PA3 결과보고서

전자공학도의 윤리 강령 (IEEE Code of Ethics)

(출처: http://www.ieee.org)

나는 전자공학도로서, 전자공학이 전 세계 인류의 삶에 끼치는 심대한 영향을 인식하여 우리의 직업, 동료와 사회에 대한 나의 의무를 짐에 있어 최고의 윤리적, 전문적 행위를 수행할 것을 다짐하면서, 다음에 동의한다.

1. **공중의 안전, 건강 복리에 대한 책임**: 공중의 안전, 건강, 복리에 부합하는 결정을 할 책임을 질 것이며, 공중 또는 환경을 위협할 수 있는 요인을 신속히 공개한다.
2. **지위 남용 배제**: 실존하거나 예기되는 이해 상충을 가능한 한 피하며, 실제로 이해가 상충할 때에는 이를 이해 관련 당사자에게 알린다. (이해 상충: conflicts of interest, 공적인 지위를 사적 이익에 남용할 가능성)
3. **정직성**: 청구 또는 견적을 함에 있어 입수 가능한 자료에 근거하여 정직하고 현실적으로 한다.
4. **뇌물 수수 금지**: 어떠한 형태의 뇌물도 거절한다.
5. **기술의 영향력 이해**: 기술과 기술의 적절한 응용 및 잠재적 영향에 대한 이해를 높인다.
6. **자기계발 및 책무성**: 기술적 능력을 유지, 증진하며, 훈련 또는 경험을 통하여 자격이 있는 경우이거나 관련 한계를 전부 밝힌 뒤에만 타인을 위한 기술 업무를 수행한다.
7. **엔지니어로서의 자세**: 기술상의 업무에 대한 솔직한 비평을 구하고, 수용하고, 제공하며, 오류를 인정하고 수정하며, 타인의 기여를 적절히 인정한다.
8. **차별 안하기**: 인종, 종교, 성별, 장애, 연령, 출신국 등의 요인에 관계없이 모든 사람을 공평하게 대한다.
9. **도덕성**: 허위 또는 악의적인 행위로 타인, 타인의 재산, 명예, 또는 취업에 해를 끼치지 않는다.
10. **동료애**: 동료와 협력자가 전문분야에서 발전하도록 도우며, 이 윤리 헌장을 준수하도록 지원한다.

위 IEEE 윤리헌장 정신에 입각하여 report를 작성하였음을 서약합니다.

학 부: 전자공학과

제출일: 2025/12/22

과목명: 전자공학과

교수명: 오영환 교수님

분 반: C061-1

학 번: 202126989, 202021025

성 명: 유현종, 안준영

1. 목표

자식을 만들어 새로운 프로세스를 생성하는 fork는 주소 공간을 복사해 사용한다. 이번 과제에서는 fork와 달리 주소 공간을 복사하지 않고 공유하는 clone 시스템 콜을 구현한다. 또한, 주소 공간을 공유하는 자식 스레드가 exit 하기를 기다리고 회수하는 join 시스템 콜을 구현한다. 이 시스템 콜들을 User가 쉽게 사용하도록 thread_create와 thread_join 유저 라이브러리를 제공한다.

2. 기능 구현 지시 사항

clone(fcn, arg1, arg2, stack)은 주소 공간을 공유하는 새로운 스레드를 만들고, 만들어지면 fcn(arg1, arg2)를 실행하도록 한다. join(void **stack)은 주소 공간을 공유하는 자식 스레드가 종료될 때까지 기다리고, 그 자식 스레드가 사용한 스택 주소를 *stack에 넣어 반환하게 구현한다. thread_create()는 스택을 할당 후 clone을 호출하도록 하며, thread_join은 join()이 반환한 스택을 free한다.

3. 세부 구현

3.1 시스템 콜 등록

- syscall.h

clone, join 시스템 콜에 22, 23의 고유 번호를 붙여준다.

```
#define SYS_clone 22
#define SYS_join 23
```

- syscall.c

유저가 SYS_clone, SYS_join의 번호를 호출했을 때, 커널의 sys_clone, sys_join 함수를 연결할 수 있도록 아래와 같이 설정한다.

```
extern int sys_clone(void);
extern int sys_join(void);
[SYS_clone]   sys_clone,
[SYS_join]    sys_join
```

- user.h

유저 프로그램이 cone, joun, thread_create, thread_join 함수를 호출할 수 있게 아래와 같이 선언을 추가한다.

