ELE3021 Project2 Wiki

Process Management and LWP

2020059152 오주영

1. 개요

구현에 앞서 확고한 디자인을 세울 수 있었다면 좋았겠지만, 이번 프로젝트의 구현 과정은 디자인 수정 및 변동의 끊임없는 반복이었다. 미처 생각치 못했던 요구 사항이나 예외 상황을 처리하기 위해서 기존의 특정 디자인을 바꾸면, 그 디자인에 의존하고 있던 디자인들에서 문제점이 발생하고, 이를 또 수정하면 역시 새로운 문제들이 꼬리에 꼬리를 물고 드러났다. 이런 문제의 고리가 길어지다 보면 내가 지금 해결하려는 문제의 근원이 무엇인지 잊어 제자리 걸음을 반복하는 일도 다반사였다. 아무튼 이런 난잡한 구현 과정 때문에 나의 최종 디자인과 이를 채택한 이유를 단번에 서술하기는 불가능에 가까울 것이다. 대신, 순서가 좀 뒤섞여도 어떤 문제 상황을 해결하기 위해 어떤 디자인을 변경했는지, 이 디자인은 어떤 문제를 고려해 채택한 것인지 등등 인과적 흐름대로 설명하고자 한다. 또한 디자인의 변경은 구현 과정의 세부적인 디테일과 관련이 크므로 디자인 설명과 구현 설명을 따로 분리하지 않고 다음 "디자인 및 구현" 하나의 섹션에서 함께 설명하고자 한다.

2. 디자인 및 구현

exec2 함수 구현

구현 과정의 시작점은 exec2이었다. 프로세스에게 스택용 페이지와 가드용 페이지를 각각 하나씩 할당해주는 기존의 exec 함수와 달리, exec2는 인자 stacksize 만큼 스택용 페이지를 할당해야 한다. 기존 exec의 페이지 할당은 아래 코드에서 확인할 수 있다:

```
// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
```

우선 총 2개의 페이지를 allocuvm을 통해 할당하고, clearpteu는 아래 페이지에 PTE_U 비트를 지워 유저 프로세스가 접근할 수 없는 가드용 페이지로 만드는 것이다. 따라서 exec2에서도 마찬가지로 할당할 스택용 페이지의 개수보다 하나 더 할당해 이를 가드용 페이지로 만들면 되므로, 위의 2 * PGSIZE를 (stacksize + 1) * PGSIZE가 되도록 구현해야 했다. 이 부분을 제외하고선 exec2와 exec은 같으므로, exec2에서 stacksize를 적절히 처리한 뒤 exec을 호출하는 방식으로 구현하고 싶었다. 따라서 proc 구조체에 stacksize 변수를 저장하고, allocproc에서 처음 생성된 프로세스는 기본적으로 이 값을 1로 가지게 구현하였다.

```
int exec2(char *path, char **argv, int stacksize)
{
  if (stacksize < 1 || stacksize > 100)
    return -1;
  myproc()->stacksize = stacksize;
  return exec(path, argv);
}
```

exec2는 인자 stacksize를 해당 프로세스의 stacksize 변수에 저장만 해주고 exec을 호출한다.

```
int stacksize = curproc->stacksize;
...
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + (stacksize + 1) * PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - (stacksize + 1) * PGSIZE));
```

그리고 기존의 exec은 위 설명했듯이 2 * PGSIZE가 아닌 (stacksize + 1) * PGSIZE만큼을 할당해주게 수정했다.

setmemorylimit 함수 구현

다음은 setmemorylimit 함수의 구현이다. 프로세스가 "추가적으로" 할당받을 수 있는 메모리의 최대치를 제한하기 위해서, 우선 프로세스가 메모리를 추가적으로 할당받는 과정을 살펴봐야했다. 이는 proc.c의 growproc 함수에서 진행된다. 기존 growproc(int n)의 경우 조건 없이 프로세스의 sz 변수를 늘려주는데(지금은 인자 n이 양수인 경우만 고려하자), 그 전에 이 프로세스의 sz와 인자 n의 합이 이 프로세스의 메모리 제한을 넘지 않는지 확인하도록 확인해줘야 한다. 따라서 각 프로세스마다 메모리 제한치 정보를 저장하기 위해 proc 구조체에 memlim 변수를 저장했다. 프로세스가 처음 생성될 때에 이 값은 0이며, 이는 제한이 없다는 뜻이다. setmemorylimit 함수는 ptable을 순회하며 인자 pid와 일치하는 프로세스를 찾고, 이 프로세스의 memlim 변수에 인자 limit을 저장해준다.

```
int
setmemorylimit(int pid, int limit)
{
  struct proc *p;

  if (limit < 0) return -1;

  acquire(&pgdirlock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){

      if (limit && p->sz > limit) break;

      p->main->memlim = limit;
      release(&pgdirlock);
      return 0;
    }
  }
  release(&pgdirlock);
  return -1;
}
```

