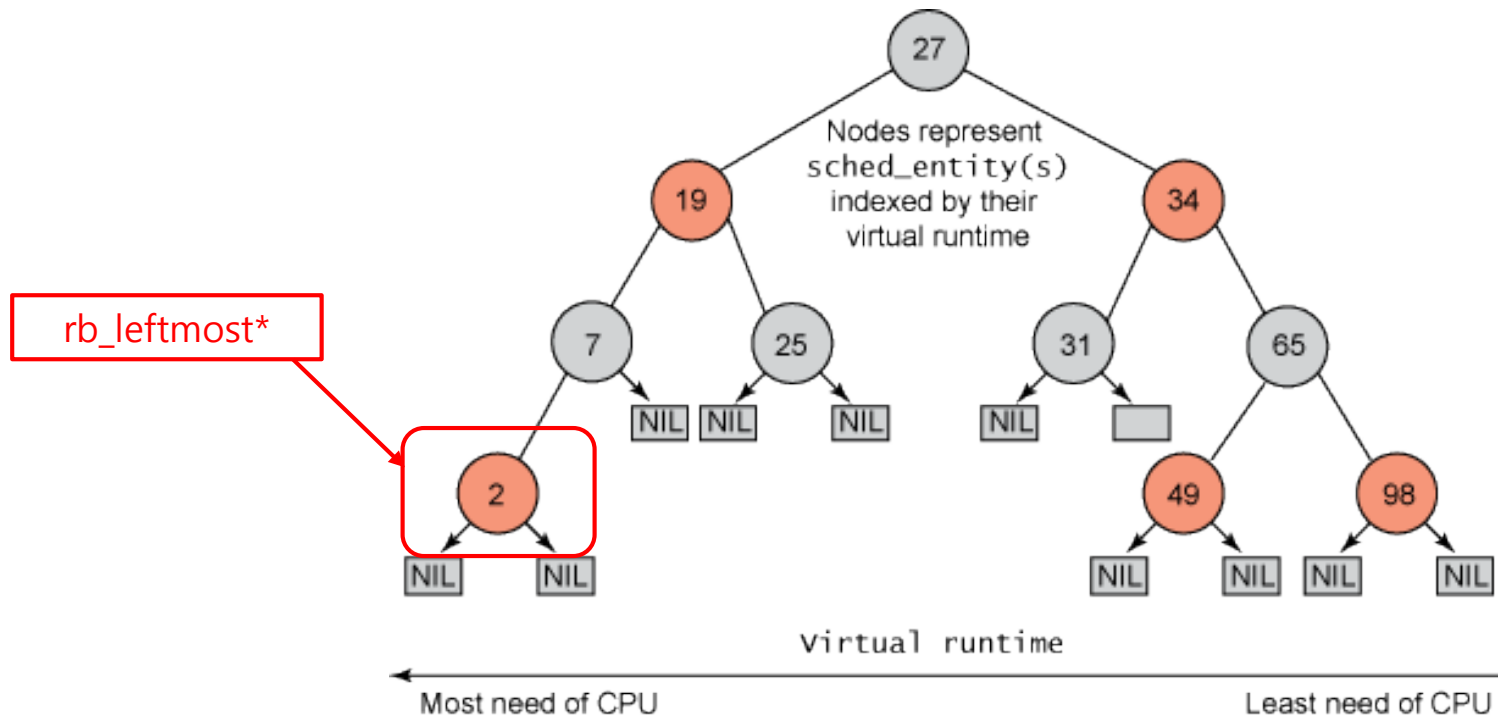
- CFS

- 프로세스들을 효율적으로 관리하기 위해 Red-Black tree 사용
 - $O(\log N)$ 으로 $O(1)$ 에 비해 느리지만 성능 상의 큰 차이는 없음
 - 다음에 실행하고자 하는 프로세스를 선택하기 쉽다.
 - virtual runtime이 제일 작은 프로세스를 실행 (tree의 가장 왼쪽 node)
 - tree의 가장 왼쪽에 있는 node에 해당하는 프로세스
 - rb_leftmost 포인터를 유지
 - 굳이 tree 를 순회하지 않아도 되도록 최적화 됨.

- rbtree 란?

