# [ELE3021] Project03 Wiki

한양대학교 컴퓨터소프트웨어학부 2020089007 김형민

**목차**

# Design

## LWP

### User adress space of xv6

xv6 프로세스의 주소공간은 논리주소 0에서부터 코드-데이터-스택-힙의 구조로 이루어져 있다. 스택은 initproc을 제외하면 2개의 페이지(4096byte)의 고정된 크기를 가진다.

스레드 개념을 도입하면서 이를 변경하려고 한다. 스레드는 각자의 스택 영역을 가지며 코드, 데이터, 힙의 3가지 영역은 피어 스레드들과 공유한다. 그리고 같은 pgdir을 갖는다.

| | |
|---|---|
| 공유 자원 | 코드(text), 데이터, 스택, 오픈 파일, 페이지 테이블 |
| 독립 자원 | 유저 스택, 커널 스택 |

주소 공간은 코드-데이터-(메인 스레의 스택)-(스택 or 힙)-(스택 or 힙)… 와 같이 바뀐다. 여기서 메인 스레드란 fork를 통해 처음 생성된 프로세스를 말한다. 메인 스레드의 스택 이후에는 프로그램마다 다음 스레드의 스택이 할당될지, 아니면 공유하는 힙 영역이 할당될지 동적으로 실행시간에 결정된다.

### struct proc

```
// proc.h
struct proc {
  ...
  thread_t tid;
  int num_lwp;
  void *retval;
}
```

- thread_t tid
  - 스레드의 ID를 나타낸다. 메인 스레드는 0, 나머지는 생성 순서에 따라 순차적으로 증가한다.
- int num_lwp
  - 해당 프로세스에 속한 스레드의 개수를 나타낸다.
- void *retval
  - thread_exit에서 인자로 받아 thread_join으로 넘겨줄 값이다.

### thread_create

여기서 수행해야할 동작들은 다음과 같다.

1. ptable을 순회하면서 UNUSED 상태인 것을 찾는다.
2. 스레드의 기본적인 상태 정보들을 설정하고 커널 스택을 할당한다.
3. 할당한 커널 스택에 trap frame과 context를 업데이트 한다.
4. 메인 스레드의 페이지 테이블, 오픈 파일, 현재 디렉토리 상태 등을 공유한다.
5. 2페이지 크기의 유저 스택을 할당한다.

6. 할당한 유저 스택에 인자로 받은 arg를 push한다.
7. trapframe의 eip와 esp를 각각 start_routine, 앞서 push한 arg에 맞게 변경한다.
8. 스레드의 상태를 RUNNABLE로 바꾼다.
9. 인자로 받은 thread에 생성된 스레드의 tid를 할당한다.

### thread_exit

여기서 수행해야할 동작들은 다음과 같다.

1. SLEEP일 가능성이 있는 메인 스레드를 wakeup1을 통해 깨운다.
2. 스레드의 state를 ZOMBIE로 변경한다.
3. 인자로 받은 retval의 값을 스레드의 retval 속성에 할당한다.
4. 스케줄러에게 실행 흐름을 넘겨준다.

### thread_join

여기서 수행해야할 동작들은 다음과 같다.

1. ptable을 순회하며 인자로 받은 tid를 가진 스레드를 찾는다.
2. 해당 스레드의 자원들을 할당 해제 및 초기화한다.
3. 인자로 받은 retval에 thread_exit에서 받은 lwp→retval 값을 넘겨준다.

## Locking

피터슨 알고리즘은 스레드가 2개일 때에만 유효하므로 사용할 수 없다.

따라서 베이커리 알고리즘과 세마포어를 이용하여 locking을 구현할 것이다.

안정성을 위해 베이커리 알고리즘으로 세마포어를 보호하는 형식으로 구현한다.

### Bakery algorithm

shared_resource는 빵집, 각각의 스레드들은 고객이라고 할 수 있다.

- 고객들은 맨 처음 번호표를 발급받고 자신의 순서가 올 때까지 대기한다.
  - 번호표는 스레드들의 우선순위를 나타낸다. 번호표의 숫자가 작을수록 임계 영역 접근에 있어서 더 높은 우선순위를 가진다.
- 빵집에 방문한 고객은 번호표를 소모하였다. 따라서 다시 빵집에 방문하고 싶다면 번호표를 재발급 받아야 한다.
  - 번호표 재발급 시, 대기중인 고객들이 가진 번호표들 중 최대값에서 1을 더한 값을 가진다.
  - 빵집 앞에 고객들이 일렬로 줄을 서고 있는 것을 상상하면 된다. 빵집 방문을 마치고 나온 고객이 다시 방문을 원한다면 줄의 맨 뒤로 가서 서야한다.

