Design

목표: `getppid` 시스템 콜을 만들고 실행할 수 있는 유저 프로그램 만들기

유저 측부터 코드를 살펴보면서 실행 흐름을 이해해보기로 했다. 우선 가장 먼저 `/user`에 실행할 사용자 프로그램을 만드는 것으로 시작해야 한다. 명세에 나와있듯이 실행 명령은 `ppid`여야 하기 때문에 사용자 프로그램을 `ppid.c`로 만든다. `Makefile`의 `UPROGS`(사용자 프로그램) 변수에 등록을 해놔야 빌드 과정에 포함되어 함께 컴파일되고 링크될 것이다.

```
UPROGS=\
  $U/_cat\
  $U/_echo\
  $U/_forktest\
  $U/_grep\
  ...
```

`/user/usys.pl` 이라는 파일은 빌드되는 과정에서 `usys.s`라는 어셈브리 스텁을 생성한다. 어셈블리 스텁은 `usys.pl`에 정의된 `entry`서브루틴을 통해 생성되고 따라서 `entry("getppid")`를 추가해 주어야 `getppid`에 해당하는 어셈블리 스텁이 만들어질 것이다.

`makefile` 내 `usys.pl`을 빌드하는 코드

```
$U/usys.S : $U/usys.pl
  perl $U/usys.pl > $U/usys.S
```

`usys.pl` 내 `entry` 서브루틴

```
sub entry {
my $name = shift;
print ".global $name\n";
print "${name}:\n";
print " li a7, SYS_${name}\n";
print " ecall\n";
print " ret\n";
}
```

각각의 시스템 콜에 대하여 `entry` 서브루틴을 실행하여 어셈블리 스텁이 만들어짐

```
entry("fork");
entry("exit");
entry("wait");
entry("pipe");
entry("read");
entry("write");
...
```

`fork` 를 예시로 들면 다음과 같은 어셈블리 스텁이 생성됨. 이 코드는 `fork`를 호출할 때 `SYS_fork`(fork 시스템 콜 번호)를 `a7` 레지스터에 로드하고 `ecall` 명령어를 통해 커널에 요청을 함.

```
.global fork
fork:
li a7, SYS_fork
ecall
ret
```

따라서 만들고자 하는 시스템 콜에 대한 `entry`서브루틴 `entry("getppid");` 를 추가해야할 것이다.

`ecall` 이 실행되면 프로세서는 트랩을 발생시켜 사용자 모드에서 커널 모드로 전환시킨다. 이때 트랩핸들러의 주소를 담고 있는 레지스터인 `stvec`이 `uservec` (유저 트랩 벡터의 시작 주소)를 가지고 있기 때문에 제어가 `uservec`이 구현되어 있는 `trampoline.s`로 넘어간다.

각종 유저 레지스터 정보들을 `Trapframe`에 저장한 뒤 `usertrap`으로 점프한다. 여기서 `trapframe`은 사용자 모드와 커널 모드 간의 전환 과정에서 사용자 상태를 복원하기 위한 저장소 역할을 하며, 커널이 관리하는 별도의 구조체이다.

```
uservec:
        csrw sscratch, a0

        li a0, TRAPFRAME

        # save the user registers in TRAPFRAME
        sd ra, 40(a0)
        sd sp, 48(a0)
        sd gp, 56(a0)
        sd tp, 64(a0)
        sd t0, 72(a0)
        sd t1, 80(a0)
        ...

        csrr t0, sscratch
        sd t0, 112(a0)

        ld sp, 8(a0)
        ld tp, 32(a0)
        ld t0, 16(a0)
        ld t1, 0(a0)
        sfence.vma zero, zero

        csrw satp, t1

        sfence.vma zero, zero

        # jump to usertrap(), which does not return
        jr t0
```