- 자가 균형 이진 탐색 트리(self-balancing binary search tree)
- 데이터를 node 에 저장
- 특정 키로 각 node 를 식별할 수 있다.
- 키 값으로 그 키 값을 가진 데이터를 효율적으로 탐색 가능.

CFS (Completely Fair Scheduler)



CFS (Completely Fair Scheduler) (6/8)

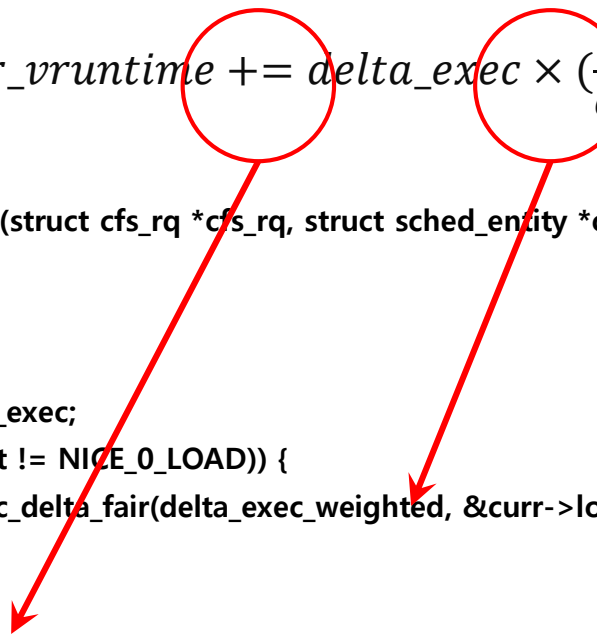
Virtual runtime

- CFS는 각 우선순위마다 가중치를 부여.
- Virtual runtime은 가중치에 따라 real runtime을 정규화한 값.
- CFS는 이 virtual runtime이 제일 작은 프로세스를 실행. (rb_leftmost)

$$curr_vruntime += delta_exec \times \left(\frac{NICE_0_LOAD}{curr_load_weight} \right)$$

```
static inline void __update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
unsigned long delta_exec)
{
    ...
    delta_exec_weighted = delta_exec;
    if (unlikely(curr->load.weight != NICE_0_LOAD)) {
        delta_exec_weighted = calc_delta_fair(delta_exec_weighted, &curr->load);
    }

    curr->vruntime += delta_exec_weighted;
    ...
}
```



CFS (Completely Fair Scheduler) (7/8)

- **Adding a process to the tree**

- 프로세스를 Run queue에 등록 시,
→ enqueue_entity() 함수 호출.
→ 몇 가지 사전작업 후, __enqueue_entity() 함수 호출.
→ rbtree에 해당 프로세스의 sched_entity 삽입.

static void

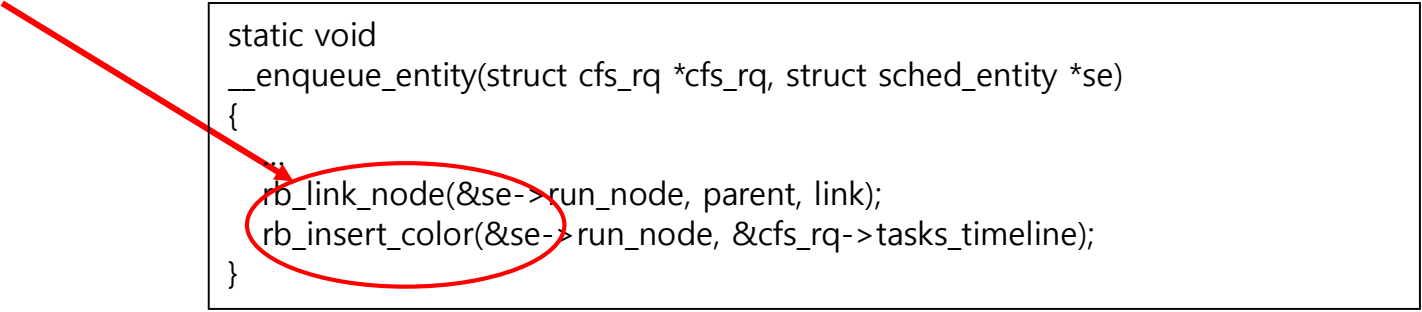
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)

{

...