실패해서 -1을 반환하는 경우는 세가지다. 첫째는 limit이 음수인 경우다. 둘째는 해당 프로세스가 기존 할당 받은 메모리보다 limit이 작은 경우인데, 이때 limit이 0인 경우는 제한을 풀겠다는 것이므로 예외다. 마지막으로 단순히 인자 pid를 가진 프로세스를 찾지 못한 경우이다.

pgdirlock, p->main 등등 위 코드의 세부적인 사항들은 다음 부분에서 설명하도록 하겠다.

proc 구조체의 main 변수

setmemorylimit의 코드에서 limit을 저장할 때, p->memlim = limit 이 아니라 p->main->memlim = limit 인 것을 볼 수 있다. 이후 훨씬 더 자세히 설명하겠지만 proc 구조체의 main 변수는 메인 쓰레드를 가리키는 포인터이다. 단일 쓰레드 프로세스나 메인 쓰레드의 경우 main 포인터는 자신을 가리킨다. 메인 쓰레드는 자식 쓰레드를 생성하는 주체이며, 자식 쓰레드들은 메인 쓰레드의 주소 공간을 공유한다(엄밀히 말하자면 부모-자식 관계는 아니지만 적절한 이름이 생각나지 않아 자식 쓰레드라고 칭하겠다). 즉 쓰레드끼리 한 주소 공간을 공유하지만 이 주소 공간의 진정한 주인은 메인 쓰레드인 셈이다. 자식 쓰레드가 growproc으로 자신의 주소 공간을 키운다는 것은 결국 메인 쓰레드의 주소 공간을 키우는 것이다. 따라서 memlim 정보는 메인 쓰레드에만 저장하기로 했다. growproc에서도 메인 쓰레드의 memlim을 참조한다.

proc 구조체의 sz 변수, pgdirlock

다시 setmemorylimit의 코드를 보면, memlim과 달리 기존 할당 받은 메모리보다 limit이 작은지 비교할 때는 메인 쓰레드의 것이 아닌 p->sz를 비교한다. 사실 p->main->sz를 비교해도 상관은 없다. 어차피 둘의 sz 값은 항상 같도록 유지해 줄 것이기 때문이다. sz 값은 접근할 수 있는 주소 공간의 크기를 의미하므로, 쓰레드끼리 한 주소 공간을 늘 공유하기 위해서는 전부 통일된 sz 값을 가져야 한다. 실제로 xv6에서 sz 변수는 정말 중요한 정보이며 잘못 건드릴 경우 페이지 폴트는 물론이고, 나는 QEMU 시스템 자체가 먹통이 되는 상황에도 부딪혔었다.

따라서 LWP를 구현할 때 이 sz 값의 일관성을 각별히 조심해야 했다. 한 프로세스의 주소공간에 대해 여러 쓰레드가 동시에 작업을 시도할 수 있기 때문이다. 이런 상황에서 Race Condition이 발생하지 않도록 proc.c에 pgdirlock이라는 spinlock 변수를 만들어 활용하였다. 왜 ptable.lock 외의 별도로 만들었는지는 이후 설명하겠다.

growproc 함수 수정

```
int
growproc(int n)
{
  struct proc *curproc = myproc();
  struct proc *main = curproc->main;
  struct proc *p;

  uint sz = curproc->sz;
  int memlim = main->memlim;

  if (!holding(&pgdirlock))
    panic("pgdirlock");

  if(n > 0)
  {
    if (!memlim || memlim >= sz + n)
    {
      if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
        return -1;
    }
    else
      return -1;
  }
  else if (n < 0)
  {
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  }

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->main == main)
      p->sz = sz;
  }

  switchuvm(curproc);
  return 0;
}
```

