

Wiki - 제출용

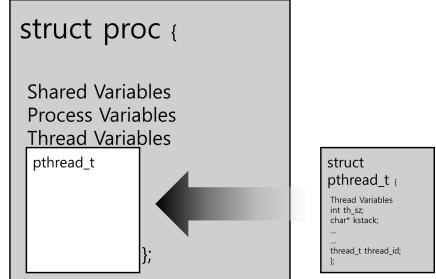
학과 : 컴퓨터소프트웨어학부

학번 : 20*****40

작성자 : 김휘수

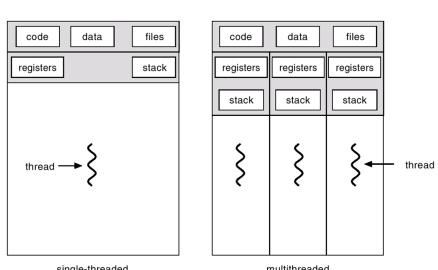
1. Design

1-1. Process - Thread : 옷 입히기



- `struct proc` 을 기준으로 xv6가 구현되어 있기 때문에, 이 부분에 대해서는 건드리지 않고 각각의 thread에 대한 새로운 구조체인 `struct pthead_t` 를 만들었다.
- process는 `ptable` 에서, thread는 `pthead_table` 에서 관리한다.
- 각각의 thread가 스케줄러 안에서 선택이 되면 process를 입고, 스케줄러를 나가서 작업을 하고, 돌아와서는 process를 벗어놓는 형태의 디자인이다.
- 비유적으로 집에서 옷을 입고, 밖에서 일을 하고, 집에 돌아와서 옷을 벗어 놓는 방식이다.
- 앞으로의 설명에서도 옷을 입다(process를 입다), 옷을 벗다(process를 벗다) 등의 표현이 자주 나오게 될 것이다.

1-1-1. 자료구조를 나눈 이유



[Thread의 구조]

- Thread는 Process와 code, data, file 등의 메모리 영역을 공유하는 부분/register, stack 등 공유하지 않는 부분으로 나누어져 있다.
- 각각의 자료를 편하게 구분하기 위해서 공유하는 부분과 공유하지 않는 부분을 나누었다.
- 공유하는 자료는 모두 `proc` 에서 관리한다.
- 공유하지 않는 자료는 모두 `pthead_t` 에서 관리한다.
- xv6의 특성상 Thread와 Process의 구분이 힘들지만, 실질적으로 구분하는 것이 프로그래밍에 좋다고 판단했다.

1-1-2. Thread Scheduling

두 가지의 Scheduling 방식을 생각할 수 있었다.

두 가지 방법 중에서 두 번째 방법이 구현도 용이하고, overhead도 적게 발생하기 때문에 더 좋은 방법이라고 생각하여, 이 방법을 이용해서 구현했다.

1. Thread 단위의 scheduling만을 고려하여, Thread가 생성된 순서를 기준으로 Scheduling한다.
2. Process 단위의 scheduling을 하고, Process 내부에서 Thread를 Scheduling한다.

ptable						
Process 1	Process 2	Process 3	Process 4 "Scheduled"	Process 5	Process 6	Process 7



pthead_table						
Process 1 Pthread 1	Process 3 Pthread 1	Process 1 Pthread 2	Process 1 Pthread 3	Process 4 Pthread 1 "Scheduled"	Process 2 Pthread 1	Process 4 Pthread 2 "(will be) Scheduled"

1. 우선적으로 Process를 RR방식으로 선정한다.
2. 해당 Process의 Thread를 `pthead_table` 에서 선택한다.
3. 선택된 Thread는 Process를 입고 자신이 하던 작업을 진행한다.
4. Scheduling이 종료되면, Process를 벗고, 정리한다.
5. 하나의 Process에서 모든 Thread를 확인했으면, 해당 Process의 Scheduling을 종료한다.
6. 위의 모든 과정을 반복한다.

1-1-3. Process - Thread의 data분리

위에서 말했듯이 Process와 Thread는 서로 공유하는 변수와, 분리되어있는 변수가 존재한다.