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

- usys.S

SYS_clone, SYS_join 번호를 레지스터에 넣고 커널을 호출하는 어셈블리 코드로 자동 변환되도록 아래와 같이 추가한다.

```
SYSCALL(clone)
SYSCALL(join)
```

- defs.h

sysproc.c의 sys_clone, sys_join이 clone, join을 호출한다. 따라서 sysproc.c가 포함하는 defs.h에 아래와 같이 선언을 추가한다.

```
int             clone(void*, void*, void*, void*);
int             join(void**);
```

3.2 함수 구현

- sysproc.c

유저가 시스템 콜로 호출할 수 있게 sys_clone, sys_join 래퍼함수를 정의한다. sys_clone에서, 유저가 호출한 clone 시스템 콜의 인자 4개를 argint로 읽어온다. 이후 읽어온 데이터를 void*로 캐스팅해 clone에 그대로 전달한다.
sys_join에서는, argint로 스택 포인터의 주소 값을 읽는다. 이 주소를 통해서 자식 스레드가 사용한 스택 주소를 다시 부모에게 돌려준다..

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

  if(argint(0, (void*)&fcn) < 0 ||
     argint(1, (void*)&arg1) < 0 ||
     argint(2, (void*)&arg2) < 0 ||
     argint(3, (void*)&stack) < 0) {
    return -1;
  }
  return clone((void*)fcn, (void*)arg1, (void*)arg2, (void*)stack);
}
```

```
int
sys_join(void)
{
  int stack_ptr;

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

  return join((void**)stack_ptr);
}
```

- proc.c

(1) proc.c - clone
clone은 fork와 유사하지만, 주소 공간을 복사하지 않고 공유하는 것이 그 차이점이다. allocproc을 통해서 새로운 프로세스 구조체를 할당받고 커널 스택을 준비한다. np->pgdir = curproc->pgdir은 부모와 자식 스레드가 같은 주소 공간을 공유하게 하는 코드이다. fork()에서는 copyuvm을 통해 메모리를 그대로 복사하지만 clone()에서는 이를 통해서 부모의 페이지 테이블 주소를 가져오는 차이가 생긴다. 그리고 메모리 크기 정보도 동일하게 설정한다.
np->tf->eip = (uint)fcn을 통해서 새 스레드가 시작할 때 인자로 받은 함수인 fcn의 주소로부터 실행하도록한다. 그리고 np->tf->esp = (uint)stack을 통해서 생성된 스레드에 clone의 리턴 값으로 0을 준다.
인자로 받은 스택 주소를 이용해서 스레드가 함수를 실행할 수 있도록 한다. 스택은 낮은 주소로 자라기 때문에 sp를 감소시키면서 값을 채워나간다. arg2, arg1을 차례대로 스택에 넣고, 함수가 종료되었을 때 돌아갈 가짜 리턴 주소를 넣는다. np->tf->esp = sp를 통해서 스택의 위치를 레지스터에 기록한다.
그리고 파일 디스크립터를 filedup을 통해서 공유한다. 설정이 끝났기 때문에 프로세스 테이블의 락을 획득하고, 실행가능 상태(RUNNABLE)로 설정해서 스케줄러가 선택할 수 있게 한다.

```
int
clone(void *fcn, void *arg1, void *arg2, void *stack)
{
  int i, pid;
  struct proc *np;
  struct proc *curproc = myproc();

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

  // fork랑 달리 페이지테이블 그대로 공유
  np->pgdir = curproc->pgdir;
  np->sz = curproc->sz;
  np->parent = curproc;
  *np->tf = *curproc->tf;

  np->tf->eax = 0;
  np->tf->eip = (uint)fcn;
  np->tf->esp = (uint)stack;
```

```
  //사용자 스택에 인자 넣기
  uint sp = (uint)stack;

  sp -= 4; *(uint*)sp = (uint)arg2;
  sp -= 4; *(uint*)sp = (uint)arg1;
  sp -= 4; *(uint*)sp = 0xffffffff;

  np->tf->esp = sp;
  np->tf->ebp = sp;

  for(i = 0; i < NOFILE; i++)
    if(curproc->ofile[i])
      np->ofile[i] = filedup(curproc->ofile[i]);
  np->cwd = idup(curproc->cwd);

