

# 배경지식

이번 과제를 수행하기에 앞서 기본적으로 알고 넘어가야 할 시스템 콜과 주요 함수들의 동작 과정 및 메모리 구조에 대해 이해해보려고 한다.

## allocproc

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

    for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state == UNUSED) {
            goto found;
        } else {
            release(&p->lock);
        }
    }
    return 0;

found:
    p->pid = allocpid();
    p->state = USED;

    // Allocate a trapframe page.
    if((p->trapframe = (struct trapframe *)kalloc()) == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    }

    // An empty user page table.
    p->pagetable = proc_pagetable(p);
    if(p->pagetable == 0){
        freeproc(p);
        release(&p->lock);
        return 0;
    }

    // Set up new context to start executing at forkret,
    // which returns to user space.
    memset(&p->context, 0, sizeof(p->context));
    p->context.ra = (uint64)forkret;
    p->context.sp = p->kstack + PGSIZE;

    return p;
}
```

**allocproc** 함수의 주요 동작은 다음과 같다.

- pid 할당하기
- trapframe 할당하기
- pagetable 할당하기
- context(커널 실행 문맥) 초기화
- context.ra를 forkret으로 설정 - scheduler가 이 프로세스를 선택한 후 switch()로 컨텍스트를 불러온 뒤 forkret부터 실행됨. forkret은 이 프로세스가 유저모드로 올바르게 진입할 수 있도록 usertrapret을 호출하여
  1. trapframe에 저장된 유저 레지스터 복원
  2. SATP를 유저 페이지 테이블로 설정
  3. sret 명령을 통해 유저모드로 점프
- 할 수 있도록 한다.
- context.sp(커널 스택 포인터)를 커널 스택의 시작 주소 + PAGESIZE로 하여 스택 포인터가 맨 꼭대기를 가리키도록 한다.  
(커널 스택은 한 페이지로 구성된다.)

## fork

만약 쓰레드가 fork를 호출하는 경우 해당 쓰레드는 부모 프로세스의 자식 프로세스가 된다.

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

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

    // Copy user memory from parent to child.
    if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
        freeproc(np);
        release(&np->lock);
        return -1;
    }
    np->sz = p->sz;

    // copy saved user registers.
    *(np->trapframe) = *(p->trapframe);

    // Cause fork to return 0 in the child.
    np->trapframe->a0 = 0;

    // increment reference counts on open file descriptors.
    for(i = 0; i < NOFILE; i++)
```

```

    if(p->ofile[i])
        np->ofile[i] = filedup(p->ofile[i]);
    np->cwd = idup(p->cwd);

    safestrcpy(np->name, p->name, sizeof(p->name));

    pid = np->pid;

    release(&np->lock);

    acquire(&wait_lock);
    np->parent = p->group;
    release(&wait_lock);

    acquire(&np->lock);
    np->state = RUNNABLE;
    release(&np->lock);

    return pid;
}

```

**fork** 함수의 주요 동작은 다음과 같다.

- allocproc으로 np 할당
- uvmcopy를 통해 부모의 메모리 정보 복사(페이지 테이블 포함)
- 부모의 sz값 복사
- 부모의 trapframe값 복사 후 a0만 0으로 설정하여 자식은 반환 값을 0으로 설정함(부모는 자식의 pid임)
- 열려 있는 파일 정보들(ofile 정보)과 현재 작업중인 디렉토리 정보(cwd) 복사
- np->parent 부모 설정, np->state RUNNABLE 상태 설정

## uvmcopy

```

int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
    pte_t *pte;
    uint64 pa, i;
    uint flags;
    char *mem;

    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walk(old, i, 0)) == 0)
            panic("uvmcopy: pte should exist");
        if((*pte & PTE_V) == 0)
            panic("uvmcopy: page not present");
        pa = PTE2PA(*pte);
        flags = PTE_FLAGS(*pte);
        if((mem = kalloc()) == 0)
            goto err;
        safestrcpy(mem, pte->pte, PGSIZE);
        pte->pte = pa | flags;
    }
}

```

```

    memmove(mem, (char*)pa, PGSIZE);
    if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
        kfree(mem);
        goto err;
    }
}
return 0;

err:
uvmunmap(new, 0, i / PGSIZE, 1);
return -1;
}

```

**uvmcopy** 함수의 주요 동작은 다음과 같다.

1. 유저 영역 크기 (sz)만큼 페이지 단위(PGSIZE)로 순회한다.
2. `walk(old, i, 0)`를 통해 부모의 페이징 테이블에서 주소 `i`에 해당하는 PTE(페이지 테이블 엔트리)를 찾아 반환하고 `PTE_V` 비트를 확인하여 페이지가 valid한지 검사한다.
3. `pa = PTE2PA(*pte)`를 통해 물리 페이지 프레임의 시작 주소를 반환한다.
4. `flags = PTE_FLAGS(*pte);`를 통해 PTE에 설정된 권한 비트(R/W/U 등)를 분리한다.
5. `kalloc()` : 커널 메모리풀에서 새 페이지 하나 할당한다.
6. `memmove` : 원래 페이지의 내용(`pa`)을 방금 할당한 `mem`으로 복사한다.
7. `mappages` : 페이지 테이블 계층(Level-2, Level-1, Level-0)을 `walk(..., alloc=1)`로 확보한다.

요약하면 0부터 sz-1까지 페이지 단위로 순회하며 기존 프로세스로부터 pa를 추출해서 mem이라는 새로운 물리 프레임에 복사한 뒤 mappages를 통해 pagetable 단계들을 매핑해주는 것이다.

## exec 앞 부분

`exec`은 이번 과제에서 개인적으로 생각하기에 가장 구현이 어려운 부분이라 정리하였다.

```

int
exec(char *path, char **argv)
{
    char *s, *last;
    int i, off;
    uint64 argc, sz = 0, sp, ustack[MAXARG], stackbase;
    struct elfhdr elf;
    struct inode *ip;
    struct proghdr ph;
    pagetable_t pagetable = 0, oldpagetable;
    struct proc *p = myproc();

    begin_op();

    if((ip = namei(path)) == 0){ // path에 해당하는 inode 가져옴
        end_op();
        return -1;
    }
}

```

```

ilock(ip); // inode 잠금

// Check ELF header
if(readi(ip, 0, (uint64)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;

if(elf.magic != ELF_MAGIC)
    goto bad;

if((pagetable = proc_pagetable(p)) == 0) // 새로운 페이지 테이블 생성
    goto bad;

// Load program into memory.
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
        continue;
    if(ph.memsz < ph.filesz)
        goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
        goto bad;
    if(ph.vaddr % PGSIZE != 0)
        goto bad;
    uint64 sz1;
    if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz, flags2perm(ph.flags))) == 0) // [sz, ph.vaddr + ph.memsz] 단위로 매핑
        goto bad;
    sz = sz1;
    if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0) // 물리 메모리에 파일 데이터 복사
        goto bad;
}
iunlockput(ip);
end_op();
ip = 0;
...

```

**exec** 시스템 콜 앞 부분의 주요 동작은 다음과 같다

- **begin\_op**는 파일 시스템 전용 로깅 관련 함수이다.
- **namei**를 통해 실행하고자 하는 파일 혹은 디렉토리의 path에 대응하는 inode를 갖고온다.
- **readi**를 통해 디스크에 저장된 파일 혹은 디렉토리의 내용을 **inode** 구조체를 통해 메모리로 읽어온다.
- **elf.magic** 값을 확인하여 실제로 커널이 읽을 수 있는 elf 포맷인지 판별한다.
- 페이지 테이블을 생성한다.
- ELF 파일의 프로그램 헤더를 하나씩 읽어, 메모리에 올려야 할 코드·데이터만 골라 페이지 단위로 매핑한 뒤, 실제 내용을 복사해서 사용자 프로그램이 곧바로 실행될 수 있도록 준비한다.

## namei 동작 로직

```

struct inode*
namei(char *path)
{
    char name[DIRSIZ];
    return namex(path, 0, name);
}

```

```

static struct inode*
namex(char *path, int nameiparent, char *name)
{
    struct inode *ip, *next;

    if(*path == '/')
        ip = igin(ROOTDEV, ROOTINO);
    else
        ip = idup(myproc()->cwd);

    while((path = skipellem(path, name)) != 0){
        ilock(ip);
        if(ip->type != T_DIR){
            iunlockput(ip);
            return 0;
        }
        if(nameiparent && *path == '\0'){
            // Stop one level early.
            iunlock(ip);
            return ip;
        }
        if((next = dirlookup(ip, name, 0)) == 0){
            iunlockput(ip);
            return 0;
        }
        iunlockput(ip);
        ip = next;
    }

    if(nameiparent){
        iput(ip);
        return 0;
    }
    return ip;
}

```

`namei("/home/inhyeok/doc.txt")` 를 예시로 들어보자.

1. 제일 위 디렉토리( / )을 열고,
2. 그 안의 `home` 폴더를 열고,
3. 다시 `inhyeok` 폴더를 열어서,
4. 그 안에서 `doc.txt`라는 이름의 문서를 찾아서,

5. 그 문서의 정보(inode)를 내게 달라

라고 부탁하는 것과 같다.

inode의 구조체는 다음과 같이 정의되어 있다.

```
struct inode {
    uint dev;           // Device number
    uint inum;          // Inode number
    int ref;            // Reference count
    struct sleeplock lock; // protects everything below here
    int valid;          // inode has been read from disk?

    short type;         // copy of disk inode
    short major;
    short minor;
    short nlink;
    uint size;
    uint addrs[NDIRECT+1];
};
```

- `dev` - 파일이 저장된 장치(디스크) 번호이다.
- `inum` - 그 장치 안에서 이 파일(또는 디렉터리)을 가리키는 고유 번호이다.
- `ref` - 커널 내부에서 이 파일 정보를 몇 군데에서 열어두고 있는지를 센 숫자이다.
- `lock` - 이 i-node 내부 정보를 바꿀 때(메타데이터 수정) 걸 수 있는 락이다.
- `valid` - 디스크에서 가져온 데이터가 메모리에 올바로 로드되었는지 표시하는 값이다. 0이면 아직 읽어오지 않은 상태, 1이면 읽은 것이다.
- `type` - 이 i-node가 가리키는 대상의 종류이다. 일반 파일(`T_FILE`), 디렉터리(`T_DIR`), 또는 특수 디바이스 파일(`T_DEV`) 중 하나이다.
- `major` - 장치 파일일 때 필요한 번호(major 번호)이다. 어떤 종류의 하드웨어 드라이버를 쓸지 결정해 준다.
- `minor` - 마찬가지로 장치 파일일 때, 하드웨어 안에서 세부 장치를 구분하는 번호이다.
- `nlink` - 이 파일을 가리키는 하드링크(디렉터리 엔트리)의 개수이다. 0이 되면 파일 삭제 조건이 된다.
- `size` - 파일의 크기(바이트 단위)이다. 디렉터리 파일이면 디렉터리 항목들의 총 크기이다.
- `addrs` - 데이터 블록 번호(physical block number)을 담는 배열이다.
  - `addrs[0] ~ addrs[NDIRECT-1]` : 직접 블록 포인터(파일 데이터를 바로 담는 블록)
  - `addrs[NDIRECT]` : 간접 블록 포인터(추가 블록 번호들이 담긴 “블록 안의 블록”을 가리킨다)

어디에 있는가: `dev + inum` 으로 디스크 상의 위치를 찾고

어떤 파일인가: `type`, `major / minor` 로 종류를 확인

얼마나 크고 몇 개 참조 중인가: `size`, `nlink`, `ref` 로 상태 파악

데이터는 어디에 있는가: `addrs[]` 에 담긴 블록 번호를 통해 실제 파일 내용을 읽거나 쓴다.

## elf(header)의 구조체를 살펴보자

```
// File header
```

```

struct elfhdr {
    uint magic; // must equal ELF_MAGIC
    uchar elf[12];
    ushort type;
    ushort machine;
    uint version;
    uint64 entry;
    uint64 phoff;
    uint64 shoff;
    uint flags;
    ushort ehsiz;
    ushort phentsize;
    ushort phnum;
    ushort shentsize;
    ushort shnum;
    ushort shstrndx;
};

};


```

- `magic` - 방금 디스크에서 읽어온 ELF 헤더의 첫 4바이트(매직 넘버)를 담고 있는 필드이다.
- `elf` - ELF 식별 정보(`e_ident`)의 나머지 부분이다.
- `type` - ELF 객체의 종류를 나타낸다. 실행 파일(`ET_EXEC`), 공유 라이브러리(`ET_DYN`), 재위치 가능 파일(`ET_REL`) 등이 있다.
- `machine` - 목표 아키텍처를 식별한다.
- `veersion` - ELF 포맷의 버전 번호이다. 현재는 보통 1 ("EV\_CURRENT")이 쓰입니다
- `entry` - 프로그램을 처음 시작할 진입점(Entry Point) 가상주소이다.
- `phoff` - 프로그램 헤더 테이블(program header table)이 파일 내에서 시작하는 바이트 오프셋이다.
- `shoff` - 섹션 헤더 테이블(section header table)이 파일 내에서 시작하는 바이트 오프셋이다.
- `flags` - 아키텍처별 추가 플래그를 담음.
- `ehsize` - 이 ELF 헤더 자체의 크기이다. 보통 `sizeof(struct elfhdr)` 와 같다.
- `phentsize` - 프로그램 헤더 항목 하나(program header entry) 의 크기이다.
- `phnum` - 프로그램 헤더 테이블에 들어 있는 항목(entry)의 개수이다.
- `shentsize` - 섹션 헤더 항목 하나(section header entry) 의 크기이다.
- `shnum` - 섹션 헤더 테이블에 들어 있는 섹션 항목의 개수입니다.
- `shstrndx` - 섹션 이름 문자열 테이블(section name string table)이 몇 번째 섹션인지 가리키는 인덱스이다.