Process와 Thread의 구조를 나누고, 서로를 인식하기 위해서 추가적인 변수들이 많지만, 제일 중요한 것들만 나타내면 아래와 같다.

실질적인 구현에 대해서는 Implementation에서 살펴본다.

공유하는 Data

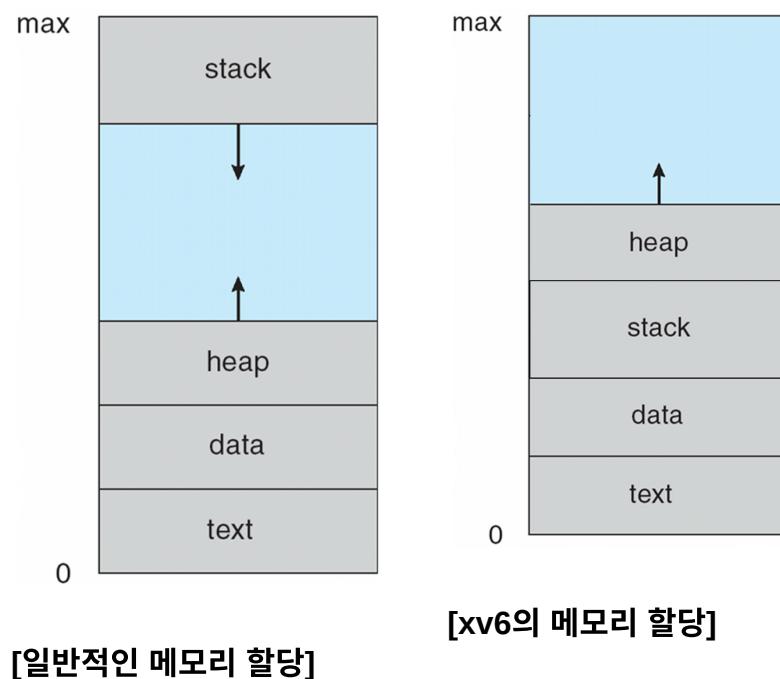
- Page Table
- Directories & Files
- Virtual Memory

공유하지 않는 Data

- State
- Trap Frame
- Context

1-2. Memory Design

xv6는 일반적인 OS와 다르게 메모리 공간을 할당한다.



- 일반적인 OS는 메모리 공간을 code, data, heap, stack으로 나눈다.
- 일반적인 OS는 heap과 stack이 서로를 바라보게 자라나고, 가변적인 크기를 갖는다.
- xv6는 메모리 공간을 code, data, fixed stack, heap으로 나눈다.
- xv6는 stack위에 heap이 존재하기 때문에, stack이 더 이상 자라날 수 없고, 크기가 고정되어있다.
- xv6의 heap은 메모리 공간의 최상단을 바라보며 자라난다.

그림과 같은 특별하게 생긴 구조를 갖고, 각 Thread는 stack만을 따로 갖기 때문에 각각의 stack을 하나의 Thread라고 생각을 하고, heap은 그 위에 존재해서 서로가 공유할 수 있도록 만들면 우리가 원하는 Thread의 구현이 가능하다고 생각했다.

1-2-1. Memory Recycle

위의 그림과 같은 xv6의 메모리 할당에 대해서, 어떤 Thread가 도중에 종료되어 정리가 되면, 해당 메모리공간이 비는 상황이 발생한다.

이렇게 비는 부분을 재사용하기 위해서는 해당 구역이 어디인지를 저장해 두었다가 새로운 메모리 할당 요청이 들어왔을 때, 비는 공간에 할당할 수 있는지를 확인하는 방법을 이용한다.

이를 이용해서 쓸데없이 낭비되는 공간을 최소화 할 수 있다. ⇒ Circular Q 구조를 이용해서 비는 공간을 저장해둔다.

Troubles

해결하지 못한 문제점으로 뒤의 Trouble Shooting에서 자세히 설명하겠지만, Heap에 대한 할당은 해결하지 못했다.

위와 같은 방법으로 해결할 수 있었던 것은 오로지 stack에 대한 부분 뿐이었다.

1-3. Pmanager

다양한 명령어를 통해 프로세스를 제어할 수 있는 프로세스이다.

