# Project 1. System Call

*2019147029 산업공학과 & 컴퓨터과학과 조은기*

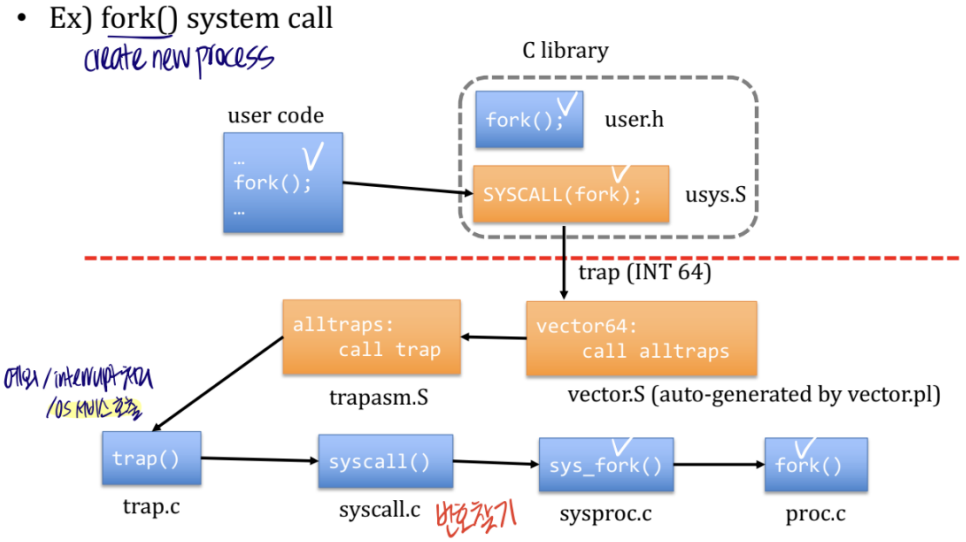
## 목표

***system call 중 ps, nice, yield 구현***

system call은 user mode에서 실행되던 application이 kernel의 권한으로 명령어를 실행하고 싶을 때 호출한다. system call이 발생하면, 실행중인 process는 cpu 상에서 user mode에서 kernel mode로 전환되어 플래그 레지스터에 반영되고, kernel mode의 privileged instructions를 실행할 수 있는 상태가 된다.

이번 과제에서 ps, nice는 직접 구현해야 하며, yield는 이미 구현된 코드를 user mode에서 호출할 수 있도록 인터페이스를 만들어주어야 한다.

## Entire System Call Flow



실질적인 system call의 동작 기능에 대한 코드는 `proc.c` 에 들어갈 것이다.

나머지 파일들은 유저 코드를 통해 받은 system call을 trap을 통해 커널 코드를 호출해, 요청이 `proc.c` 코드를 정확하게 호출할 수 있도록 연결해 주어야 한다.

# User Mode System Call Flow

## ✔️ Makefile

shell에 명령어를 입력했을 때, 해당 명령어를 파싱해서 올바른 로직과 매핑하려면 내가 구현한 명령어들을 Makefile에 추가해야 한다.

현재 개발할 내용들은 유저 프로그램이 될 것이기 때문에 `UPROGS` 부분에 명령어를 추가한다.

`Makefile`

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
    _swaptest\
    _ps\
    _yield\
    _nice\
```

*(맨아래 세줄)*

해당 파일에서 정의한 ps, yield, nice 명령어에 대해 각각이 호출되었을 때 실행될 c 파일을 생성했다.

## ✔️ ps.c & yield.c & nice.c

`ps.c`

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(int argc, char** argv)
{
    ps();
    exit();
}
```

`yield.c`

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int main(int argc, char** argv)
{
    yield();
    exit();
}
```

`nice.c`

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fcntl.h"

int
main(int argc, char *argv[]) {
    int value = atoi(argv[2]);
    nice(value);
    exit();
}
```

## ✔ user.h

- `user.h`: user mode의 기능 목록(method, system call 등등,,)을 정의한 헤더 파일

`user.h`

```
// system calls
int ps(void);
int yield(void);
int nice(int);
```

system call을 구현한 것이 때문에 system call 주석 아래에 새로 구현할 3가지 메서드를 추가했다.