### Semaphore

정수 변수 하나(S)와 여기에 접근 가능한 유이한 함수 두 개(P(S), V(S))를 제공한다.

```
Initially S = 1;
P(S):
  while(S <= 0);
  S--;

V(S):
  S++;

P(S)
critical section
V(S)
```

n개의 스레드가 임계 영역에 진입을 원한다고 가정할 때, 하나의 스레드가 먼저 P(S)를 호출하고 임계 영역에 진입했다면 S는 0이 될 것이므로 나머지 n-1개의 스레드는 진입한 스레드가 작업을 마치고 나와 V(S)를 통해 S를 다시 1로 만들 때까지 P(S)의 while에서 busy waiting을 해야한다.

### Conclusion

싱글 코어 시스템에서는 베이커리 알고리즘과 세마포어 둘 중 하나만 사용하여 lock과 unlock을 구현하여도 스레드 간 동기화를 달성할 수 있다. 하지만 최근의 거의 모든 PC는 멀티 코어 시스템을 사용하므로 예측할 수 없는 영역에서 race condition이 발생할 위험이 존재한다. 따라서 더 높은 안정성을 위해 베이커리 알고리즘과 세마포어를 혼합하여 lock과 unlock을 구현할 것이다.

# Implement

## LWP

### thread_t

```
// types.h
...
typedef uint thread_t;
```

### alloclwp

```
// lwp.c
static struct proc*
alloclwp(void)
{
  ...
found:
  lwp->state = EMBRYO;
  lwp->pid = curproc->pid;

  // Initialize lwp attribute.
```

```
  lwp->tid = ++curproc->num_lwp;
  ...

  return lwp;
}
```

ptable에 새로운 스레드를 할당하는 함수이다.

1. ptable을 순회하며 빈 공간을 찾고, 그 곳에 스레드를 할당한다.
2. 스레드의 pid는 메인 스레드(curproc)의 pid를 그대로 사용한다.
3. 스레드는 0이 아닌 tid를 사용한다.
4. 스레드에 커널 스택을 할당한다.
5. 할당한 커널 스택에 trapframe과 context를 push하고 이들을 초기화한다.

### dealloclwp

```
// lwp.c
void
dealloclwp(struct proc *lwp)
{
  kfree(lwp->stack);
  lwp->kstack;
  lwp->pid;
  lwp->parent->num_lwp--;
  lwp->parent = 0;
  lwp->pgdir = 0;
  lwp->name[0] = 0;
  lwp->killed = 0;
  lwp->tf = 0;
  lwp->context = 0;
  lwp->state = UNUSED;

  // Disconnect from open files.
  for(int fd = 0; fd < NOFILE; fd++){
    if(lwp->ofile[fd]){
      lwp->ofile[fd] = 0;
    }
  }
  lwp->cwd = 0;
}
```

스레드의 자원을 할당 해제하는 함수이다.

1. 커널 스택을 회수한다.
2. 스레드의 상태 정보를 초기화한다.
3. 파일 정보를 초기화한다.

오픈 파일 리소스 접근에 있어 스레드가 파일의 refcnt를 증가시키는 것이 아니라 단순히 메인 스레드의 참조를 이용하는 방법으로 구현하였기 때문에 fileclose를 사용할 필요 없이 그냥 참조 정보를 0으로 초기화 해주면 된다.

### thread_create

```
// lwp.c
int thread_create(thread_t *thread, void *(*start_routine)(void *), void *a
rg)
{
  ...
  // Allocate lwp.
  if((lwp = alloclwp()) == 0)
    return -1;

  // Avoid remap conflict with growproc.
  acquire(&ptable.lock);

  // Allocate 2 pages of stack to lwp.
  sz = PGROUNDUP(curproc->sz);
  if((sz = allocuvm(curproc->pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(curproc->pgdir, (char*)(sz - 2*PGSIZE));
  curproc->sz = sz;
  sp = sz;

  // For sharing heap segment, We need to expand lwp's size.
  lwp->parent = curproc;
  for(t = ptable.proc; t < &ptable.proc[NPROC]; t++){
    if(t->tid == 0 || t->parent != curproc)
      continue;
    t->sz = curproc->sz;
  }
  ...
}
```

