

# 운영체제(가) 3차 과제

일어일문학과

20181755 이건희

2023년 11월



# 1. Development Environment

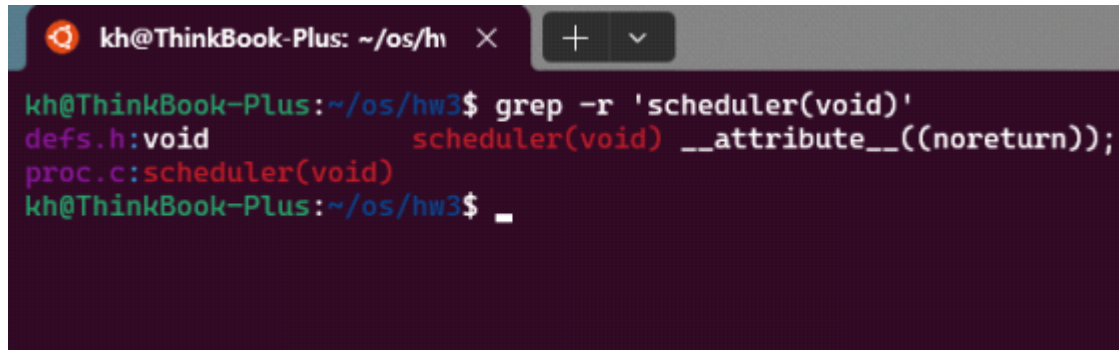
2차과제를 진행했을때와 동일하다.



<https://github.com/kitsune03k/Vim8.2CheatSheet>

다만 이제는 nano를 사용하지 않고 vim으로만 작업한다. 중간고사 기간에 공부가 안되고 심심해서 위의 자료를 만들었는데, 만들면서 vim에 대한 이해가 많이 늘었다. vim은 키들이 난해하다는 평들이 많은데, 이는 사람 입장에서나 난해한것이지 직접 하나하나 찾아보니 이유가 없는 것이 없었다. 일례로 ^, \$는 regular expression에서 가져온 것이다. 또한 모든 명령어는 숫자+키로 몇 번 반복실행할지 조합이 가능했다. 이런 뛰어난 에디터를 두고 nano 같은 것을 쓴 자신에게 부끄럽다. 본 과제를 진행하면서 vim, cat 이외의 방법으로는 절대 코드를 보지 않았다.

## 2. Scheduler?



```
kh@ThinkBook-Plus: ~/os/hw3$ grep -r 'scheduler(void)'
defs.h:void scheduler(void) __attribute__((noreturn));
proc.c:scheduler(void)
kh@ThinkBook-Plus:~/os/hw3$
```

`scheduler()`는 `proc.c`안에 있다. 이를 분석하기 위해서는 구조체 `proc`와 `cpu`의 원형이 필요한데, 이들은 `proc.h`에 정의되어있다. 코드 전문의 텍스트나 사진은 보고서 용량의 낭비기에 생략한다.

여하튼 이들을 참고하여 `scheduler()`를 사람이 알아들을 수 있는 말로 번역해보겠다.

```
=====
// cpu당 프로세스 스케줄러이다
// 각 cpu는 자신을 세팅한 이후 scheduler()를 불러온다
(참고로 struct cpu는 proc.h에 정의되어있다)
// 스케줄러는 절대 return하지 않는다, 무한반복하면서 아래와 같은것들을 한다
// - 실행할 프로세스를 선택한다
// - 그 선택한 프로세스를 실행하기 위해 swtch()한다
(swtch()가 이에 해당한다, 이는 C코드가 아닌 어셈블리 코드(.S)인데, swtch.S에 나와있다. 내용상으로 보았을
때는 컨텍스트 스위칭인데, 안타깝게도 필자는 어셈블리를 모른다.)
// - 결국 그 프로세스는 스케줄러에 대한 swtch()를 통해 스케줄러에 컨트롤을 다시 보낼 것이다
```

```
void scheduler(void)
{
    구조체 proc포인터 p;
    구조체 cpu포인터 c = mycpu();
```

`mycpu()`는 `proc.c`에 정의되어있다. 이 기능을 간단히 표현하자면, 구조체 `cpu` 스스로의 `apicid`를 가져와서 이를 `cpu[]`에서 찾아, `apicid`가 일치하는 구조체를 찾으면 이의 주소(`struct cpu*`)를 `return`하고, 없으면 `kernel panic`을 일으킨다.

xv6의 `make`과정을 유심히 보니 기본옵션으로는 2개의 `processor`를 가지게끔 `make`된다.

현재 `cpu`의 `proc`을 0으로 바꾼다;

```
// 프로세스 테이블을 계속 돌고돌아 실행가능한 프로세스를 찾는다
for 문 무한 루프{
    // 현재 프로세서에 인터럽트를 가능하게 한다
    sti(); (함수 원형은 asm volatile("sti");)
```

xv6의 소스코드에서 자주 보이는 cli, sti는 어셈블리어라  
필자가 겁을 먹었는데, 검색 결과에 의하면 이는 CLear  
Interrupt, SeT Interrupt라 한다

```
ptable을 lock하여 다른 함수가 접근하지 못하게 막는다;
for(proc포인터 p는 ptable.proc의 첫주소부터 ; p가 ptable.proc의 마지막까지; p++){
    (여하튼 ptable내의 proc에 대해 접근한다는 소리다)
    if(proc의 상태가 실행가능(RUNNABLE)하지 않을 때
        continue;
```

```
// 선택한 process로 교체한다
// ptable.lock을 release하고, jumping back 하기전에 다시 acquire하는 것은
// 프로세스가 해야 할 일이다
```

```
(이렇게 패스 되다보면, 실행가능한 상태의 process가 ptable.proc에 있을 것이다)
cpu의 proc은 runnable한 process의 주소이다;
user virtual memory를 switch한다;
p의 상태를 실행중(RUNNING)으로 바꾼다;
```

```
cpu의->scheduler의 주소, runnable한 process의 context)간 Context Switch한다;
(proc.h에 struct proc의 구성원중 context의 주석을 보면 swtch() here to run process)
kernel virtual memory를 바꾼다;
```

```
// 프로세스의 동작이 지금 끝났다
// 여기로 돌아오기전에 실행했던 process의 상태가 바뀌어있을 것이다
```