__enqueue_entity(cfs_rq, se);

}



```
static void
__enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    ...
    rb_link_node(&se->run_node, parent, link);
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
```

rb_link_node : 새로 추가할 프로세스를 자식 node로 만든다.

rb_insert_color : tree의 자가 균형 속성 유지하게 한다.

CFS (Completely Fair Scheduler) (8/8)

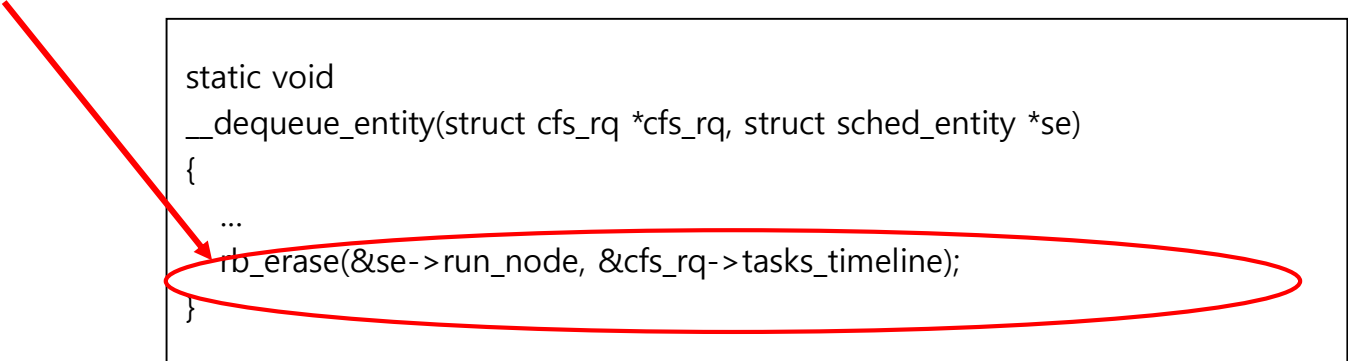
- **Removing a process to the tree**

- 프로세스를 Run queue에서 해제 시,
→ `dequeue_entity()` 함수 호출.
→ 몇 가지 사전작업 후, `__dequeue_entity()` 함수 호출.
→ Red black tree로부터 해당 프로세스의 `sched_entity` 해제.

static void

`dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int sleep)`

```
{  
    ...  
    __dequeue_entity(cfs_rq, se);  
    ...  
}
```



```
static void  
__dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)  
{  
    ...  
    rb_erase(&se->run_node, &cfs_rq->tasks_timeline);  
}
```

`rb_erase` : tree에서 프로세스 제거

CFS Scheduler

■ CFS Overview

```
struct task_struct {  
    ...  
    const struct sched_class *sched_class;  
    struct sched_entity se;  
    unsigned int policy;  
    unsigned int time_slice;  
    ...  
};
```

SCHED_NORMAL
SCHED_BATCH
SCHED_IDLE

SCHED_FIFO
SCHED_RR

**sched_class를 변경하여
스케줄링 정책 변경 가능.(유지보수)**

```
struct sched_entity {  
    struct load_weight load;  
    struct rb_node run_node;  
    ...  
#ifdef CONFIG_FAIR_GROUP_SCHED  
    struct sched_entity *parent;  
    struct cfs_rq *cfs_rq;  
    struct cfs_rq *my_rq;  
#endif  
};
```

**pick_next_task 함수는
다음에 스케줄링 될 태스크를 반환.**

```
static const struct sched_class fair_sched_class = {  
    .next = &idle_sched_class,  
    .enqueue_task = enqueue_task_fair,  
    .dequeue_task = dequeue_task_fair,  
    .pick_next_task = pick_next_task_fair,  
    .put_prev_task = put_prev_task_fair,  
    ...  
#ifdef CONFIG_SMP  
    .load_balance = load_balance_fair,  
    .move_one_task = move_one_task_fair,  
#endif  
    .set_curr_task = set_curr_task_fair,  
    ...  
};
```

```
const struct sched_class rt_sched_class = {  
    .next = &fair_sched_class,  
    .enqueue_task = enqueue_task_rt,  
    .dequeue_task = dequeue_task_rt,  
    .pick_next_task = pick_next_task_rt,  
    .put_prev_task = put_prev_task_rt,  
    ...  
#ifdef CONFIG_SMP  
    .load_balance = load_balance_rt,  
    .move_one_task = move_one_task_rt,  
#endif  
    .set_curr_task = set_curr_task_rt,  
    ...  
};
```

2019년 2학기 운영체제

Thread

System Software Laboratory

School of Computer and Information Engineering

Kwangwoon Univ.