## elf.magic

`exec` 내에서 `elf.magic`과 `ELF_MAGIC`을 비교한다.

```

#define ELF_MAGIC 0x464C457FU // "\x7FELF" in little endian

if (elf.magic != ELF_MAGIC)
    goto bad;

```

`exec()` 하려고 연 파일이 진짜 실행 가능한 ELF인지 단순한 텍스트나 다른 바이너리가 아니라, 커널이 이해할 수 있는 포맷인지 판별한다.

## 프로그램 헤더 루프

다음은 ELF 파일의 프로그램 헤더를 하나씩 읽어, 메모리에 올려야 할 코드·데이터만 골라 페이지 단위로 매핑한 뒤, 실제 내용을 복사해서 사용자 프로그램이 곧바로 실행될 수 있도록 준비하는 루프이다.

```
// Load program into memory.
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
        continue;
    if(ph.memsz < ph.filesz)
        goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
        goto bad;
    if(ph.vaddr % PGSIZE != 0)
        goto bad;
    uint64 sz1;
    if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz, flags2perm(ph.flags))) == 0) // [sz, ph.vaddr + ph.memsz] 단위로 매핑
        goto bad;
    sz = sz1;
    if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0) // 물리 메모리에 파일 데이터 복사
        goto bad;
}
```

우선 `ph`에 대한 구조체를 살펴보자.

```
// Program section header
struct proghdr {
    uint32 type;
    uint32 flags;
    uint64 off;
    uint64 vaddr;
    uint64 paddr;
    uint64 filesz;
    uint64 memsz;
    uint64 align;
};
```

`type` - 헤더의 역할

`flags` - 세그먼트가 매핑될 때 부여할 권한

`off` - ELF 파일에서 이 세그먼트 데이터가 시작되는 위치

`vaddr` - 세그먼트를 메모리에 로드할 때 어떤 가상 주소에 배치할지를 나타냄

`paddr` - 물리 주소를 지정하는 필드

`filesz` - 파일에서 실제로 읽어올 데이터 크기(byte 단위)

`memsz` - 메모리에서 차지할 전체 크기(byte 단위), `memsz` 가 `filesz` 보다 크다면, 파일로부터 복사한 뒤 남는 부분은 0으로 채워(BSS)야 함.

`align` - 정렬 요구사항

이제 코드를 하나하나 뜯어보자.

```
readi(ip, 0, (uint64)&ph, off, sizeof(ph))
```

파일 i-node `ip` 가 가리키는 파일의, 파일 시작점에서 `off` 바이트 떨어진 위치부터 `sizeof(ph)` 바이트를 읽어서 커널 메모리 상의 `ph` 구조체(`&ph`)에 복사한다는 의미이다.

```
if(ph.type != ELF_PROG_LOAD)
    continue;
if(ph.memsz < ph.filesz)
    goto bad;
if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
if(ph.vaddr % PGSIZE != 0)
    goto bad;
```

가져온 `ph` 값에 대한 유효성 검사이다.

```
if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz, flags2perm(ph.flags))) == 0) // [sz, ph.vaddr + ph.memsz] 단위로 매핑
    goto bad;
sz = sz1;
```

`sz` 부터 `ph.vaddr+ph.memsz` (새로운 `sz`)까지의 가상 주소 범위에, `flags2perm(ph.flags)` 권한을 주고 물리 페이지를 할당 후 페이지 테이블에 매핑하는 코드이다.

```
if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0) // 물리 메모리에 파일 데이터 복사
    goto bad;
```

파일 `ip` 의 `ph.off` 위치부터 `ph.filesz` 바이트를 읽어 가상주소 `ph.vaddr`에 복사하는 함수이다.

## exec 뒷 부분

```
p = myproc();
uint64 oldsz = p->sz;

// Allocate some pages at the next page boundary.
// Make the first inaccessible as a stack guard.
// Use the rest as the user stack.
sz = PGROUNDUP(sz);
uint64 sz1;
```

```

// 유저 스택 메모리 할당
if((sz1 = uvmalloc(pagetable, sz, sz + (USERSTACK+1)*PGSIZE, PTE_W)) == 0)
    goto bad;
sz = sz1;

// guard 페이지 할당
uvmclear(pagetable, sz-(USERSTACK+1)*PGSIZE);
sp = sz;
stackbase = sp - USERSTACK*PGSIZE;

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
        goto bad;
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16; // riscv sp must be 16-byte aligned
    if(sp < stackbase)
        goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
        goto bad;
    ustack[argc] = sp;
}
ustack[argc] = 0;

// push the array of argv[] pointers.
sp -= (argc+1) * sizeof(uint64);
sp -= sp % 16;
if(sp < stackbase)
    goto bad;
if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)) < 0)
    goto bad;

// arguments to user main(argc, argv)
// argc is returned via the system call return
// value, which goes in a0.
p->trapframe->a1 = sp;

// Save program name for debugging.
for(last=s=path; *s; s++)
    if(*s == '/')
        last = s+1;
safestrcpy(p->name, last, sizeof(p->name));

// Commit to the user image.
oldpagetable = p->pagetable;
p->pagetable = pagetable;
p->sz = sz;
p->trapframe->epc = elf.entry; // initial program counter = main
p->trapframe->sp = sp; // initial stack pointer
proc_freepagetable(oldpagetable, olds);

```

```

return argc; // this ends up in a0, the first argument to main(argc, argv)

bad:
if(pagetable)
    proc_freepagetable(pagetable, sz);
if(ip){
    iunlockput(ip);
    end_op();
}
return -1;
}

```

**exec** 시스템 콜 뒷 부분의 주요 동작은 다음과 같다

- 유저 스택 메모리 할당
- guard 페이지 할당
- 명령행 인자 문자열들을 ustack에 저장 (각 `argv[i]` 문자열을 스택에 `copyout` 하며 주소를 `ustack[i]`에 저장)
- PCB 갱신

```

// 유저 스택 메모리 할당
if((sz1 = uvmalloc(pagetable, sz, sz + (USERSTACK+1)*PGSIZE, PTE_W)) == 0)
    goto bad;
sz = sz1;

```

`sz`부터 `sz + (USERSTACK+1)*PGSIZE`(새로운 `sz`)까지의 가상 주소 범위에, `PTE_W` 권한을 주고 물리 페이지를 할당 후 페이지 테이블에 매핑하는 코드이다.

```

// guard 페이지 할당
uvmclear(pagetable, sz-(USERSTACK+1)*PGSIZE);
sp = sz;
stackbase = sp - USERSTACK*PGSIZE;

```

이전에 만든 `USERSTACK+1` 만큼의 페이지에서 stack base를 맨 아래 한 페이지를 남겨둔 채 한 칸 위로 설정하여 맨 아래 페이지를 가드 페이지로 설정한것이다. `uvmclear`에서 `PTE_U` 비트만 꺼주어 유저 모드에서 접근이 불가능하도록 만들어준다. (페이지 가드)

```

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
        goto bad;
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16; // riscv sp must be 16-byte aligned
    if(sp < stackbase)
        goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
        goto bad;
    ustack[argc] = sp;
}
ustack[argc] = 0;

```

각 명령행 인자 문자열을 16바이트 정렬된 스택 공간에 복사한 뒤 그 주소를 `ustack` 배열에 순서대로 저장하고 마지막에 NULL 을 추가한다.

```

// push the array of argv[] pointers.
sp -= (argc+1) * sizeof(uint64);
sp -= sp % 16;
if(sp < stackbase)
    goto bad;
if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)) < 0)
    goto bad;

```

스택에 `ustack`에 모은 인자 포인터 배열(`argc+1` 개)을 16바이트 정렬된 위치에 복사한다.

```

// arguments to user main(argc, argv)
// argc is returned via the system call return
// value, which goes in a0.
p->trapframe->a1 = sp;

```

유저 모드 진입 시 `argv` 포인터(`sp`)를 두 번째 인자 레지스터 `a1`에 설정한다.

```

// Save program name for debugging.
for(last=s=path; *s; s++)
    if(*s == '/')
        last = s+1;
safestrcpy(p->name, last, sizeof(p->name));

// Commit to the user image.
oldpagetable = p->pagetable;
p->pagetable = pagetable;
p->sz = sz;
p->trapframe->epc = elf.entry; // initial program counter = main
p->trapframe->sp = sp; // initial stack pointer
proc_freepagetable(oldpagetable, oldsiz);

return argc; // this ends up in a0, the first argument to main(argc, argv)

```

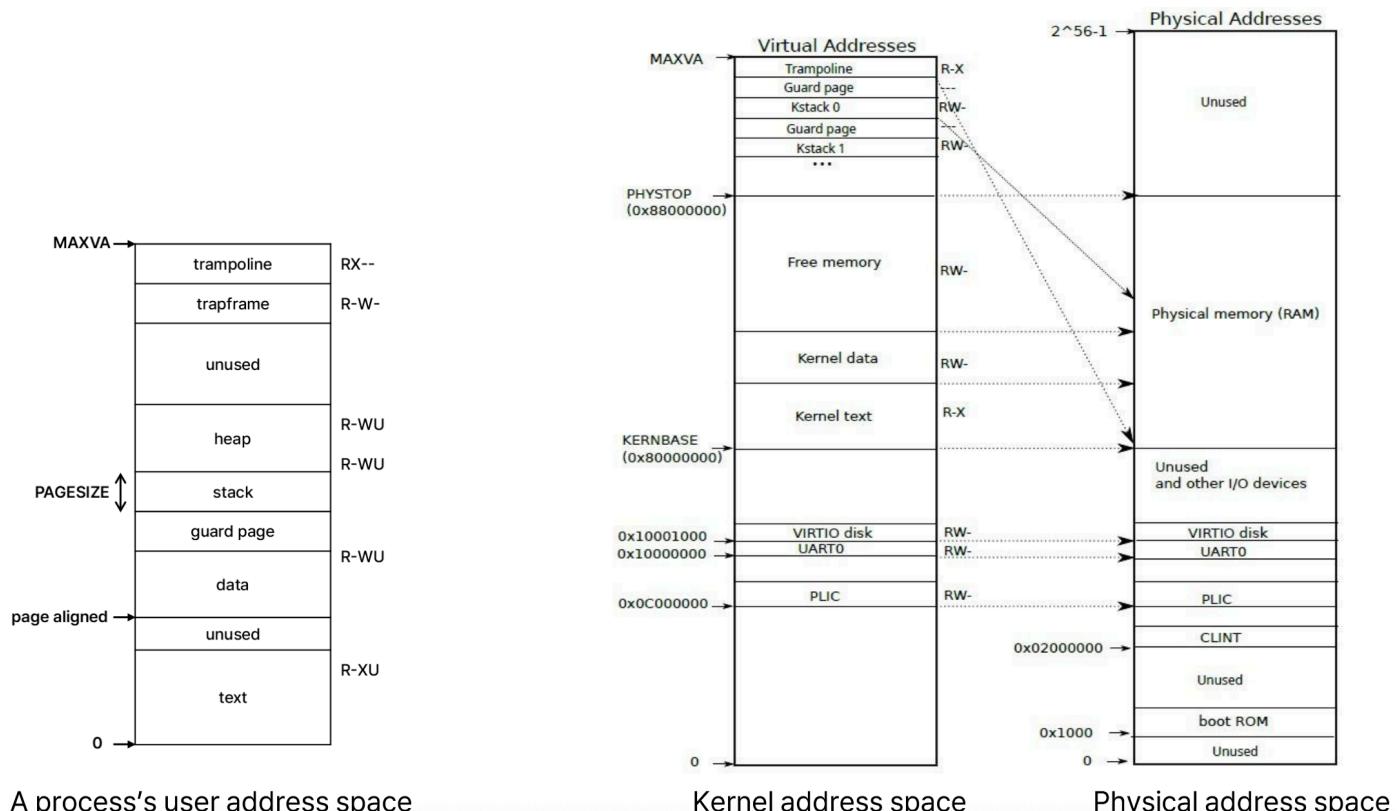
새 프로그램 이름을 `p->name`에 저장하고, 새 페이지 테이블, sz, 초기 PC(엔트리 포인트), 스택 포인터를 프로세스에 적용한 뒤 옛 페이지 테이블을 해제하고 `argc`를 반환한다.

## 메모리 구조와 페이지테이블에 대한 탐구

유저 페이지테이블이 로드된 상태에선 유저 메모리(예: 프로그램 코드·데이터·힙·유저 스택)에 연결되고,

커널 페이지테이블이 로드된 상태에선 커널 메모리(예: 커널 텍스트·데이터·물리 메모리 전역·각 프로세스의 커널 스택 등)에 연결된다.

따라서 같은 가상 주소라고 하더라도 페이지 테이블이 다르기 때문에 실제 물리 주소를 참조하는 방식이 달라지는 것이다.



페이지 테이블을 어떻게 바꾸는지 탐구해보자.

## satp 분석

xv6는 **SATP**(Supervisor Address Translation and Protection) 레지스터 값을 보고 어느 페이지 테이블(유저, 커널)을 사용할지 결정한다.

### 유저 모드에서 커널모드로 진입할 때

아래는 uservec의 코드 일부분이다.