이 주석의 이해가 확실치 않아서, proc.c의 yield()를 보니, 상태가  
running에서 runnable로 바뀌고, sched()로 다시 스케줄러로 컨텍스트  
스위칭이 된다. 이런 경우가 해당 주석이 뜻하는 바이다.

```
c->proc = 0;
}
ptable의 lock을 해제한다;
}
}
```

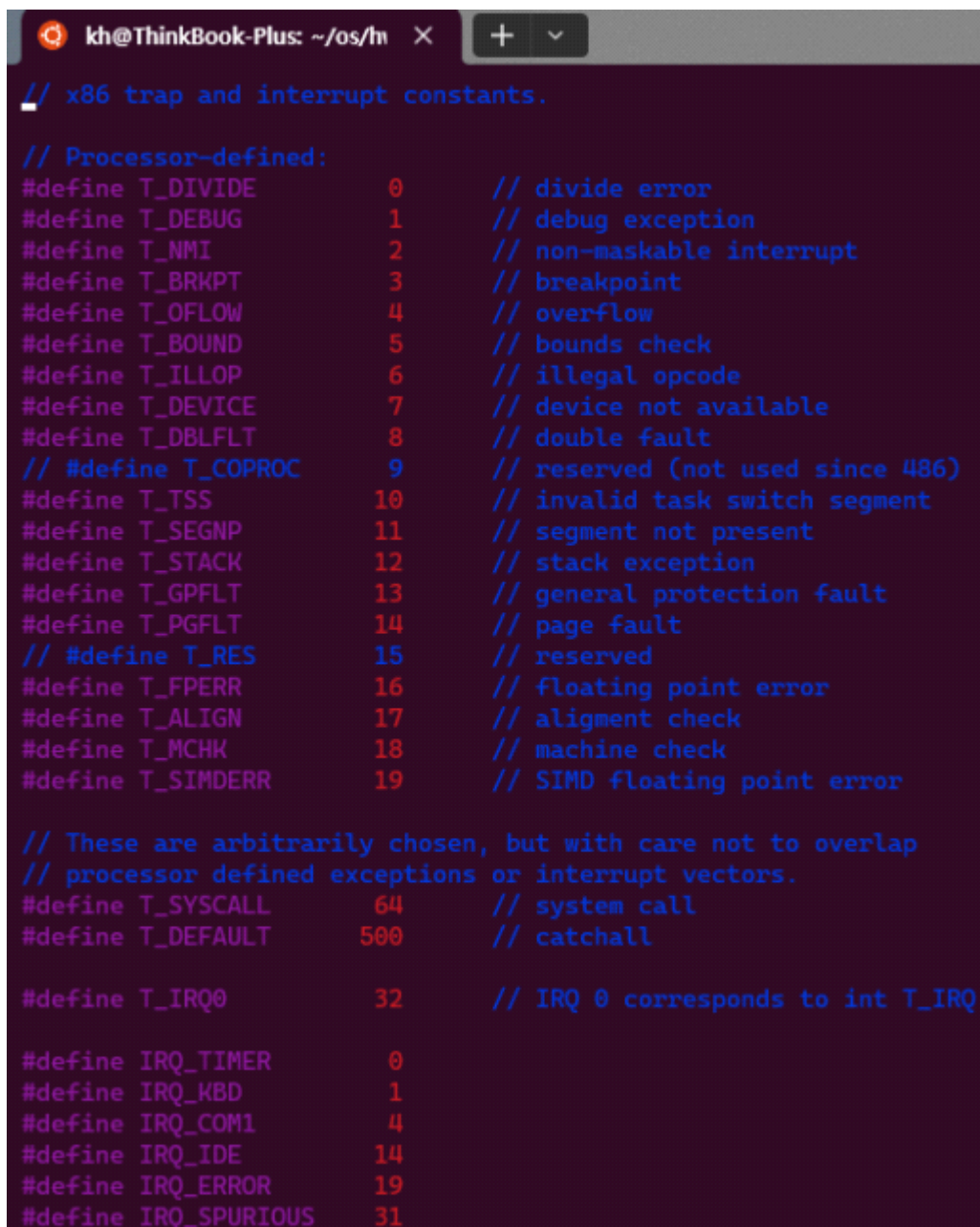
=====

스케줄러의 코드만 보면 RR이라 받아들이기가 힘들어서, 시간 관련된 내용을 찾아보았다. xv6은 'tick' 이라는 개념을 사용하는데, 이는 이 한 tick 은 10ms임을 현실의 시계와 같이 uptime syscall을 사용한 uptimetest.c 로 알아냈다.

```
trap.c: struct spinlock tickslock;
trap.c: uint ticks;
trap.c: initlock(&tickslock, "time");
trap.c: acquire(&tickslock);
trap.c: ticks++;
trap.c: wakeup(&ticks);
trap.c: release(&tickslock);
trap.c: // Force process to give up CPU on clock tick.
grep: trap.o: binary file matches
grep: sysproc.o: binary file matches
kh@ThinkBook-Plus: ~/os/hw1$
```

grep으로 찾아보니 trap.c에서 tick 관련한 내용을 다루고 있다. 이를 자세히 살펴보겠다.

trap.c의 내용을 보니 traps.h의 내용을 먼저 신는게 맞을 것 같아서 아래의 사진을 먼저 넣었다.



```
// x86 trap and interrupt constants.

// Processor-defined:
#define T_DIVIDE 0 // divide error
#define T_DEBUG 1 // debug exception
#define T_NMI 2 // non-maskable interrupt
#define T_BRKPT 3 // breakpoint
#define T_OFLOW 4 // overflow
#define T_BOUND 5 // bounds check
#define T_ILLOP 6 // illegal opcode
#define T_DEVICE 7 // device not available
#define T_DBLFLT 8 // double fault
// #define T_COPROC 9 // reserved (not used since 486)
#define T_TSS 10 // invalid task switch segment
#define T_SEGNP 11 // segment not present
#define T_STACK 12 // stack exception
#define T_GPFLT 13 // general protection fault
#define T_PGFLT 14 // page fault
// #define T_RES 15 // reserved
#define T_FPERR 16 // floating point error
#define T_ALIGN 17 // alignment check
#define T_MCHK 18 // machine check
#define T_SIMDERR 19 // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL 64 // system call
#define T_DEFAULT 500 // catchall

#define T_IRQ0 32 // IRQ 0 corresponds to int T_IRQ

#define IRQ_TIMER 0
#define IRQ_KBD 1
#define IRQ_COM1 4
#define IRQ_IDE 14
#define IRQ_ERROR 19
#define IRQ_SPURIOUS 31
```