수정한 growproc은 이 프로세스가 n만큼 메모리 할당을 받았을 때 자신의 메모리 제한을 넘지 않는지 확인한다. 성공적으로 할당을 해주었으면 이제 ptable을 순회하며 같은 주소 공간에 포함되는 프로세스들의 sz도 똑같이 갱신해준다. 이 모든 과정은 pgdirlock으로 보호되어야 하는데, growproc이 pgdirlock을 직접 선점하지는 않는 것을 볼 수 있다. 단지 호출 시점에 이것의 선점 여부를 따지고 선점되어 있지 않다면 panic을 발생시키므로 growproc을 호출하기 전에 선점을 해주어야 한다. 이런 구현의 이유는 다음에 설명할 sys_sbrk, 그리고 thread_create 함수와 관련이 있다.

sys_sbrk 함수 수정

기존 sys_sbrk 함수는 growproc을 호출하기 전 프로세스의 sz를 반환한다. 따라서 여기에도 Race Condition에 대한 보호가 되도록 아래와 같이 수정해 주었다.

```
extern struct spinlock pgdirlock;

int
sys_sbrk(void)
{
  ...
  acquire(&pgdirlock);
  addr = myproc()->sz;

  if(growproc(n) < 0)
  {
    release(&pgdirlock);
    return -1;
  }

  release(&pgdirlock);
  return addr;
}
```

결론부터 말하자면, 이렇게 보호를 해주어도 주소 공간에 대한 Race Condition의 발생은 끝내 해결하지 못했다. 조교님이 제공해주신 테스트 코드 중 thread_sbrk 테스트를 돌리면 열에 한번 꼴로 Failed나 페이지 폴트가 발생한다. 그래도 위 코드 디자인의 의도를 설명하고자 한다.

기존 sys_sbrk 처럼 growproc 하기 전의 sz를 반환하려면 growproc의 수행을 pgdirlock으로 보호하는 것만으로는 부족하다고 판단했다. addr = myproc()->sz를 읽어오고 growproc을 호출하려는 직전에 또 다른 곳에서 growproc이 호출되는 상황을 생각해보면 알 수 있다. 따라서 sz를 읽어오는 것과 growproc을 호출하는 것 모두 보호가 필요한 것이다. thread_create를 구현할 때도 비슷한 문제가 생겼다. thread_create 함수에서도 growproc을 호출하는데, growproc 수행이 끝나고도 주소 공간의 보호는 이어져야 했다. 반대로 growproc에서 pgdirlock을 선점하고 종료하기 전에 이를 풀어버린다면 호출 전후의 보호를 보장할 수 없게 된다고 생각했다. 따라서 growproc을 호출하는 함수가 자유롭게, 더 광범위하게 pgdirlock을 활용할 수 있도록 growproc에서 이를 선점하지 않게 구현한 것이다. 또 이를 위해서는 sys_sbrk도 spinlock을 써야하므로 ptable.lock과 별개로 pgdirlock을 만든 것이다.

쓰레드와 프로세스의 관계, 추가한 변수

쓰레드를 구현함에 있어 가장 첫 난관은 쓰레드와 프로세스의 관계를 이해하는 것이 아닐까 싶다. 명세에 등장하는 "thread_t"라는 쓰레드 자료형(구조체)를 따로 만들고 이를 기존 proc 구조체랑 달리 관리해야 한다는 생각을 깨는 것이 쓰레드 구현의 시작이었다. 쓰레드도 프로세스이므로, 새로 쓰레드 자료형을 만들지 않고 기존 proc 구조체에 정보를 추가하기로 했다. 쓰레드를 위해 추가한 정보(변수)는 다음과 같다.

- **thread_t tid**

  쓰레드끼리 식별하기 위한 정보이다. 기존 pid와 마찬가지로 thread_t 자료형은 단순히 int 자료형이며, types.h에 정의해두었다. 쓰레드가 아닌 프로세스의 경우 tid 값을 0으로 가지는데, 이것으로 쓰레드 여부를 확인하지는 않는다.

- **struct proc *main**

  앞서 설명했듯이 메인 쓰레드를 가리키는 포인터이다. 초기 디자인에는 고려하지 못했으나, 구현 과정에서 한 프로세스의 쓰레드들을 관리하는 일종의 중앙 집중된 "대표" 프로세스의 필요를 느껴 추가했다.