구조적으로 Shell과 다른 것이 없고, 입력으로 들어오는 문자열 파싱만 잘해주어 인자로 전달하면 된다.

1-3-1. 부가적인 system call

- `int exec2(char* path, char** argv, int stacksize);`
 - 스택용 페이지를 여러 개 할당받도록 하는 system call
 - xv6에서는 기본적으로 2개의 stackpage를 할당받아서, 하나는 data를 저장하기 위한 용도로, 다른 하나는 guard page로 사용하는 데 이러한 stackpage를 여러 개 만든다는 의미이다.
 - `exec` system call과 다른 부분이 거의 없다.
- `int setmemorylimit(int pid, int limit);`
 - 특정 프로세스의 memory limit정하여, 그 이상의 메모리를 할당받지 못하도록 만드는 system call이다.
 - 메모리를 할당할 때마다 정해진 한도 이상이 필요한지 확인하여, 한도가 넘게되면 메모리를 할당받아서는 안된다.
 - Process안에 추가적인 변수를 두어서, 메모리 할당할 때 마다 확인하면 된다.

1-4. System calls

기본적으로 Process를 위해서 구현되어 있는 system call이나 함수들을 Thread를 위해서 추가하거나 약간의 수정이 필요하다.

1-4-1. thread_create

Thread를 생성하기 위해서 필요한 API

Thread를 생성하고, Thread 함수를 실행하도록 하는 역할을 한다.

구현이 `fork` 와 `exec` 을 합친 것과 비슷하다. ⇒ `fork` 를 통해서 Thread를 새로 생성 & `exec` 을 통해서 새로운 start point 설정

생성된 Thread는 생성한 Thread에 의해서 정리되어야 한다.

1-4-2. thread_exit

Thread를 종료하고 반환값을 돌려주기 위해서 필요한 API

Thread를 종료하는 명령이기 때문에 `exit` 과 비슷하다.

만약 `thread_exit` 을 호출한 thread가 Main Thread라면 `kill` 을 통해서 process를 죽이도록 한다. (일반적이지 않은 경우)

1-4-3. thread_join

종료된 Thread의 메모리를 정리하고, 저장한 반환값을 받기 위해서 필요한 API

Thread를 정리하는 명령이기 때문에 `wait` 과 비슷하다.

- `thread_join() & thread_create()` 의 주체

초기에는 `thread_join` 은 오직 Main Thread만 실행가능한 구현을 생각했었다.

왜냐하면, `thread_join()` 을 악의적인 사용으로, 누군가가 deadlock을 발생시킬 수 있다고 생각했기 때문이다.

그러나 이는 올바르지 않은 구현이었다.

임의의 thread가 `thread_create()` 를 호출할 수 있는 상황이라면, 사용자가 그 thread를 생성하고, return값을 받기 위해서는 `thread_join()` 을 사용해야 한다.

그러나 초기에 생각했던 구현은, Main Thread가 아닌 Sub Thread가 thread를 생성했을 때 Main Thread에 종속되는 방식으로 구현했었는데, 이렇게 하면 Main Thread에서 의도치 않은 thread를 정리하는 경우가 발생하기 때문이다.

그래서 새롭게 이 thread를 join해주기 위한 `joiner_thread` 를 따로 관리해주었다.

1-4-4. fork

Process를 복사하기 위해서 필요한 system call

현재 실행 중인 Process의 정보를 그대로 복사한 Virtual Memory를 생성한다.

이 때, `fork` 를 실행한 주체가 누군가에 따라서 나올 수 있는 경우의 수가 두 가지 존재한다.

1. Main Thread가 fork를 실행한 경우

Main Thread는 곧 Process와 같기 때문에, Thread를 포함한 Process의 모든 정보를 그대로 복사해준다.

그 결과 $P_1, T_1 = \{t_1, t_2, t_3, \dots\}$ 가 복사된 $P'_1, T'_1 = \{t'_1, t'_2, t'_3, \dots\}$ 가 생긴다.

2. 그 외의 Thread가 fork를 실행한 경우

이 경우에는 실행 중인 Thread와 Thread가 입고 있는 Process의 정보만을 복사해준다.

