

# Project 3 - Light Weight Process

## Design

### Thread

- User Stack : LWP( Thread) 拥有自己的用户栈空间。每个 Thread 拥有自己的用户栈空间，并拥有自己的 page table。同时，拥有 file descriptor。
- Thread Id : 每个 Thread 拥有自己的 id。同时，Thread 拥有自己的 (fork 的父进程) Main Thread 的 id，Main Thread 的 id 为 0。
- Main Thread : 每个 Thread 拥有 Main Thread，每个 Thread 拥有自己的 Thread 的 id。每个 Thread 拥有 Main Thread 的 id，同时拥有自己的 id。每个 Thread 拥有自己的 Thread 的 id，同时拥有自己的 Thread 的 id。
- Exec & Exit : Thread 拥有 exec 的函数，同时拥有 exit 的函数。同时，每个 Thread (Main Thread) 拥有 exec 的函数，同时拥有 exit 的函数。同时，每个 Thread 拥有自己的 Thread 的 id (exec 的函数)。

### System Call

- Thread 拥有自己的系统调用。Thread 拥有自己的系统调用，同时拥有自己的系统调用。同时，每个 Thread 拥有自己的系统调用。

## Implement

### Thread

- 每个 Thread 拥有自己的 thread.h 的函数。
- thread\_t : Thread 的 id。
- 每个 Thread 拥有 Thread 的 id，Main Thread 的 id，Thread 的 id 为 0。

```
// proc.h

// Per-process state
struct proc {
    ...
    int tid; // Thread ID
    struct proc *main; // Main thread
    void* retval; // Return value
};
```

### Thread Creation

return	name	arguments	description
int	_thread_create	thread_t *thread, void *(*start_routine)(void *), void *arg	创建 Thread 的函数。

- `thread` 返回 Thread 的 id 值。
  - `start_routine` Thread 的入口函数。
  - `arg` `start_routine` 的参数。
1. `myproc` 函数在子进程中运行。
  2. `allocproc` 函数在子进程中分配 Thread 的堆空间。(通常由父进程调用 Thread 的堆空间。)
  3. 返回 Thread 的 `pid`, `tid`, `main`, `pgdir`, `parent`, `tf` 值。
  4. 返回 Thread 的堆空间地址，并设置 Thread 的 `sz` 值。
  5. 返回 Thread 的 `arg` 值，`tf` 的 `eip` 指向 `start_routine` 函数，返回 `esp` 值。
  6. 返回 Thread 的 `sz` 值，并设置 Thread 的 `sz` 值。
  7. 返回子进程的堆空间。
  8. Thread 的堆空间地址，返回 Thread 的 id 值 `thread`。
  9. 返回 Thread 的 `state` 值 `RUNNABLE`，并设置子进程的堆空间。

Thread Termination

return	name	arguments	description
void	<code>_thread_exit</code>	<code>void *retval</code>	返回 Thread 的堆空间。

- `retval` Thread 的 `start_routine` 的返回值。
1. `myproc` 函数在子进程中运行。
  2. 返回 Thread 的 Main Thread 的堆空间，并设置 Thread 的堆空间。
  3. Thread 的堆空间地址。
  4. 返回 Thread 的堆空间地址 `initproc` 函数。
  5. 返回 Thread 的 Main Thread 的堆空间，并设置 Main Thread 的堆空间。
  6. 返回 Thread 的 `retval` 值，`state` 值 `ZOMBIE`。
  7. 返回子进程的堆空间。

Thread Waiting

return	name	arguments	description
int	<code>_thread_join</code>	<code>thread_t thread, void **retval</code>	返回 Thread 的堆空间。

- `thread` 返回 Thread 的 id 值。
  - `retval` 返回 Thread 的堆空间。
1. `myproc` 函数在子进程中运行。
  2. 返回 Thread 的堆空间。
  3. 返回 Thread 的堆空间地址，并设置 Thread 的 `SLEEPING` 值，并设置 Thread 的堆空间。
  4. 返回 Thread 的堆空间地址，`retval` 返回 Thread 的堆空间，并设置 Thread 的堆空间。

proc.c

initproc

- `thread.c` 函数在子进程中运行 `initproc` 函数，并设置 `extern` 值。

## wakeup1

- `thread.c`에 `initproc`를 `extern`으로 선언.

## allocproc

- `thread.c`에 `initproc`를 `extern`으로 선언.
- `Thread`에 `proc`를 `struct`로 선언.

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

    // EDITED : Thread
    p->tid = 0;
    p->main = p;
    ...
}
```

## growproc

- `Thread`에 `proc`를 `sz`로 선언, `critical section`을 처리.