1. 앞서 구현한 alloclwp를 통해 ptable에 스레드를 할당하고 관련 작업을 해준다.
2. 스레드에 스택을 할당하기 이전에, 프로세스에 힙을 할당하는 growproc과 충돌이 일어나는 것을 방지하기 위해 ptable에 대한 락을 획득한다.
3. 2개의 페이지를 스레드에 할당한다. 이는 프로세스의 페이지 테이블을 통해 접근한다.
4. 페이지 테이블을 모든 스레드가 공유하므로 사이즈 변경이 일어날 경우 이를 해당 프로세스의 모든 스레드들이 반영한다.

엄밀히 말하면 스레드에 스택을 할당한다기 보다는, 프로세스에 스택을 할당하고 이에 대한 접근 정보를 해당 스레드에게 넘겨준다는 표현이 더 정확하다. 이에 대한 구현은 이후 자세히 나올 것이다.

```
// lwp.c
  ...
  // Just share main lwp's(process) pgdir.
  lwp->pgdir = curproc->pgdir;

  *lwp->tf = *curproc->tf;

  release(&ptable.lock);

  // Push arg to user stack.
  ustack[0] = 0xffffffff; // fake return PC
  ustack[1] = (uint)arg;

  sp -= 8;
  if(copyout(lwp->pgdir, sp, ustack, 8) < 0)
    goto bad;

  // Assign a start point and stack pointer.
  lwp->tf->eip = (uint)start_routine;
  lwp->tf->ebp = sp + 8;
  lwp->tf->esp = sp;

  ...

  acuqire(&ptable.lock);
  lwp->state = RUNNABLE;
  release(&ptable.lock);

  // Pass ID of lwp.
  *thread = lwp->tid;

  return 0;
  ...
}
```

1. 페이지 테이블을 새로 만들지 않고, 프로세스 내의 모든 스레드가 공유하도록 한다.
2. 인자로 받은 arg를 ustack에 push하고 이를 앞서 할당한 유저 스택에 복사한다.
3. lwp→tf→eip를 통해 유저 모드에서 이 스레드가 start_routine을 실행하도록 한다.
4. lwp→tf→ebp, lwp→tf→esp를 통해 스레드가 유저 스택에 대한 접근 정보를 갖도록 한다.
5. 스레드가 스케줄링 될 수 있도록 상태를 RUNNABLE로 변경한다.
6. 인자로 받은 thread에 할당한 스레드의 id 값을 저장해준다.

앞서 말했듯 생성된 스레드는 esp, ebp를 통해 유저 스택에 대한 접근 정보를 갖게 된다.

### thread_exit

```
// lwp.c
void
thread_exit(void *retval)
{
  ...
  acquire(&ptable.lock);

  // Main lwp might be sleeping in wait().
  wakeup1(lwp->parent);

  // Pass retval to thread_join().
  lwp->retval = retval;

  // Jump into the scheduler, nerver to return.
  lwp->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

1. thread_join을 통해 자원을 회수해야할 메인 스레드가 SLEEP일 경우를 생각해서 깨워준다.
2. thread_join을 통해 받아갈 retval을 저장해준다.
3. 종료되었지만 자원 회수가 이루어지지는 않았음을 알리기 위해 상태를 ZOMBIE로 바꿔준다.
4. 스케줄러에게 실행 흐름을 넘겨준다.

### thread_join

```
// lwp.c
int
thread_join(thread_t thread, void **retval)
{
  ...
  acquire(&ptable.lock);

  for(;;){
    // Scan through table looking for exited lwp.
    havekids = 0;
    for(lwp->ptable.proc; lwp < &ptable.proc[NPROC]; lwp++){
      if(lwp->tid != thread || lwp->parent != curproc)
        continue;

      havekids = 1;
      // lwp exited.
      if(lwp->state == ZOMBIE){
        dealloclwp(lwp);
        *retval = lwp->retval;
        release(&ptable.lock);
```

```
        return 0;
      }
    }
  ...