`kernel/trap.c`의 `usertrap`함수가 실행된다. 만약 `r_scause`값이 8이라면 시스템 콜(`syscall`)이 호출되며 아닌 경우에서 타이머 인터럽트 또는 외부 장치 인터럽트를 나타내는 디바이스 인터럽트(`devintr`)가 발생한다. 마지막에는 `usertrapret`가 호출되며 제어가 `trap.c`의 `usertrapret`으로 넘어간다.

```
void
```

```
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0) //사용자 모드에서 트랩이 발생한 것이 맞는지 확인
    panic("usertrap: not from user mode");

  w_stvec((uint64)kernelvec); //stvec 레지스터에 값을 씀(트랩 발생 시 제어가 넘어갈 핸들러의 주소). 커널 모드에서 발생한 트랩을 kernelvec으로 처리해야하기 때문

  struct proc *p = myproc(); //현재 프로세스 포인터

  // save user program counter.
  p->trapframe->epc = r_sepc(); //트랩이 발생하기 직전의 프로그램 카운터 저장

  if(r_scause() == 8){
    // system call

    if(killed(p))
      exit(-1);

    p->trapframe->epc += 4; //시스템 콜이 끝난 이후 사용자 모드로 돌아갈 때 다음 명령어 실행 위치 (4바이트 명령어 크기)

    intr_on(); //인터럽트 온

    syscall(); //시스템 콜 수행
  } else if((which_dev = devintr()) != 0){ //타이머 인터럽트 또는 외부장치 인터럽트
    // ok
  } else {
    printf("usertrap(): unexpected scause 0x%lx pid=%d\n", r_scause(), p->pid);
    printf("            sepc=0x%lx stval=0x%lx\n", r_sepc(), r_stval());
    setkilled(p);
  }

  if(killed(p))
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret(); //트랩이 발생하기 전의 레지스터 상태 복원
}
```

`scause`가 8인 상황, 즉 시스템 콜이 호출되는 상황을 살펴보자. 아래는 `/kernel/syscall.c`에 정의된 `syscall` 함수이다. `Trapframe`의 `a7` 레지스터에서 해당하는 시스템 콜의 번호를 가져와 a0(사용자에게 반환될 레지스터)에 시스템 콜 함수를 넣어준다.

```
void
syscall(void)
```

```
{
  int num; //시스템 콜 번호
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num](); //사용자에게 반환될 레지스터인 a0에 해당 시스템콜 함수를 넣음
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

시스템 콜 함수 배열인 `syscalls`는 아래와 같이 정의되어 있으며 우리가 만들 시스템콜을 여기에 정의해줘야 한다.

```
static uint64 (*syscalls[])(void) = { //반환이 uint64, 인자가 없는 함수 포인터 배열, 배열의 크기는 자동으로 생성
[SYS_fork]    sys_fork, //시스템 콜 번호 SYS_fork(예: 1)을 배열 인덱스로 하고, 해당 배열 위치에 sys_fork 함수의 주소를 할당
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
...
};
```

또한 함수 정보를 참조할 수 있어야 하기에 우리가 만들 시스템 콜함수를 여기에도 추가해야 한다. 해당 시스템 콜 함수(부모 프로세스id를 반환하는 `sys_getppid`)는 `/kernel`하위에 만들면 될 것이다.

```
extern uint64 sys_fork(void);
extern uint64 sys_exit(void);
extern uint64 sys_wait(void);
extern uint64 sys_pipe(void);
extern uint64 sys_read(void);
extern uint64 sys_kill(void);
extern uint64 sys_exec(void);
...
```

`usertrapret`을 통해 이제 유저 모드로 갈 준비를 한다. 유저모드로 돌아갈 것이기 때문에 `stvec` 값을 `uservec`으로 설정하고 다음에 커널 모드로 진입할 때 필요한 정보들을 `trapframe`에 기록해놓는다. 사용자 모드의 프로그램 카운터 또한 복원시킨다. `userret`으로 제어를 넘기기 전에 필요한 각종 레지스터 정보들을 기록해놓고 `trampoline.s` 에 있는 `userret`으로 제어를 넘긴다.

```
//
// return to user space
//
void
```

```
usertrapret(void)
{
  struct proc *p = myproc();

  intr_off(); //인터럽트 끄기 -> kerneltrap에서 usertrap으로 바뀌는 과정에서 예기치 못한 예외가 발생하면 안됨

  // send syscalls, interrupts, and exceptions to uservec in trampoline.S
  uint64 trampoline_uservec = TRAMPOLINE + (uservec - trampoline); //trampoline_uservec의 주소를 계산, uservec - trampoline은 uservec함수가 trampoline 내에서 얼마나 떨어져 있는지 나타냄
  w_stvec(trampoline_uservec); //stvec 레지스터에 값을 씀 (트랩 발생 시 제어가 넘어갈 핸들러의 주소). stvec을 설정해야 추후에 유저모드로 복귀하고 다시 시스템 콜이 발생했을 때 CPU가 점프해야 될 위치를 알 수 있음
  //커널모드는 kernelvec으로 트랩을 처리하지만 유저모드는 uservec으로 트랩을 처리하기 때문에 모드 변경 과정에서 stvec이 갖고 있는 주소를 항상 바꿔줘야됨

