project03 wiki - 2022028522

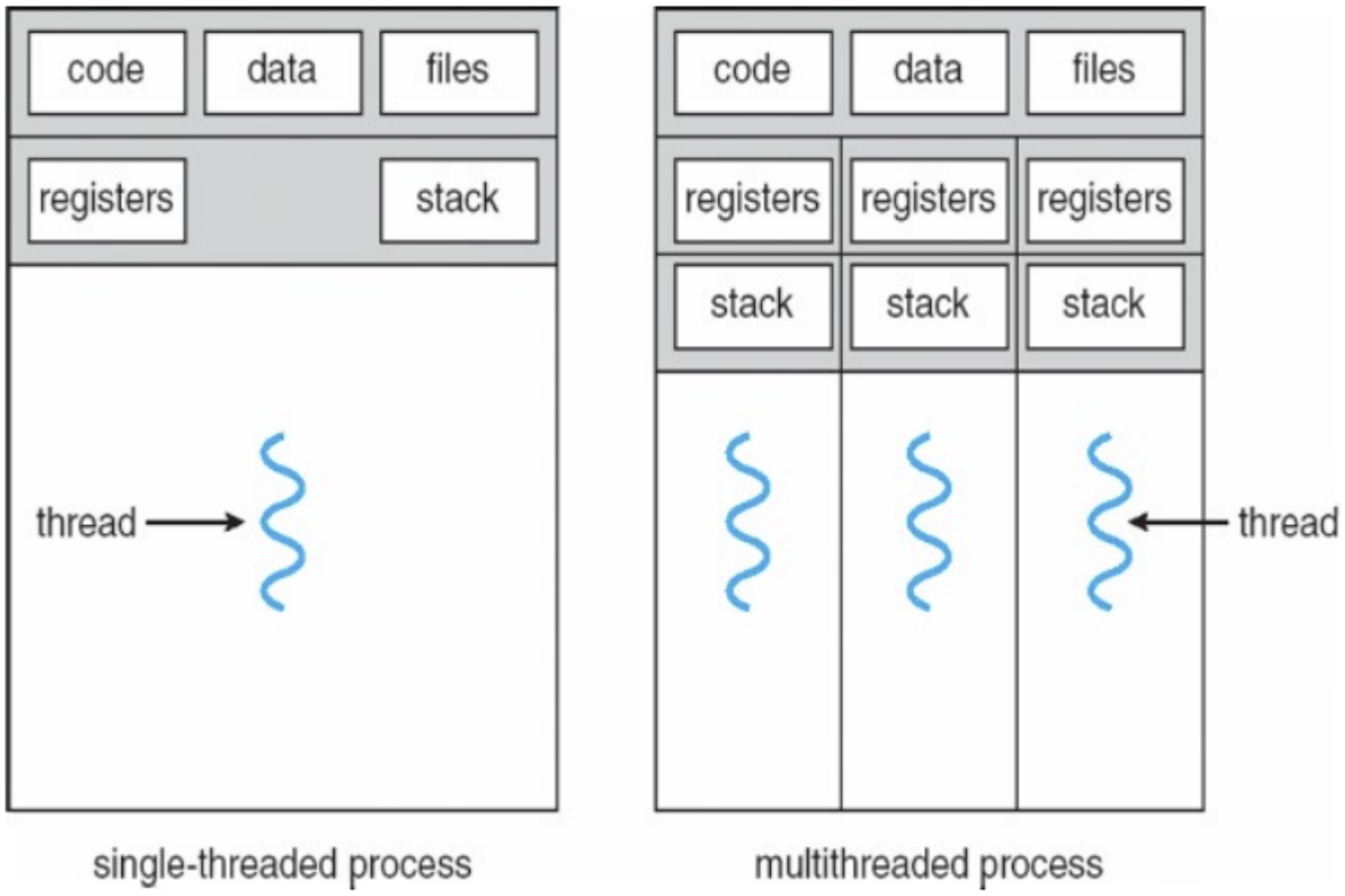
1. Design

> 📌 명세에서 요구하는 조건에 대해서 어떻게 구현할 계획인지, 어떤 자료구조와 알고리즘이 필요한지, 자신만의 디자인을 서술합니다.

Light-Weight Process

Thread

과제를 진행하기에 앞서, 기본적인 스레드의 개념에 대해 알아보았습니다.



각각의 스레드들은 data, text를 서로 공유하며, stack과 register context를 고유하게 갖습니다.