Contents

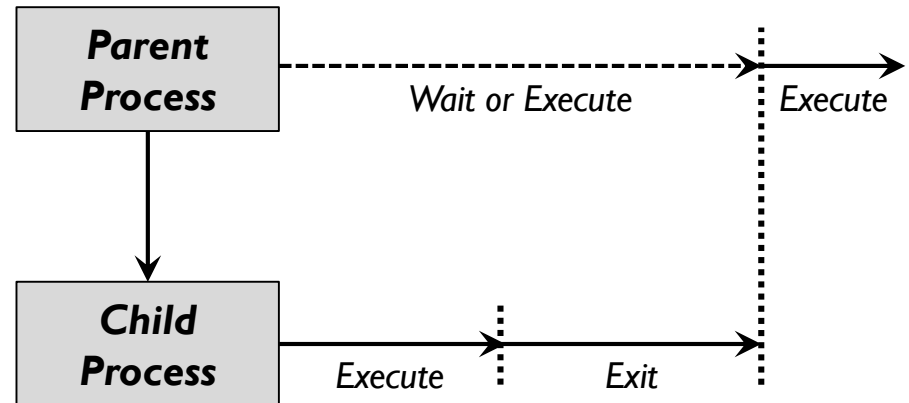
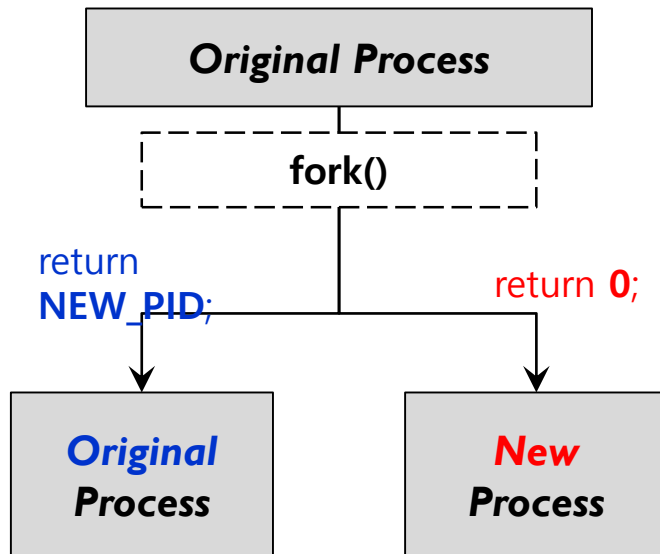
- Process Creation API
- 실습 1. Process Creation
- Process wait
- 실습 2. Process wait

- Thread의 이해
- POSIX Thread
- 실습 3. POSIX Thread

Process Creation API

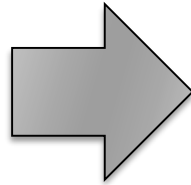
▪ fork()

- 새로운 프로세스는 부모 프로세스로부터 생성
 - 생성된 프로세스 : 자식 프로세스 (child process)
 - fork()를 호출한 프로세스 : 부모 프로세스 (parent process)
- 이 시점에서 두 프로세스가 동시 작업 수행



실습 1. Process Creation

```
1 #include <stdio.h>
2 #include <sys/types.h>
3
4 #define MAX 5
5
6 void child();
7 void parent();
8
9 int main()
10 {
11     pid_t pid;
12     if( (pid = fork()) < 0 )
13         return 1;
14     else if( pid == 0 )
15         child();
16     else
17         parent();
18     return 0;
19 }
20
21 void child()
22 {
23     int i;
24     for( i=0 ; i<MAX ; ++i, sleep(1) )
25         printf("child %d\n", i);
26     printf("child done\n");
27 }
28
29 void parent()
30 {
31     int i;
32     for( i=0 ; i<MAX ; ++i, sleep(1) )
33         printf("parent %d\n", i);
34     printf("parent done\n");
35 }
36
process.c
```



```
ssangkong@ssangkong-sslab:~/process$ make
cc -c -o process.o process.c
cc process.o -o process
ssangkong@ssangkong-sslab:~/process$ ./process
parent 0
child 0
parent 1
child 1
child 2
parent 2
child 3
parent 3
child 4
parent 4
parent done
child done
```