트랩 발생 후 유저모드에서 커널모드로 진입하는 초기에 uservec이 호출되고 satp값에 t1(커널 페이지 테이블 값)을 쓴다.

```
uservec:
#
# trap.c sets stvec to point here, so
```

```

# traps from user space start here,
# in supervisor mode, but with a
# user page table.
#
...
# fetch the kernel page table address, from p->trapframe->kernel_satp.
ld t1, 0(a0)

# wait for any previous memory operations to complete, so that
# they use the user page table.
sfence.vma zero, zero

# install the kernel page table.
csrw satp, t1

# flush now-stale user entries from the TLB.
sfence.vma zero, zero

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

```

### 커널 모드에서 유저모드로 복귀할 때

프로세스의 pagetable필드는 프로세스의 유저 페이지 테이블 정보이다. 커널->유저 모드로 복귀할 때 usertrapret에서 satp 값을 유저 페이지 테이블로 바꾼다.

```

void
usertrapret(void)
{
...
// tell trampoline.S the user page table to switch to.
uint64 satp = MAKE_SATP(p->pagetable);
...
((void (*)(uint64, uint64))trampoline_userret)(satp, p->trapframe_va);
}

```

### 유저 페이지 테이블 생성

**proc\_pagetable**을 통해 유저 페이지 테이블을 생성해준다. **uvmccreate**를 통해서 최상위 페이지(level2)를 물리 메모리에 생성해준다. 기본적으로 트램폴린·트랩프레임 엔트리만 매핑한다. **mappages**를 통해 엔트리를 할당하는 과정에서 level1과 level0 페이지 테이블을 생성해준다.

pagetable\_t

```

proc_pagetable(struct proc *p)
{
    pagetable_t pagetable;

    // An empty page table.
    pagetable = uvmcreate(); // 최상위(level 2) 페이지 테이블 생성
    if(pagetable == 0)
        return 0;

    // map the trampoline code (for system call return)
    // at the highest user virtual address.
    // only the supervisor uses it, on the way
    // to/from user space, so not PTE_U.
    if(mappages(pagetable, TRAMPOLINE, PGSIZE, // mappages의 walk를 통해 level 1과 level 0 페이지 테이블을 생성함
                (uint64)trampoline, PTE_R | PTE_X) < 0){
        uvmfree(pagetable, 0);
        return 0;
    }

    // map the trapframe page just below the trampoline page, for
    // trampoline.S.
    if(mappages(pagetable, p->trapframe_va = TRAPFRAME, PGSIZE,
                (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
        uvmunmap(pagetable, TRAMPOLINE, 1, 0);
        uvmfree(pagetable, 0);
        return 0;
    }

    return pagetable;
}

```

## 커널 페이지 테이블 생성

**main** 함수 초기에 **kvminit**이라는 함수가 호출된다.

```

// start() jumps here in supervisor mode on all CPUs.
void
main()
{
    if(cpuid() == 0){
        ...
        kvminit();           // create kernel page table
        ...
    } else {
        ...
    }

    scheduler();
}

```

**kvminit**은 **kvmmake**를 통해 커널 페이지 테이블을 생성해준다.

```
// Initialize the one kernel_pagetable
void
kvminit(void)
{
    kernel_pagetable = kvmmake();
}
```

**kvmmake**에서 **kalloc**을 통해 실제 물리 메모리를 생성하고 **kvmmmap**을 통해 커널 페이지 테이블의 엔트리들을 물리주소로 매핑해준다.

```
// Make a direct-map page table for the kernel.
pagetable_t
kvmmake(void)
{
    pagetable_t kpgtbl;

    kpgtbl = (pagetable_t) kalloc();
    memset(kpgtbl, 0, PGSIZE);

    // uart registers
    kvmmmap(kpgtbl, UART0, UART0, PGSIZE, PTE_R | PTE_W);

    // virtio mmio disk interface
    kvmmmap(kpgtbl, VIRTIO0, VIRTIO0, PGSIZE, PTE_R | PTE_W);

    // PLIC
    kvmmmap(kpgtbl, PLIC, PLIC, 0x4000000, PTE_R | PTE_W);

    // map kernel text executable and read-only.
    kvmmmap(kpgtbl, KERNBASE, KERNBASE, (uint64)etext-KERNBASE, PTE_R | PTE_X);

    // map kernel data and the physical RAM we'll make use of.
    kvmmmap(kpgtbl, (uint64)etext, (uint64)etext, PHYSTOP-(uint64)etext, PTE_R | PTE_W);

    // map the trampoline for trap entry/exit to
    // the highest virtual address in the kernel.
    kvmmmap(kpgtbl, TRAMPOLINE, (uint64)trampoline, PGSIZE, PTE_R | PTE_X);

    // allocate and map a kernel stack for each process.
    proc_mapstacks(kpgtbl);

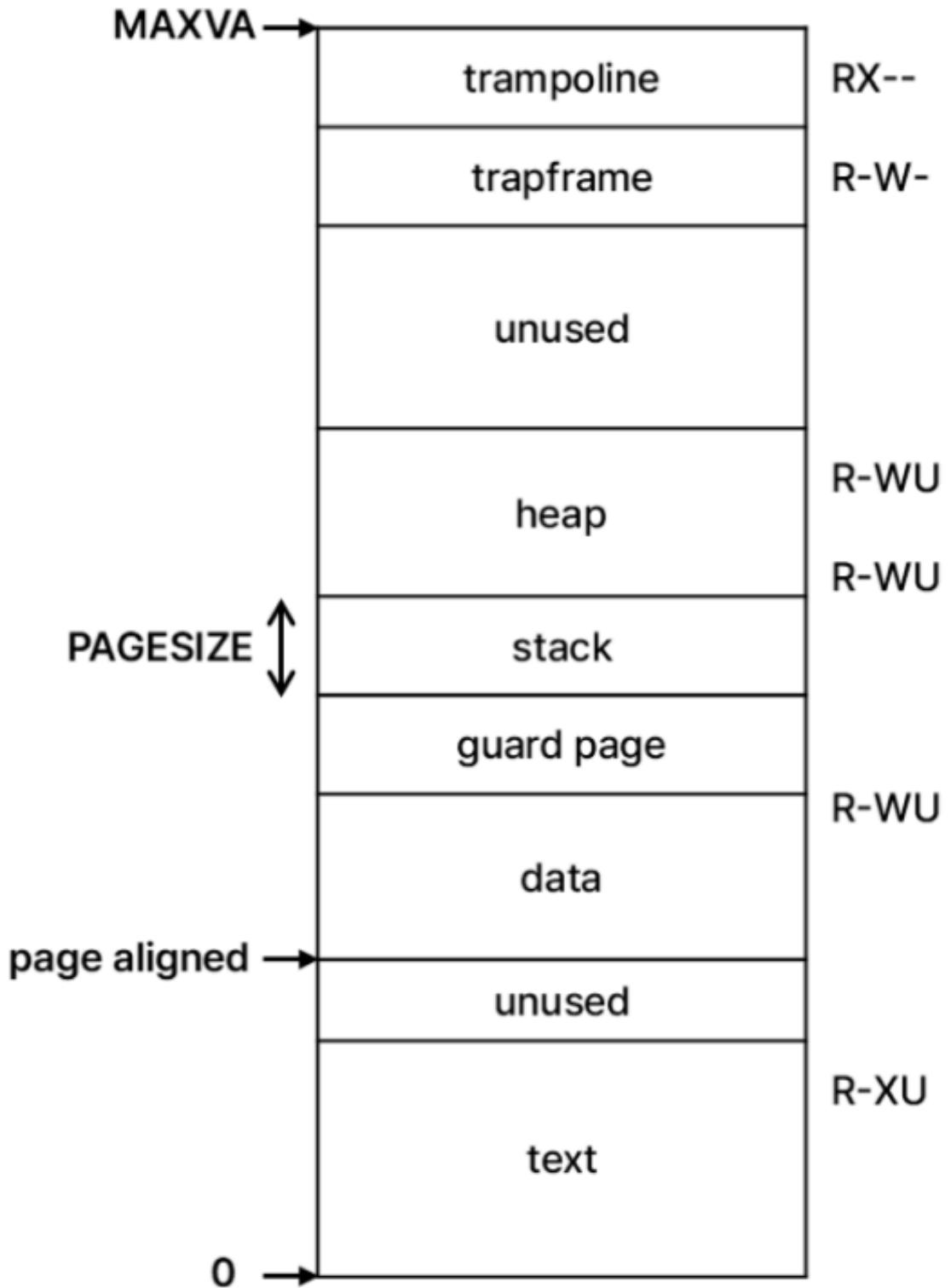
    return kpgtbl;
}
```

**Kvmmmap**을 통해 커널 페이지 테이블의 엔트리들을 물리주소로 매핑해주는 코드이다.

```
// add a mapping to the kernel page table.  
// only used when booting.  
// does not flush TLB or enable paging.  
  
void  
kvmmmap(pagetable_t kpgtbl, uint64 va, uint64 pa, uint64 sz, int perm)  
{  
    if(mappages(kpgtbl, va, sz, pa, perm) != 0)  
        panic("kvmmmap");  
}
```

## 유저 메모리(user address space)의 구성

유저 메모리는 다음과 같이 구성된다. 각 영역들에 대한 가상 주소가 어떻게 페이지 테이블에 매핑되는지 살펴보자.



## A process's user address space

가장 위의 **trampoline**과 **trapframe**이 페이지테이블에 매핑되는 과정을 먼저 살펴보자. 아까 봤던 유저 페이징 테이블을 생성하는 코드를 다시 한 번 확인해보면 페이징 테이블을 생성하면서 페이징 테이블의 **trampoline** 가상 주소와 **trapframe** 가상 주소를 유저 페이징 테이블에 매핑해주는 것을 확인해볼 수 있다.

```

#define TRAMPOLINE (MAXVA - PGSIZE)
#define TRAPFRAME (TRAMPOLINE - PGSIZE)

// Create a user page table for a given process, with no user memory,
// but with trampoline and trapframe pages.
pagetable_t
proc_pagetable(struct proc *p)
{
    pagetable_t pagetable;

    // An empty page table.
    pagetable = uvmcreate(); // 최상위(level 2) 페이지 테이블 생성
    if(pagetable == 0)
        return 0;

    // map the trampoline code (for system call return)
    // at the highest user virtual address.
    // only the supervisor uses it, on the way
    // to/from user space, so not PTE_U.
    if(mappages(pagetable, TRAMPOLINE, PGSIZE, // mappages의 walk를 통해 level 1과 level 0 페이지 테이블을 생성함
                (uint64)trampoline, PTE_R | PTE_X) < 0){
        uvmfree(pagetable, 0);
        return 0;
    }

    // map the trapframe page just below the trampoline page, for
    // trampoline.S.
    if(mappages(pagetable, p->trapframe_va = TRAPFRAME, PGSIZE,
                (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
        uvmunmap(pagetable, TRAMPOLINE, 1, 0);
        uvmfree(pagetable, 0);
        return 0;
    }

    return pagetable;
}

```

이제 **text**영역과 **data**영역이 어떻게 페이지 테이블에 매핑되는지 확인해보자. 이 부분을 확인하기 위해서는 프로세스가 만들어지는 과정을 살펴봐야한다.

### 최초의 프로세스가 **text**영역과 **data**영역을 페이지 테이블에 매핑하는 과정

우선 `userinit`에서 `uvmfirst` 함수를 살펴봐야한다. 일단 `allocproc`에서 `proc_pagetable`호출을 통해 기본적인 페이지 테이블이 만들어지고 **trampoline**과 **trapframe**영역은 매핑이 된 상태이다.

```

// Set up first user process.
void
userinit(void)
{

```

```

struct proc *p;

p = allocproc();
initproc = p;

// allocate one user page and copy initcode's instructions
// and data into it.
uvmfirst(p->pagetable, initcode, sizeof(initcode));
...
}

```

`uvmfirst`에서 유저 메모리 0번지부터 **PGSIZE**까지 **initcode**바이너리를 옮겨준다.

```

// a user program that calls exec("/init")
// assembled from ../user/initcode.S
// od -t xc ../user/initcode
uchar initcode[] = {
    0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x45, 0x02,
    0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x35, 0x02,
    0x93, 0x08, 0x70, 0x00, 0x73, 0x00, 0x00, 0x00,
    0x93, 0x08, 0x20, 0x00, 0x73, 0x00, 0x00, 0x00,
    0xef, 0xf0, 0x9f, 0xff, 0x2f, 0x69, 0x6e, 0x69,
    0x74, 0x00, 0x00, 0x24, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00
};