그 결과 현재 실행 중인 Thread T_1 과 Process P_1 을 복사한 새로운 Process P'_1 과 Thread T'_1 이 생성되고 T'_1 이 P'_1 의 새로운 Main Thread가 된다.

⇒ 두 경우 process를 정리하는 주체는 fork를 실행한 thread가 된다.

1-4-5. exec

Process를 새롭게 만드는 데에 필요한 system call

현재 실행 중인 Process의 정보를 모두 지우고, 새로운 Process 정보를 올려 완전히 새로운 Virtual Memory를 생성한다.

이 때, 어떤 Thread가 `exec`을 호출했느냐에 상관없이, 성공적인 실행에 대해서 다른 모든 Thread를 지우고 호출한 Thread가 새로운 Process의 Main Thread가 된다.

만약에 다른 Thread에 Child Process가 존재하는 경우, 그 Process들을 모두 남아있는 Main Thread의 Child로 만들어 준다.

1-4-6. sbrk

Process의 Heap을 할당하는 system call

여러 Thread의 호출에 대해서 알맞게 공간을 할당해주기 위해서 내부에 Lock 구현을 추가해야하며, 할당된 공간은 모든 Thread가 공유해야 한다.

기본적으로 `sbrk`는 `growproc` 함수와 관련이 있기 때문에, `growproc` 부분만 수정하면 된다.

1-4-7. kill

Process를 강제종료하는 system call

어떤 Thread가 `kill`을 호출했느냐에 상관없이, 모든 Thread를 종료하고 메모리를 정리해야 한다.

1-4-8. sleep

Process를 일시적으로 중지하는 system call

`sleep`을 호출한 Thread만 중지되어야 하며, 같은 Process 내의 다른 Thread는 정상적으로 동작해야 한다.

2. Implement

2-1. Process & Thread

2-1-1. Process Structure

```
1 struct proc {
2     // Shared Variable
3     pde_t* pgdir;           // Page table
4     char name[55];          // Process name (debugging)
5     struct proc* parent_proc; // Parent process who calls fork()
6     struct pthread_t*
7         parent_pthread;      // Parent pthread who calls fork()
8     struct inode* cwd;       // Current directory
9     struct file* ofile[NFILE]; // Open files
10    uint memlimit;          // Memory limitation (default 0)
11    uint thread_assign;     // Thread ID to Assign (start to 1)
12    uint stacksz;           // # of Stack Page (default 1)
13
14    // process Variable
15    uint freed_sz[NTHRD];   // Queue for freed Stack. LOL
16    int front;              // front of Q
17    int back;               // back of Q
18    int freed;               // size of Q
19    int pid;                // Process ID
20    uint sz;                 // Size of Thread memory (bytes)
21    int main_thread_idx;     // pthread table index of main thread
22
23    // Thread Variable
24    uint th_sz;              // Size of Thread memory (bytes)
25    char* kstack;             // Bottom of kernel stack for this process
26    enum proctype state;     // Process state
27    void* chan;               // If non-zero, sleeping on chan
28    struct trapframe* tf;     // Trap frame for current syscall
29    struct context* context;  // swtch() here to run process
30    int killed;               // If non-zero, have been killed
31    thread_t thread_id;       // thread ID
32    struct pthread_t*
33        joiner_pthread;      // Parent pthread who calls thread_create()
34    int thread_idx;           // Thread table index
35 };
```

- xv6에서의 Process를 나타내는 `proc` 자료구조이다.
- 다양한 부분이 추가되고 분류에 따라서 위치를 변경하였기 때문에, 메모리 측면에서 비효율적일 수 있다.
- `parent_proc, parent_pthread`
해당 Process를 생성한, 부모의 역할을 하는 `pthread` 와 `proc` 을 나타내는 포인터이다.
Process, Thread의 정리를 할 때에 이 변수들을 이용한다.
- `memlimit`
Process가 가질 수 있는 최대 메모리의 한계를 나타낸다.
0인 경우에는 제한이 없음을 나타낸다. 그렇지 않은 경우, 메모리가 이 이상 할당될 수 없다.
- `thread_assign`
생성되는 Thread의 id를 만들어주기 위한 변수이다.
Process안에서 Thread의 id를 만들어 주기 때문에, 다른 Process의 Thread와 값이 같을 수 있다.
- `stacksz`
Process에 할당되는 Stack Page의 수를 나타낸다. Thread도 동일한 수의 Stack Page를 갖는다.
- `freed_sz[], front, back, freed`
Thread가 Free되면 VM에 빈 공간이 생겼음을 저장하는 자료구조이다.
Circular Q로 구현했으며, 자료구조의 유지를 위한 `front, back, freed` 변수가 존재한다.
- `main_thread_idx`
Process의 주인인 Main Thread가 저장되어 있는 Table의 index를 나타낸다.