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

  pid = np->pid;

  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);

  return pid;
}
```

(2) proc.c - join
wait() 구조를 베이스로 하되, 주소 공간을 공유하는 자식 스레드만 대상으로 설정했다. ptable을 순환할 포인터 p, 기다릴 대상이 있는지 체크할 havekids, pid, 그리고 현재 스레드(프로세스)를 나타낼 curporc를 셋팅한다.
proc 테이블을 훑을 동안은 동기화가 필요해 락을 acquire한다. 이후 본격적으로 테이블을 훑는다. wait과 같이 무한루프를 돌며 종료한 자식이 나타날 때 까지 비지웨이팅한다. 매 루프마다 havekids를 초기화 한 후, proc 테이블을 돌며 현재 프로세스가 부모인지, clone을 통해서 생성된 페이지 테이블을 공유하고 있는 스레드인지를 조건으로 확인한다. 이를 통해 현재 프로세스와 메모리 공간을 공유하는 자식 스레드인지 찾을 수 있다.
havekid = 1로 해서 join할 대상이 적어도 하나 있다고 나타내고, if(p->state == ZOMBIE) 조건으로 종료된 스레

드를 발견하면 부모에게 그 스레드가 쓰던 스택 주소를 돌려주고 FREE한다. 이때, 주의할 점은 wait에서처럼 freevm하면 같은 주소 공간을 쓰는 스레드들이 전부 다 터진다. 이후 proc 슬롯을 빈 슬롯으로 초기화해서 재사용 가능하게 한다. 메타 데이터 역시 정리한다. 완료했으면 락을 풀고 pid를 반환한다.
자식 스레드가 하나도 없다면 기다릴 필요가 없기 때문에 -1을 반환하고 종료한다. 아직 종료된 자식 스레드가 없다면 부모는 sleep 상태로 들어가서 CPU를 양보한다.

```
int
join(void **stack)
{
  struct proc *p;
  int havekids, pid;
  struct proc *curproc = myproc();

  acquire(&ptable.lock);
  for(;;){
    // 공유하는 자식 thread 확인
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc || p->pgdir != curproc->pgdir)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // 종료된 thread(zombie) 발견 시 정리
        pid = p->pid;
        *stack = (void*)p->tf->ebp; //clone에서 설정했던 스택주소 다시 부모에 알려줌
        kfree(p->kstack);
        p->kstack = 0;
        //pgdir 공유중이므로 freeuvm 안 씀
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;
        release(&ptable.lock);
        return pid;
      }
    }

    // 자식 없거나 죽은 자식 없으면 대기
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
}
```

- ulib.c

thread_create는 유저 스택을 malloc으로 할당한 후 clone을 호출한다. 스택은 낮은 주소로 자라기 때문에 메모리의 위쪽 끝 지점을 넘겨주기 위해서 stack+PGSIZE를 clone에 넘겨준다.
thread_join()은, join으로 스택 주소를 받은 후 free하고 pid를 반환한다. free할 때는 원래 시작 주소로 돌아가기 위해서 PGSIZE만큼 빼고 해제한다.

```
int
thread_create(void (*fcn)(void *, void *), void *arg1, void *arg2)
{
  void *stack = malloc(PGSIZE);
  if (stack == 0) return -1;

  // 할당된 메모리 넘기면서 clone 호출
  int pid = clone(fcn, arg1, arg2, stack + PGSIZE);

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

int
thread_join(void)
{
  void *stack;
  // join 호추랄면서 종료된 thread의 스택 주소 받아옴
  int pid = join(&stack);