위와 같은 개념을 생각해보았을 때 제가 스레드를 구현하기 위해 생각한 방법은 process를 스레드처럼 동작하게 만들자는 것이었습니다. 어차피 스레드는 process와 기본적인 구조가 비슷하고, 다른 스레드와 data와 text만 공유할 수 있게 하면 된다고 생각했습니다.

proc 구조체

스레드를 process처럼 동작하게 하기 위해서는, `proc.h`의 `proc` 구조체에 새로운 멤버변수를 추가해야합니다.

```
// proc.h

struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
```

```
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

    // 추가된 부분
  thread_t tid;
  struct proc *main;
  void *retval;
};
```

`thread_t tid`는 스레드의 id를 저장하는 변수이고, `void *retval`은 스레드가 반환하는 값으로 `thread_exit`과 `thread_join`을 구현하기 위해 추가하였습니다. `struct proc *main`은 최초의 스레드를 저장하는 포인터이며, `allocproc()`로 할당받게 되는 프로세스를 모두 `main`으로 지정해주었습니다. 이후 `thread_create`에서 스레드를 통해 스레드를 생성하는 경우 새로 생성된 스레드의 `main`을 그 스레드를 생성한 스레드의 `main` 값으로 저장할 수 있게 하였습니다.

```
// types.h

typedef unsigned int   uint;
typedef unsigned short ushort;
typedef unsigned char  uchar;
typedef uint pde_t;

// types.h에서 thread_t 추가
typedef int thread_t;
```

이때 `thread_t` 자료형은 원래 없는 자료형이므로, `types.h`에서 `typdef int thread_t`를 추가해줌으로써 `thread_t`가 자료형으로 동작할 수 있게 만들었습니다.

API

과제에서 명시된 추가해야할 API는 다음 세 개이고, 이들을 어떻게 구현해야할지 대략적으로 디자인 해보았습니다.

1. `int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)`
   - 현재 실행 중인 프로세스 혹은 스레드로부터 새로운 스레드를 만드는 함수입니다. 그러므로 기존의 `fork` 함수와 `exec` 함수를 참고하였습니다.
   - 새로 만들어진 스레드는 메모리 공간을 다른 스레드들과 공유해야하므로, `main`이 그로부터 파생된 모든 스레드들을 관리하는 느낌으로 `main`의 `pgdir`를 공유하는 방식으로 구현하였습니다.
   - 새로 만들어진 스레드는 프로세스가 아니기 때문에, `allocproc`을 사용하되 `pid`가 증가하지 않게 하여 생성된 스레드는 그 스레드를 생성한 스레드와 같은 `pid` 값을 가질 수 있도록 해야 합니다. 대신, 스레드들을 구분할 수 있게 `tid`는 다른 값이 되어야 합니다.
   - 스레드가 생성되면 스레드가 `start_routine`에서 시작하게 하기 위해 `trapframe`에서 수정해야할 부분을 수정하였습니다.
2. `void thread_exit(void *retval)`
   - 스레드를 종료하는 함수이므로 기존의 `exit` 함수를 참고하였습니다.
   - 다만 `exit` 함수와 다르게 자식 프로세스는 관리할 필요가 없으니 자식 프로세스를 다루는 부분을 생략하고, 부모 프로세스 대신 `main`을 `wakeup`하여 `thread_join`을 통해 반환값을 받아가도록 하였습니다.
   - `retval`을 `thread_join` 함수에게 넘여주기 위해 `proc` 구조체에서 선언해둔 멤버변수 `retval`를 이용하면 됩니다.
3. `int thread_join(thread_t thread, void **retval)`
   - 스레드가 종료되기를 기다리는 함수이므로 기존의 `wait` 함수를 참고하였습니다.
   - 다만 `wait` 함수와 다르게 다른 스레드가 공유 자원을 사용하고 있을 수 있으므로 프로세스의 `pgdir` 전체를 `freevm`하는 과정을 생략해야 합니다.

기존 system call들과의 호환성

스레드는 프로세스를 기반으로 작동하기 때문에 기존의 system call들과 호환 가능하게 기존의 system call들을 살짝 수정해줄 필요가 있습니다. 수정이 필요한 system call들은 아래와 같습니다.

- `fork`: 새로운 프로세스를 생성할 때 `curproc`의 `sz`가 아닌 `curproc`의 `main`의 `sz`를 이용하도록 하였고, 메인 스레드가 아닌 스레드가 다른 프로세스의 부모가 되는 것을 막기 위한 과정을 추가해야합니다.
- `exit`: 스레드가 `exit`을 호출하게 되면 해당 스레드가 속한 모든 스레드가 종료되어야 하기 때문에 이에 대한 수정이 필요합니다. 이는 `pid`가 같으면 같은 프로세스의 스레드라는 의미이므로 `ptable.proc` 배열을 순회하며 `curproc`과 같은 `pid`를 가진 스레드를 찾는 방식으로 구현할 수 있습니다.
- `sbrk`: `sbrk`를 호출하면 내부적으로 `growproc`을 호출하여 메모리 사용 영역의 크기를 늘려줍니다. 이때 스레드들은 `main`을 통해 자원을 공유하므로 메모리 영역의 사이즈는 항상 `main`의 `sz`를 사용하도록 해야 합니다.
- `exec`: 스레드가 `exec`을 호출하면 자신 외의 다른 스레드를 모두 정리하고 해당 스레드의 정보가 새로운 프로세스도 덮어씌워지게 됩니다. 그러므로 `exec`을 실행한 스레드가 메인 스레드가 아닐 경우 메인 스레드의 `parent`를 상속하고 `tid`를 `0`으로 설정하여 스스로 메인 스레드가 되는 과정과 다른 스레드들을 정리하는 과정이 추가로 필요합니다.