Process wait

- 자식 프로세스가 부모 프로세스에게 자원 반납
- 좀비 프로세스를 만들지 않기 위함
- 자식프로세스가 종료될 때까지 sleep
- 사용 함수: wait()

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

- int *status : status를 통해 자식 프로세스의 상태를 전달 받음
- return : 종료된 프로세스의 pid

Process wait

- 특정 프로세스가 종료될 때까지 sleep

- 사용 함수: waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

- pid_t pid : 종료한 프로세스의 pid
- int *status : status를 통해 자식 프로세스의 상태를 전달 받음
- int option : waitpid의 옵션
- return : 종료된 프로세스의 pid

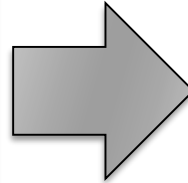
실습 2. Process Wait

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <wait.h>

#define MAX 5

int main(){
    pid_t pid;
    int i, a;

    for(i=0;i<MAX;i++){
        if( (pid = fork()) < 0)
            return 1;
        else if(pid == 0)
            exit(i);
    }
    for(i=0;i<MAX;i++){
        wait(&a);
        printf("original exit variable : %d\n",a);
        printf("shift exit variable   : %d\n\n",a>>8);
    }
    return 0;
}
```



```
os2019110613@ubuntu:~/test$ ./a.out
original exit variable : 0
shift exit variable   : 0