```

```

// Load the user initcode into address 0 of pagetable,
// for the very first process.
// sz must be less than a page.
void
uvmfirst(pagetable_t pagetable, uchar *src, uint sz)
{
    char *mem;

    if(sz >= PGSIZE)
        panic("uvmfirst: more than a page");
    mem = kalloc();
    memset(mem, 0, PGSIZE);
    mappages(pagetable, 0, PGSIZE, (uint64)mem, PTE_W|PTE_R|PTE_X|PTE_U);
    memmove(mem, src, sz);
}

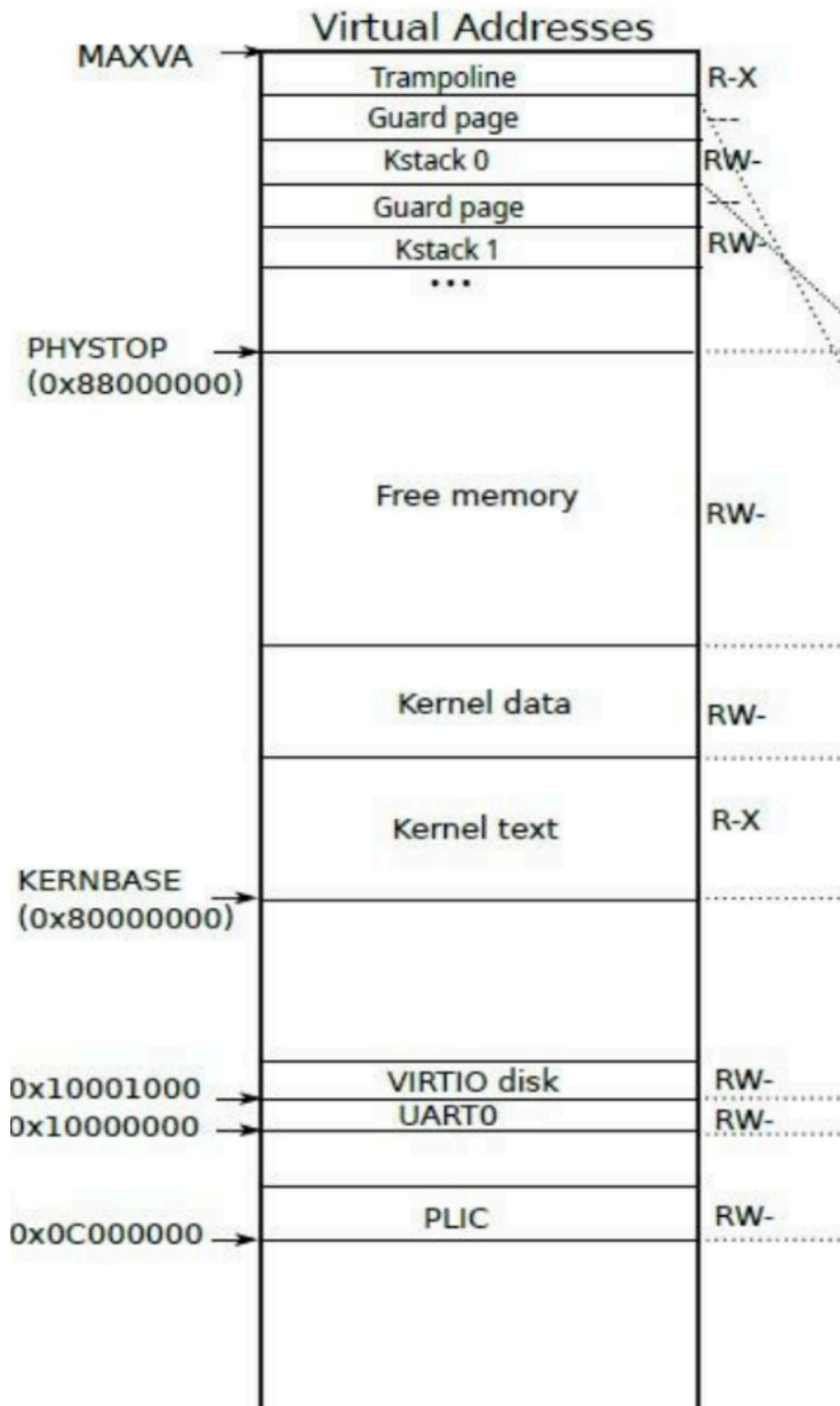
```

이후에 **initcode**부분이 실행되면서 `exec("/init")` 호출을 통해 두 번째 프로세스부터는 **exec**을 통한 **text/data**영역 매핑이 일어나는 것이다.

즉, 최초의 프로세스인 **initcode**프로세스는 텍스트 영역에 **initcode** 한페이지 밖에 존재하지 않는 상황인 것이다.

## 커널 메모리(kernel address space)의 구성

유저 메모리는 다음과 같이 구성된다. 각 영역들에 대한 가상 주소가 어떻게 페이지 테이블에 매핑되는지 살펴보자.





# Kernel address space

## 메모리의 할당과 해제

우리의 xv6는 riscv's sv39 page table scheme 을 참고하고 있다.

가상 주소

| 38     | 30 29  | 21 20  | 12 11       | 0 |
|--------|--------|--------|-------------|---|
| VPN[2] | VPN[1] | VPN[0] | page offset |   |
| 9      | 9      | 9      | 12          |   |

페이지 테이블 엔트리

| 63       | 54 53  | 28 27  | 19 18  | 10 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|--------|--------|--------|------|---|---|---|---|---|---|---|---|---|
| Reserved | PPN[2] | PPN[1] | PPN[0] | RSW  | D | A | G | U | X | W | R | V |   |
| 10       | 26     | 9      | 9      | 2    | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |   |

메모리 할당 과정

요약

- `kalloc`을 통해 `kmem.freelist`로부터 비어있는 물리 프레임 하나를 반환받는다.
- `kalloc`으로 할당받은 물리 프레임을 `mappages`를 통해 페이지 테이블에 맵핑한다.

`kalloc`

`kmem.freelist`로부터 비어있는 물리 프레임 하나를 반환받음. `junk 0x05` 값으로 채우는 이유는 내가 나중에 디버깅하면서 초기화 하지 않은 값을 확인해볼 수 있기 때문.

```
void *
kalloc(void)
{
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r)
        kmem.freelist = r->next;
    release(&kmem.lock);
```

```

if(r)
    memset((char*)r, 5, PGSIZE); // fill with junk
return (void*)r;
}

```

## mappages

가상주소 `va`부터 길이 `size`만큼을 페이지 단위로 순회하면서, 물리주소 `pa`에서 시작하는 메모리 블록을 `perm|PTE_V` 권한으로 매핑하고, 중간에 필요한 페이지 테이블이 없으면 할당하며, 성공 시 0, 실패 시 -1을 반환한다. `walk`를 통해 페이지 테이블이 없는 경우 생성해준다. 마지막 페이지 레벨의 엔트리를 물리 프레임 값을 가리키도록 한다.

```

int
mappages(pagetable_t pagetable, uint64 va, uint64 size, uint64 pa, int perm)
{
    uint64 a, last;
    pte_t *pte;

    if((va % PGSIZE) != 0)
        panic("mappages: va not aligned");

    if((size % PGSIZE) != 0)
        panic("mappages: size not aligned");

    if(size == 0)
        panic("mappages: size");

    a = va; // 현재 처리할 가상 페이지의 시작주소
    last = va + size - PGSIZE; // 마지막으로 매핑할 페이지의 시작주소
    for(;;){
        if((pte = walk(pagetable, a, 1)) == 0) // level 0 페이지 테이블에서의 엔트리 주소
            return -1;
        if(*pte & PTE_V)
            panic("mappages: remap");
        *pte = PA2PTE(pa) | perm | PTE_V; // 엔트리 값을 물리 프레임 값으로 매핑함
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}

```

## walk

주어진 페이지 테이블과 가상주소 `va`에 대해 PTE(페이지 테이블 엔트리)의 물리 프레임의 위치를 찾기 위한 함수이다.

`alloc=1`일 때는, 해당 엔트리가 속한 중간 페이지 테이블 페이지가 아직 없으면 새로 할당해 준다.

`alloc=0`일 때는 없으면 그냥 `NULL`을 리턴한다.

```

pte_t *
walk(pagetable_t pagetable, uint64 va, int alloc) // 3단계 페이징 기법
{
    if(va >= MAXVA)
        panic("walk");

    for(int level = 2; level > 0; level--) {
        pte_t *pte = &pagetable[PX(level, va)]; // 해당 레벨의 인덱스에 대응하는 엔트리 추출
        if(*pte & PTE_V) { // PTE valid
            pagetable = (pagetable_t)PTE2PA(*pte); // 다음 레벨 페이지테이블의 물리 프레임을 포인팅
        } else { // PTE invalid, alloc이 1인 경우에만 페이지 테이블 생성
            if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
                return 0;
            memset(pagetable, 0, PGSIZE); // 할당한 페이지 테이블의 모든 엔트리를 0으로 초기화
            *pte = PA2PTE(pagetable) | PTE_V; // 엔트리 값 방금 만든 페이지에 연결해줌
        }
    }
    return &pagetable[PX(0, va)]; // 마지막 단계까지 내려옴 -> level 0에 대응하는 엔트리 추출
}

```

## 메모리 할당 관련 상위 레벨 api인 uvmalloc

`uvmalloc`은 현재 프로세스 브레이크(sz)로부터 확장할 브레이크(sz)까지 페이지를 추가할 때 사용된다. `sbrk(n)` 시스템 콜이 호출될 경우 `growproc(n)` 함수가 `uvmalloc`을 호출하여 힙 브레이크를 확장해준다.

```

// Allocate PTEs and physical memory to grow process from oldsz to
// newsz, which need not be page aligned. Returns new size or 0 on error.
uint64
uvmalloc(pagetable_t pagetable, uint64 oldsz, uint64 newsz, int xperm)
{
    char *mem;
    uint64 a;

    if(newsz < oldsz)
        return oldsz;

    oldsz = PGROUNDUP(oldsz);
    for(a = oldsz; a < newsz; a += PGSIZE){ // newsz는 PGROUNDUP하기 이전의 oldsz + n인데 n이
4096이상이어야 for문 조건이 참이 됨. n이 4096이상 8192미만이면 1번, 8192이상 12288미만이면 2번 ~~
        mem = kalloc(); // 성공시 커널 가상 공간에서 페이지 시작점 반환
        if(mem == 0){ // 할당 실패
            uvmdealloc(pagetable, a, oldsz);
            return 0;
        }
        memset(mem, 0, PGSIZE);
        if(mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_R|PTE_U|xperm) != 0){
            kfree(mem);
            uvmdealloc(pagetable, a, oldsz);
            return 0;
        }
    }
}

```

```

    return newsz;
}

```

## 메모리 해제 과정

### 요약

- uvmunmap을 통해 va부터 npages만큼의 페이지들의 물리 프레임들을 kfree해준다.
- freewalk를 통해 페이지 테이블들에 대한 물리 프레임들을 kfree해준다.

## kfree, freerange

kfree()는 kmem.freelist에 free한 물리 프레임을 추가하고 junk 0x01로 채운다. kfree()를 영역 단위로 실행하는 게 freerange()이다.

```

void
kfree(void *pa)
{
    struct run *r;

    if(((uint64)pa % PGSIZE) != 0 || (char*)pa < end || (uint64)pa >= PHYSTOP)
        panic("kfree");

    // Fill with junk to catch dangling refs.
    memset(pa, 1, PGSIZE);

    r = (struct run*)pa;

    acquire(&kmem.lock);
    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
}

void
freerange(void *pa_start, void *pa_end)
{
    char *p;
    p = (char*)PGROUNDUP((uint64)pa_start);
    for(; p + PGSIZE <= (char*)pa_end; p += PGSIZE)
        kfree(p);
}

```

## uvmunmap

가상 주소 va가 page aligned되어 있는지 검사 후 va부터 npages만큼의 가상 주소 범위의 물리 프레임들을 kfree해준다.

```

void

```

```

uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
    uint64 a;
    pte_t *pte;

    if((va % PGSIZE) != 0)
        panic("uvmunmap: not aligned");

    for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
        if((pte = walk(pagetable, a, 0)) == 0)
            panic("uvmunmap: walk");
        if((*pte & PTE_V) == 0)
            panic("uvmunmap: not mapped");
        if(PTE_FLAGS(*pte) == PTE_V)
            panic("uvmunmap: not a leaf");
        if(do_free){
            uint64 pa = PTE2PA(*pte);
            kfree((void*)pa);
        }
        *pte = 0;
    }
}

```

## freewalk

재귀적으로 페이지테이블에 할당된 페이지들을 차례차례 해제해준다.

```

// Recursively free page-table pages.
// All leaf mappings must already have been removed.
void
freewalk(pagetable_t pagetable)
{
    // there are 2^9 = 512 PTEs in a page table.
    for(int i = 0; i < 512; i++){
        pte_t pte = pagetable[i];
        if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0){
            // this PTE points to a lower-level page table.
            uint64 child = PTE2PA(pte);
            freewalk((pagetable_t)child);
            pagetable[i] = 0;
        } else if(pte & PTE_V){
            panic("freewalk: leaf");
        }
    }
    kfree((void*)pagetable);
}

```

## uvmfree

가상 주소 0번지부터 sz번지까지의 페이지들을 `uvmunmap`을 통해 `kfree`해주고 `freewalk`를 통해 페이지 테이블들을 `kfree`해준다.

```

// Free user memory pages,
// then free page-table pages.

void
uvmfree(pagetable_t pagetable, uint64 sz)
{
    if(sz > 0)
        uvmunmap(pagetable, 0, PGROUNDUP(sz)/PGSIZE, 1);
    freewalk(pagetable);
}

```

## Context Switch 절차(TRAPFRAME 중심)

context switch가 발생하는 과정을 **trapframe** 중심으로 살펴보자

유저 코드가 실행되는 도중에 타이머 인터럽트가 발생하여 **context switch**가 발생하는 과정을 확인해보자.