  // Wait for thread to exit.
  sleep(curproc, &ptable.lock);
}
```

1. 인자로 받은 thread에 해당하는 tid를 가진 스레드가 exit하였는지 확인한다.
    a. exit하였을 경우 dealloclwp를 통해 자원을 회수하고 exit에서 받아간 retval을 넘겨준다.
    b. 아직 exit하지 않았을 경우 sleep한다.

스레드가 개인적으로 가진 자원은 커널 스택과 유저 스택이기 때문에 개념적으로는 유저 스택도 이 단계에서 반납받아야 한다. 그러나 주소 공간의 경우의 수가 복잡하기 때문에 구현이 어렵다.

- 코드-데이터-스택-스택-힙-스택-힙…
- 코드-데이터-스택-힙-스택-스택-힙…

위와 같이 프로그램의 구조에 따라서 스택과 힙 영역이 완벽하게 분리되지 않고 복합적으로 나타날 수 있기 때문에 프로그램이 완전히 종료되지 않은 단계에서 자원을 회수하는 것은 위험하다.

하나의 스택을 할당 해제하고 나머지를 앞으로 당기는 방식으로 구현할 수도 있겠으나, 이 또한 논리 주소가 변하게 되므로 segmentation fault 혹은 다른 스레드의 스택 영역을 침범할 위험이 있다.

따라서 thread_join을 통해서는 커널 스택 자원만 회수해주고, 유저 스택은 이후 메인 스레드가 종료하면서 페이지 테이블, 코드, 데이터, 힙 등의 모든 자원을 회수할 때 한 번에 회수해줄 것이다.

### fork

```
// proc.c
int
fork(void)
{
  ...
  // It becomes main lwp.
  np->tid = 0;
  ...
}
```

메인 스레드가 아닌 서브 스레드 중에서도 fork를 호출할 가능성이 있기 때문에 메인 스레드로 만들어준다.

### growproc

```
// proc.c
int
growproc(int n)
{
```

```
  // Avoid remap conflict with thread_create.
  acquire(&ptable.lock);

  // We need to distinguish between sub lwp and main lwp.
  p = curproc->tid == 0 ? curproc : curproc->parent;

  ...

  // Heap is shared segment by all lwps.
  for(lwp = ptable.proc; lwp < &ptable.proc[NPROC]; lwp++){
    if(lwp->tid == 0 || lwp->parent != p)
      continue;
    lwp->sz = sz;
  }

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

1. thread_create와의 충돌을 피하기 위해 ptable에 대한 락을 획득한다.
2. 서브 스레드들은 parent 속성을 통해 메인 스레드를 가리키고 있으므로 힙 자원 할당은 메인 스레드를 기준으로 일어나도록 해준다.
3. 힙은 프로세스 내 모든 스레드들이 공유해야하므로 sz 속성을 변경하여 모든 스레드들이 힙 영역에 접근할 수 있도록 해준다.

### exec

```
// exec.c
int
exec(char *path, char **argv)
{
  ...
  // It need to be adopted by the parent of main lwp if it is sub lwp.
  if(curproc->tid != 0)
    curproc->parent = curproc->parent->parent;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == curproc->pid){
      if(p->tid == curproc->tid)
        continue;
      deallocLwp(p);
    }
  }
  ...
  curproc->tid = 0;
  curproc->num_lwp = 0;
```

```
  switchuvm(curproc);
  freevm(oldpgdir);

  return 0;
  ...
}
```

1. exec을 호출한 스레드가 메인 스레드가 아닐 경우, 이것의 parent 정보는 메인 스레드의 parent, 즉 이전에 프로세스를 생성한 프로세스의 메인 스레드를 가리키도록 한다.
    a. sh(main lwp) <- thread_test(main lwp) <- thread_test(sub1 lwp)의 구조로 이루어져 있으며 sub1 lwp가 exec을 호출했다고 가정하자. 이때 sub1 lwp의 parent 속성은 sh(main lwp)를 가리키도록 변경되어야 한다.
2. 이후 프로세스 내 모든 스레드들을 dealloclwp를 통해 정리한다.
3. 최종적으로 freevm(oldpgdir)을 통해 모든 자원들을 회수한다.