2-1-2. Thread Structure

```
1 struct pthread_t {
2     uint th_sz;              // Size of Thread memory (bytes)
3     int pid;                // process ID owned thread
4     char* kstack;             // Bottom of kernel stack for this process
5     enum proctype state;     // Process state
6     void* chan;               // If non-zero, sleeping on chan
7     struct trapframe* tf;     // Trap frame for current syscall
8     struct context* context;  // swtch() here to run process
9     int ptable_idx;           // Proc table index
10    thread_t thread_id;       // thread ID
11    struct pthread_t*
12        joiner_pthread;      // Parent pthread who calls thread_create()
13    void* retval;              // Save retval to return in thread_join
14 };
```

- xv6에서의 Thread를 나타내는 `pthread_t` 자료구조이다.
- `th_sz`
Thread Stack의 top을 나타내는 변수이다.
- `pid`
어떤 Process의 Thread인지를 나타내는 변수이다.
- `ptable_idx`
`pid`에 대응되는 Process의 `ptable` 에서의 위치를 나타내는 변수이다.
- `thread_id`
Process 내의 각각 Thread의 고유한 id이다.
- `joiner_pthread`
Thread를 만들어 준 Thread로, 어떤 Thread가 `thread_join()` 을 통해서 return값을 받아줄 것인지를 나타낸다.
- `retval`
Thread 종료 후 return 값을 저장할 변수이다