Locking

C library에서 제공하는 동기화 API를 사용하지 않고 `lock`과 `unlock`을 구현하기 위해서 가장 먼저 든 생각은 소프트웨어적인 알고리즘만으로 상호 배제 문제를 해결할 수 있는 peterson algorithm을 이용하는 것이었습니다. 그러나 peterson algorithm은 두 개의 프로세스일 때만 적용되는 문제가 있었고, 이를 여러 개의 프로세스로 확장시키려면 for문 등으로 다른 프로세스의 flag를 순회하는 방식을 써야 하는데, 여기서도 race condition이 발생할 수 있다는 문제점이 있었습니다. 그래서 이 문제에서는 peterson algorithm은 쓰면 안 되겠다고 생각하였습니다.

그래서 다른 알고리즘을 찾아봤더니, 빵집을 비유로 고객들에서 번호표를 부여하는 식으로 race condition을 방지하는 bakery algorithm이라는 것을 알게 되었습니다. 이 알고리즘을 참고하여 실제로 `lock`과 `unlock`을 구현해보았을 때, race condition이 발생하지 않고 값이 제대로 출력되었었습니다.

그러나 `NUM_ITERS`와 `NUM_THREADS`가 `10000` 정도로 커지면 프로그램이 아예 멈추거나, `1000` 정도로만 커져도 50번에 한 번 꼴로 프로그램이 작동하지 않는 문제점이 발생하였습니다. 프로그램이 아예 멈춘 것처럼 보여도 아주 오래 기다리면 결과가 나오기도 하나, 확실히 문제가 있다고 느꼈습니다. 원인이 뭘까 생각해보니, 소프트웨어 기반 동기화 알고리즘은 busy waiting을 하면서 CPU 자원을 많이 소모할 수 있어 시스템 전체 성능을 저하시키는 게 문제점이라고 생각하였습니다. 그래서 소프트웨어 기반 동기화 알고리즘 말고 하드웨어 지원 동기화를 쓰면 이 문제를 해결할 수 있을 것 같아, 원자적 연산을 보장하는 하드웨어 명령어 중 하나인 Compare-and-Swap을 이용해서 `lock`과 `unlock`을 구현해보자는 생각을 하였습니다.

Test-and-Set보다 Compare-and-Swap을 채택한 이유는, TAS를 사용한 스핀락은 상태 변수를 락이 걸렸는지 안 걸렸는지만을 나타내는 경우에만 유용하지만 CAS의 경우에는 기대하는 값과 일치하면 새로운 값으로 설정하여 단순히 상태를 전환하는 것 이상의 기능을 제공하므로 더 유연한 동기화 기능을 구현할 수 있다고 생각했기 때문입니다.

---

2. Implement

📌 실제 구현 과정에서 변경하게 되는 코드영역이나 작성한 자료구조 등에 대한 설명을 구체적으로 서술합니다

Light-Weight Process

allocproc()

`proc.c`의 `allocproc()` 함수를 다음과 같이 수정하였습니다.

```
static struct proc*
allocproc(void)
{
...
found:
  p->state = EMBRYO;
```

```
  p->pid = nextpid++;
  p->main = p;
  p->tid = 0;
...
}
```

프로세스를 할당할 때 그 프로세스는 파생된 스레드가 아닌 최초의 스레드임을 알 수 있기 위해 `main`을 자기 자신으로 설정하고, `tid`를 `0`으로 설정하는 과정을 추가하였습니다.

growproc()

`proc.c`의 `growproc()` 함수를 다음과 같이 수정하였습니다.

```
int
growproc(int n)
{
...
  sz = curproc->main->sz;
...
  curproc->main->sz = sz;
...
}
```

스레드들은 `main`을 통해 자원을 공유하므로 기존의 `curproc->sz` 부분을 `curproc->main->sz`로 수정하였습니다.

fork()

`proc.c`의 `fork()` 함수를 다음과 같이 수정하였습니다.

```
int
fork(void)
{
...
  // Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->main->sz)) == 0){
...
  np->sz = curproc->main->sz;
...

  if (np->tid == 0){
    np->parent = curproc;
  }
  else{
    np->parent = curproc->main;
  }
...
}
```

`growproc()`과 마찬가지로 기존의 `curproc->sz` 부분을 `curproc->main->sz`로 수정하고, 메인 스레드가 아닌 스레드가 다른 프로세스의 부모가 되는 것을 막기 위해 `np`의 `tid`가 `0`이 아닌 경우 `np`의 `parent`를 `curproc`의 `main`으로 지정합니다.

exit()

`proc.c`의 `exit()` 함수를 다음과 같이 수정하였습니다.

```
void
exit(void)
{
...
  acquire(&ptable.lock);
```

```
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    // pid는 같지만 curproc이 아 경우
    if(p->pid == curproc->pid && p != curproc){
      kfree(p->kstack);
      p->kstack = 0;
      p->pid = 0;
      p->tid = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
    }
  }
  release(&ptable.lock);
...
}
```

해당 스레드가 속한 모든 스레드가 종료되게 하기 위해 `ptable.proc` 배열을 순회하며 `curproc`과 같은 `pid`를 가진 스레드들을 찾고, 그 스레드들을 종료하는 과정을 추가합니다.