### exit

```
// proc.c
void
exit(void)
{
  ...
  // Pass abandoned lwps to parent.
  for(lwp = ptable.proc; lwp < &ptable.proc[NPROC]; lwp++){
    if(lwp-> tid != 0 && lwp->pid == curproc->pid){
      lwp->state = ZOMBIE;
      lwp->parent->state = ZOMBIE;
      lwp->parent = lwp->parent->parent;
    }
  }
  ...
}
```

서브 스레드는 반드시 thread_exit를 통해서 종료되어야 한다. 따라서 exit를 통해 종료되는 상황은 있어서는 안되는 예외적인 상황이다. 그러나 이에 대해서도 대응해야 한다.

1. wait을 통해 자원이 회수될 수 있도록 서브 스레드들의 상태를 모두 ZOMBIE로 바꾼다.
2. 서브 스레드에서 exit가 호출되는 경우를 고려하여 lwp→parent→state 또한 ZOMBIE로 바꿔주고, 서브 스레드를 메인 스레드의 parent에게 입양시킨다.
    a. 여기서 lwp→parent는 메인 스레드를 뜻한다.

### wait

```
// proc.c
int
```

```
wait(void)
{
  ...
  for(lwp = ptable.proc; lwp < &ptable.proc[NPROC]; lwp++){
    if(lwp->tid == 0 || lwp->parent != curproc)
      continue;
    if(lwp->state == ZOMBIE)
      dealloclwp(lwp);
  }
  ...
}
```

1. 메인 스레드의 경우 추가적인 자원 회수가 요구되므로 이후 과정에서 처리되도록 한다. 기존에 구현되어있는 wait 루틴에 따라 처리된다.
2. 모든 서브 스레드들의 자원을 회수한다.

### kill

```
// proc.c
int
kill(int pid)
{
  ...
  for(lwp = ptable.proc; lwp < &ptable.proc[NPROC]; lwp++){
      if(lwp->pid == pid){
        if(lwp->tid == 0){
          lwp->killed = 1;
          rv = 0;
          // Wake process from sleep if necessary.
          if(lwp->state == SLEEPING)
              lwp->state = RUNNABLE;
        }
        else
          dealloclwp(lwp);
      }
  }
  ...
}
```

1. 메인 스레드는 기존 루틴에 따라 처리된다.
2. 서브 스레드들의 자원을 회수한다.

## Locking

```
// pthread_lock_linux.c
...
```

```
int n = NUM_THREADS;

// 번호표를 발급중인지 확인. 0은 발급 완료, 1은 아직.
int choice[NUM_THREADS] = { 0 };

// 번호표, 0은 발급 안받았음. 숫자가 작을수록 우선순위가 높음
int ticket[NUM_THREADS] = { 0 };

// shared_resource에 대해 동기화하는 세마포어 변수
int S = 1;

// 현재 가장 낮은 우선순위를 가진 번호표를 반환
int max(void)
{
  int rv = ticket[0];
  for (int i = 1; i < NUM_THREADS; i++) {
    if (ticket[i] > rv)
      rv = ticket[i];
  }
  return rv;
}

void lock(int tid)
{
  // 데드락 발생을 감지하기 위한 변수.
    long cnt;

  // 고객이 번호표를 발급받는 중.
  choice[tid] = 1;

  // 고객은 제일 낮은 우선순위의 번호표를 받음(줄의 맨 뒤로 감).
  ticket[tid] = 1 + max();

  // 고객이 번호표 발급을 완료함.
  choice[tid] = 0;

  // 1. 번호표를 발급중인 고객이 있으면 대기
  // 2. 나보다 높은 우선순위의 번호표를 가진 고객이 있을 경우 대기(내 앞에 사람 있음)
  // 3. 나와 번호가 같은데 나이가 더 많은 경우(i < tid) 대기
  // 3번의 경우 max 함수에 대한 race condition이 발생할 수 있으므로 고려해야함.
  for (int i = 0; i < NUM_THREADS; i++) {
    if (i == tid) continue;

    cnt = 0;
    while (choice[i]) {
      if (++cnt > NUM_ITERS * NUM_THREADS) {
        printf("Deadlock occured.\n");
        pthread_exit(NULL);
```

```
      }
    }

    cnt = 0;
    while (
      (ticket[i] != 0) &&
      (ticket[i] < ticket[tid] || (ticket[i] == ticket[tid] && i < tid))
    ) {
      if(++cnt > NUM_ITERS * NUM_THREADS) {
          printf("Deadlock occured.\n");
          pthread_exit(NULL);
      }
    }
  }

  cnt = 0;
  while (S <= 0) {
    if (++cnt > NUM_ITERS * NUM_THREADS) {
      printf("Deadlock occured.\n");
      pthread_exit(NULL);
    }
  }
  S--;
}

void unlock(int tid)
{
  S++;

  // 용건이 끝났으면 번호표는 사라짐.
  ticket[tid] = 0;
}

...
```

코드에 대한 상세한 설명은 주석에 적었다. 몇 가지 중요한 점들만 짚고 넘어가겠다.