위 내용 중 IRQ에 관심이 생겨 검색을 해본 결과

- x86부터 intel의 8259 pic(Programmable Interrupt Controller)가 두 개 사용되었다.<sup>1)</sup>
- 첫 pic가 master, 두 번째 pic가 slave이며, 첫번째 pic만 cpu에 직접적인 신호를 보낼 수 있다.
- 이보다 더 발전된 것은 apic(advanced pic)이며, p54c(pentium mmx)부터 cpu에 내장되었다<sup>2)</sup>.

더 자세한 정보는 주석으로 달아놓은 위키백과 링크를 참고하면 된다.

이제 trap.c를 중요한 부분만 보겠다.

=====

...

```
void trap(struct trapframe *tf){
if(tf->trapno == 64일 때 (= t_syscall일 때)){
    현재 프로세스가 죽었으면 exit;
    현재 프로세스->tf = 매개변수 tf;
    syscall();
```

syscall()은 syscall.c에 정의되어있는데, 이의 내용은 아래와 같다 1. 현재 proc->tf->eax 레지스터에서 num을 가져온다 2. num에 해당하는 syscall의 함수의 주소를 eax 레지스터에 넣어준다 (syscall시 컨텍스트 스위칭)
---

```
    현재 프로세스가 죽었으면 exit;
    return;
}
```

(아래 스위치는 trapno가 64가 아닌 경우이다)

```
switch(tf->trapno){
    32+0 = 32일 때(= 타이머 인터럽트일 때):
        cpuid가 0이면
            ticks++;
            wakeup(&ticks);
    32+14 = 46일 때(= IDE(저장장치) 슬롯 0 인터럽트일 때): ...
    32+14+1 = 47일 때(= IDE(저장장치) 슬롯 1 인터럽트일 때): ...
    32+1 = 33일 때(= 키보드 인터럽트일 때): ...
    32+4 = 36일 때(= 시리얼 통신 인터럽트일 때): ...
    32+7일 때(= line print 시리얼 인터럽트일 때(프린터 관련)): 아무것도 안함 (당연히 xv6는 교육용 os니..)
    32+31일 때(= 거짓 인터럽트일 때): ...
```

---

1) [https://en.wikipedia.org/wiki/Interrupt\\_request](https://en.wikipedia.org/wiki/Interrupt_request)

2) [https://en.wikipedia.org/wiki/Advanced\\_Programmable\\_Interrupt\\_Controller](https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller)

switch의 case들은 공통적으로 lapiceoi()를 실행하고 break된다.  
lapiceoi()의 원형은 lapic.c에 정의되어있다.

lapic.c는 처음의 주석에 스스로를 다음과 같이 소개한다:  
local APIC는 non-io 인터럽트를 관리한다

이의 내용을 간단히 살펴보면 lapiceoi는 lapicw(EOI,0)을 실행한다.  
lapicw(위치, 값);은 lapic[위치] = 값; 이다  
(uint\* lapic는 mp.c에서 초기화되었다는 주석이 있다)

EOI는 End of Interrupt의 약자로써, pic에게 인터럽트 처리가 완료되었음을 알리는 신호이다. lapic.c에서 EOI는  
#define EOI (0x00B0/4) // EOI  
와 같이 정의되어있고, 0xB0/4 = 2C이다.

종합해서, lapiceoi()는 메모리주소중 2C의 값을 0으로 바꿔준다. 결과적으로 다음 인터럽트를 받을 준비가 되었음을 알려준다.

default:

...  
예외처리  
...

if(현재 프로세스가 죽은상태이고, tf->cs와 3을 비트연산한 값이 DPL\_USER이면) exit();

DPL\_USER = 3임이 mmu.h에 define되어있다.  
DPL은 Description Privilege Level로써 (ia-32 기준) 4가지의 ring이 있는데, ring 3는 user mode이다.  
따라서 tf->cs가 usermode이면 trap함수를 exit()한다고 이해하면 된다.

if(현재 프로세스가 작동중인 상태이고, tf->trapno가 32+0이면 = timer interrupt이면) yield();

if(현재 프로세스가 죽은상태이고, tf->cs와 3을 비트연산한 값이 DPL\_USER이면) exit();

}

=====



여기까지 대강 trap.c를 관찰하고나서, 궁금한 부분들에 대해 더 자세히 보자.

첫 번째로, trap은 언제 실행되는지이다.

```
traprunning          1 1 512
traprunning          traprunning
traprunning          ..          1 1 512
traprunning          README      2 2 2286
traprunning          cat          2 3 15520
traprunning          echtraprunning
1 traprunning          o          2 4 14404
traprunning          forktest     2 5 8848
traprunning          grep          2 6 18364
traprunning          init          2 7 15024
traprunning          traprunning
5 traprunning          kill          2 8 14488
traprunning          ln            2 9 14384
traprunning          ls            2 10 16952
traprunning          traprunning
traprunning          mkdir          2 11 14512
traprunning          rm            2 12 14492
traprunning          sh            2 13 28548
traprunning          strestraprunning
traprunning          sfs           2 14 15420
traprunning          usertests     2 15 62920
traprunning          wc            2 16 15948
traprunning          traprunning
traprunning          zombie        2 17 14068
traprunning          newsyscalltest 2 18 14628
.          1 1 512          uptimetest 2 1traprunning
traprunning          9 14408
```

trap에 `cprint("traprunning");` 한 줄을 추가하고, xv6를 실행하니 끊임없이 tick마다 호출되는 것을 확인했다. 필자는 어셈블리를 몰라서 정확한 원리는 솔직히 아직도 모르겠으나 xv6는 RR방식의 스케줄러를 가지고 있다 할 수 있겠다.