sbrk

`sysproc.c`에 있는 `sys_sbrk()` 함수를 다음과 같이 수정하였습니다.

```
int
sys_sbrk(void)
{
...
  if (myproc()->tid == 0 || !(myproc()->main)){
    addr = myproc()->sz;
  }
  else{
    addr = myproc()->main->sz;
  }
...
}
```

스레드들은 `main`을 통해 자원을 공유하므로 메모리 영역의 사이즈는 항상 `main`의 `sz`를 사용하도록 수정해주었습니다. `myproc()`의 `tid`가 `0`이라 `myproc()`이 메인 스레드이거나, `myproc()`의 `main`이 존재하지 않을 때만 `addr`에 `myproc()`의 `sz`를 할당해주고, 그게 아니라면 `myproc()`이 메인 스레드가 아닌 스레드라는 뜻이므로 `myproc()`의 `main`의 `sz`를 `addr`에 할당해주었습니다.

exec

`exec.c`의 `exec()` 함수를 다음과 같이 수정하였습니다.

```
int
exec(char *path, char **argv)
{
...
  if(curproc->tid > 0){
    curproc->main->tid = curproc->tid;
    curproc->tid = 0;
    curproc->parent = curproc->main->parent;
    curproc->main = 0;
  }

  // Exit all the threads
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
```

```
    if(p->pid == curproc->pid && p->tid > 0){
      kfree(p->kstack);
      p->kstack = 0;
      p->pid = 0;
      p->tid = 0;
      p->parent = 0;
      p->name[0] = 0;
      p->killed = 0;
      p->state = UNUSED;
    }
  }
...
}
```

`curproc()`이 메인 스레드가 아닐 경우, 즉 `curproc()`의 `tid`가 `0`보다 클 경우 메인 스레드의 `parent`를 상속하고 `tid`를 `0`으로 설정하여 스스로 메인 스레드가 되는 과정을 추가하였습니다. 그리고 다른 스레드들을 정리하기 위해 `ptable.proc`을 순회하면서 정리해주었는데, `exec.c`에서 이에 접근하기 위해 `exec.c`의 첫 부분에 아래의 내용들을 추가하였습니다.

```
// exec.c
...
#include "spinlock.h"

struct PTABLE {
  struct spinlock lock;
  struct proc proc[NPROC];
};

extern struct PTABLE ptable;
```

int thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)

새 스레드를 생성하고 시작하는 함수로, 각 인자들은 다음을 의미합니다.

`thread`: 스레드의 id를 지정합니다.

`start_routine`: 스레드가 시작할 함수를 지정하며, 새로 생성된 스레드는 `start_routine`이 가리키는 함수에서 시작하게 됩니다.

`arg`: 스레드의 `start_routine`에 전달하는 인자입니다.

`return`: 스레드가 성공적으로 만들어졌으면 `0`, 에러가 있다면 `-1`을 반환합니다.

`thread_create` 함수의 동작 원리는 아래와 같습니다.

```
int
thread_create(thread_t *thread, void *(*start_routine)(void *), void *arg)
{
...
  // Allocate thread
  if((np = allocproc()) == 0){
    cprintf("thread allocate failed!\n");
    return -1;
  }
  nextpid--;
```

우선, 기존의 프로세스를 생성할 때 `allocproc()` 함수를 썼던 것처럼 `allocproc()` 함수를 써서 새로운 프로세스를 생성하여 이게 스레드처럼 동작하게 합니다. `allocproc()` 함수를 써서 프로세스를 생성했지만 프로세스가 아니기 때문에 `pid`가 증가하지 않도록 `nextpid--;`로 `thread_create`를 호출한 스레드와 `pid`가 같아지게 합니다.

```
// Find main thread
if (curproc->main){
  main_thread = curproc->main;
}
else{
  main_thread = curproc;
}
```

이후에 `main_thread`를 지정해줍니다. 기본적으로 `main_thread`는 `curproc`의 `main`으로 할당하나, `curproc`의 `main`이 존재하지 않을 경우 `main_thread`를 `curproc`으로 지정해주었습니다.

```
// Share page table
np->pgdir = main_thread->pgdir;

// Thread initialization
np->tid = nexttid;
*thread = np->tid;
nexttid++;
np->sz = main_thread->sz;
np->parent = main_thread->parent;
np->main = main_thread;
np->pid = main_thread->pid;
*np->tf = *main_thread->tf;

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

np->cwd = idup(main_thread->cwd);

safestrcpy(np->name, main_thread->name, sizeof(main_thread->name));
```

`np`가 스레드로 동작할 수 있기 위해 `np`를 설정해줍니다. 새 스레드는 `np`가 `main_thread`와 `pgdir`를 공유할 수 있게 설정해주고, 스레드들을 구별할 수 있게 `tid`를 설정해주는데, 구별을 위해 `tid`는 할당해준 뒤 `nexttid++;`로 값을 늘려줍니다. 그 외에 `sz`, `parent`, `main`, `pid`, `tf` 등등을 `main_thread`에 기반하여 설정합니다.

```
pgdir = main_thread->pgdir;
sz = main_thread->sz;
sp = sz;

// Allocate stack, stack은 공유 안 함
// 2개의 PGSIZE 할당. 하나는 user stack, 하나는 guard page
if ((sz = allocuvm(pgdir, sz, sz + 2 * PGSIZE)) == 0)
{
  cprintf("ERROR: thread stack allocate failed!\n");
  np->state = UNUSED;
  release(&ptable.lock);
  return -1;
}

// guard page에는 접근 불가능
clearpteu(pgdir, (char *)(sz - 2 * PGSIZE));
```