  // set up trapframe values that uservec will need when
  // the process next traps into the kernel.
  // 커널 모드로 다시 복귀할 때 필요한 값들 저장
  p->trapframe->kernel_satp = r_satp();         // kernel page table
  p->trapframe->kernel_sp = p->kstack + PGSIZE; // process's kernel stack
  p->trapframe->kernel_trap = (uint64)usertrap; // 다음에 트랩이 발생했을 때 호출할 함수(여기서는 usertrap)의 주소를 저장합니다. (유저모드에서 호출할 것이기 때문)
  p->trapframe->kernel_hartid = r_tp();         // hartid for cpuid() 프로세스 코어 id

  // set up the registers that trampoline.S's sret will use
  // to get to user space.

  // set S Previous Privilege mode to User.
  unsigned long x = r_sstatus();
  x &= ~SSTATUS_SPP; // clear SPP to 0 for user mode
  x |= SSTATUS_SPIE; // enable interrupts in user mode
  w_sstatus(x); //sstatus 레지스터는 현재 실행 모드, 인터럽트 상태 등 정보를 담고 있음

  // set S Exception Program Counter to the saved user pc.
  w_sepc(p->trapframe->epc); //트랩이 발생할 때, 사용자 모드의 프로그램 카운터(현재 실행 중이던 명령어 주소)가 sepc에 저장됨. sret 명령어를 통해 사용자 모드로 전환 시 정확히 이 주소부터 실행을 재개할 수 있게 함

  // tell trampoline.S the user page table to switch to.
  uint64 satp = MAKE_SATP(p->pagetable); //사용자 페이지 테이블 준비하기

  // jump to userret in trampoline.S at the top of memory, which
  // switches to the user page table, restores user registers,
  // and switches to user mode with sret.
  uint64 trampoline_userret = TRAMPOLINE + (userret - trampoline); //userret 주소 계산
  ((void (*)(uint64))trampoline_userret)(satp);
}
```

`userret`에서는 `trapframe`으로부터 유저 모드 레지스터 정보들을 복구시킨뒤 `sret`을 통해 유저 모드로 모드를 바꾼다.

```
.globl userret
userret:
```

```
        sfence.vma zero, zero
        csrw satp, a0
        sfence.vma zero, zero

        li a0, TRAPFRAME

        # restore all but a0 from TRAPFRAME
        ld ra, 40(a0)
        ld sp, 48(a0)
        ld gp, 56(a0)
        ld tp, 64(a0)
        ld t0, 72(a0)
        ld t1, 80(a0)
        ld t2, 88(a0)
        ...

  # restore user a0
        ld a0, 112(a0)

        # return to user mode and user pc.
        # usertrapret() set up sstatus and sepc.
        sret
```

Implementation

유저 부분

실행할 사용자 프로그램부터 만들어 보았다. `/user/ppid.c` 파일을 만들고 명세에 따라 출력에 요구되는 값들을 작성하였다.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(void) {
    int my_pid = getpid();
    int parent_pid = getppid();
    printf("My student ID is 2021058995\n");
    printf("My pid is %d\n", my_pid);
    printf("My ppid is %d\n", parent_pid);
    exit(0);
}
```

우리가 만들 프로그램이 같이 빌드될 수 있도록 `makefile`의 `UPROGS`(사용자 프로그램 변수)에 `ppid` 프로그램을 추가해주었다.

```
UPROGS=\
  $U/_cat\
  $U/_echo\
  $U/_forktest\
  $U/_grep\
  $U/_init\
  $U/_kill\
  ...
  $U/_ppid\
```

새롭게 구현할 `getppid` 함수를 `user.h` 헤더파일에 등록을 해주었다.

```
// system calls
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
...
int getppid(void); //usys.pl 스크립트가 생성하는 어셈블리 스텁이 getppid를 호출하고(시스템 콜 번호를 레지스터에 설정) 트랩을 통해 커널 모드로 변경 후 내부 커널 함수인 sys_getppid를 호출하도록 구성되어 있음
```