두 번째로, switch문 중 `case 32+0(= 타이머 인터럽트)의 cpuid == 0`일 경우이다. 이 경우에 `ticks++`, `wakeup(&ticks)`를 실행한다. `wakeup` 함수는 `proc.c`에 정의되어있는데, `ptable`에서 `sleeping`이고, `p->chan`이 입력받은 `chan`인 프로세스를 `runnable`로 만들어준다. 이 `chan`은 `channel`같은데, 이 역시나 정확하게 뭔지를 모르겠다.

세 번째로 `yield`가 정확히 무엇을 하는지다. `yield()`의 원형은 `proc.c`에 있고, 이의 내용은 아래와 같다.

ptable의 락을 얻는다;  
RUNNING 상태의 proc의 상태를 RUNNABLE 상태로 바꾼다;  
sched()함수를 실행한다;  
얻었던 ptable의 락을 해제한다;

중간에 실행되는 `sched()`의 원형은 `proc.c`의 `yield()`의 바로 위에 있는데, 이의 내용은 아래와 같다.

```
=====
// 스케줄러에 입장한다. ptable의 락만 가지고있어야하며, proc의 상태는 바뀌었어야 한다.
(yield()에서 sched()이전에 둘 다 실행됨)
// intena를 저장하고 복구하는데, 그 이유는 intena는 이 cpu가 아닌 이 커널 쓰레드의 속성이기 때문이다.
(intena는 struct cpu의 멤버로 있다.)
// 저장되고 복구되는 것은 proc->intena나, proc->ncli여야 하는데,
// 이는 락이 걸린 몇군데에서 깨질수도 있지만 여기에는 프로세스가 없다(따라서 상관없다).
```

```
void
sched(void)
{
int intena;
struct proc *p = myproc;

if(lock을 가지고있지 않을 때) panic;
if(ncli가 1이 아닐 때) panic;
if(p->state가 1이 아닐 때) panic;
if(readeflags()와 FL_IF의 비트and연산의 값이 참일 때) panic;

intena = 현재 cpu의 intena;
swtch(p->context의 주소, 현재cpu->스케줄러의 주소)
현재cpu의 intena = intena;
```

intena는 interrupt enable의 줄임말이다. 0은 거짓이고, 1은 참이다.  
 2-1.에서 보았겠지만, sti는 인터럽트 설정, cli는 인터럽트 해제이다.  
 예를 들어, cli 이후에는 intena = 0이 되어야 할 것이다.

```
=====

yield와 sched에서, 프로세스를 runnable 상태로 만들고, 프로세스에서 스케줄러로 컨텍스트 스위칭 함을 알 수 있다.
```

어셈블리어를 모르니, 존재 자체는 아나 존재 이유를 모르는 상황이다. 본 보고서를 쓰면서 수박 겉핥기식으로 알고 쓰는게 참 찝찝한 것이 사실이다. 그럼에도 불구하고 trap.c에서 얻어가는것이 없는건 아니였기에 할 가치는 충분했다 생각한다.

### 3. Add the concept of “priority” to xv6

3-1.

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

-- INSERT --
```

struct proc에 int priority를 멤버로 추가해주었다.

3-2.

```
int
fork(void)
{
    int i, pid;
    struct proc *np;
    struct proc *curproc = myproc();

    // Allocate process.
    if((np = allocproc()) == 0){
        return -1;
    }

    // Copy process state from proc.
    if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
        kfree(np->kstack);
        np->kstack = 0;
        np->state = UNUSED;
        return -1;
    }
    np->sz = curproc->sz;
    np->parent = curproc;
    *np->tf = *curproc->tf;
    np->priority = curproc->priority;
}
```

fork()에 child의 priority는 parent의 priority와 같은 값을 가지게끔 수정해주었다.

```
kh@ThinkBook-Plus:~/os/hw3$ vi proc.h
kh@ThinkBook-Plus:~/os/hw3$ make qemu
qemu-system-i386 -serial mon:stdio -dr
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 r
init: starting sh
$ _
```

혹시나 오류가 생길까봐 여기까지 진행 후 make를 해보았는데, 이정도 추가하는거로는 문제가 되지 않는 것을 확인했다.

### 3-3.

본 단계를 진행 전에 먼저 allocproc, userinit 함수를 보는 것이 맞을거 같아 이의 원형을 관찰하겠다. allocproc()과 userinit() 둘 다 proc.c에 정의되어있는데, userinit에서 allocproc이 사용되는 것을 볼 수 있었다. 해당 함수들을 위에서처럼 번역해보겠다.

```
=====
static struct proc* allocproc(void)
{
    struct proc* p;
    char* sp;

    ptable의 락을 얻는다

    for(p = ptable을 처음부터, 마지막 프로세스까지 살펴보는데){
        상태가 한번도 사용되지않음(UNUSED)인 프로세스를 찾으면
            found로 간다
    }

    (found로 가지못한 상황은 UNUSED 상태의 process가 없는 상황이다)
    ptable의 락을 해제한다
    return 0;

    found: (여기서 다루는 p는 위의 for문에서 찾은 한번도 사용되지 않은 프로세스이다)
    프로세스 상태는 배아(EMBRYO) 상태이다;
    프로세스의 pid는 nextpid++이다; (nextpid는 1로 시작함이 proc.c의 처음에 정의되어있다)

    ptable의 락을 해제한다;

    // 커널 스택을 할당한다
    if(p의 커널스택을 kalloc으로 할당에 실패하면){
        프로세스의 상태를 다시 한번도 사용되지 않음(UNUSED)으로 바꾼다;
        return 0;
    }
    (여기서부터는 p의 커널스택이 kalloc으로 정상적으로 할당 된 경우이다)
    sp는 프로세스의 커널스택 + 커널스택사이즈이다;
    (다시말해, sp는 프로세스의 커널스택이 끝나는 그 지점을 가리킨다)