original exit variable : 256
shift exit variable   : 1

original exit variable : 1024
shift exit variable   : 4

original exit variable : 512
shift exit variable   : 2

original exit variable : 768
shift exit variable   : 3
```

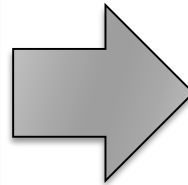
실습 2. Process Wait

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

void main(void)
{
    int pid1, pid2, status;
    pid_t child_pid;
    int a = 0;
    int b = 0;

    if ((pid1=fork()) == -1)
        perror("fork failed");

    if (pid1 == 0) {
        a = 1;
        printf("child_pid 1 = %d\n", getpid());
        if((pid2 = fork()) == 0){
            a = 2;
            printf("child_pid 2 = %d\n\n", getpid());
            exit(a);
        }
        else{
            child_pid = waitpid(pid2,&status,0);
            printf("child_pid : %d\n",child_pid);
            printf("original status : %d\n",status);
            printf("shift status : %d\n\n",status >> 8);
            exit(a);
        }
    }
    else if(pid1 != 0){
        child_pid = waitpid(pid1,&status,0);
        printf("child_pid : %d\n",child_pid);
        printf("original status : %d\n",status);
        printf("shift status : %d\n\n",status >> 8);
    }
}
```



```
os2019110613@ubuntu:~/test$ ./a.out
child_pid 1 = 9541
child_pid 2 = 9542

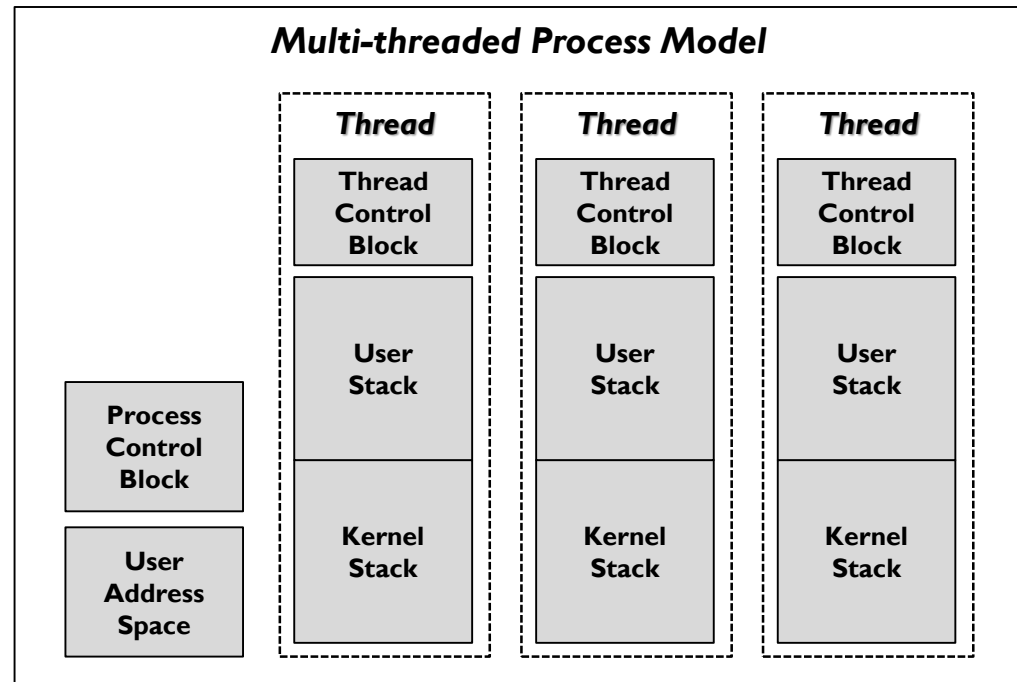
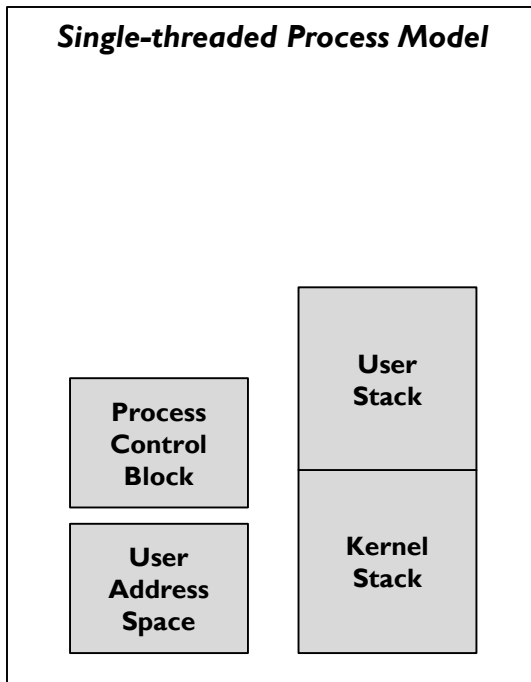
child_pid : 9542
original status : 512
shift status : 2

child_pid : 9541
original status : 256
shift status : 1
```

Thread의 이해

▪ Thread

- 특정 Process 내에서 실행되는 하나의 흐름을 나타내는 단위
- 독립된 program counter를 갖는 단위
- 독립된 register set과 stack을 가짐
- 비동기적인(asynchronous) 두 개의 작업이 서로 독립적으로 진행 가능
 - 처리를 위해 조건 변수나 mutex, semaphore와 같은 방법을 사용함



POSIX Thread

- **POSIX**

- 이식 가능 운영 체제 인터페이스(Portable Operating System Interface)
- 서로 다른 UNIX OS의 공통 API를 정리하여 이식성이 높은 유닉스 응용 프로그램을 개발하기 위한 목적으로 IEEE가 책정한 애플리케이션 인터페이스 규격

- **POSIX Thread**

함수명	설명
pthread_create	새로운 Thread를 생성함
pthread_detach	Thread가 자원을 해제하도록 설정
pthread_equal	두 Thread의 ID 비교
pthread_exit	Process는 유지하면서 지정된 Thread 종료
pthread_kill	해당 Thread에게 Signal을 보냄
pthread_join	임의의 Thread가 다른 Thread의 종료를 기다림
pthread_self	자신의 Thread id를 얻어옴