유저->커널 진입 과정(uservec)에서 이전 프로세스의 TRAPFRAME을 저장하고 usertrap으로 점프한다

```

.section trampsec
.globl trampoline
.globl usertrap
trampoline:
.align 4
.globl uservec
uservec:
#
# trap.c sets stvec to point here, so
# traps from user space start here,
# in supervisor mode, but with a
# user page table.
#
.csrrw a0, sscratch, a0

# save the user registers in TRAPFRAME
sd ra, 40(a0)
sd sp, 48(a0)
...
# save the user a0 in p->trapframe->a0
csrr t0, sscratch
sd t0, 112(a0)

...
# install the kernel page table.
csrw satp, t1

# flush now-stale user entries from the TLB.
sfence.vma zero, zero

```

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

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

    if((r_sstatus() & SSTATUS_SPP) != 0)
        panic("usertrap: not from user mode");

    // send interrupts and exceptions to kerneltrap(),
    // since we're now in the kernel.
    w_stvec((uint64)kernelvec);

    // 부모 프로세스가 clone syscall을 호출하여 usertrap에 진입한 시점에 myproc을 하면 쓰레드의 정보임
    struct proc *p = myproc();

    p->trapframe->epc = r_sepc();

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

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

        // sepc points to the ecall instruction,
        // but we want to return to the next instruction.
        p->trapframe->epc += 4;

        // an interrupt will change sepc, scause, and sstatus,
        // so enable only now that we're done with those registers.
        intr_on();

        syscall();
    } else if((which_dev = devintr()) != 0){
        //timer/device interrupt
        // 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();  
}
```

## 타이머 인터럽트의 경우 yield 호출

```
// Give up the CPU for one scheduling round.  
void  
yield(void)  
{  
    struct proc *p = myproc();  
    acquire(&p->lock);  
    p->state = RUNNABLE;  
    sched();  
    release(&p->lock);  
}
```

## swtch를 통해 이전 프로세스의 context -> 스케줄러의 context

```
// Switch to scheduler. Must hold only p->lock  
// and have changed proc->state. Saves and restores  
// intena because intena is a property of this  
// kernel thread, not this CPU. It should  
// be proc->intena and proc->noff, but that would  
// break in the few places where a lock is held but  
// there's no process.  
void  
sched(void)  
{  
    int intena;  
    struct proc *p = myproc();  
  
    if(!holding(&p->lock))  
        panic("sched p->lock");  
    if(mycpu()->noff != 1)  
        panic("sched locks");  
    if(p->state == RUNNING)  
        panic("sched running");  
    if(intr_get())  
        panic("sched interruptible");  
  
    intena = mycpu()->intena;  
  
    swtch(&p->context, &mycpu()->context);  
    mycpu()->intena = intena;  
}
```

## swtch를 통해 스케줄러의 context -> RUNNABLE 프로세스의 context

```
void
```

```

scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();

    c->proc = 0;
    for(;;){
        // The most recent process to run may have had interrupts
        // turned off; enable them to avoid a deadlock if all
        // processes are waiting.
        intr_on();

        int found = 0;
        for(p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock);
            if(p->state == RUNNABLE) {
                // Switch to chosen process. It is the process's job
                // to release its lock and then reacquire it
                // before jumping back to us.
                p->state = RUNNING;
                c->proc = p;
                swtch(&c->context, &p->context);

                // Process is done running for now.
                // It should have changed its p->state before coming back.
                c->proc = 0;
                found = 1;
            }
            release(&p->lock);
        }
        if(found == 0) {
            // nothing to run; stop running on this core until an interrupt.
            intr_on();
            asm volatile("wfi");
        }
    }
}

```

## usertrapret

```

// 
// return to user space
//
void
usertrapret(void)
{
    struct proc *p = myproc();

    // //추가추가
    // if(p->isThread){
    //     uvmunmap(p->pagetable, TRAPFRAME, 1, 0);

```

```

//    mappages(p->pagetable, TRAPFRAME, PGSIZE, (uint64)p->trapframe, PTE_R | PTE_W);
// }

// //추가추가

// we're about to switch the destination of traps from
// kerneltrap() to usertrap(), so turn off interrupts until
// we're back in user space, where usertrap() is correct.
intr_off();

// send syscalls, interrupts, and exceptions to uservc in trampoline.S
uint64 trampoline_uservc = TRAMPOLINE + (uservc - trampoline);

w_stvec(trampoline_uservc);

// set up trapframe values that uservc will need when
// the process next traps into the kernel.
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;
p->trapframe->kernel_hartid = r_tp();           // hartid for cpuid()

// 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);

// set S Exception Program Counter to the saved user pc.
w_sepc(p->trapframe->epc);

asm volatile("csrw sscratch, %0" : : "r" (p->trapframe_va));

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

// 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);

((void (*)(uint64, uint64))trampoline_userret)(satp, p->trapframe_va);
}

```

## userret

```
.globl userret
userret:
    # userret(pagetable, trapframe_va)
    # called by usertrapret() in trap.c to
    # switch from kernel to user.
    # a0: user page table, for satp.
    # a1: trapframe virtual address.

    # switch to the user page table.
    sfence.vma zero, zero
    csrw satp, a0
    sfence.vma zero, zero

    mv a0, a1

    # 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)
    ld s0, 96(a0)
    ld s1, 104(a0)
    ld a1, 120(a0)
    ld a2, 128(a0)
    ld a3, 136(a0)
    ld a4, 144(a0)
    ld a5, 152(a0)
    ld a6, 160(a0)
    ld a7, 168(a0)
    ld s2, 176(a0)
    ld s3, 184(a0)
    ld s4, 192(a0)
    ld s5, 200(a0)
    ld s6, 208(a0)
    ld s7, 216(a0)
    ld s8, 224(a0)
    ld s9, 232(a0)
    ld s10, 240(a0)
    ld s11, 248(a0)
    ld t3, 256(a0)
    ld t4, 264(a0)
    ld t5, 272(a0)
    ld t6, 280(a0)

    # restore user a0
    ld a0, 112(a0)

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

# Design

## trapframe 관리 방법

이번 과제에서 가장 핵심은 쓰레드의 **trapframe**을 어떻게 관리할 것인가이다.

프로세스의 user address space와 그 프로세스의 쓰레드들의 user address space는 동일해야하는데, 프로세스와 쓰레드의 **trapframe\_va**가 같은 위치(TRAPFRAME)를 가리키면 어떻게 문제가 되는지 고민해봐야한다.

우리의 이번 과제에서는 커널로 진입하는 두 가지의 경우를 살펴봐야한다.

1. clone system call을 만들었을 때 system call 호출로 인한 커널 진입
2. 타이머 인터럽트가 발생했을 때 커널로 진입

1번의 경우 유저 레벨-> **clone syscall**(커널 진입) -> **uservec**에서 트랩프레임 저장 -> **usertrap** -> **clone** -> **usertrapret**에서 트랩프레임 복원 -> 유저 레벨 복귀

2번의 경우 유저 레벨 -> 타이머 인터럽트 발생 -> **uservec**에서 트랩프레임 저장 -> **usertrap** -> **yield** -> **scheduler**에서 프로세스 선택 -> **usertrapret**에서 트랩프레임 복원 -> 유저 레벨 복귀

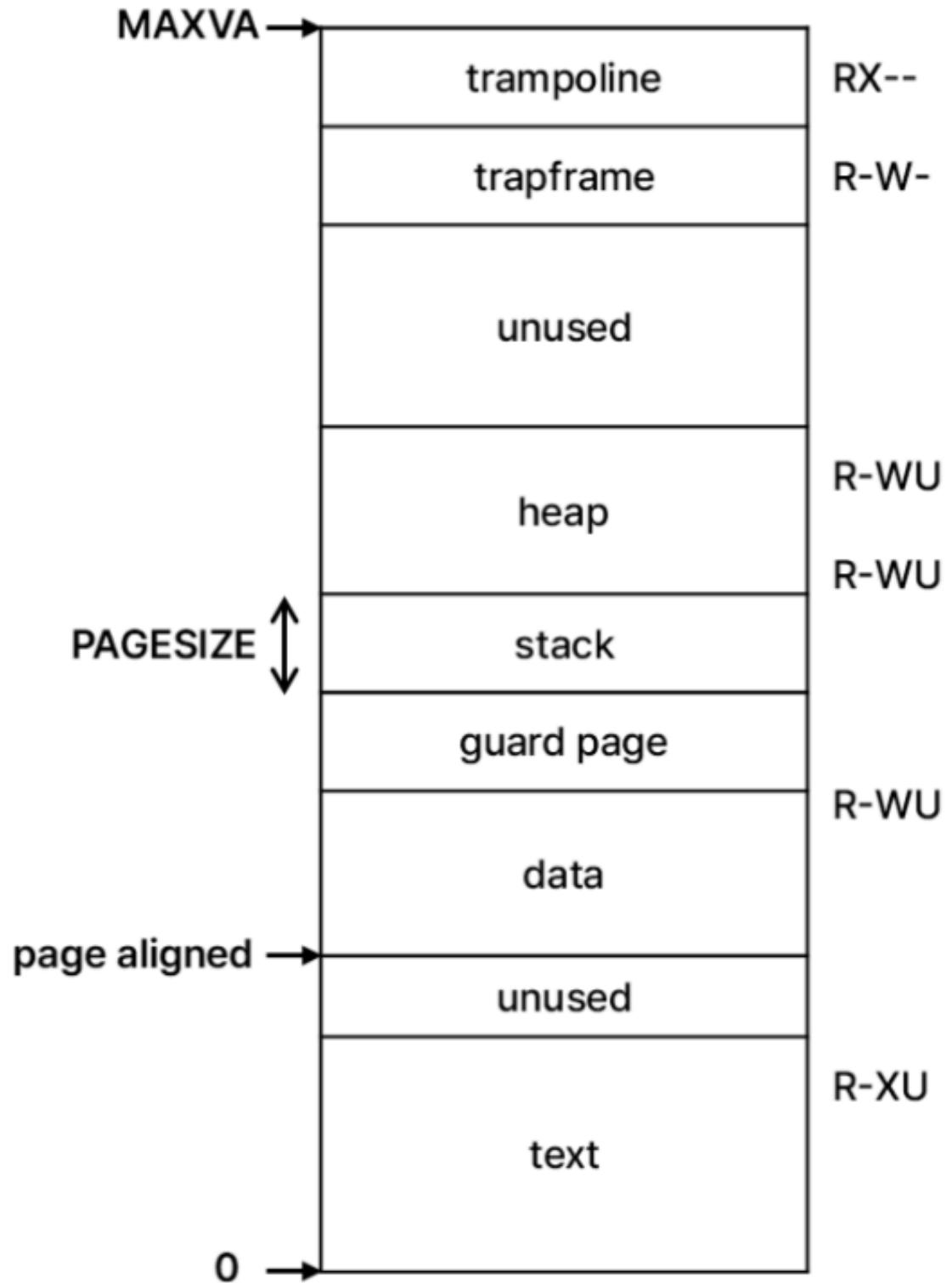
**usertrapret**에서 트랩프레임을 복원하는 부분을 살펴보면 **satp**를 유저 페이지 테이블로 기록하고 트랩프레임을 복원한다. 즉, 트랩프레임을 불러오는 과정이 **trapframe\_va**로 유저 페이지 테이블 엔트리를 통해 불러온다는 것이다.

만약 내가 **clone** 과정에서 쓰레드의 물리 프레임을 할당하고 프로세스와 동일한 TRAPFRAME 위치를 **trapframe\_va**가 가리키도록 만든다고 해보자. 나중에 실행할 때 타이머 인터럽트로 인한 context switch가 발생하여 쓰레드가 **usertrapret**에서 트랩프레임을 복원하면 프로세스와 쓰레드의 **trapframe\_va**가 동일하고 유저 페이징 테이블도 동일하기 때문에 결국 같은 물리 프레임을 가리켜 쓰레드가 복원하는 트랩프레임 값이 프로세스가 커널 모드를 진입하면서 저장한 트랩프레임 값이 되어버린다.

사실 이 뿐만이 아니라 **clone** 과정에서 물리 프레임을 할당하고 mappages를 통해 페이지 테이블에 맵핑할 때마다 TRAPFRAME이 실제로 가리키는 물리프레임 위치는 매번 바뀌게 되는데, 결국 첫 context switch 과정에서 프로세스가 트랩프레임을 저장하는 위치가 자신의 물리 프레임 위치가 아닌 마지막으로 **clone**한 쓰레드의 물리 프레임 위치가 되어버린다.

"그래서 어떻게 해야될까?"

USER ADDRESS SPACE를 보면 TRAPFRAME(이 부분을 프로세스의 트랩프레임으로 설정하고)아래에 unused 영역이 있다. 이 부분에 쓰레드별로 한 페이지씩 쓰레드의 **trapframe\_va**가 가리키도록 만들면 된다.



## A process's user address space

**stack 관리 방법**

프로세스와 쓰레드는 동일한 user address space를 공유한다.

그리고 clone system call의 형태가 다음과 같은 모습인 것으로 보아 유저 영역에서 stack 영역을 할당해주고 인자로 넘겨주어야 하는 것 같다.

```
int clone(void (*fcn)(void *, void *), void *arg1, void *arg2, void *stack);
```

따라서 쓰레드의 stack 값은 malloc을 통해 힙 영역에 할당하기로 계획하였다.

## fork와 clone 비교

| 비교 항목        | fork()   | clone()   |
|--------------|--|---|
| 주소 공간        | uvmcopy()로 부모의 유저 메모리(코드·데이터·힙·스택)를 완전 복사                          | 부모의 pagetable을 공유   |
| 사용자 스택       | 부모와 완전히 독립된 스택   | 호출자가 alloc_stack()로 미리 할당한 페이지 한장을 새 쓰레드 스택으로 사용              |
| 실행 진입점 (EPC) | 부모와 같은 trapframe->epc에서 시작   | clone(fcn,...)의 fcn으로 설정하여 지정한 함수에서 시작                        |
| 반환값 (a0)     | 자식 프로세스에서 a0 = 0   | 새 스레드에서 a0 = 0  |
| 부모 설정        | wait_lock 아래에서 np->parent = p                                      | wait_lock 아래에서 np->parent = currproc                          |
| 락 사용         | allocproc() 가 잡은 p->lock 해제 후<br>ptable.lock 아래에서 state = RUNNABLE | allocproc() 가 잡은 np->lock 해제 후 np->lock 아래에서 state = RUNNABLE |
| 파일·CWD 복제    | filedup() / idup()로 부모의 열린 파일·cwd 참조 카운트 증가                        | fork와 동일  |
| 프로세스 이름      | safestrcpy(np->name, p->name)                                      | safestrcpy(np->name, currproc->name)                          |
| 커널 컨텍스트      | np->context.ra = forkret np->context.sp = kstack+PGSIZE            | fork와 동일  |
| 리소스 해제       | proc_freepagetable(oldpagetable, oldsz)로 옛 페이지테이블 해제               | 주소 공간은 공유하므로 해제하지 않음; join()에서 trapframe·kstack만 해제           |

## wait과 join 비교

| 특징             | <code>wait(uint64 addr)</code>  | <code>join(void **stack)</code>   |
|----------------|---|---|
| 대상             | <code>fork()</code> 로 생성된 자식 프로세스   | <code>clone()</code> 으로 생성된 자식 스레드(같은 pagetable)                                  |
| 주소 공간          | 완전히 독립된 자식 주소 공간  | 부모와 공유하는 주소 공간  |
| 식별 조건          | <code>pp-&gt;parent == cur</code>   | <code>p-&gt;parent == cur &amp;&amp; p-&gt;pagetable == cur-&gt;pagetable</code>  |
| 종료 탐지 상태       | <code>pp-&gt;state == ZOMBIE</code>   | <code>p-&gt;state == ZOMBIE</code>  |
| 반환값            | 종료된 자식의 PID(성공) / -1(실패)  | 종료된 스레드의 PID(성공) / -1(실패)   |
| 종료 코드 전달       | <code>copyout()</code> 로 사용자 메모리( <code>addr</code> )에 <code>xstate</code> 복사     | 사용자 스택 주소( <code>*stack = p-&gt;ustack</code> )로 반환                               |
| 해제되는 자원        | <code>freeproc()</code> → trapframe, pagetable, ustack, kstack 등 모두 해제            | trapframe, kstack 해제, ustack 반환, pagetable은 그대로 유지                                |
| 락(lock)        | <code>wait_lock</code> 로 자식 목록 전체 보호  | 동일하게 <code>wait_lock</code> 사용  |
| 재대기(sleep)/깨우기 | <code>sleep(cur, &amp;wait_lock) ← exit() 시</code><br><code>wakeup(parent)</code> | <code>sleep(cur, &amp;wait_lock) ← exit() 시</code><br><code>wakeup(parent)</code> |
| 사용 예시          | 독립 실행 중인 자식 프로세스가 종료될 때까지 대기  | 같은 프로세스 안에서 생성된 스레드가 종료될 때까지 대기   |

# Implementation

## allocthread

`clone`을 구현하기 위해 쓰레드를 생성해주는 함수 `allocthread`를 우선적으로 구현해야한다. `allocthread(struct proc *p)`의 인자로 들어가는 `p`는 무조건 쓰레드가 아닌 프로세스여야 하도록 구현하였다.