```
int growproc(int n)
{
    ...
    acquire(&ptable.lock);
    ...
    release(&ptable.lock);
    ...
}
```

## exit

- `Thread(Main Thread)`에 `exit`를 호출, `Thread`를 종료.

```
void exit(void){
    ...
    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->pid == curproc->pid)
        {
            p->state = ZOMBIE;
        }
    }
}
```

```

    p->cwd = 0;

    for(int fd = 0; fd < NOFILE; fd++){
        if(p->ofile[fd]){
            p->ofile[fd] = 0;
        }
    }
}
}
...
}

```

## wait

- **ZOMBIE** 프로세스(Thread)가 자식 프로세스를 생성한다.

```

int wait(void){
    ...
    for (struct proc *p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    {
        if (p->state == ZOMBIE && p->parent == curproc)
        {
            p->state = UNUSED;
            p->pid = 0;
            p->parent = 0;
            p->name[0] = 0;
            p->killed = 0;
            p->main = 0;
            p->tid = 0;
            p->retval = 0;
            for(; p < &ptable.proc[NPROC]; p++)
            {
                if (p->parent != curproc)
                    continue;
                if(p->state == ZOMBIE){
                    kfree(p->kstack);
                    if (p->tid == 0)
                    {
                        freevm(p->pgdir);
                    }
                    p->kstack = 0;
                    p->pid = 0;
                    p->parent = 0;
                    p->name[0] = 0;
                    p->killed = 0;
                    p->state = UNUSED;
                }
            }
        }
        release(&ptable.lock);
        return pid;
    }
}

```

```
}  
...  
}
```

## Result

### thread\_test

#### Test 1

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

- Thread 0 1 0 0 0 0, Thread 0 0 0 0 0 1 0 0 Thread 1 0 0 0 0.

#### Test 2

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

- Thread의 `fork`는 부모 프로세스와 자식 프로세스를 생성하며, 부모 프로세스는 `(int status)`를 반환하여 상태를 반환.

### Test 3

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

- `Thread`의 `exec`는 새로운 프로세스를 생성하며, 부모 프로세스는 `exec`를 호출한 후 종료.

### `thread_exec`

```
$ thread_exec
Thread exec test start at 21
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 start
Thread 4 start
Executing...
Hello, thread!
```

- Thread의 `exec`는 새로운 프로세스를 생성하며, 부모 프로세스는 `exec`를 호출한 후 종료.

### `thread_exit`

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

- Thread의 `exit`는 프로세스를 종료하며, 부모 Thread는 `exit`를 호출한 후 종료.

### `thread_kill`

```
$ thread_kill
Thread kill test start
Killing process 34
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.
This code should be executed 5 times.
kill test finished
```

- Thread `kill` 0000 00 00 0000 0, 00 Thread 000000 00000.

## Trouble Shooting

### `_thread_create`

- Thread 000000 0000 0000 000000, 00 Thread 0000 0 0000 000000 0000 0000 `allocproc`0000 0000. 00 0 `proc` 0000 0000 Thread 0000 0000 0000 0000 000000 0000. 0000 `allocproc` 0000 0000 0000 `nextpid` 00 00000 0000, 00000 00 `pid` 00000.
- Thread Stack 0000 0, 2 x PGSIZE 000000 00. 00 Guard Page 00000 00000. 00 `clearpteu` 0000 00 Guard Page 000000.

### `_thread_exit`

- 00 Main Thread 0 0000 00000, 00 Thread 00000 00 0000 0000 00. 00 00 `kill` 0000 0000 0000 00 Thread 0 `killed` 1 00000, trap 00000 `exit` 0000 000000 00.

### `_thread_join`

- 00 0000 `wait` 0000 00000.

### `ofile & cwd`

- Thread 000000 0000 00 00000000 000000, `ofile` `cwd` 00000.
- 0000 0000 Thread `exit`0000 0000 00 Thread 000000, `exec` 0000 0000 0 0000 Thread 00 0000, `ofile` `cwd` 0000 0 `filedup`00 `idup` 00000 0000. 0 00 000000 00000, 00000 `ref` 0000 000000 0000. 00 Thread 0000 0, `fileclose` 0000 `iput` 0000 0 0000 00000.

## Locking

- 000000 c inline assembly 0000 atomic swap 0000 0 00.
- x86 0000 `xchg` 00000 0000 atomic swap 0000 0 00, arm64 0000 `ldxr` `stxr` 00000 0000 atomic swap 00 00 00.
- atomic swap 0000 `compare_and_swap` 0000 00000, 0 0000 00 spinlock 0000 0 00.