2-1-3. Tables

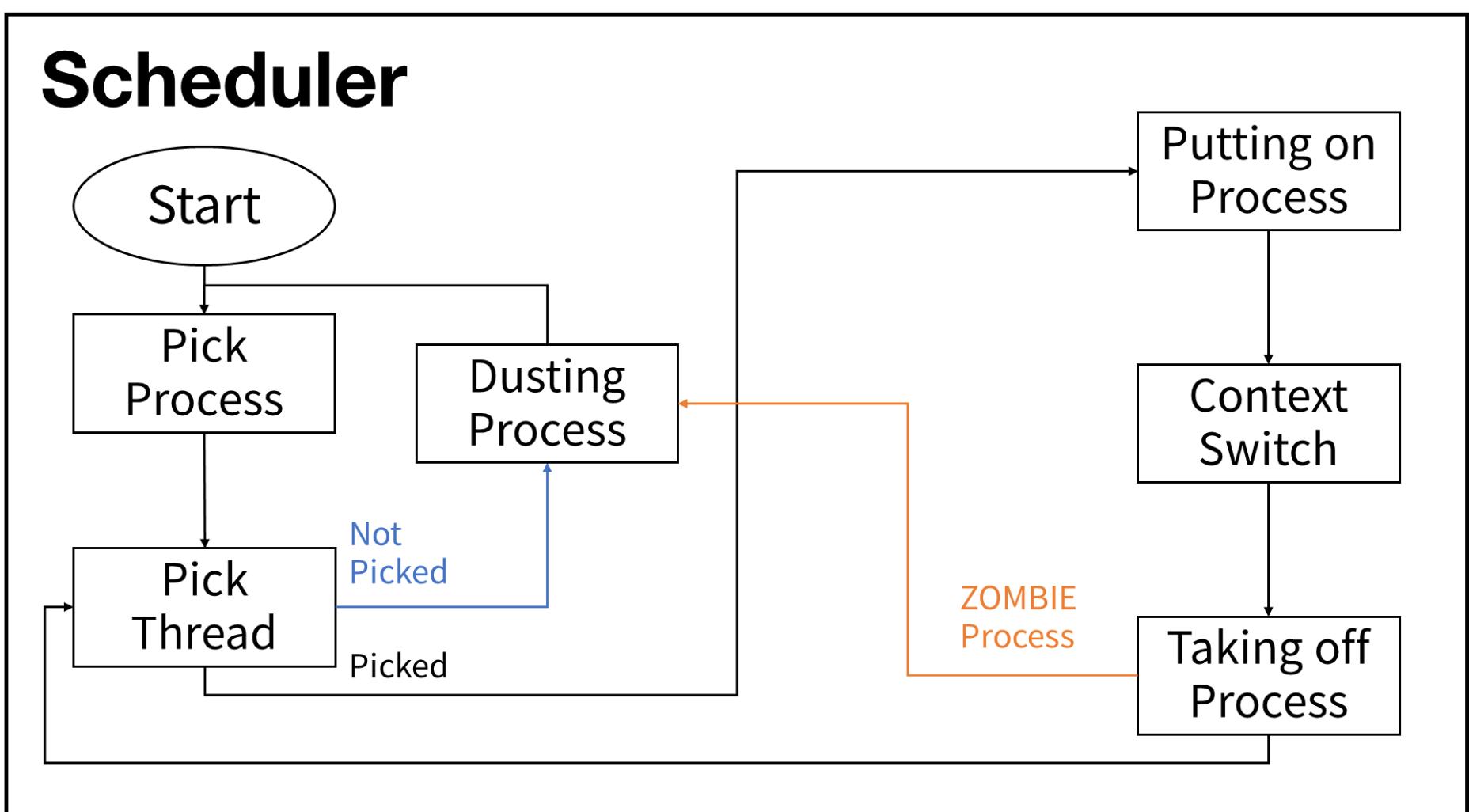
```
1 // Process is A cloth for Threads.
2 struct {
3     struct spinlock lock;
4     struct proc proc[NPROC];
5 } ptable;
6
7 // pthread table that contains Actual information of Threads.
8 struct {
9     struct spinlock lock;
10    struct pthread_t pthread[NTHR];
11 } pthread_table;
```

각각의 생성된 Process와 Thread를 구분하여 Table에 저장한다.

생성된 Process는 `proc` 자료구조에 담겨, `ptable`에 저장된다.

생성된 Thread는 `pthread_t` 자료구조에 담겨, `pthread_table`에 저장된다.

2-2. Scheduler()



위의 Scheduler 디자인과 실제 작성 코드를 보면서 진행한다. (코드의 line number가 실제 line number가 아님에 유의)

2-2-1. Pick Process

Scheduler가 Scheduling할 Process를 선택한다.

```
1 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
2     if (p->state != RUNNABLE) {
3         continue;
4     }
```

2-2-2. Pick Thread

선택된 Process에 대해서 Scheduling할 Thread를 선택한다.

- Thread가 선택되면 Dressing Process로 이동한다.
- Thread가 선택되지 않으면 모든 Thread를 Scheduling한 것이므로 Dusting Process로 이동한다.

2-2-3. Putting on Process

Thread가 Process를 입는 과정이다.

Thread의 정보를 Process로 복사하여, Process가 Thread인 것처럼 동작한다.

mvThreadToProc() 구현

```
1 void mvThreadToProc(int thread_table_idx, struct proc* p) {
2     p->th_sz = pthread_table.pthread[thread_table_idx].th_sz;
3     p->pid = pthread_table.pthread[thread_table_idx].pid;
4     p->kstack = pthread_table.pthread[thread_table_idx].kstack;
5     p->state = pthread_table.pthread[thread_table_idx].state;
6     p->chan = pthread_table.pthread[thread_table_idx].chan;
7     p->tf = (pthread_table.pthread[thread_table_idx].tf);
8     p->context = (pthread_table.pthread[thread_table_idx].context);
9     p->thread_id = pthread_table.pthread[thread_table_idx].thread_id;
10    p->joiner_pthread = pthread_table.pthread[thread_table_idx].joiner_pthread;
11    p->thread_idx = thread_table_idx;
12 }
```

Thread의 정보를 Process에 옮겨주는 작업이다.