  if (pid > 0) {
    free(stack - PGSIZE);
  }
  return pid;
}
```

3.3 Makefile

```
UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\
	_test_thread\
```

Makefile에 _test_thread₩를 추가한다.

4. 테스트 과정 및 결과

test_thread에서는 스레드 f1, f2, f3를 만들고 3번 조인을 실행한다, 락이 있을 때랑 락이 없을 때 버전으로 출력을 테스트한다. main에서 arg1을 1로하면 각 f1,f2,f3에서 락이 실행되고 0이면 락을 거는 코드가 if(num)에 걸려서 제외된다.
락이 있는 버전의 출력에 대해서, 스핀 락 lock_acquire으로 락을 소유한 동안에는 다른 스레드들이 test_thread.c의 printf 구간에 못들어온다. 따라서 f1, f2, f3의 각 printf가 온전히 출력됨이 보장된다. 반면 락이 없을 때(arg1 = 0)는 printf가 실행되는 중에도 스케줄링 되므로 마치 번갈아가며 출력되는 듯한 출력이 나타난다.

thread_create가 실행되면 스택을 할당하고 clone으로 스레드를 RUNNABLE 상태로 만든다. 스케줄러가 main에서 만든 3개의 스레드를 번갈아가며 돌린다. 락을 걸면 직렬화되어 출력이 깔끔해지고 락을 안걸면 인터리빙되며 프린트가 진행되어 출력이 섞인다. 작업이 끝난 스레드는 비지웨이팅하는 join에 의해 수거되며 스택이 free된다.

아래는 9번의 test_thread 실행 결과이다. 락이 있는 버전은 깔끔히 출력되고, 락이 없는 버전은 출력이 뒤섞임을 볼 수 있다. 9번의 테스트에서 출력이 의도와 같은 것을 확인해 통과했기에, 구현한 clone, join과 pthread_create, pthread_join이 정상적으로 작동하며 이번 과제의 목표를 성공적으로 구현했다고 볼 수 있다.

<table>
<tr>
<td>

```
$ test_thread
below should be sequential print statements:
1. sleep for 100 ticks
3. sleep for 100 ticks
2. sleep for 100 ticks
below should be a jarbled mess:
123. sleep . sleep for 100 tifcks
or 100. ticks
 sleep for 100 ticks
```

</td>
<td>

```
$ test_thread
below should be sequential print statements:
1. sleep for 100 ticks
3. sleep for 100 ticks
2. sleep for 100 ticks
below should be a jarbled mess:
1.3. s2. sleep forleep for  100 tick100 ticks
s
 sleep for 100 ticks
```

</td>
<td>

```
$ test_thread
below should be sequential print statements:
1. sleep for 100 ticks
3. sleep for 100 ticks
2. sleep for 100 ticks
below should be a jarbled mess:
1.3. slee2. sleepp for 100 tic ks
s forleep for 100 ticks
 100 ticks
```

</td>
</tr>
<tr>
<td>

```
$ test_thread
below should be sequential print statements:
1. sleep for 100 ticks
2. sleep for 100 ticks
3. sleep for 100 ticks
below should be a jarbled mess:
1.2. sle3 seleep for 100p fo ticks
r .100 ticks
 sleep for 100 ticks
```

</td>
<td>

```
$ test_thread
below should be sequential print statements:
1. sleep for 100 ticks
3. sleep for 100 ticks
2. sleep for 100 ticks
below should be a jarbled mess:
12. sleep for. sleep for3 100 tick. sleep fs
or 100  ti100 tickcks
s
```

</td>
<td>

```
$ test_thread
below should be sequential print statements:
1. sleep for 100 ticks
2. sleep for 100 ticks
3. sleep for 100 ticks
below should be a jarbled mess:
13.. 2 sleep sleep for 100 ticks
for 100 tick. sleep fs
or 100 ticks
```

</td>
</tr>
<tr>
<td>

```
$ test_thread
below should be sequential print statements:
2. sleep for 100 ticks
1. sleep for 100 ticks
3. sleep for 100 ticks
below should be a jarbled mess:
123. sleep for. sleep  100 tic. sleep forfor 100 tic 100 ticks
ks
ks
```

</td>
<td>

```
$ test_thread
below should be sequential print statements:
1. sleep for 100 ticks
2. sleep for 100 ticks
3. sleep for 100 ticks
below should be a jarbled mess:
1. sl2. slee3eep for 100 ticks
. sleepp for 100 ticks
 for 100 ticks
```

</td>
<td>

```
$ test_thread
below should be sequential print statements:
1. sleep for 100 ticks
3. sleep for 100 ticks
2. sleep for 100 ticks
below should be a jarbled mess:
1.23. sleep for 1. slee sle00 ticks
epp for 100 ticks
 for 100 ticks
```

</td>
</tr>
</table>