    // 트랩프레임을 위한 공간을 둔다
    sp -= sizeof(프로세스의 트랩프레임);
    p의 트랩프레임 = sp;
```

// traplet에 return할, forklet에서 실행되는 새로운 컨텍스트를 설정한다

sp -= 4;

(여기서 한가지 추측하자면, sp는 한 바이트 단위씩 접근해야하기 때문에 char\*형으로 선언한 것이다. sp가 가리키는 내용을 접근할 때 마다 casting으로 접근하는 것을 보고 약한 추측을 했는데, 여기서 숫자단위로 더하고 빼주는 것을 보고 확실해졌다.)

(uint\*) sp의 내용 = (uint)trapret;

sp -= sizeof(프로세스의 컨텍스트);

프로세스의 컨텍스트는 (context\*)sp;

메모리초기화(컨텍스트, 0, sizeof(컨텍스트가 가리키는 내용));

프로세스의 컨텍스트의 eip레지스터는 (uint)forkret;

return p; (태아상태로 바뀐 프로세스의 시작주소)

=====

요컨대 allocproc()은 ptable에서 unused상태의 프로세스에 커널스택을 할당해주고, 트랩프레임을 지정해주고, 배아상태로 만들어주는 것이다.

userinit()은 중요한 부분이 눈에 확 들어와서, 생략할 것은 어느정도 생략하고 지나가겠다.

=====

// 첫 유저 프로세스를 설정한다.

void

userinit(void)

{

struct proc\* p;

p = allocproc();

(p는 embryo 상태일 것이다)

initproc = p;

(initproc은 proc.c의 초반에 static struct proc\* initproc;으로 정의되어있다.)

if(p의 페이지주소를 setupkvm()으로 할당한 것이 오류가 날 때)

    패닉(out of memory?);

유저버추얼메모리init(...);

...

프로세스의 트랩프레임 지정 관련

...

페이지테이블의 락을 얻는다;

p의 상태는 runnable이다;

페이지테이블의 락을 해제한다;

}

=====

여하튼 새로운 프로세스는 allocproc()와 userinit()을 통해 unused->embryo->runnable 상태가 되었고, 스케줄러에서 runnable->running 상태가 될 것이다.

```
allocproc(void)
{
    struct proc *p;
    char *sp;

    acquire(&ptable.lock);

    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
        if(p->state == UNUSED)
            goto found;

    release(&ptable.lock);
    return 0;

found:
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->priority = 5;

    release(&ptable.lock);
}
```

따라서 프로세스의 상태가 unused->embryo가 되는 allocproc()에서 p.priority를 5로 지정해주었다.

### 3-4. get\_proc\_priority(이하 get\_), set\_proc\_priority(이하 set\_)

과제2에서 새로운 syscall을 만든 것처럼 새로운 syscall 2개를 추가해주어야 하는데, 그 전에 먼저 두 syscall 함수의 원형을 생각해보았다. 먼저 get\_은 ptable에서 pid가 일치하는 proc을 찾아 이의 priority를 return해야 할 것이다. 따라서 get\_의 원형은 “int get\_proc\_priority(int pid)”가 되겠다. 다음으로 set\_은 ptable에서 pid가 일치하는 proc을 찾아 이의 priority를 이를 매개변수로 받은 값으로 수정해야 할 것이다. return값은 다음에 get\_을 또 실행하든, set\_안에 get\_을 실행하든 하면 되기 때문에 set\_의 return값은 void 여도 된다. 뭐 취향껏 제대로 변경이 되었음을 알리는 return값이 있어도 상관은 없으나 개인적으로는 별 필요가 없어보인다. 따라서 필자가 생각한 set\_의 원형은 “void set\_proc\_priority(int pid, int newpriority)”가 되겠다.

```
int
get_proc_priority(int pid)
{
    int priority;
    struct proc* p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            priority = p->priority;
            release(&ptable.lock);
            return priority;
        }
    }
    return 0; // 오류시
}

void
set_proc_priority(int pid, int newpriority)
{
    struct proc* p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            p->priority = newpriority;
            release(&ptable.lock);
            return;
        }
    }
}
```

이를 함수로 구현하면 위 사진과 같다.

추가적으로, 해당 두 syscall 함수의 원형을 정의한 proc.c에는 myproc()이라는 함수도 정의되어있다. 현재 프로세스의 주소를 return해주는 함수이다. 만약 명세에 매개변수 pid를 받지 않는다 하면 이를 이용했을 것이다.

그리고 과제2와 똑같이 새로운 syscall을 추가해주었다. 과정에서 wrapping 함수를 추가하는데 문제가 있어서, 이를 해결한 과정을 Troubleshooting의 5-1.에 적어두었다.

```

h0@ThinkBook-Plus: ~/os/hw1$ vi proc.c
h0@ThinkBook-Plus: ~/os/hw1$ make qemu
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o console.o console.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o exec.o exec.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o fs.o fs.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o ide.o ide.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o main.o main.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o mp.o mp.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o pipe.o pipe.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o proc.o proc.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o sleeplock.o sleeplock.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o spinlock.o spinlock.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o syscall.o syscall.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o sysfile.o sysfile.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o sysproc.o sysproc.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o trap.o trap.c
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o uart.o uart.c
ld -m elf_i386 -T kernel.ld -o kernel entry.o bio.o console.o exec.o file.o fs.o ide.o ioapic.o kalloc.o kbd.o lapic.o log.o main.o mp.o picirq.o pipe.o proc.o sleeplock.o spinlock.o
pass.o trap.o uart.o vectors.o vm.o -b binary initcode entryother
objdump -S kernel > kernel.asm
objdump -t kernel | sed '1,/SYMBOL TABLE/d; s/ . * / /; /$$/d' > kernel.sym
dd if=/dev/zero of=xv6.img count=10000
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB, 4.9 MiB) copied, 0.0112785 s, 454 MB/s
dd if=bootblock of=xv6.img conv=notrunc
1+0 records in
1+0 records out
512 bytes copied, 0.0017658 s, 290 kB/s
dd if=kernel of=xv6.img seek=1 conv=notrunc
396+1 records in
396+1 records out
202768 bytes (203 kB, 198 MiB) copied, 0.0003036 s, 668 MB/s
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ _

```

여기까지 진행하고 make를 하니 오류 없이 성공하였고, 이제 syscall을 테스트하는 프로그램을 만들어보겠다.



### 3-5. newsyscalltest.c

```
kh@ThinkBook-Plus: ~/os/m × + v
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void){
    int pid, priority1, priority2;
    pid = getpid();
    printf(1, "current process's pid : %d\n", pid);

    priority1 = get_proc_priority(pid);
    printf(1, "before set priority of pid(%d) = %d\n", pid, priority1);

    set_proc_priority(pid, 10);
    priority2 = get_proc_priority(pid);
    printf(1, "after set priority of pid(%d) = %d\n", pid, priority2);

    exit();
}
```

helloworld를 만드는 1주차 과제처럼 만들었다. set\_이 자꾸 오작동을 하여 지금까지 수정한 파일들을 전부 살펴보니 wrapping 함수의 내용에 문제가 있었다. 이를 해결한 과정을 Troubleshooting의 5-2.에 적어두었다.