- 프로세스 테이블을 순회하며 **UNUSED**인 프로세스를 찾으면 해당 프로세스(쓰레드)의 **pid, state, is\_thread**를 설정해 준다.
- 쓰레드의 **pagetable**이 프로세스의 **pagetable**을 가리키도록 만들었다.
- `kalloc`을 통해 트랩프레임을 물리 프레임에 할당하고 **np->trapframe**이 트랩프레임이 실제로 저장된 물리 프레임을 가리키도록 만들었다.
- 만약 `kalloc`을 실패하면 `freethread(np)`가 실행된다.
- 쓰레드가 만들어졌으니 프로세스의 **p->thread\_count**를 하나 증가시켜준다.

- `np->trapframe_va`가 `TRAPFRAME - PGSIZE*(p->thread_count)`를 가리키도록 한다. 이것은 기존 프로세스의 user address space 트랩프레임의 아래에 쓰레드의 트랩프레임을 매핑해주는 코드이다. 처음 쓰레드가 생성된 경우는 `thread_count`가 1이므로 바로 아래에 생성된다. 그 이후부터 쭉 이어서 아래에 생성되는 구조이다. `np->trapframe_va`가 유저 페이지 테이블을 통해 생성된 물리 프레임 `np->trapframe`을 참조할 수 있도록 `mappages` 를 통해 유저 페이지 테이블에 매핑해주자.

```

struct proc*
allocethread(struct proc *p) // 프로세스를 인자로 받음
{
    struct proc *np;

    for(np = proc; np < &proc[NPROC]; np++) {
        acquire(&np->lock);
        if(np->state == UNUSED) {
            goto found;
        } else {
            release(&np->lock);
        }
    }
    return 0;
}

found:
    np->pid = allocpid();
    np->state = USED;
    np->is_thread = 1;

    // process's user page table.
    np->pagetable = p->pagetable;

    // Allocate a trapframe page.
    if((np->trapframe = (struct trapframe *)kalloc()) == 0){
        freethread(np);
        release(&np->lock);
        return 0;
    }

    // 쓰레드 수 증가
    p->thread_count++;
    // 쓰레드의 트랩프레임 매핑
    if(mappages(np->pagetable, np->trapframe_va = TRAPFRAME - PGSIZE*(p->thread_count),
PGSIZE, (uint64)(np->trapframe), PTE_R | PTE_W) < 0){
        panic("panic: thread trapframe mapping failed");
        return 0;
    }

    // Set up new context to start executing at forkret,
    // which returns to user space.
    memset(&np->context, 0, sizeof(np->context));
    np->context.ra = (uint64)forkret;
    np->context.sp = np->kstack + PGSIZE;

```

```
    return np;
}
```

만약 `kalloc`이 실패하면 `freethread`를 호출한다.

## freethread

`allocthread`에서 호출되는 경우는 `p->trapframe`이 이미 0인 상태이다. `allocthread`가 호출하는 경우는 필드들을 정리해준다는 것에 의미가 있다.

```
void
freethread(struct proc *p)
{
    // 1) 사용자 트랩프레임 매핑 해제
    if (p->trapframe_va) {
        uvmunmap(p->pagetable, p->trapframe_va, 1, 0);
        p->trapframe_va = 0;
    }
    // 2) 커널 측 trapframe 해제
    if (p->trapframe) {
        kfree((void*)p->trapframe);
        p->trapframe = 0;
    }
    // 3) 스레드 슬롯 UNUSED로
    acquire(&p->lock);
    p->state = UNUSED;
    release(&p->lock);
    // 4) 메타데이터 초기화
    p->ustack      = 0;
    p->group       = 0;
    p->gprev       = p->gnext = 0;
    p->is_thread   = 0;
    p->thread_count = 0;
}
```

## thread\_create

새로운 쓰레드를 만들기 위해 `alloc_stack()`을 사용하여 스택을 할당하고, `clone()`을 호출하여 새로운 쓰레드를 생성해주는 유저 라이브러리 함수이다.

```
int
thread_create(void (*start_routine)(void*, void*), void *arg1, void *arg2)
{
    void *stack = alloc_stack();
    if (!stack)
        return -1;
    int pid = clone(start_routine, arg1, arg2, stack);
```

```

if (pid < 0) {
    free_stack(stack);
    return -1;
}
return pid;
}

```

## alloc\_stack

아래 그림을 통해 쉽게 이해해볼 수 있다.

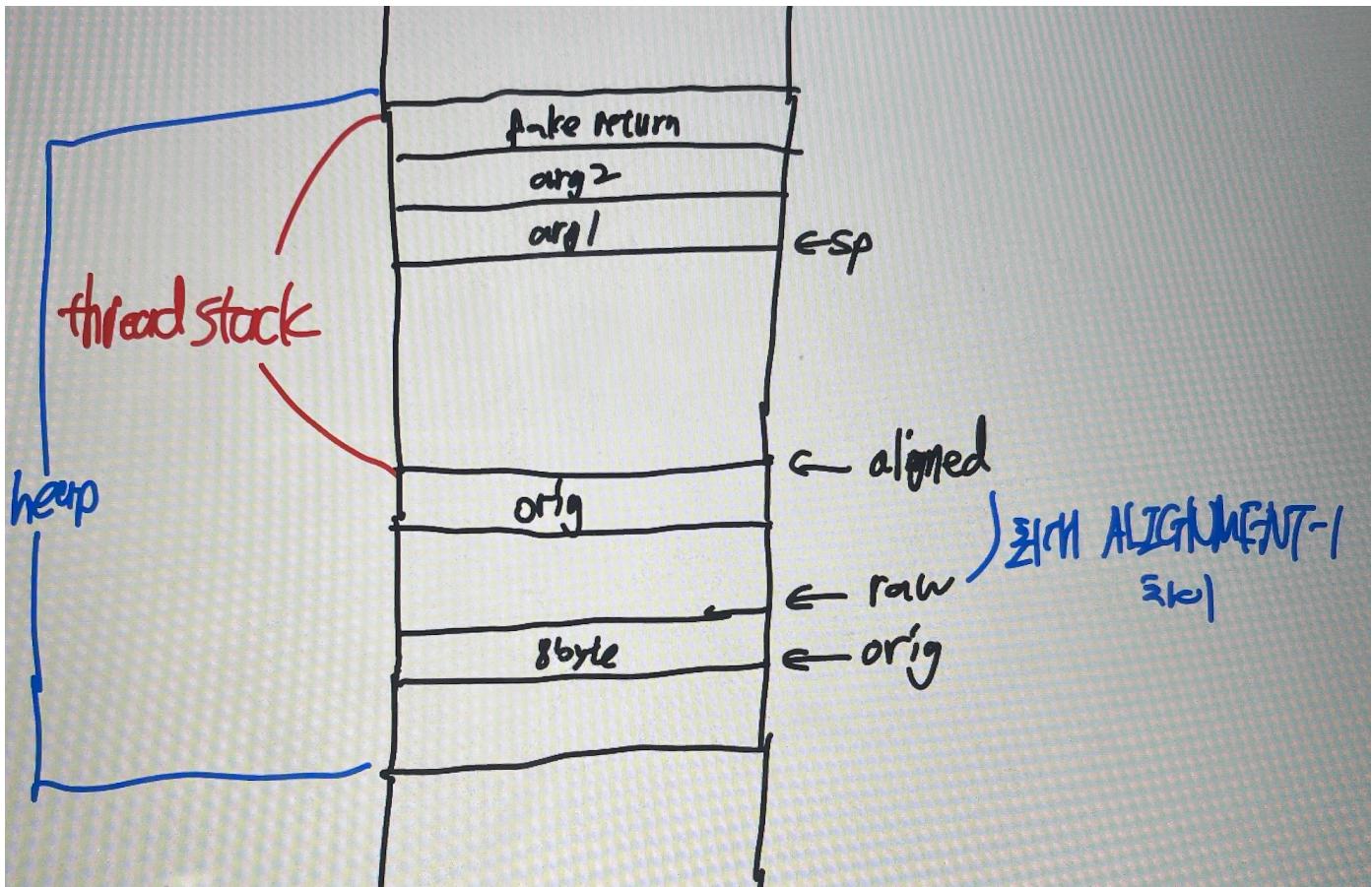
```

static void *
alloc_stack(void)
{
    // 스택 크기 + 정렬 오버헤드 + 원본 포인터 저장 공간
    uint64 tot = STACK_SIZE + ALIGNMENT - 1 + sizeof(void*);
    void *orig = malloc(tot);
    if (!orig)
        return 0;

    // orig 바로 다음 바이트를 기준으로 페이지 경계 위쪽으로
    uint64 raw = (uint64)orig + sizeof(void*);
    uint64 aligned = (raw + ALIGNMENT - 1) & ~(uint64)(ALIGNMENT - 1);

    // aligned 바로 앞에 orig 저장
    ((void**)aligned)[-1] = orig;
    return (void*)aligned;
}

```



## freestack

할당된 스택 영역을 free해주는 함수이다.

```

static void
free_stack(void *stack) {
    if (stack) {
        /* aligned 바로 앞에 저장된 orig를 꺼내서 free */
        void *orig = ((void**)stack)[-1];
        free(orig);
    }
}

```

## clone

아래는 `clone` 함수이다. `void(*fcn)(void *, void *)`인자는 생성된 쓰레드가 시작할 위치이다. `arg1`과 `arg2`는 쓰레드가 `fcn`에서 시작할 때 가질 인자 값들이다. `stack`은 생성될 쓰레드의 스택 베이스 주소가 인자로 넘어오도록 하였다. 쓰레드가 `clone`을 호출하는 경우도 있으므로 처음에 `proc`값을 `myproc()->group`(프로세스는 자기 자신을 가리키고, 쓰레드는 자신을 만든 프로세스를 가리킴)으로 하여 프로세스를 가리키도록 하자.

- `np`가 `allocthread(proc)`을 통해 생성된 쓰레드를 가리키도록 한다.
- 프로세스와 쓰레드는 힙 영역을 공유하므로 프로세스의 `sz`(힙 브레이크)를 쓰레드의 `sz`값에 매핑한다. 정의된 `group`은 프로세스의 쓰레드를 관리하는 그룹으로 `doubly linked list`방식을 사용했다. 새로운 쓰레드(노드)를 `group`에 추가해주었다.

- `kalloc`으로 만들어진 **trapframe** 영역은 **junk 값(0x05)**들로 채워져 있다. `memset(np->trapframe, 0, sizeof(*np->trapframe));`을 통해 0으로 세팅해준다.
- 쓰레드의 **epc(진입점)**을 **fcn**으로 놓고 전달되는 인자 **arg1, arg2**를 각각 **a0, a1**에 넣어준다.
- **ustack**이 쓰레드 스택 베이스를 가리키도록 만들고 **sp**를 쓰레드 스택 탑을 가리키도록 한다.
- 이제 CPU가 마치 `call fcn(arg1,arg2)`을 수행한 것과 같이 가짜 스택 프레임을 만들어주어야 한다. 쓰레드의 스택 탑에 차례대로 **fake return address, arg1, arg2**를 넣어주고 **sp**를 갱신한뒤 **trapframe->sp**에 넣어준다. 이때 `copyout`이 쓰이는데 쓰이는 이유를 아래 이어서 설명을 하려고 한다.
- 프로세스의 열린 파일 디스크립터와 **CWD**를 참조 카운트 증가하면서 복제
- 쓰레드의 **name, parent, state**를 갱신해준다.
- **쓰레드ID**를 반환한다.

```

int
clone(void(*fcn)(void *, void *), void *arg1, void *arg2, void *stack)
{
    int i;
    struct proc *np; //쓰레드
    int tid; //쓰레드 id
    struct proc *proc = myproc()->group; // 프로세스
    uint64 stack_base = (uint64)stack; // 스택의 시작되는 주소

    // 새로운 프로세스 영역 할당
    if ((np = allocthread(proc)) == 0){
        return -1;
    }

    np->sz = proc->sz;

    // doubly linked list로 group 관리
    np->group = proc;
    np->gnext = proc->gnext;
    if(proc->gnext) {
        proc->gnext->gprev = np;
    }
    proc->gnext = np;
    np->gprev = proc;

    // 트랩프레임 영역 초기화
    memset(np->trapframe, 0, sizeof(*np->trapframe));

    // 진입점 설정
    np->trapframe->epc = (uint64)fcn;
    // a0/a1: fcn(arg1,arg2)
    np->trapframe->a0 = (uint64)arg1;
    np->trapframe->a1 = (uint64)arg2;

    // 쓰레드 스택 주소 보관 (join에서 사용) 새로 할당된 페이지의 top 부분임
    np->ustack = stack_base;
}