이후 내용은 시간이 부족한 관계로 코드와 함께 간략히 작성하겠다

thread_create 함수 구현

```
int
thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)
{
  struct proc *main = myproc()->main;
  struct proc *p;
  uint sp, ustack[2];
  pde_t *pgdir;

  acquire(&pgdirlock);

  if ((p = allocproc()) == 0)
    return -1;

  if (growproc(2 * PGSIZE) == -1)
  {
    cprintf("growproc fail\n");
    thread_free(p);
    return -1;
  }

  pgdir = p->pgdir = main->pgdir;
  sp = p->sz = main->sz;

  clearpteu(pgdir, (char*)(sp - 2 * PGSIZE));

  ...
```

allocproc을 호출해 ptable의 빈칸을 찾아 p를 할당해준다. allocproc에서 p의 kstack을 초기화해주고 여기에 trapframe과 context 같은 자료를 할당해준다.

growproc을 통해 2개의 페이지를 할당받는다. exec과 유사하게 아래 페이지는 가드용 페이지로 쓰기 위함이다. 또 p의 pgdir와 sz 정보를 main의 것과 통일시킨다. pgdir의 경우 주소값이 복사되므로 shallow copy를 해주는 셈이다.

또 sp = main->sz로 sp가 이제 할당된 스택용 페이지의 가장 윗부분을 가리키게 한다. 스택은 높은 주소에서 낮은 주소로 커지므로, 이제 sp를 감소시켜가며 스택에 원하는 내용을 푸쉬해야 한다.

```
*p->tf = *main->tf;
 p->tf->eip = (uint)start_routine;

 ustack[0] = 0xffffffff;
 ustack[1] = (uint)arg;

 sp -= 8;
 if (copyout(pgdir, sp, ustack, 8) < 0)
 {
   cprintf("ustack fail?\n");
   thread_free(p);
   release(&ptable.lock);
   return -1;
 }
 ...
```

다음은 p가 start_routine을 실행하도록 p의 trapframe과 스택 페이지에 작업을 해주어야 한다. trapframe은 커널 모드에서 유저 모드로 돌아갈 때를 위해 유저 모드에서의 레지스터 값들을 저장한 것이다. allocproc에서는 커널 스택에 return 주소로 trapret의 주소를 푸쉬해주고, context의 eip를 forkret으로 설정한다. 덕분에 이 프로세스로 context switching이 발생할 때 forkret을 실행하고, forkret은 trapret으로 return하는데, trapret에서는 trapframe에 저장해둔 정보를 레지스터들에 다시 복구해주는 작업을 한다. 따라서 trapframe의 eip, 즉 PC를 start_routine으로 설정해준다. 다음은 인자와 return 주소를 푸쉬해 준다.

```
   ...
   p->tf->esp = sp;
   p->pid = main->pid;
   p->tid = *thread = nextpid;
   p->parent = main->parent;
   p->main = main;

   release(&pgdirlock);
   ...
```

trapframe의 esp, 즉 스택의 top 포인터를 sp로 맞춰준다. pid는 메인 쓰레드와 공유하고 tid 값은 nextpid로 저장한다. nextpid는 원래 프로세스끼리 구분하기 위함이지만, allocproc이 호출될 때마다 증가하므로 tid로 써도 겹침 없이 식별자로 무방하다고 판단했다. parent와 main은 main의 것을 공유한다. 즉 한 프로세스의 모든 쓰레드에는 하나의 공통된 parent와 main이 유지된다. 여기까지의 과정을 pgdirlock으로 보호해준다. sz와 pgdir에 대한 작업은 물론, p->main = main까지 보호해주어야 다른 쓰레드가 growproc을 호출했을 때 자신의 sz까지 갱신될 것이기 때문이다.

```
  ...
  for(int i = 0; i < NOFILE; i++)
    if(main->ofile[i])
      p->ofile[i] = filedup(main->ofile[i]);
  p->cwd = idup(main->cwd);

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

  acquire(&ptable.lock);

  p->state = RUNNABLE;

  release(&ptable.lock);

  return 0;
}
```

main의 파일과 디렉토리 정보 또한 공유한다. filedup과 idup 함수는 사본을 만드는 것이 아닌, 각각에 대해 ref count 변수를 둬 현재 이것을 참조하는 프로세스의 수를 기록한다. 준비를 마쳤으니 state를 RUNNABLE로 바꿔주고 종료한다.

thread_exit 함수 구현

```
void
thread_exit(void *retval)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;
```

thread_exit은 기존 exit과 매우 유사하다. 앞서 언급한 filedup, idup 함수처럼 fileclose, iput 함수도 실제 파일을 닫는것이 아니라 ref count를 감소시키고, 이것이 0이 될때야 실제로 닫는 작업을 한다. 그러므로 thread의 exit에서도 이 과정을 진행한다.

```
  acquire(&ptable.lock);

  wakeup1(curproc->main);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p == p->main && p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  curproc->state = ZOMBIE;

  sched();
  panic("thread zombie exit");
}
```

exit과 달리, 자원 회수의 주체는 main이므로 main에 대해 wakeup1을 해준다. 그리고 고아 프로세스를 initproc에게 넘겨줄 때도 자식이 쓰레드인 경우를 고려해야한다. 고아 메인 쓰레드인 경우에만 initproc을 깨워준다. 이유는 wait 함수의 수정 부분에서 설명한다.