```
$ newsyscalltest
current process's pid : 4
before set priority of pid(4) = 5
newpriority = 10
after set priority of pid(4) = 10
$ newsyscalltest
current process's pid : 5
before set priority of pid(5) = 5
newpriority = 10
after set priority of pid(5) = 10
$ newsyscalltest
current process's pid : 6
before set priority of pid(6) = 5
newpriority = 10
after set priority of pid(6) = 10
$ _
```

새로 만든 두 개의 syscall이 의도한 대로 잘 작동을 한다. 작동 확인 이후에는 디버그 코드를 지웠다.

## 4. New scheduler (1)

과제의 명세에는 ‘선점형 방식’의 ‘우선순위’ 스케줄러를 제작하라 되어있다. 거기에 더해 모든 프로세스가 scheduler 함수에서 선택되어 실행한 횟수를 기억하고, 이를 통해 starvation이 절대 발생하면 안된다 한다.

스케줄러를 만들면서 다음과 같은 생각을 해볼 수 있다.

1. priority가 높다는건 단순히 우선순위가 높다는 것 외에 어떤 의미가 있을까?

-> process가 시작부터 우선순위가 높을수도 있지만, aging에 의해 우선순위가 높아질 수도 있다. 전자의 경우에는 사용자가 계속 높은 우선순위로 돌아가기를 바랄 것이다. 후자의 경우에는 일반 우선순위로 실행됐다가, 이게 해당 실행에서 전부 끝나지를 않으니 점점 우선순위를 높혀가면서 끝내려는 느낌이다.

priority의 시작을 5로 하니, 1~5는 process가 일반적으로 움직일 수 있는 priority, 6~10은 background process같은 중요하지 않은 process가 사용하게끔 만들겠다.

2. priority가 고정될 필요가 있는 프로세스도 있지 않을까?

```
struct file *ofile[NFILE];
struct inode *cwd;
char name[16];
int priority;
int isconst;
int runcount[10];
};
```

-> struct proc에 isconst라는 멤버를 하나 추가했다.

```
int
get_proc_isconst(int pid){
    int isconst;
    struct proc* p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            isconst = p->isconst;
            release(&ptable.lock);
            return isconst;
        }
    }
    return -1; // 오류시
}

void
set_proc_isconst(int pid, int newisconst){
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->pid == pid){
            p->isconst = newisconst;
            release(&ptable.lock);
            return;
        }
    }
}
```

이 flag를 수정하고, 확인하는 syscall들 역시나 만들어야한다. proc.c에 get\_proc\_isconst, set\_proc\_isconst로 구현했으며, 작동방식은 위에서 만든 priority 관련 syscall과 동일하다.

```

np->sz = curproc->sz;
np->parent = curproc;
*np->tf = *curproc->tf;
np->priority = curproc->priority;
np->isconst = curproc->isconst;
for(int x=0; x<10; x++){
    np->runcount[x] = 0;
}

```

fork의 경우 child는 새로운 프로세스이고, 부모의 우선순위를 가져가야 할 것 같아서 isconst를 parent로부터 가져오게끔 했다. 또한, 부모의 실행횟수를 알 필요가 없기에 runcount[]를 0으로 초기화했다.

```

found:
p->state = EMBRYO;
p->pid = nextpid++;
p->priority = 5;
p->isconst = 0;
for(int x=0; x<10; x++){
    p->runcount[x] = 0;
}
release(&ptable.lock);

```

allocproc의 경우, 프로세스의 isconst를 0(거짓)으로 시작하게끔 만들었다. 사용자가 필요하다 하면 set\_proc\_isconst(1)(참)을 이용하여 스케줄러가 priority를 처리하지 않게끔 만들 수 있다. 또한, 아직 시작도 안한 상태의 프로세스기에 runcount[]를 0으로 초기화했다.

1.에서 시작부터 우선순위가 높은 경우는 사용자가 직접 isconst를 1로 설정해줘야한다.

3. proc이 어느 정도의 실행을 하고나서 priority를 올려줘야할까?

-> 사실 이는 설정하기 나름이다. os가 무당도 아니고, 어떤 프로세스가 얼마나 오래걸릴지 정확하게 아는 것은 불가능하다. 어떻게 보면 숫자 끼워맞추기일수도 있지만, 이때문에 필자의 경우에는 테스트 프로그램들의 실행 시간을 priority의 큐가 다룰만하게끔 맞출 것이다. 시스템마다 주로 하는 작업의 크기가 다르다면 시스템마다 priority를 올려주는 스케줄러의 threshold도 달라져야한다. 일단은 100번 실행되면 priority가 1만큼 높아지게 (p->priority--) 설정했다. 다음장의 관찰때는 달라질 수도 있음을 미리 알린다.

```

void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;
    for(;;){
        sti();

        // Loop over process table looking for process to run.

        for(int i=1; i<=10; i++){
            acquire(&ptable.lock);
            // high priority일 수록 많이 실행해준다.
            for(int j=10; j>=i; j--){
                for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
                    //
                    if(p->state != RUNNABLE || p->priority != i)
                        continue;

                    // Switch to chosen process. It is the process's job
                    // to release ptable.lock and then reacquire it
                    // before jumping back to us.
                    c->proc = p;
                    switchuvm(p);
                    p->state = RUNNING;

                    swtch(&(c->scheduler), p->context);
                    switchkvm();

                    prioritymanager(p);

                    // Process is done running for now.
                    // It should have changed its p->state before coming back.
                    c->proc = 0;
                }
            }
            release(&ptable.lock);
        }
    }
}

```

```

void
prioritymanager(struct proc *p)
{
    int idx = p->priority-1;
    // idx + 1 = priority
    p->runcount[idx]++;
    if(p->runcount[idx] >= 100){
        p->runcount[idx] = 0;
        // isconst = true의 경우 아래의 priority 조작을 하지 않고, 여기서 그냥 끝낸다
        if(p->isconst == 1)
            return;