스택을 할당하는 과정입니다. 2개의 페이지를 할당하여 하나는 user stack으로, 하나는 guard page로 사용합니다. 이때 guard page에는 접근할 수 없도록 설정합니다. `allocuvm` 함수를 호출하여 새로운 스택 메모리를 할당하는데, 할당 실패 시 스레드를 `UNUSED` 상태로 만든 후 `-1`을 반환합니다.

```
// user stack 설정
ustack[0] = 0xFFFFFFFF;
ustack[1] = (uint)arg;
sp -= 8;

if (copyout(pgdir, sp, ustack, 8) < 0){
  cprintf("ERROR: stack copy failed!\n");
  np->state = UNUSED;
  release(&ptable.lock);
  return -1;
}
```

thread를 실행하기 위해 `ustack`을 설정하고, 새 스레드의 스택 영역에 `copyout` 함수를 통해 복사합니다. `copyout` 함수가 실패하면 스레드를 `UNUSED` 상태로 만든 후 `-1`을 반환합니다.

```
  main_thread->sz = sz;

np->sz = sz;
np->pgdir = pgdir;

np->tf->eax = 0;
np->tf->eip = (uint)start_routine;
np->tf->esp = sp;

*thread = np->tid;

np->state = RUNNABLE;

release(&ptable.lock);

return 0;
```

`main_thread`의 stack에 스레드들이 쓰는 stack을 쌓습니다. 실행할 명령 주소인 `start_routine`을 `tf`의 `eip`에 설정해주고, `esp`에는 `sp`를 저장해줍니다. 이후 유저 프로그램에서 스레드를 관리할 수 있도록 함수 인자인 `thread_t *thread`에 생성된 스레드의 `tid`를 넘겨주고, 스레드의 상태를 `RUNNABLE`로 바꾸어 스케줄러에 의해 스케줄되도록 합니다. 이 모든 과정을 마치면 `thread_create`가 정상적으로 수행되었다는 뜻이므로 `0`을 반환합니다.

void thread_exit(void *retval)

스레드를 종료하고 값을 반환하는 함수입니다. `thread_exit` 함수의 동작 원리는 기존의 `exit()` 함수와 거의 같고, `exit()` 함수와 달라진 점은 다음과 같습니다.

```
// main thread가 wait되었을 수 있음.
if(curproc->main){
  wakeup1(curproc->main);
}
else{
  wakeup1(curproc->parent);
}
```

부모 프로세스 대신 `curproc`의 `main`을 `wakeup`하나, main thread가 `wait`로 인해 대기하는 경우가 있을 수 있으므로 이 경우에는 `curproc`의 `parent`를 `wakeup`합니다.

```
  // retval 저장
  curproc->retval = retval;
```

이후 `curproc`의 `retval`을 함수 인자인 `retval`로 설정해줍니다.

또한 기존의 `exit()` 함수와 다르게 자식 프로세스를 다루는 부분이 생략되었습니다.

int thread_join(thread_t thread, void **retval)

해당 스레드의 종료를 기다리고, 스레드가 `thread_exit`을 통해 반환한 값을 반환하는 함수입니다. `thread_join` 함수의 동작 원리는 기존의 `wait()` 함수와 거의 같고, `wait()` 함수와 달라진 점은 다음과 같습니다.

```
  int havethread;
```

자식 프로세스가 있는지를 찾는 게 아니라 스레드가 있는지를 찾으므로 기존의 `havekids`를 가독성을 위해 `havethread`로 수정하였습니다.

```
for(;;){
    havethread = 0;
    // Scan through table looking for thread
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->tid != thread)
        continue;
      havethread = 1;
      if(p->state == ZOMBIE){
        // Found one.
        *retval = p->retval;

        // Clear thread
        kfree(p->kstack);
        p->kstack = 0;
        p->pid = 0;
        p->tid = 0;
        p->parent = 0;
        p->main = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;

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

`ptable.proc`을 순회하면서 스레드를 찾고, 스레드가 있으면 `havethread`의 값을 `1`로 설정합니다. `thread_exit`되어 `retval`을 저장해둔 채 `ZOMBIE`가 되어 `join`되기를 기다리고 있는 스레드가 있다면 해당 `retval`을 저장해두고 스레드를 정리합니다. 이때 스레드를 정리할 때 다른 스레드에서 `pgdir`을 사용하고 있을 수 있기 때문에 `pgdir`은 정리하지 않습니다.

Locking

int compare_and_swap(volatile int *ptr, int old, int new)

`compare_and_swap` 함수는 x86 아키텍처의 `cmpxchgl` 어셈블리 명령어를 사용해서 원자적 비교 및 교환을 수행합니다. 이 함수의 동작 원리는 아래와 같습니다.