## ✔ usys.S

- `usys.S`: 유저 프로그램에서 호출하는 각각의 system call을 운영 체제에서 정의한 시스템 콜을 호출하는 어셈블리어로 매핑하는 파일 (매핑 도구로 커널 모드의 코드인 `syscall.h`에서 정의한 system call number를 활용한다)

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```

위와 같이 커널 모드의 코드를 호출할 수 있는 어셈블리어로 매핑해 준다.

따라서 아래에 계속해서 나열된 SYSCALL에 내가 새롭게 정의할 3가지 system call을 추가해준다.

`usys.S`

```
SYSCALL(ps)
SYSCALL(yield)
SYSCALL(nice)
```

이제 커널 모드를 호출하게 되므로, 아래부터 커널 모드의 system call flow를 구성한다.

# Kernel Mode System Call Flow

## ✔ trap.c

유저 모드에서 시스템 호출을 하면 운영 체제는 해당 시스템 호출을 처리하기 위해 커널 모드로 전환된다. 이때 보통 `trap`이나 `interrupt` 메커니즘이 사용된다.

해당 파일에서 system call을 처리하는 부분은 여기다.

```
//PAGEBREAK: 41
void
trap(struct trapframe *tf)
{
```

```
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

trap의 코드가 SYSTEM CALL로 지정해놓은 코드와 동일하다면, 별 일이 없는 한 (프로세스가 곧바로 죽었다던가,,) `syscall()`을 호출한다.

이 `syscall()` 메서드는 `syscall.c` 파일에 정의되어 있다.

## ✔ syscall.h

system call의 코드를 정의하는 파일이다.

이미 있는 것들의 뒤에 오름차순으로 고유한 코드를 설정한다.

```
#define SYS_ps     24
#define SYS_yield  25
#define SYS_nice   26
```

## ✔ syscall.c

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();
```

```
  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

이 코드는 현재 실행 중인 프로세스가 요청한 시스템 콜을 실제로 처리하고, 그 결과를 `%eax` 레지스터에 반환해, 실질적으로 system call을 처리하는 부분이다.

좀 자세히 보면, if문을 통해서 유효한 시스템 콜인지(코드가 존재하는지 등등 검사,,) 검증한 후에 유효하다면, `curproc->tf->eax = syscalls[num]();` 를 통해 매핑된 커널 모드의 시스템 콜 메서드를 호출한다.

이때 사용되는 `syscalls[num]` 배열에 매핑되기 위해서는 내가 정의한 system call들도 모조리 추가해야만 한다.

`syscall.c`

```
static int (*syscalls[])(void) = {
 // ...
[SYS_ps]     sys_ps,
[SYS_yield]   sys_yield,
[SYS_nice]    sys_nice,
};
```

그리고, 살짝 위를 보면 여기서 매핑한 메서드들을 실제로 전역 메서드(`extern`)로 정의하고 있는데 거기에 맞춰서 3가지 메서드도 모두 추가해준다.

`syscall.c`

```
extern int sys_ps(void);
extern int sys_yield(void);
extern int sys_nice(void);
```

이제 `syscall.c` 파일에 정의된 함수들을 어딘가에는 실제로 구현해야 하는데, 그 파일이 `sysproc.c` 파일이다.

## ✔ sysproc.c

`syscall.c`에서 호출할 3가지 메서드를 모두 구현해 놓았다.

`sysproc.c`

```
int
sys_ps(void)
{
    return ps();
}

int
sys_yield(void)
{
    yield();
    return 0;
}

int
sys_nice(void)
{
    int value;
    if(argint(0, &value) < 0)
        return -1;