        // priority가 2-10이면 우선순위를 올려준다
        if(idx > 0)
            p->priority--;
    }
}

```

그렇게 해서 완성된 스케줄러 함수와, priority를 규칙에 맞춰 수정하는 함수이다.

## 5. New scheduler (2)

```
kh@ThinkBook-Plus: ~/os/mn × + v
#define MAXCPROC 10
#define LASTPID (mainpid+MAXCPROC)

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

int main(void) {
    //int mainpid = getpid();
    int p;
    for (int i = 0; i < MAXCPROC; i++) {
        p = fork();
        // child
        if (p == 0) {
            int cpid = getpid();

            set_proc_priority(cpid, i%10+1);

            int pri = get_proc_priority(cpid);
            printf(1, "child %d created with priority(%d)!\n", cpid, pri);

            int t_start = uptime();
            int t_end, t_elipsed;
            while(1){
                int newpri = get_proc_priority(cpid);
                if(newpri != pri){
                    t_end = uptime();
                    t_elipsed = t_end-t_start;
                    printf(1, "child %d priority changed from %d to %d in %d ticks\n", cpid, pri, newpri, t_elipsed);
                    t_start = t_end;
                }
                pri = newpri;
            }
        }
    }
    while(1); // parent never end
}
```

테스트 프로그램은 이렇게 생겼다. 이를 실행한 결과는 출력전환하여 result.txt에 저장하였다.

과제의 명세에 starvation이 발생하지 않아야 한다 되어있다. 이 starvation의 기준이 궁금하다. 적당한 시간 내에 실행이 되지 않는 것을 starvation 기준이라 하면, 필자의 스케줄러는 starvation이 존재한다. 하지만 절대로 실행되지 않는 것을 기준이라 하면, 필자의 스케줄러는 starvation이 존재하지 않는다. result.txt를 보면 알겠지만, 시간이 약인것처럼 언젠가는 결국 실행되기 때문이다. 이 적당하다는 기준은 무엇인가? 이 애매한 starvation의 기준은 어떤 시스템이 어떤 사양을 가지고있는지, 주로 어떤 용도로 쓰이는지 등에 따라 os가 능동적으로 정해줘야 한다.

사실 테스트 프로그램같은 경우, 새로운 child 프로세스 10개 추가 이후 다른 프로세스의 추가가 없는 실험 상황에서 starvation이 없는 것이 확인된것이며, 실험 상황이 아닌 널리 사용되는 실제 os라 하면 계속 높은 priority의 프로세스들만 추가되는 상황이 없다할 수 없다. 따라서 필자가 만든 스케줄러는 절대 타인이 쓰는 os에 들어가면 안되는 멍청한 스케줄러이다.

이 스케줄러의 단점은, priority가 올라가는 것이 정말 느려서 낮은 priority를 가진 process의 priority를 급격하게 올려줘야 하는 상황에 대응을 전혀 못한다는 것이다. 스케줄러를 보면, priority가 1일 때 ptable을 10번 탐색, 2일 때 9번 탐색, ..., 10일 때 1번 탐색한다. 지금처럼 각 priority별로 실행중인 프로세스가 하나밖에 없는 경우 한 사이클은 프로세스 55번 실행이라 할 수 있는데( $10+9+\dots+1$ ), pid(4)는 10번, pid(5)는 9번, ..., pid(13)은 1번 실행되는 것이다. 55번중에 1번 실행이면 한 사이클 내 약 1.8%의 비율로 pid(13)이 실행되는 것이다. p(13)이 급격한 실행이 필요한데, 이를 위해서는 ptable을 54번 탐색하고 난 뒤에야 pid(13)을 위한 ptable 탐색이 시작되고, pid(13)으로 context switching이 되었다 한들 주어진 time quantum은 10ms(1tick) 뿐이다. priority 조작과 관련한 함수는 prioritymanager() 이외에는 존재하지 않는데, 해당 함수의 작동원리 상 대처가 상당히 느리다. 따라서 이 단점을 해결하기 위해서는 prioritymanager()을 override하는 priority를 빠르게 조작 가능한 다른 함수(혹은 syscall)이 필요하다.

새로운 함수를 추가하지 않는 다른 방법들은 아래와 같다.

1. 필자가 사용하는 구형 레노버 노트북을 사양이 좋은 최신식으로 바꾼다.
2. Makefile을 수정하여 qemu를 돌리는 코어의 개수를 늘린다.
3. priority의 depth를 낮춘다.
4. depth를 유지한다면, 두 번째 for문( $j=\dots$ )의 숫자를 잘 조정하여 한 사이클 내 실행 비율의 격차를 줄인다.

## 6. Trouble Shooting

### 6-1. wrapping 함수 (1)

```
sysproc.c:97:10: error: 'pid' is used uninitialized [-Werror=uninitialized]
  97 |     return get_proc_priority(pid);
      |
sysproc.c: In function 'sys_set_proc_priority':
sysproc.c:105:10: error: 'pid' is used uninitialized [-Werror=uninitialized]
 105 |     return set_proc_priority(pid, newpriority);
      |
sysproc.c:105:10: error: 'newpriority' is used uninitialized [-Werror=uninitialized]
cc1: all warnings being treated as errors
make: *** [builtin]: sysproc.o] Error 1
kh@ThinkBook-Plus:~/os/hw3$ make qemu-nox
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o sysproc.o sysproc.c
sysproc.c:97:10: error: 'pid' is used uninitialized [-Werror=uninitialized]
  97 |     return get_proc_priority(pid);
      |
sysproc.c: In function 'sys_set_proc_priority':
sysproc.c:105:10: error: 'pid' is used uninitialized [-Werror=uninitialized]
 105 |     return set_proc_priority(pid, newpriority);
      |
sysproc.c:105:10: error: 'newpriority' is used uninitialized [-Werror=uninitialized]
cc1: all warnings being treated as errors
make: *** [builtin]: sysproc.o] Error 1
kh@ThinkBook-Plus:~/os/hw3$

int
sys_get_proc_priority(void)
{
    int pid;
    return get_proc_priority(pid);
}

void
sys_set_proc_priority(void)
{
    int pid;
    int newpriority;
    return set_proc_priority(pid, newpriority);
}

~
~
"sysproc.c" 106L, 1294B written
```

과제2처럼 syscall의 wrapping 함수를 짰는데, make시 변수를 초기화하지 않고 사용되었다고 경고가 나왔다. 문제 해결을 위해, syscall을 위한 wrapping 함수를 살펴보기로 했다.