```

```

// 쓰레드 스택 초기화
uint64 sp = stack_base + PGSIZE;

// 스택에 값 담기
// *(uint64*)sRet = 0xFFFFFFFF;
uint64 sRet = sp - 3*sizeof(uint64); // va
uint64 tmp_ret = 0xFFFFFFFF;
if (copyout(np->pagetable, sRet, (char *)&tmp_ret, sizeof(tmp_ret)) < 0)
    panic("clone: copyout failed for ret");

// *(uint64*)sArg1 = (uint64)arg1;
uint64 sArg1 = sp - 2*sizeof(uint64); // va
uint64 tmp_arg1 = (uint64)arg1;
if (copyout(np->pagetable, sArg1, (char *)&tmp_arg1, sizeof(tmp_arg1)) < 0)
    panic("clone: copyout failed for arg1");

// *(uint64*)sArg2 = (uint64)arg2;
uint64 sArg2 = sp - sizeof(uint64); // va
uint64 tmp_arg2 = (uint64)arg2;
if (copyout(np->pagetable, sArg2, (char *)&tmp_arg2, sizeof(tmp_arg2)) < 0)
    panic("clone: copyout failed for arg2");

np->trapframe->sp = sp - 3*sizeof(uint64);

// 열린 파일과 cwd(current working directory) 복제
for (i = 0; i < NOFILE; i++) // 파일 디스크립터 테이블 순회
    if (proc->ofile[i]) // 파일이 열려 있다면
        np->ofile[i] = filedup(proc->ofile[i]); // 복사
np->cwd = idup(proc->cwd); // 현재 작업 중인 디렉토리 복사

// 프로세스 이름 복사
safestrcpy(np->name, proc->name, sizeof(proc->name));
tid = np->pid;

release(&np->lock);

// 부모 프로세스 설정
acquire(&wait_lock);
np->parent = proc;
release(&wait_lock);

// RUNNABLE 상태로 만들
acquire(&np->lock);
np->state = RUNNABLE;
release(&np->lock);

return tid;
}

```

## copyout

Kernel address space에 있는(**src**)에서 User address space(**dstva**)로 데이터를 안전하게 복사해 주는 역할을 한다. 절차는 다음과 같다.

- **destva** 값이 참조하는 물리 프레임의 주소를 구한다. -> **pa0**
- **memmove** 를 통해 **src**의 값을 **pa0**에 쓴다.

`clone`이 실행되는 도중 `copyout`을 해야되는 이유는 쓰레드들의 스택에 값들을 유저 페이지 테이블을 참조한 올바른 물리 페이지에 기록해야하기 때문이다.

```
int
copyout(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
{
    uint64 n, va0, pa0;
    pte_t *pte;

    while(len > 0){
        va0 = PGROUNDDOWN(dstva);
        if(va0 >= MAXVA)
            return -1;
        pte = walk(pagetable, va0, 0);
        if(pte == 0 || (*pte & PTE_V) == 0 || (*pte & PTE_U) == 0 ||
           (*pte & PTE_W) == 0)
            return -1;
        pa0 = PTE2PA(*pte);
        n = PGSIZE - (dstva - va0);
        if(n > len)
            n = len;
        memmove((void *)(pa0 + (dstva - va0)), src, n);

        len -= n;
        src += n;
        dstva = va0 + PGSIZE;
    }
    return 0;
}
```

## thread\_join

쓰레드들을 수거하기 위해 `join`을 호출하고 `free_stack`을 통해 자원을 반환하는 유저 라이브러리 함수이다.

```

int
thread_join(void)
{
    void *stack;
    int pid = join(&stack);
    if (pid < 0)
        return -1;

    free_stack(stack);
    return pid;
}

```

## join

**join** 시스템콜의 동작 로직은 다음과 같다.

- **join** 함수는 자식 스레드가 종료될 때까지 계속 실행됩니다. 루프 안에서 부모 프로세스가 자식 스레드를 기다리고, 자식 스레드가 종료되면 자원을 정리하고 종료된 스레드의 PID를 반환한다.
- 모든 프로세스를 순차적으로 탐색하며, **parent**가 수거를 하고 있는 프로세스인지 확인하고 쓰레드가 **zombie**인 경우 수거 한다.

```

int
join(void **stack)
{
    struct proc *p;
    struct proc *cur = myproc();
    int haveThread;

    acquire(&wait_lock);
    for (;;) {
        haveThread = 0;

        // Scan all procs for threads of this parent in same pagetable
        for (p = proc; p < &proc[NPROC]; p++) {
            acquire(&p->lock);

            if (p->parent != cur) {
                release(&p->lock);
                continue;
            }

            haveThread = 1;
            // Found a zombie thread

            if (p->state == ZOMBIE) {
                int pid = p->pid;
                void *tstack = (void*)p->ustack;

                // Decrement thread count
                cur->thread_count--;
            }
        }
    }
}

```

```

    // Use freethread to cleanup thread-specific resources
    release(&p->lock);
    *stack = tstack;
    release(&wait_lock);
    if(p->is_thread){
        freethread(p);
    }else{ //exec을 호출한 경우임
        freeproc(p);
    }

    if(cur->thread_count == 0 && cur->temp_killed){
        cur->temp_killed = 0;
        cur->killed = 1;
    }
    return pid;
}
release(&p->lock);
}

// 2) 더 이상 남은 스레드가 없으면 -1 리턴
if (!haveThread || killed(p)) {
    release(&wait_lock);
    return -1;
}

sleep(cur, &wait_lock);
}
}

```

## exit

수정된 **exit** 코드는 다음과 같이 동작한다.

### 쓰레드인 경우

- 그룹 리스트에서 해당 쓰레드를 분리한 후, **p->grev**와 **p->gnext**를 초기화해준다.
- 부모를 wakeup한다.

### 프로세스인 경우

- 열린 파일을 모두 닫고 cwd를 받납한다. 그룹을 순회하며 자식들의 killed를 1로 세팅한다. 자식들은 **context switch** 된 이후 **usertrap**에서 죽게 된다.
- 만약 지금 종료되는 프로세스가 **fork**로 만든 자식이 존재한다면 **reparent**를 통해 **init**에 입양시킨다.
- 부모를 wakeup한다.

```

void
exit(int status)
{
    struct proc *p = myproc();
    struct proc *parent = p->parent;

```

```

int is_thread = (p->group != p);

if (is_thread) {
    // — 쓰레드 종료: 그룹 리스트에서 분리하고 join_parent()만 깨움 ————
    if (p->gprev) p->gprev->gnext = p->gnext;
    if (p->gnext) p->gnext->gprev = p->gprev;
    p->gprev = p->gnext = 0;

    // join() 대기 중인 부모(루트) 깨우기
    acquire(&wait_lock);
    wakeup(parent);
    release(&wait_lock);
} else {
    // — 프로세스 종료: 공유 자원 정리 ——————
    if (p == initproc)
        panic("init exiting");

    // 열린 파일 모두 닫기
    for (int fd = 0; fd < NOFILE; fd++) {
        if (p->ofile[fd]) {
            fileclose(p->ofile[fd]);
            p->ofile[fd] = 0;
        }
    }
    // cwd 반납
    begin_op();
    iput(p->cwd);
    end_op();
    p->cwd = 0;

    // — 자식 스레드들 종료 신호 ——————
    // 같은 그룹에 속한 모든 쓰레드에 killed 세팅
    for (struct proc *q = p->gnext; q; q = q->gnext) {
        acquire(&q->lock);
        q->killed = 1;
        if (q->state == SLEEPING)
            q->state = RUNNABLE;
        release(&q->lock);
    }

    // 자식 프로세스 및 남은 쓰레드를 init으로 재배치 후 부모 깨우기
    acquire(&wait_lock);
    reparent(p);
    wakeup(parent);
    release(&wait_lock);
}

// — 공통: ZOMBIE 설정 및 스케줄링 ——————
acquire(&p->lock);
p->xstate = status;
p->state = ZOMBIE;

```

```

// 컨텍스트 스위치: 더 이상 돌아오지 않음
sched();
panic("exit should not return");
}

```

## kill

**kill**의 로직은 다음과 같다.

- 인자로 넘겨받은 **pid**에 대응하는 쓰레드/프로세스의 **root** 값에 **temp\_killed** 세팅을 한다. 만약 쓰레드였다면 부모 프로세스가 **temp\_killed** 되고 프로세스였다면 자신이 **temp\_killed** 된다. **temp\_killed** 란 프로세스가 자식을 아직 다 수거하지 못한 상태에서 죽는 것을 방지하기 위해 도입한 필드이다.
- 일단 **root**에는 **temp\_killed** 플래그를 세우지만, 자식들에는 **killed** 플래그를 세우지 않는다. 그 이유는 지금 현재 **kill** 을 호출한 쓰레드가 유저 모드로 한 번 복귀 했다가 다음 **usertrap**에서 죽어야 하지만 **usertrap**의 로직을 보면 **syscall** 호출 이후 **killed** 플래그가 세워진 쓰레드를 죽이기 때문에 해당 쓰레드가 바로 죽어버린다.
- 그렇기 때문에 **root**로 context switch 된 시점에서 만약 **temp\_killed** 플래그가 세워져 있다면 그 때 자식 쓰레드들의 **killed** 플래그를 세워주고 본인은 자식을 전부 수거하고 죽어야하기 때문에 만약 자식이 없는 경우에 본인에게 **killed** 플래그를 세운 뒤 죽는다.

```

int
kill(int pid)
{
    struct proc *p, *target = 0;

    // 1) pid에 해당하는 proc 찾기
    for(p = proc; p < &proc[NPROC]; p++){
        acquire(&p->lock);
        if(p->pid == pid){
            target = p;
            release(&p->lock);
            break;
        }
        release(&p->lock);
    }
    if (!target)
        return -1;

    // 2) 같은 그룹의 루트(root) 결정
    struct proc *root = (target->group == target
                         ? target
                         : target->group);

    // 3) 루트만 temp_killed 플래그 세팅
    acquire(&root->lock);
    root->temp_killed = 1;
    if(root->state == SLEEPING)
        root->state = RUNNABLE;
    release(&root->lock);
}

```

```
    return 0;
}
```

**root**가 **temp\_flag**가 세워진 시점부터 지속적으로 자식이 수거되었는지 검사하는 로직은 `usertrap`에 구현하는 것이 가장 적합하다.

```
if (p == p->group && p->temp_killed) {
    // 1) 자식 쓰레드들에 실제 killed 플래그 세팅
    for (struct proc *q = p->gnext; q; q = q->gnext) {
        acquire(&q->lock);
        q->killed = 1;
        if (q->state == SLEEPING)
            q->state = RUNNABLE;
        release(&q->lock);
    }
    // 2) 자식 카운트가 0이면 자기 자신도 killed(=exit) 플래그
    if (p->thread_count == 0) {
        p->killed = 1;
        if (p->state == SLEEPING)
            p->state = RUNNABLE;
    }
    // 3) 더 이상 반복하지 않게 temp_killed 클리어
    p->temp_killed = 0;
}
```

## exec

**exec**을 설명하기에 앞서 내 전체적인 구현 로직은 다음과 같은 순서를 무조건 지켰다.

1. 프로세스가 쓰레드든 프로세스든 자식을 만든다
2. 자식이 무엇을 하던 종료된 이후에 부모가 모두 수거한다. 즉 부모가 자식보다 먼저 죽는(고아 프로세스) 상황을 절대 만들지 않는다.

이러한 기준에 따라 exec을 구현하였고, 바로 직전에 설명했던 **kill**도 그래서 자식을 모두 죽인 뒤 -> 부모가 join을 함에 따라 자식의 자원을 모두 회수하고->부모를 죽이는 순서로 구현한 것이다.

이것은 자원 해제 로직을 오로지 join 또는 wait에서만 구현함으로써 모듈화를 확실하게 할 수 있었다.

**exec**은 일단 쓰레드가 호출하는 경우 문제가 발생하는 상황이 많이 생긴다. 만약 원래 프로세스에 대한 **exec**처럼 동작한다고 가정하면 **oldpagetable**을 free해버리게 되고 이것은 부모 프로세스가 참조하는 페이지테이블이기 때문에 오류가 발생한다. 따라서 exec을 호출한 쓰레드에 대해서 새로운 페이징 테이블을 할당하기 이전에 `uvmunmap(p->pagetable, p->trapframe_va, 1, 0);`을 통해 트랩프레임 영역만 우선 해제하고, join에서는 **exec**을 호출한 쓰레드를 위한 기준만 따로 추가해주었다.