```
int compare_and_swap(volatile int *ptr, int old, int new) {
    int ret;
    __asm__ __volatile__(
        "lock cmpxchgl %2, %1"
        : "=a"(ret), "+m"(*ptr)
        : "r"(new), "a"(old)
        : "memory");
    return ret;
}
```

- `ptr`이 가리키는 메모리 위치의 값이 `old`과 같은지 비교하여, 같다면 `new` 값을 메모리 위치에 저장하고 다르다면 아무 작업도 하지 않고 메모리 위치의 현재 값을 `ret`에 저장합니다.
- `lock cmpxchgl %2, %1` : `new` 값인 `%2`를 `ptr`이 가리키는 메모리 위치인 `%1`의 값과 비교하고, 같으면 `%1`에 `%2` 값을 저장합니다. 결과는 `ret`인 `%0`에 저장됩니다.
- `: "=a"(ret), "+m"(*ptr)` : `"=a"(ret)`는 `ret`가 결과를 저장할 레지스터임을 나타내고, `"+m"(*ptr)`는 메모리 위치의 값을 읽고 수정할 수 있음을 나타냅니다.
- `: "r"(new), "a"(old)` : `"r"(new)`는 `new` 값을 레지스터에 저장하고, `"a"(old)`는 `old` 값을 `EAX` 레지스터에 저장합니다.
- `: "memory"` : 메모리 배리어로, 컴파일러와 프로세서가 메모리 접근 순서를 재정렬하지 않도록 합니다.

void lock(int tid)

critical section에 진입하는 스레드가 하나만 있도록 보장하기 위해 critical section에 진입하기 전 스레드들은 `lock` 함수를 통과해야 합니다. `lock` 함수의 동작 원리는 아래와 같습니다.

```
void lock(int tid)
{
    int backoff = 1;  // 초기 백오프 시간
    while (compare_and_swap(&lock_var, 0, 1) != 0) {
        // busy-wait를 피하기 위해 백오프
        usleep(backoff);
        backoff = backoff * 2;  // 백오프 시간 증가
        if (backoff > 1000) {
            backoff = 1000;  // 최대 백오프 시간 제한
        }
    }
}
```

- `compare_and_swap(&lock_var, 0, 1) != 0` : `lock_var`가 `0`이면 `1`로 설정하고 락을 획득합니다. `lock_var`가 `0`이 아니면 락을 획득하지 못하고 `compare_and_swap` 함수는 `lock_var`의 현재 값을 반환합니다.
- `usleep(backoff)` : 락을 획득하지 못했을 때, `usleep` 함수를 호출하여 백오프 시간 동안 대기합니다. 이를 통해 busy-waiting을 줄이고, CPU 자원을 절약할 수 있습니다.
- `backoff = backoff * 2` : 백오프 시간을 두 배로 증가시킵니다.
- `if (backoff > 1000) { backoff = 1000; }` : 백오프 시간이 너무 길어지지 않도록 최대 값을 설정합니다. 이 경우에는 1ms 이상 기다리지 않도록 하였습니다.

void unlock(int tid)

critical section에서 빠져나온 스레드가 호출하여 다른 스레드가 critical section에 진입할 수 있게 해주는 함수입니다.

```
void unlock(int tid)
{
    lock_var = 0;
}
```

`lock_var`를 `0`으로 설정하여 락을 해제하면 다른 스레드가 `lock_var`를 `1`로 설정하여 락을 획득할 수 있습니다.

---

3. Result

> 📌 컴파일 및 실행 과정과, 해당 명세에서 요구한 부분이 정상적으로 동작하는 실행 결과를 첨부하고, 동작 과정에 대해 설명합니다.

Light-Weight Process

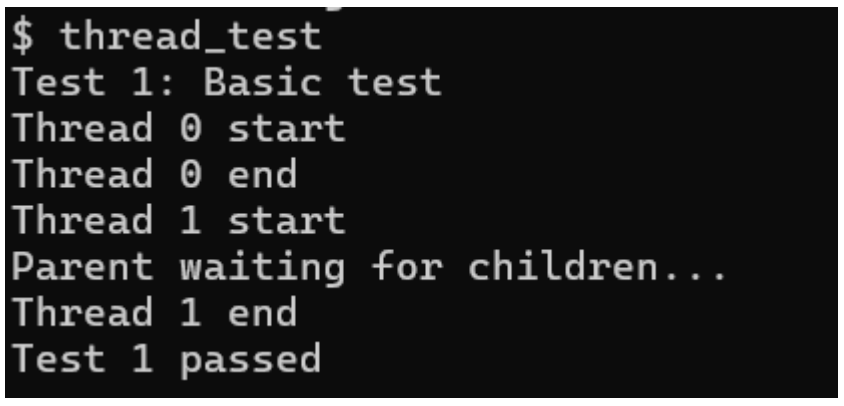
xv6-public 디렉토리 안에서 다음과 같은 명령어를 차례대로 입력합니다.