    return nice(value);
}
```

사실상 구체적인 구현 내용은 `proc.c`에 모두 들어있어서, 불필요한 Layer를 추가했다는 생각도 들었다. 아마도 이 파일이 존재하는 이유는, `syscall.c` 파일은 커널모드의 입장에서는 서버의 presentation layer과 비슷한 역할을 하기 때문에, system call을 정의한 인터페이스가 변화하더라도 구체적인 구현 내용인 `proc.c`의 내용에는 변경을 최소화하기 위한 것이라고 생각한다.

이제 남은건, `proc.c`에 정말로 구체적인 내용을 정의하는 일이다.

그치만 그에 앞서 `proc.c`에 구현할 메서드들을 미리 `defs.h` 파일에 메서드 시그니처만 정의해 주어야 한다. 메서드의 관리를 쉽게 하기 위해서 os 내부적으로 유저 모드의 메서드는 `user.h`에, 커널 모드의 메서드는 모두 `defs.h`에 모아두는 컨벤션이 있는 것 같다.

## ✔ defs.h

- `defs.h`: kernel mode의 기능 목록(method 등등,,)을 정의한 헤더 파일

세 메서드 모두 process 관리와 관련된 기능이기 때문에, proc.c라고 주석이 되어 있는 곳 아래에 선언했다.

`defs.h`

```
//PAGEBREAK: 16
// proc.c
void            yield(void);
int             ps(void);
int             nice(int value);
```

이제 정말로 `defs.h`를 include하는 proc.c 파일에 구체적인 내용을 구현만 하면 된다!

## ✔ nice 구현

nice는 현재 진행중인 프로세스의 우선순위를 변경하는 system call이다.

현재 PCB(Process Control Block)에는 우선순위와 관련된 내용이 없으므로, 정의해준다. 해당 내용은 `proc.h`에 정의되어 있다.

`proc.h`

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (byt
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for
  enum procstate state;        // Process state
  int pid;                     // Process ID
  int ticks;
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current sysc
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on ch
  int killed;                  // If non-zero, have been kill
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
  int priority;
};
```

위처럼 마지막 줄에 `priority`를 추가해 주었다.

`proc.c`

```
int
nice(int priority) {
    struct proc *curproc = myproc();

    if (curproc == 0) {
        return -1;
    }
```

```
    curproc->priority += priority;

    if (curproc->priority < -5) {
        curproc->priority = -5;
    }

    if (curproc->priority > 4) {
        curproc->priority = 4;
    }

    return curproc->priority;
}
```

- 현재 실행중인 프로세스의 priority를 수정하는 작업이기에, 현재 진행중인 프로세스가 없는 경우 예외로 보기로 하고 -1을 반환했다.
- 현재 프로세스의 priority 값에 파라미터로 받은 값을 합한 값을 priority로 업데이트했다.
    - 제약 조건에 따라서 -5 미만, 4 초과의 값에 대해서는 바운더리의 값으로 priority 값을 대체했다.
- 변경된 priority을 반환했다.

## ✔ ps 구현

마찬가지로, `defs.h` 에 정의한 ps는 **프로세스 관리와 관련된 기능**이기 때문에, `proc.c`에 구현했다.

`proc.c`

```
int state(enum procstate state) {
    switch (state) {
        case UNUSED:
            return 0;
        case EMBRYO:
            return 1;
        case SLEEPING:
```

```
            return 2;
        case RUNNABLE:
            return 3;
        case RUNNING:
            return 4;
        case ZOMBIE:
            return 5;
        default:
            return -1;
    }
}

int
ps() {
    struct proc *p;

    acquire(&ptable.lock);

    cprintf("name \t pid \t state \t nice \t ticks \t ticks: 
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        int stateNumber = state(p->state);
        if (stateNumber == 2 || stateNumber == 3 || stateNumb
            cprintf("%s \t %d \t %d \t %d \t %d \n ", p->name
        }
    }

    release(&ptable.lock);

    return 0;
}
```

- 전체 프로세스의 context switching 횟수를 의미하는 ticks는 `trap.c`에 정의되어 있는데, 해당 변수를 `defs.h`에서 전역변수(`extern`)으로 선언해주었기 때문에 그냥 사용했다.
- 나머지는 PCB에서 제공하는 정보로, `ptable`에 락을 취득한 뒤 읽기 작업을 통해, 필요한 정보를 모두 화면에 출력했다.

- 해당 과제에서는 state를 숫자로 표시하기에, 매핑하는 메서드를 별도로 생성해 주었다.