우리가 실행할 프로그램(ppid)의 `getppid`는 빌드과정에서 다음과 같은 `entry`서브루틴을 통해 어셈블리 스텁이 생성되어야한다. 즉 우리가 `ppid.c`에서 호출하는 함수인 `getppid`는 아래와 같은 어셈블리 스텁이 실행되는거다. 아래 코드는 `a7`레지스터에 `SYS_getppid`(시스템 콜 번호)를 로드하고 ecall을 실행한 뒤 반환하는 코드이다.

```
.global getppid
getppid:
li a7, SYS_getppid
ecall
ret
```

따라서 `usys.pl`에 `getppid`에 대한 `entry`서브루틴을 추가해준다. 이제 커널 부분으로 넘어가자.

```
entry("fork");
entry("exit");
entry("wait");
...
entry("getppid");
```

커널 부분

어셈블리 스텁의 `SYS_getppid`(시스템 콜 번호)가 얼마인지는 `syscall.h`에 등록을 해주어야 한다. 마지막 번의 다음인 23번으로 등록해준다.

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
...
#define SYS_myfunction 22
#define SYS_getppid 23
```

이제 시스템 콜 함수를 구현할 차례이다. `kernel/getppid.c` 를 만들고 아래와 같이 함수를 구현해준다. `myproc` 은 현재 프로세스의 구조체 포인터를 반환해주는 함수이고 `proc.c` 에 구현이 되어있다. 부모 프로세스가 있는 경우에 부모 프로세스의 id를, 없는 경우에는 -1을 반환하도록 구현하였다.

```
#include "types.h"
#include "param.h"
#include "riscv.h"
#include "spinlock.h"
#include "defs.h"
#include "proc.h"

uint64 sys_getppid(void) { //argument를 받고 있지 않기 때문에 따로 wrapper function은 불필요
    struct proc *p = myproc(); //현재 프로세스 구조체 포인터
    if(p->parent){ //부모 프로세스가 있다면
        return p->parent->pid;
    }else{ //부모 프로세스가 없다면
        return -1; //-1 반환
    }
}
```

`kernel/defs.h` 헤더파일에 구현한 함수를 등록해준다.

```
// getppid.c
uint64 sys_getppid(void);
```

`kernel/syscall.c` 에서 이제 `syscalls` 함수 포인터 배열에 우리가 구현한 sys_getppid함수를 `SYS_getppid(23번)` 인덱스에 할당해준다. `extern` 으로 선언해준 부분은 외부에서 정의된 함수임을 명시하기 위함이다. 해당 syscalls함수는 나중에 `trap.c` 에서 `syscall()` 이 호출 될 경우 사용되며, 어셈블리 스텝에서 저장한 시스템 콜 번호인 `trapframe` 의 `a7` 레지스터의 정보를 통해 `syscalls` 의 인덱스로 접근하여 시스템 콜을 호출하는 방식으로 동작한다.

```
// Prototypes for the functions that handle system calls.
extern uint64 sys_fork(void);
extern uint64 sys_exit(void);
extern uint64 sys_wait(void);
extern uint64 sys_pipe(void);
extern uint64 sys_read(void);
...
extern uint64 sys_getppid(void);

// An array mapping syscall numbers from syscall.h
```

```
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = { //반환이 uint64, 인자가 없는 함수 포인터 배열, 배열의 크기는 자동으로 생성
[SYS_fork]    sys_fork, //시스템 콜 번호 SYS_fork(예: 1)을 배열 인덱스로 하고, 해당 배열 위치에 sys_fork 함수의 주소를 할당
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
...
[SYS_getppid] sys_getppid,
};
```

시스템 콜 호출 로직은 `syscall.c`에 정의되어 있다.

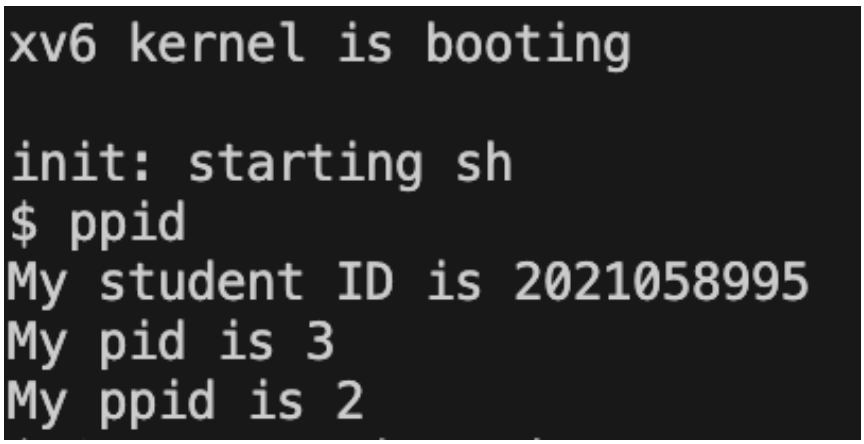
```
void
syscall(void)
{
  int num; //시스템 콜 번호
  struct proc *p = myproc();

  num = p->trapframe->a7;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num](); //사용자에게 반환될 레지스터인 a0에 해당 시스템콜 함수를 넣음
  } else {
    printf("%d %s: unknown sys call %d\n",
            p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