```
$ make clean
$ make
$ make fs.img
…
$ ./bootxv6.sh
```

```
$ thread_test
$ thread_exec
$ thread_exit
$ thread_kill
```

테스트 결과는 아래와 같습니다.

thread_test - Test 1



스레드 API의 기본적인 기능(create, exit, join)과 스레드 사이에서 메모리가 잘 공유되고 있음을 확인할 수 있었습니다. 스레드0은 곧바로 종료하고, 스레드1은 2초 후에 종료합니다.

thread_test - Test 2

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

스레드 내에서 fork를 했을 때 부모 프로세스는 기존 프로세스의 주소 공간에서 계속 동작하고, 자식 프로세스는 분리된 주소 공간에서 올바르게 동작함을 확인할 수 있었습니다.

thread_test - Test3

```
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!
$
```

여러 스레드들이 각자 메모리를 할당받아도 겹치는 주소에 중복 할당되지 않고, 동시에 할당받고 해제해도 문제가 발생하지 않아 스레드에서 sbrk가 올바르게 동작함을 확인할 수 있었습니다.

thread_exec

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

스레드 중 하나가 exec로 hello_thread 프로그램을 실행하고, 다른 스레드들은 모두 종료되어 exec가 올바르게 동작함을 확인할 수 있었습니다.

thread_exit

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

하나의 스레드에서 exit이 호출되고 Exiting... 출력 후 바로 쉘로 빠져나가 그 프로세스 내의 모든 스레드가 종료되어 exit가 올바르게 동작함을 확인할 수 있었습니다.

thread_kill

```
$ thread_kill
Thread kill test start
Killing process 4
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
This code should be executed 5 times.
Kill test finished
$
```

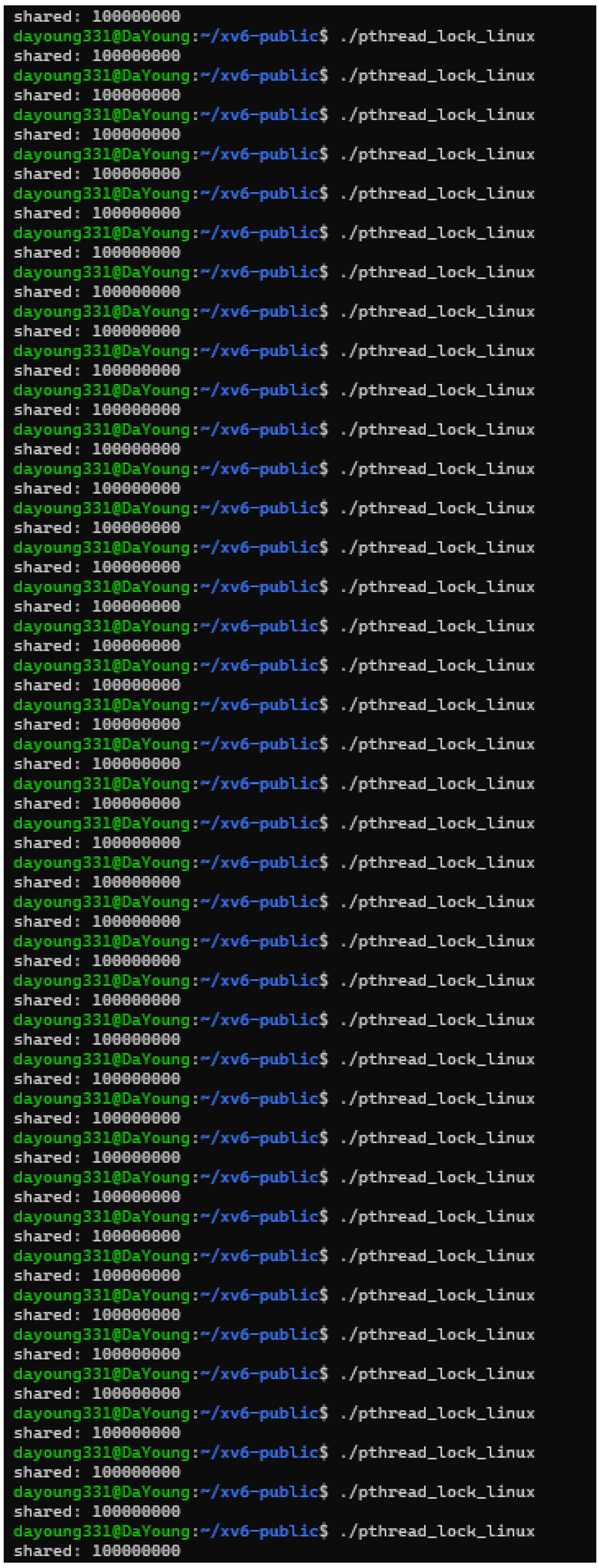
fork를 통해 부모 프로세스와 자식 프로세스가 나누어지고, 부모 프로세스 쪽의 스레드 하나가 자식 프로세스를 kill 합니다. 부모 프로세스의 스레드는 kill에 영향을 받지 않고, 자식 프로세스들의 스레드들은 모두 즉시 올바르게 종료됨을 확인할 수 있었습니다.

Locking

디렉토리 안에서 다음과 같은 명령어를 차례대로 입력합니다.