- 컴파일시 -pthread 옵션 추가
 - e.g. \$ gcc -pthread thread_test.c

POSIX Thread: Creation

- Thread는 `pthread_t` 타입의 thread ID로 처리
- POSIX thread는 사용자가 지정한 특정 함수를 호출함으로써 시작
 - 이 thread 시작 function은 `void*` 형의 인자를 하나 취한다
- 사용 함수: `pthread_create()`

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);
```

- `pthread_t *thread` : Thread ID
- `const pthread_attr_t *attr` : Thread 속성 지정. 기본값은 NULL.
- `void *(*start_routine)(void*)` : 특정 함수(**start_routine**)를 호출함으로써 thread가 시작
- `void *arg` : start 함수의 인자

POSIX Thread: Termination

- Process는 유지 하면서 pthread_exit() 함수를 호출하여 thread 자신을 종료
- 단순히 thread를 종료 하는 역할만 수행
 - 단, thread의 resource가 완전히 정리되지 않음

```
#include <pthread.h>  
  
void pthread_exit(void *retval);
```

- void *retval : Return value가 저장. 사용하지 않으면, NULL

POSIX Thread: Detach and Join

- **Join: 결합**
 - 생성된 thread가 pthread_join()을 호출한 thread에게 반환값을 전달하고 종료
- **Detach: 분리**
 - Process와 thread가 분리되면서 종료 시 자신이 사용했던 자원을 바로 반납
- 즉, thread를 종료 할 때 결합 혹은 분리가 필요

POSIX Thread: Join

- 다른 thread가 **thread_join()**을 반드시 호출해야 함
 - Thread의 memory resource가 완전히 정리되지 않음
- **pthread_join()**
 - 지정된 thread가 종료될 때까지 호출 thread의 수행을 중단

```
#include <pthread.h>

int pthread_join (pthread_t thread, void **retval);
```

- waitpid()의 역할과 유사
- void **retval : thread의 종료코드가 저장될 장소, 사용하지 않으면 NULL
- Return value
 - 성공 시: 0
 - 실패 시: 0이 아닌 오류 코드

POSIX Thread: Detach

- **결합 가능(joinable)한 상태의 thread**

- 분리되지 않은 thread
- 종료되더라도 자원이 해제되지 않음

- **pthread_detach()**

- Thread 종료 시 자원을 반납하도록 지정된 thread를 분리(detach) 상태로 만든다.

```
#include <pthread.h>

int pthread_detach (pthread_t thread);
```

- pthread_t thread : 스레드 식별자 thread
- Return value
 - 성공 시: 0
 - 실패 시: 0이 아닌 오류 코드

POSIX Thread: Thread Cleanup Handler

- **Thread cleanup handler 등록**

- thread 종료 시 호출되는 특정 함수 등록
- 하나의 thread에 둘 이상의 handler를 두는 것도 가능
 - 여러 handler는 하나의 스택에 등록

- **pthread_cleanup_push()**

- 지정된 마무리 함수를 **스택**에 등록

```
#include <pthread.h>
```

```
void pthread_cleanup_push(void(*routine)(void*), void* arg);
```

- routine : cleanup handler function
- void* arg: routine 함수의 인자
- handler 호출 조건
 - thread가 pthread_exit() 호출
 - thread가 pthread_cancel()에 반응

- 인

```
int pthread_cancel(pthread_t thread);
```

- thread가 execute 인수에 0이 아닌 값을 넣어 pthread_cleanup_pop()을 호출

POSIX Thread: Thread Cleanup Handler

- **Thread cleanup handler 제거**
 - 스택에 등록된 cleanup handler를 제거

- **pthread_cleanup_pop()**
 - 지정된 마무리 함수를 스택에서 제거

```
#include <pthread.h>

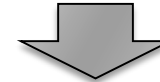
void pthread_cleanup_pop(int execute);
```

- execute : 값이 0일 경우 등록된 handler를 호출하지 않고 삭제함
값이 1일 경우 등록된 handler를 호출하고 삭제함
- cleanup handler는 스택에 등록된 반대 순서로 호출됨
- 이들은 매크로로 구현될 수 있기 때문에, push-pop의 호출은 반드시 한 thread 범위 안에서 짝을 맞춰 주어야 함
 - push가 { 문자를 포함하고, pop이 } 문자를 포함

실습 2. POSIX Thread