이제 `makefile`의 `OBJS`변수에 우리가 구현한 함수의 오브젝트 파일을 추가해주어 링커에 전달된 뒤, 하나의 실행 가능한 커널 바이너리로 결합될 수 있도록 한다. 여기까지 실행을 위한 기본 세팅은 끝난다.

```
OBJS = \
  $K/entry.o \
  $K/start.o \
  $K/console.o \
  $K/printf.o \
  $K/uart.o \
  $K/kalloc.o \
  $K/spinlock.o \
  $K/string.o \
  $K/main.o \
  ...
  $K/getppid.o
```

Result

아래는 실행 결과물이다.



위에서 정리했던 내용들 바탕으로 실행 과정을 간단히 요약해보면 다음과 같다.

1. 사용자 모드에서 `ppid` 프로그램이 실행되고 `getppid()` 호출 시 어셈블리 스텁이 실행된다.
2. 어셈블리 스텁은 시스템 콜 번호 `SYS_getppid`를 `a7` 레지스터에 설정하고 `ecall`을 실행한다.
3. ecall에 의해 사용자 모드에서 커널 모드로 트랩이 발생한다.
4. 트랩 핸들러(`uservec`)가 호출되어 사용자 레지스터 상태를 `Trapframe`에 저장하고 `usertrap`으로 제어를 넘긴다.
5. `usertrap`은 트랩 원인을 분석하여 `syscall()`을 호출한다.
6. `syscall()` 함수는 syscalls 배열에서 시스템 콜 번호를 이용해 `sys_getppid()`를 호출한다.
7. `sys_getppid()` 함수는 현재 프로세스의 부모 프로세스 ID를 반환한다.
8. 트랩 처리가 완료되면 `usertrapret`과 `userret`을 통해 사용자 모드로 복귀하고, 사용자 프로그램에서 결과가 출력된다.

Troubleshooting

1. 구현을 끝내고 `make qemu` 를 했을 때 다음과 같은 오류가 발생했었다. 내용을 살펴보면 `proc.h` 에 있는 구조체들의 타입이 문제가 있다는 것이었다.

```
In file included from kernel/getppid.c:3:
kernel/proc.h:86:19: error: field 'lock' has incomplete type
   86 |   struct spinlock lock;
      |                   ^~~~
kernel/proc.h:101:3: error: unknown type name 'pagetable_t'
  101 |   pagetable_t pagetable;         // User page table
      |   ^~~~~~~~~~~
make: *** [kernel/getppid.o] Error 1
```

`getppid.c`를 살펴본 뒤 원인을 알 수 있었다. 헤더파일을 추가할 때는 항상 순서가 중요한 법인데 내가 `proc.h`를 `spinlock.h`보다 먼저 선언하는 바람에 `spinlock.h`에서 사용되는 정보들이 `proc.h`에서 찾지 못하는 오류였다. `proc.h`를 맨 마지막에 선언하여 문제를 해결하였다.

```
#include "types.h"
#include "param.h"
#include "proc.h"
#include "riscv.h"
#include "spinlock.h"
#include "defs.h"
```

2. 컴파일 시 `undefined reference to sys_getppid` 오류가 발생했다.

```
riscv64-unknown-elf-ld: warning: kernel/kernel has a LOAD segment with RWX permissions
riscv64-unknown-elf-ld: kernel/syscall.o:(.rodata+0xd0): undefined reference to `sys_getppid'
make: *** [kernel/kernel] Error 1
```

원인을 찾아본 결과 `makefile`의 OBJS 변수에 `$K/getppid.o` 가 빠져있어서 해당 원인이 발생한 것이었다. 링커가 최종 커널 바이너리를 만들 때 `getppid.o`를 포함하지 않으면 `sys_getppid()` 함수의 실제 정의를 찾을 수 없기