```
$ gcc -o pthread_lock_linux pthread_lock_linux.c -lpthread
$ ./pthread_lock_linux
```

테스트 결과는 아래와 같습니다.



`NUM_ITERS` 와 `NUM_THREADS` 를 각각 `10000` 으로 정의했을 때, 몇 십 번을 실행해도 오래 기다리지 않고 `shared` 값이 항상 `100000000` 으로 제대로 출력되는 것을 볼 수 있습니다.

```
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
shared: 900000000
```

`NUM_ITERS` 와 `NUM_THREADS` 를 각각 `30000` 까지 늘렸을 때도, 결과가 나오기까지 몇 초 정도씩 걸리긴 하지만 오래 기다리지 않고 `900000000` 이 항상 제대로 나오는 모습을 확인할 수 있었습니다.

---

4. Trouble Shooting

> 📌 과제 수행 과정에서 겪은 문제가 있다면, 해당 문제와 해결 과정을 서술합니다. 해결하지 못했다면 어떤 문제였고 어떻게 해결하려 했는지 서술합니다.

Light-Weight Process

panic: remap

```
Test 3: Sbrk test
Thread 0 start
lapicid 0: panic: remap
 80106fc7 801073f9 80103bf7 80105d8f 8010508d 80106261 80105fa4 0 0 0
```

remap은 이미 할당된 메모리에 재할당할 때 생기는 trap이라는 사실을 생각해보며 원인을 찾아보았습니다. 이 문제의 원인은 `growproc()` 함수에 있었으며, `growproc()` 에서 메모리를 할당할 때 메인 스레드를 통해서 메모리를 할당해주도록 바꾸었더니 이 오류가 해결되었습니다.

trap 14

```
Test 3: Sbrk test
Thread 0 start
pid 3 thread_test: trap 14 err 7 on cpu 0 eip 0xb98 addr 0x14004--kill proc
$ 
```

이 오류의 원인은 `sbrk` 함수에 있었습니다. `sysproc.c` 의 `sys_sbrk` 함수에서 `addr` 를 할당할 때 위의 `remap` 문제와 비슷하게 메인 스레드를 통해서 할당해주도록 바꾸었더니 이 문제가 해결되었습니다.

thread_exit에서 쉘로 빠져나가지 않는 문제

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

위와 같이 Exiting... 이후 $가 뜨지 않아 프로그램이 끝나지 않는 문제가 발생하였었습니다. 이 문제의 원인은 아직 찾지 못하였으나, 위의 remap 문제와 trap 14 문제를 해결하는 과정에서 자연스럽게 같이 해결되었습니다.

Permission denied

```
dayoung331@DaYoung:~/xv6-public$ make
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -Werrc
gcc -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -WerrS
ld -m    elf_i386 -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
objdump -S bootblock.o > bootblock.asm
objcopy -S -O binary -j .text bootblock.o bootblock
./sign.pl bootblock
make: execvp: ./sign.pl: Permission denied
make: *** [Makefile:109: bootblock] Error 127
```

Permission denied라며 make가 되지 않는 문제가 발생하였었는데, 다음과 같은 명령어로 실행 권한을 설정하고 다시 make를 하였더니 정상적으로 작동하였습니다.

```
$ chmod 777 -R xv6-public
```

Locking

대기 시간 문제

```
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
```

`NUM_ITERS`와 `NUM_THREADS`의 값을 `10000`으로 설정했을 때 프로그램을 실행하면 몇 초 정도의 대기 후에 `shared: 100000000`이 뜨는 것을 확인할 수 있었습니다. 소프트웨어 기반 동기화 알고리즘을 사용했을 때보다는 훨씬 양호하지만, 그래도 여전히 busy waiting 때문에 오버헤드가 발생하는 것 같아 지수 백오프 알고리즘을 도입하여 busy waiting을 최소화하는 방법을 사용하였습니다. 이 방법을 도입하자 몇 초 정도의 대기 시간이 1초도 안 되는 대기 시간으로 확실하게 줄어든 모습을 확인할 수 있었습니다.

Segmentation fault

```
dayoung331@DaYoung:~/xv6-public$ ./pthread_lock_linux
Segmentation fault
```

`NUM_ITER`와 `NUM_THREADS`의 값을 `100000`으로 설정했을 때 프로그램을 실행하면 오랜 대기 후에 `Segmentation fault` 오류가 발생하며 프로그램이 실행되지 않았습니다. `Segmentation fault`는 잘못된 메모리 접근으로 인해 발생하는 오류입니다. 매우 큰 숫자를 사용하는 경우, 메모리 할당 실패, 스택 오버플로, 힙 오버플로 등의 원인으로 `Segmentation fault`가 발생할 수 있습니다.

```
$ ulimit -u
```

위와 같은 명령어를 입력하면 현재 사용자 세션에서 생성할 수 있는 최대 스레드 수를 알 수 있는데, 제 컴퓨터에서는 이 값이 31110이라고 뜹니다. 그러므로 스레드 개수를 이보다 많이 설정한다면 `Segmentation fault` 오류가 생길 가능성이 있습니다.

이는 컴퓨터상의 문제이고 제가 짠 코드상의 문제는 아니기 때문에 `NUM_ITERS`와 `NUM_THREADS`의 값을 적당히 너무 크지 않게 설정하는 것으로 타협했습니다.