```
int sys_forknexec(void) {
    const char *path;
    const char **argv;

    if ( argstr(0, (void*)&path ) < 0 || argptr(1, (void*)&argv, sizeof(*argv)) < 0){
        return -1;
    }
    return forknexec(path, argv);
}

454,5      99%
```

지난 과제로 제출한 forknexec()의 wrapping 함수를 살펴보았다.

```
int
sys_kill(void)
{
    int pid;

    if(argint(0, &pid) < 0)
        return -1;
    return kill(pid);
}
```

다른 wrapping 함수를 살펴보았다. 이중 sys\_kill함수가 필자처럼 pid를 사용했다. 필자가 만든 wrapping 함수와 사실상 같은 상황인데도 오류가 안나는 것을 보고, wrapping 함수에서 처음 나온 변수들은 모종의 과정을 통과를 한 뒤에야 비로소 사용이 가능하다는 생각이 들었다.

```

int sys_forknexec(void) {
    const char *path;
    const char **argv;
    /**
     * if ( argstr(0, (void*)&path ) < 0 || argptr(1, (void*)&argv, sizeof(*argv)) < 0){
         return -1;
     }
     */
    return forknexec(path, argv);
}

```

456,1

Bot

이것이 사실인지 검증하기 위해, 지난 과제의 forknexec의 wrapping 함수의 모종의 과정에 주석처리를 하고 make 해보겠다.

```

kh@ThinkBook-Plus:~/os/hw2$ make qemu-nox
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werror -fno-omit-frame-pointer -fno-stack-protector -fno-pie -no-pie -c -o sysfile.o sysfile.c
sysfile.c: In function 'sys_forknexec':
sysfile.c:455:12: error: 'path' is used uninitialized [-Werror=uninitialized]
  455 |         return forknexec(path, argv);
      |         ^
sysfile.c:455:12: error: 'argv' is used uninitialized [-Werror=uninitialized]
cc1: all warnings being treated as errors
make: *** [<built-in>: sysfile.o] Error 1
kh@ThinkBook-Plus:~/os/hw2$ _

```

유레카! 원인은 이거였다.

```

//
// File-system system calls.
// Mostly argument checking, since we don't trust
// user code, and calls into file.c and fs.c.
//

```

굳이 os를 이렇게 만든 이유로는, sysfile.c의 주석의 내용처럼 사용자의 코드를 믿지 않기에 입력값의 검증을 필수적으로 하게끔 하려는 의도로 보인다. argint, argstr, argptr 이 셋이 입력값 검증을 해주는 것 같은 느낌이 들어서, chatgpt에게 물어보았다. 참고로 이들은 syscall.c에 정의되어있다.



1. `'argint'`:
  - `'argint'` is a function used to extract integer arguments from the command-line arguments of a user-level program.
  - It takes the argument index (position), and a pointer to an integer as arguments.
  - The function converts the corresponding command-line argument (usually provided as a string) to an integer and stores the result in the provided integer pointer.
2. `'argstr'`:
  - `'argstr'` is a function used to extract string (character array) arguments from the command-line arguments of a user-level program.
  - It takes the argument index (position) and a pointer to a character array as arguments.
  - The function copies the corresponding command-line argument (a string) into the provided character array.
3. `'argptr'`:
  - `'argptr'` is a more general function that can be used to extract a pointer to some data structure from the command-line arguments.
  - It takes the argument index (position) and a pointer to a data structure as arguments.
  - This function typically doesn't do any data conversion. Instead, it is used when you need to pass a pointer to a data structure as an argument to your program.

These functions are typically used when writing user-level programs in xv6 to retrieve and parse command-line arguments that are passed to the program when it is executed. They help simplify the process of parsing and extracting different types of data from the command-line arguments.

gpt는 argint가 사용자 수준 프로그램의 명령줄에서 정수 인수를 추출하는데 사용되는 함수라 한다. 생각이 반쯤 맞고 반쯤 틀렸다.

```
int
sys_get_proc_priority(void)
{
    int pid;

    if(argint(0, &pid) < 0)
        return -1;

    return get_proc_priority(pid);
}

void
sys_set_proc_priority(void)
{
    int pid;
    int newpriority;

    if( (argint(0, &pid) < 0) || argint(0, &newpriority) < 0 )
        return;
    return set_proc_priority(pid, newpriority);
}
```

따라서 wrapping 함수를 이렇게 수정했다.

## 6-2. wrapping 함수 (2)

5-1.의 수정 후 새로운 syscall들을 시험하기 위한 newsyscalltest.c를 만들고 make하여 xv6를 실행했다.

```
$ prioritytest
current process's pid : 3
before set priority of pid(3) = 5
newpriority = 3
p->priority = 5
p->priority = 3
after set priority of pid(3) = 3
$ prioritytest
current process's pid : 4
before set priority of pid(4) = 5
newpriority = 4
p->priority = 5
p->priority = 4
after set priority of pid(4) = 4
$ _
```

제대로 작동이 안되서 중간에 디버그코드를 집어넣고 출력하는 모습입니다. set\_함수에서 두 번째 인자로 받은 newpriority가 어떤 이유에 의해 이상한 값이 되버렸다. 예상컨대 pid로 덮어씌워진 듯 하다.

```
int
sys_get_proc_priority(void)
{
    int pid;

    if(argint(0, &pid) < 0)
        return -1;

    return get_proc_priority(pid);
}

void
sys_set_proc_priority(void)
{
    int pid;
    int newpriority;

    if( (argint(0, &pid) < 0) || argint(0, &newpriority) < 0 )
        return;
    return set_proc_priority(pid, newpriority);
}
```

문제가 되는 부분은 사진과 같았다. 두 페이지 전의 설명을 다시 보자, argint는 argument index와 pointer를 argv로 받는다. argument index를 중복시키는 바보짓을 했다. 따라서 이를 1로 수정했다.