thread_join 함수 구현

```
int
thread_join(thread_t thread, void **retval)
{
  struct proc *p;
  struct proc *curproc = myproc();
  int found;

  acquire(&ptable.lock);

  for (;;)
  {
    found = 0;

    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
      if (p->tid != thread)
        continue;

      if (p->state == ZOMBIE)
      {
        *(uint*)retval = *(uint*)(p->tf->esp + 4);

        thread_free(p);

        release(&ptable.lock);
        return 0;
      }

      found = 1;
    }

    if (!found || curproc->killed)
    {
      release(&ptable.lock);
      return -1;
    }

    sleep(curproc, &ptable.lock);
  }
}
```

wait과 유사하게 ptable을 순회하며 ZOMBIE 상태인 프로세스를 찾는다. retval의 회수도 caller frame을 거슬러 올라가 가져올 수 있다. 정상적으로 종료한 쓰레드라면 thread_exit을 통해 종료했을 것이므로, thread_exit의 인자로 retval을 스택에 푸쉬하기 때문이다.

wait 함수 수정

기존 wait 함수는 현재 프로세스의 자식 중 ZOMBIE인 것을 찾아 자원을 회수한다. 하지만 쓰레드끼리 같은 부모를 공유하게끔 구현하였고, 자식 쓰레드의 자원회수는 메인 쓰레드가 하도록 구현하였으므로 wait은 꼭 메인 쓰레드인 프로세스들만을 찾도록 수정하였다.

3. 테스트 결과

```
$ thread_exec
Thread exec test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Thread 0 Executing...
Hello, thread!
```

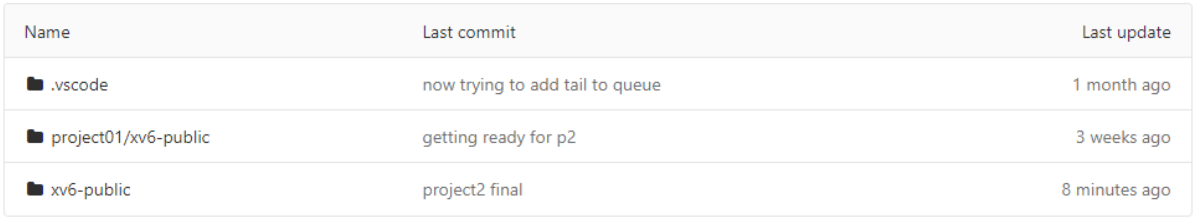
```
$ thread_kill
Thread kill test start
Killing process 10
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 timeThis code should be executed 5 times.
This code should be executes.
d 5 times.
Kill test finished
```

```
$ thread_exit
Thread exit test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Exiting...
```

```
$ thread_test
Test 1: Basic test
Thread 0 start
Thread 0 end
Thread 1 start
Parent waiting for children...
Thread 1 end
Test 1 passed

Test 2: Fork test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Child of thread 1 start
Child of thread 3 start
Child ofChild of thread 2 start
Child of thread 4 start
 thread 0 start
Child of thread 1 end
Child of thread 3 end
Thread 1 end
Thread 3 end
Child of thread 2 end
Child of thread 4 enThread 2 end
Child of thread 0 end
d
Thread 4 end
Thread 0 end
Test 2 passed

Test 3: Sbrk test
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Test 3 passed

All tests passed!
$
```

... 보시다시피 시간에 쫓겨 많이 부족한 위키를 제출하게 되었습니다. gitlab에 올린 코드에도 주석이 너무 많아 깔끔하게 정리해서 최종 제출을 하고자 했으나 이마저도 못하게 되었습니다. 정말 죄송합니다.

기존 프로젝트 1 소스 코드를 project01 폴더에 정리하고 새로 clone을 받아와 프로젝트를 진행했습니다.



Name	Last commit	Last update
.vscode	now trying to add tail to queue	1 month ago
project01/xv6-public	getting ready for p2	3 weeks ago
xv6-public	project2 final	8 minutes ago

여기서 xv6-public 폴더에 있는 것이 이번 프로젝트 2 소스 파일입니다.