아래는 `join`의 일부이다. **zombie**상태를 수거하는 로직에 들어오는건 부모 프로세스의 쓰레드였던 자식 쓰레드들만 해당되지만 **exec**을 호출한 쓰레드는 **is\_thread** 값이 0이기 때문에 `freeproc`을 통해 자원을 해제해주었다.

```

if(p->is_thread){
    freethread(p);
}else{ //exec을 호출한 경우임
    freeproc(p);
}

```

`exec` 코드는 트랩프레임 언맵하는 부분만 맨 앞에, 페이지테이블을 할당하기 이전에 작성한 것을 제외하고는 마지막 실질적 값들을 할당하는 케이스에서 아예 프로세스와 쓰레드를 분기했다. 쓰레드가 `exec`을 호출하는 케이스에서 자신과 부모를 제외한 쓰레드들을 `killed` 표시하여 해당 쓰레드들이 다음 스케줄링 과정에서 `usertrap`에 진입했을 때 죽을 수 있도록 만들었고, `exec`을 호출한 본인은 호출이 끝나면 죽게 되며 나중에 자식이 다 죽었을 때는 부모가 회수하는 과정에서 자원들이 해제되게끔 만들었다.

```

int
exec(char *path, char **argv)
{
    char *s, *last;
    int i, off;
    uint64 argc, sz = 0, sp, ustack[MAXARG], stackbase;
    struct elfhdr elf;
    struct inode *ip;
    struct proghdr ph;
    pagetable_t pagetable = 0, oldpagetable;
    struct proc *p = myproc();

begin_op();

if((ip = namei(path)) == 0){
    end_op();
    return -1;
}
ilock(ip);

// Check ELF header
if(readi(ip, 0, (uint64)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;

if(elf.magic != ELF_MAGIC)
    goto bad;

if(p->is_thread){ //쓰레드가 호출한 경우 미리 언맵해야됨
    uvmunmap(p->pagetable, p->trapframe_va, 1, 0);
}

if((pagetable = proc_pagetable(p)) == 0)
    goto bad;

// Load program into memory.
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, 0, (uint64)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
        continue;
}

```

```

if(ph.memsz < ph.filesz)
    goto bad;
if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
if(ph.vaddr % PGSIZE != 0)
    goto bad;
uint64 sz1;
if((sz1 = uvmalloc(pagetable, sz, ph.vaddr + ph.memsz, flags2perm(ph.flags))) == 0)
    goto bad;
sz = sz1;
if(loadseg(pagetable, ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}

iunlockput(ip);
end_op();
ip = 0;

p = myproc();
uint64 oldsz = p->sz;

// Allocate some pages at the next page boundary.
// Make the first inaccessible as a stack guard.
// Use the rest as the user stack.
sz = PGROUNDDUP(sz);
uint64 sz1;
if((sz1 = uvmalloc(pagetable, sz, sz + (USERSTACK+1)*PGSIZE, PTE_W)) == 0)
    goto bad;
sz = sz1;
uvmclear(pagetable, sz-(USERSTACK+1)*PGSIZE);
sp = sz;
stackbase = sp - USERSTACK*PGSIZE;

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
        goto bad;
    sp -= strlen(argv[argc]) + 1;
    sp -= sp % 16; // riscv sp must be 16-byte aligned
    if(sp < stackbase)
        goto bad;
    if(copyout(pagetable, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
        goto bad;
    ustack[argc] = sp;
}
ustack[argc] = 0;

// push the array of argv[] pointers.
sp -= (argc+1) * sizeof(uint64);
sp -= sp % 16;
if(sp < stackbase)
    goto bad;
if(copyout(pagetable, sp, (char *)ustack, (argc+1)*sizeof(uint64)) < 0)

```

```

goto bad;

// arguments to user main(argc, argv)
// argc is returned via the system call return
// value, which goes in a0.
p->trapframe->a1 = sp;

// Save program name for debugging.
for(last=s=path; *s; s++)
    if(*s == '/')
        last = s+1;
safestrcpy(p->name, last, sizeof(p->name));

if(!p->is_thread){ // 프로세스의 경우
    // Commit to the user image.
    oldpagetable = p->pagetable;
    p->pagetable = pagetable;
    p->sz = sz;
    p->trapframe->epc = elf.entry; // initial program counter = main
    p->trapframe->sp = sp; // initial stack pointer
    proc_freepagetable(oldpagetable, olds, p);
}else{ // 쓰레드의 경우
    struct proc *root = p->group;
    // ===== THREAD & ROOT CLEANUP =====
    // exec을 호출한 쓰레드가 남아 있어야 함.
    for (struct proc *q = root; q; ) {
        struct proc *next = q->gnext;
        if (q != p && q != root) { // 본인, 루트 외에 다 제거
            q->killed = 1;
        }
        q = next;
    }
    root->gprev = root->gnext = 0;

    // 그룹 링크 완전 초기화: 호출 쓰레드가 유일한 멤버
    p->gprev = p->gnext = 0;
    p->is_thread = 0;
    p->thread_count = 0;
    // ===== /THREAD & ROOT CLEANUP =====

    // Commit to the user image.
    p->pagetable = pagetable;
    p->sz = sz;
    p->trapframe->epc = elf.entry; // initial program counter = main
    p->trapframe->sp = sp; // initial stack pointer
}

return argc; // this ends up in a0, the first argument to main(argc, argv)

bad:
if(pagetable)

```

```

    proc_freepagetable(pagetable, sz, p);
    if(ip){
        iunlockput(ip);
        end_op();
    }
    return -1;
}

```

## 메모리 할당 관련 설정

`sbrk` 시스템 콜을 호출하는 과정에서 쓰레드들이 필드로 갖고 있는 `sz` 값들이 모두 업데이트되어야 한다. 따라서 `group`을 순회하며 `sz`값을 업데이트 해주었다.

```

// Grow or shrink user memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
    uint64 sz;
    struct proc *p = myproc();

    //추가추가
    struct proc *root = p->group;
    //추가추가

    sz = p->sz;
    if(n > 0){
        if((sz = uvmalloc(p->pagetable, sz, sz + n, PTE_W)) == 0) {
            return -1;
        }
    } else if(n < 0){
        sz = uvmdealloc(p->pagetable, sz, sz + n);
    }
    // p->sz = sz;

    // update the canonical sz
    root->sz = sz;
    // now propagate that new size into every member's p->sz
    for(struct proc *q = root; q; q = q->gnext){
        q->sz = sz;
        // since the list is finite, you'll stop once gnnext==NULL
    }

    return 0;
}

```

## 메모리 해제 관련 설정

### proc\_freepagetable

원래와는 다르게 TRAPFRAME을 해제하는 과정에서 쓰레드들의 TRAPFRAME도 같이 해제하도록 구현한다.

```

// Free a process's page table, and free the
// physical memory it refers to.
void
proc_freepagetable(pagetable_t pagetable, uint64 sz, struct proc *p)
{
    uvmunmap(pagetable, TRAMPOLINE, 1, 0);
    uvmunmap(pagetable, TRAPFRAME - (p->thread_count * PGSIZE), 1 + p->thread_count, 0);
    uvmfree(pagetable, sz);
}

```

## freeproc

```

static void
freeproc(struct proc *p)
{
    // 1) 루트 프로세스이면 소속된 쓰레드 모두 해제
    if (p->group == p) {
        for (struct proc *q = p->gnext; q; q = q->gnext) {
            // 각 쓰레드 freethread 호출
            freethread(q);
        }
    }
    // 2) 루트의 페이지테이블 전체 해제
    if (p->pagetable) {
        proc_freepagetable(p->pagetable, p->sz, p);
        p->pagetable = 0;
    }
    // 4) 루트까지 해제 후 메타데이터 리셋
    p->sz      = 0;
    p->pid     = 0;
    p->parent   = 0;
    p->name[0] = '\0';
    p->chan     = 0;
    p->killed   = 0;
    p->xstate   = 0;
    p->state    = UNUSED;
    p->group    = 0;
    p->gprev   = p->gnext = 0;
    p->is_thread = 0;
    p->thread_count = 0;
    p->ustack   = 0;
}

```

## freethread

```

void
freethread(struct proc *p)
{
    // 1) 사용자 트랩프레임 매핑 해제

```

```

if (p->trapframe_va) {
    uvmunmap(p->pagetable, p->trapframe_va, 1, 0);
    p->trapframe_va = 0;
}
// 2) 커널 측 trapframe 해제
if (p->trapframe) {
    kfree((void*)p->trapframe);
    p->trapframe = 0;
}
// 3) 스레드 슬롯 UNUSED로
acquire(&p->lock);
p->state = UNUSED;
release(&p->lock);
// 4) 메타데이터 초기화
p->ustack = 0;
p->group = 0;
p->gprev = p->gnext = 0;
p->is_thread = 0;
p->thread_count = 0;
}

```

## 필드 관련 부가 설명

추가한 필드들은 다음과 같다.

### **thread\_count, is\_thread, ustack, group, gprev, gnex**

thread\_count는 부모 프로세스가 가지고 있는 쓰레드의 수(메인쓰레드 제외) 값이다.

is\_thread는 쓰레드인지 여부를 나타낸다.

ustack은 유저 스택 주소를 가리킨다.

group은 doubly linked list에서 루트(부모 프로세스)를 가리킨다.

gprev, gnex는 doubly linked list에서 이전과 이후를 나타내는 값이다.

```

// Per-process state
struct proc {
    struct spinlock lock;

    // p->lock must be held when using these:
    enum procstate state;          // Process state
    void *chan;                    // If non-zero, sleeping on chan
    int temp_killed;               // temp killed
    int killed;                    // If non-zero, have been killed
    int xstate;                    // Exit status to be returned to parent's wait
    int pid;                       // Process ID
    int thread_count;              // Number of threads
    int is_thread;                 // 쓰레드인지 여부. 메인쓰레드는 0임
}

```

```

// wait_lock must be held when using this:
struct proc *parent;           // Parent process

// these are private to the process, so p->lock need not be held.
uint64 ustack;                // user stack
uint64 kstack;                // Virtual address of kernel stack
uint64 sz;                     // Size of process memory (bytes)
pagetable_t pagetable;         // User page table
uint64 trapframe_va;          // virtual address of the trapframe
struct trapframe *trapframe;   // data page for trampoline.S
struct context context;        // swtch() here to run process
struct file *ofile[NFILE];    // Open files
struct inode *cwd;             // Current directory
char name[16];                // Process name (debugging)

// sbrk를 위한 필드
struct proc *group;           // 루트를 가리킴. 쓰래드 뮐음에서 메인 쓰래드를 가리킴
struct proc *gprev, *gnext;    // doubly-linked list
};


```

## Result

첫 번째 테스트 결과이다.

```

[TEST#1]
Thread 0 start
Thread 1 start
Thread 1 end
Thread 2 start
Thread 2 end
Thread 3 start
Thread 3 end
Thread 4 start
Thread 4 end
Thread 0 end
TEST#1 Passed

```

두 번째 테스트 결과이다.

```
[TEST#2]
Thread 0 start, iter=0
Thread 0 end
Thread 1 start, iter=1000
Thread 1 end
Thread 2 start, iter=2000
Thread 2 end
Thread 3 start, iter=3000
Thread 3 end
Thread 4 start, iter=4000
Thread 4 end
TEST#2 Passed
```

세 번째 테스트 결과이다.

```
[TEST#3]
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Child of thread 0 start
Child of thread 1 start
Child of thread 2 start
Child of thread 3 start
Child of thread 4 start
Child of thread 0 end
Child of thread 1 end
Child of thread 2 end
Child of thread 3 end
Child of thread 4 end
Thread 0 end
Thread 1 end
Thread 2 end
Thread 3 end
Thread 4 end
TEST#3 Passed
```

네 번째 테스트 결과이다.

#### [TEST#4]

```
Thread 0 sbrk: old break = 0x00000000000015000
Thread 0 sbrk: increased break by 14000
new break = 0x00000000000029010
Thread 1 size = 0x00000000000029010
Thread 2 size = 0x00000000000029010
Thread 3 size = 0x00000000000029010
Thread 4 size = 0x00000000000029010
Thread 0 sbrk: free memory
Thread 0 end
Thread 1 end
Thread 2 end
Thread 3 end
Thread 4 end
TEST#4 Passed
```

다섯 번째 테스트 결과이다.

#### [TEST#5]

```
Thread 0 start, pid 29
Thread 1 start, pid 29
Thread 2 start, pid 29
Thread 3 start, pid 29
Thread 4 start, pid 29
Thread 0 end
TEST#5 Passed
```

여섯 번째 테스트 결과이다.

```
[TEST#6]
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Executing...
Thread exec test 0
TEST#6 Passed
```

```
All tests passed. Great job!!
TEST#6 Passed
```

```
All tests passed. Great job!!
```

여섯 번째 테스트는 출력이 두번 나온다. 이 이유는 내 구현은 `kill`을 세팅한 이후 유저모드 복귀 후 다음 스케줄링 루틴에서 죽기 때문이다. 사실 이건 5번째 테스트 코드 `kill`에서도 마찬가지이다.

바로 죽이지 않았던 이유는 역할에 구현을 확실하게 하기 위해서였다. 코드에 대한 모듈화가 제대로 이루어지려면 부모를 죽이더라도 "자식들이 모두 죽고", "부모가 자식을 수거하고", "부모가 죽고"와 같은 순서를 지켰다. 자식에 대한 자원을 해제하는 것이 `wait` 와 `join`에서만 구현이 되고 그 어떤 위치에서도 구현하지 않아 일부러 코드의 모듈화를 확실하게 하였다. 이러한 구현을 바탕으로 test code를 일부 수정할 필요가 있었는데 `fork`가 된 자식은 본인의 실행을 마칠 때 `exit(0)`을 반드시 호출하도록 하는 규약을 지켜야했다.

다음은 수정된 테스트에 따른 출력 결과이다.

```
[TEST#6]
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Executing...
Thread exec test 0
TEST#6 Passed
```

```
All tests passed. Great job!!
$ █
```

## Troubleshooting

---

아무래도 트랩프레임에 대한 문제가 제일 컸다. 처음에 아무런 생각 없이 유저스페이스의 trapframe\_va를 모든 쓰레드가 동일하게 TRAPFRAME영역을 가리키도록 했었고 위에 설명했던 원인으로 인한 문제가 발생하였다 ([design](#)의 트랩프레임 관리 방법 참고).

이번 과제에서는 메모리에 대한 관리가 제일 중요한 부분이었던 것 같다. 프로세스와 쓰레드가 동일한 user address space를 참 고하고 있다보니 clone, join 또는 fork, wait 과정에서 쓰레드 및 프로세스에 대한 자원 + 페이지테이블에 대한 자원이 올바르게 회수되지 않는 경우 메모리 관련 panic이 발생하였다. 이 부분에서 모듈화의 중요성을 다시 한 번 깨달았던 것 같다. `join`과 `wait`에서만 자원을 회수하는 로직을 구현하여 문제가 발생하여도 쉽게 찾아 해결할 수 있었다.