```
ssangkong — ssangkong@ssangkong-sslab: ~/thread — ssh — 80x53
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <linux/unistd.h>
6
7 void* thread_func(void *arg);
8 void cleanup_func(void *arg);
9 pid_t gettid(void);
10
11 int main()
12 {
13     pthread_t tid[2];
14
15     pthread_create(&tid[0], NULL, thread_func, (void*)0);
16     pthread_create(&tid[1], NULL, thread_func, (void*)1);
17
18     printf("main    gettid = %ld\n", (unsigned long)gettid());
19     printf("main    getpid = %ld\n", (unsigned long)getpid());
20
21     pthread_join(tid[0], NULL);
22     pthread_join(tid[1], NULL);
23
24     return 0;
25 }
26
27 void* thread_func(void *arg)
28 {
29     int i;
30     pthread_cleanup_push(cleanup_func, "first cleanup");
31     pthread_cleanup_push(cleanup_func, "second cleanup");
32     printf("$tid[%d] start\n", (int)arg);
33     printf("$tid[%d] gettid = %ld\n", (int)arg, (unsigned long)gettid());
34     printf("$tid[%d] getpid = %ld\n", (int)arg, (unsigned long)getpid());
35     for( i=0 ; i<0x40000000 ; ++i );
36     if( (int)arg == 0 )
37         pthread_exit(0);
38     pthread_cleanup_pop(0);
39     pthread_cleanup_pop(1);
40     return (void*)1;
41 }
42
43 void cleanup_func(void *arg)
44 {
45     printf("%s\n", (char*)arg);
46 }
47
48 pid_t gettid(void)
49 {
50     return syscall(__NR_gettid);
51 }
thread.c 1,1 Top
:wq
```

```
1 LDFlags=-pthread
2
3 thread:thread.o
4
5 clean:
6 $(RM) thread thread.o
```



```
ssangkong@ssangkong-sslab:~/thread$ make
cc -c -o thread.o thread.c
cc -pthread thread.o -o thread
ssangkong@ssangkong-sslab:~/thread$ ./thread
main    gettid = 4933
main    getpid = 4933
$tid[0] start
$tid[0] gettid = 4934
$tid[0] getpid = 4933
$tid[1] start
$tid[1] gettid = 4935
$tid[1] getpid = 4933
^Z
[1]+  Stopped                  ./thread
ssangkong@ssangkong-sslab:~/thread$ ps -L
  PID   LWP  TTY          TIME CMD
 3857   3857 pts/0        00:00:00 bash
 4933   4933 pts/0        00:00:00 thread
 4933   4934 pts/0        00:00:00 thread
 4933   4935 pts/0        00:00:00 thread
 4936   4936 pts/0        00:00:00 ps
ssangkong@ssangkong-sslab:~/thread$ fg
./thread
second cleanup
first cleanup
first cleanup
```


실습 3. POSIX Thread

```
1 LDFLAGS=-pthread
2
3 thread:thread.o
4
5 clean:
6     $(RM) thread thread.o
```

→ Linking시 자동으로 포함되는 변수

```
ssangkong@ssangkong-sslab:~/thread$ make
cc -c -o thread.o thread.c
cc -pthread thread.o -o thread
ssangkong@ssangkong-sslab:~/thread$ ./thread
main    gettid = 4933
main    getpid = 4933
$tid[0] start
$tid[0] gettid = 4934
$tid[0] getpid = 4933
$tid[1] start
$tid[1] gettid = 4935
$tid[1] getpid = 4933
^Z
[1]+  Stopped                  ./thread
ssangkong@ssangkong-sslab:~/thread$ ps -L
  PID  LWP  TTY          TIME CMD
 3857  3857 pts/0        00:00:00 bash
 4933  4933 pts/0        00:00:00 thread
 4933  4934 pts/0        00:00:00 thread
 4933  4935 pts/0        00:00:00 thread
 4936  4936 pts/0        00:00:00 ps
ssangkong@ssangkong-sslab:~/thread$ fg
./thread
second cleanup
first cleanup
first cleanup
```

→ Ctrl + z키를 누름. SIGSTOP Signal을 보냄.

→ LWP(Light-Weight Process) : Thread를 의미

→ fg명령어. SIGCONT Signal을 보냄.