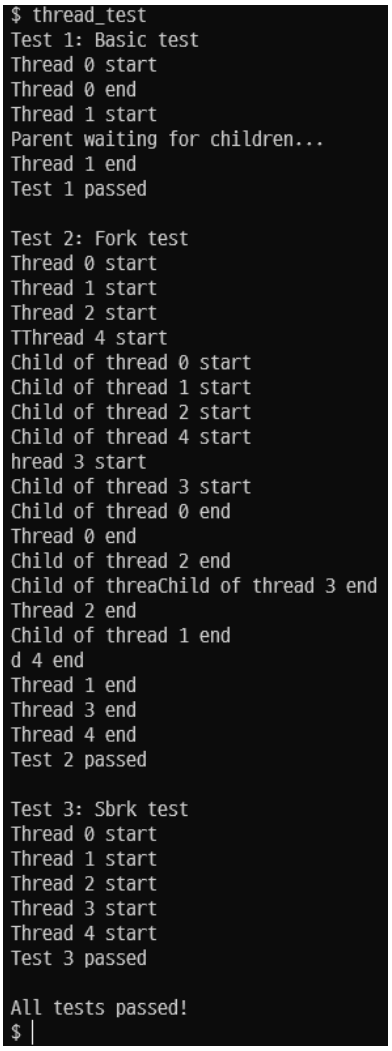
- 베이커리 알고리즘으로 세마포어 변수 S를 동기화하고, P에 해당하는 lock, V에 해당하는 unlock을 통해서 shared_resource 변수에 대한 동기화를 구현하였다.
- 즉 이중 동기화인 것이다. 세마포어 변수 S에 대한 접근을 동기화하여 세마포어를 구현하고, 구현한 세마포어를 통해서 shared_resource에 대한 동기화를 구현한다.
- 베이커리 알고리즘의 (ticket[i] == ticket[tid] && i < tid)에 대해 설명하겠다.
  - max 함수 또한 임계 영역에 해당될 수 있음에도 이에 대해 동기화 알고리즘을 적용하지 않았기 때문에 race condition이 발생하여 여러 개의 스레드의 번호표가 같은 우선순위를 가질 수 있고, 이는 곧 shared_resource에 대한 race condition으로 이어질 수 있다.
  - 따라서 max 함수에 대한 race로 인해 같은 우선순위의 번호표를 가지는 경우가 발생했다면 더 먼저 생성된, tid가 작은 스레드에게 임계 영역에 대한 접근 권한을 부여한다.

- 멀티 코어 시스템이므로 데드락이 발생할 위험이 존재한다. 따라서 무한반복의 가능성이 있는 곳마다 반복의 횟수가 임계점을 넘어가면 데드락이라 판단하고 프로세스를 종료하도록 한다.

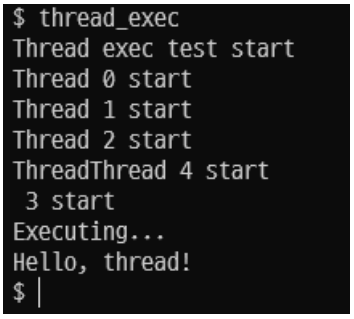
# Result

## LWP

### thread_test

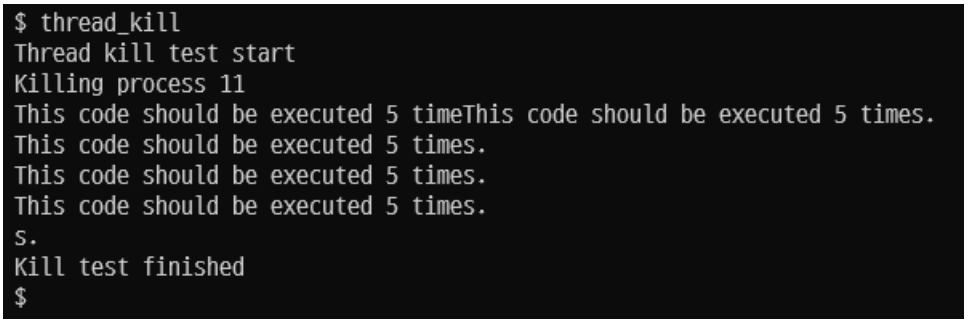



### thread_exec



### thread_exit


### thread_kill




### Comment

모든 테스트를 정상적으로 통과하였다. 설계대로 동작한 것으로 보인다.

thread_join시 유저 스택이 곧바로 회수되지 않는다는 점이 아쉽다. 하지만 최종적으로 프로세스가 종료되는 시점에서는 커널 스택, 유저 스택, 힙, 코드, 데이터, 오픈 파일 정보 등의 모든 자원들이 회수되기 때문에 이는 구현상의 차이일 뿐 명세를 위반한다고 볼 수는 없다.

## Locking

### Race condition

NUM_THREADS와 NUM_ITERS를 바꿔가며 테스트를 진행하는 동안 race condition 발생하지 않았다.

### Dead lock

NUM_THREADS = 5000, NUM_ITERS = 50000 정도의 규모로 키워서 테스트를 진행했더니 데드락이 빈번하게 발생하기 시작했다. 프로세스는 종료되지 않고, CPU 사용량만 100%에 가깝게 올라가는 것으로 보아 데드락이 확실했다.

멀티 코어 시스템이므로 어디서 어떻게 데드락이 발생할지 예측하는 것이 어렵다. 따라서 공유 변수에 접근하여 데드락이 발생할 가능성이 있는 곳마다  반복 횟수의 upper bound를 정하였다. 따라서 최악의 경우에도 프로그램은 종료하게 된다.

# Trouble shooting

## LWP

### Trouble 1

thread_join에서 xv6가 계속 재부팅되는 문제를 겪었다. 초기에 기능을 부분적으로 추가해나가면서 함수를 완전히 완성시키지 않았기에 발생한 일이었다. retval에 대해서 설정해주지 않았었는데, 이것은 메인 스레드가 서브 스레드들보다 먼저 종료되어서 참조하는 pgdir이 날아가 발생하는 문제인 것처럼 보였다. 아직 덜 만들었기에 발생한 문제라는 것을 깨닫고 개발을 완료하고 난 뒤에는 같은 문제가 발생하지 않았다.

### Trouble 2

스레드에 할당한 유저 스택 영역을 해제하는 과정에서 어려움을 겪었었다. xv6의 주소 공간 구조에서는 스레드별로 스택 영역을 할당 해제하는 것은 복잡함을 깨달아서 커널 스택 영역을 할당해제 하는 것으로 만족하고, 유저 스택은 메인 스레드 종료 시 같이 회수하는 것으로 가닥을 잡았다.

### Trouble 3

malloc을 통해 힙을 할당하는 과정에서 sbrk remap 에러를 여러 번 겪었다. 이는 스레드 생성과 힙 할당 사이에 동기화가 이루어지지 않았기에 remap 충돌이 발생해 일어난 것이었다. allocuvm에서 로그를 출력하는 디버깅을 통해 해당 문제를 확인했다. growproc, thread_create 각자의 시스템콜에서 ptable에 대한 lock을 획득하여 인터럽트를 비활성화 했고, 이를 통해 스레드 생성과 힙 할당 사이의 동기화를 이룰 수 있었다.

## Locking

### Trouble 4

초기에 베이커리 알고리즘만을 적용하여 구현하였을 때 race condition이 빈번하게 발생하는 문제를 겪었었다. 디버깅을 통해 같은 우선순위를 가지는 스레드들이 존재함을 깨닫고, max 함수는 동기화가 이루어지지 못해 race condition이 발생할 수 있다는 것을 알게되었다. 따라서 우선순위가 같은 스레드들 사이에서도 실행 순서를 정해줌으로써 여기서 발생하는 race condition 문제를 해결할 수 있었다. 또, 세마포어 개념을 추가하니 race condition은 보지 못했다.

### Trouble 5

race condition 문제를 해결하고 나니 데드락이 발목을 잡았다. 머리를 굴려봤지만 해결 방법이 보이지 않아 반복 횟수에 upper bound를 정해주는 것으로 결정하였다. 데드락이 발생하여도 종료는 된다.