Thread만이 가지고 있는 고유한 정보이기 때문에, 거의 대부분을 복사한다.

2-2-4. Context Switch

```
1 // Context Switch
2 // Switch to chosen process. It is the process's job
3 // to release ptable.lock and then reacquire it
4 // before jumping back to us.
5 c->proc = p;
6 switchuvmp(p);
7 p->state = RUNNING;
8 swtch(&(c->scheduler), p->context);
```

- process의 State를 RUNNING으로 바꾸어준다.

- swtch()를 통해서 Context Switch 실행.

2-2-5. Taking off Process

```
1 mvProcToThread(p - ptable.proc, th);
2 if (p->state == ZOMBIE) {
3     break;
4 }
```

Thread가 Process를 벗는 과정이다.

Process의 정보를 Thread로 복사하여, 지금까지 작업한 정보를 Thread에 기록한다.

만약 돌아온 Process의 상태가 ZOMBIE라면, 더 이상 process가 동작해서는 안되므로, Dusting Process로 이동한다.

mvProcToThread() 구현

```
1 void mvProcToThread(int ptable_idx, struct pthread_t* th) {
2     th->th_sz = ptable.proc[ptable_idx].th_sz;
3     th->pid = ptable.proc[ptable_idx].pid;
4     th->kstack = ptable.proc[ptable_idx].kstack;
5     th->state = ptable.proc[ptable_idx].state;
6     th->chan = ptable.proc[ptable_idx].chan;
7     th->tf = (ptable.proc[ptable_idx].tf);
8     th->context = (ptable.proc[ptable_idx].context);
9     th->thread_id = ptable.proc[ptable_idx].thread_id;
10    th->joiner_pthread = ptable.proc[ptable_idx].joiner_pthread;
11    th->ptable_idx = ptable_idx;
12 }
```

Process의 정보를 Thread에 옮겨주는 작업이다.

Thread만이 가지고 있는 고유한 정보만을 복사한다.

2-2-6. Dusting Process

```
1 c->proc = 0;
2 clearProc(p);
```

```
1 void clearProc(struct proc* p) {
2     if (p->state != ZOMBIE)
3         p->state = RUNNABLE;
4     p->chan = 0;
5 }
```

- Thread가 입고 나갔던 옷인 Process의 먼지를 털어주는 작업이다.

- 대부분 Thread의 정보를 복사할 것이기 때문에, Scheduling을 위해서 필요한 state 와 의도치않은 wakeup을 방지하기 위해서 chan 만을 초기화 해준다.

2-3. allocproc()

```

1 static struct proc*
2 allocproc(void)
3 {
4     struct proc* p;
5     struct pthread_t* th;
6     char* sp;
7
8     acquire(&ptable.lock);
9     // Allocate New Process.
10    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
11        if (p->state == UNUSED)
12            goto find_thread;
13
14    release(&ptable.lock);
15    return p;
16
17 find_thread:
18     // Allocate Thread.
19    if ((th = allocthread()) != null)
20        goto found;
21
22    release(&ptable.lock);
23    return p;
24
25 found:
26
27    p->state = EMBRYO;
28    p->pid = nextpid++;
29    th->pid = p->pid;
30
31    // assign process kernel stack with thread kernel stack.
32    if ((p->kstack = th->kstack) == 0) {
33        // Impossible Condition
34        p->state = UNUSED;
35        release(&ptable.lock);
36        return p;
37    }
38
39    // set process information as default.
40    sp = p->kstack + KSTACKSIZE;
41
42    // Leave room for trap frame.
43    sp -= sizeof * p->tf;
44    p->tf = (struct trapframe*)sp;
45
46    // Set up new context to start executing at forkret,
47    // which returns to trapret.
48    sp -= 4;
49    *(uint*)sp = (uint)trapret;
50
51    sp -= sizeof * p->context;
52    p->context = (struct context*)sp;
53    memset(p->context, 0, sizeof * p->context);
54    p->context->eip = (uint)forkret;
55    th->context->eip = p->context->eip;
56    p->sz = 0;
57    p->memlimit = 0;
58    p->thread_assign = 1;
59
60    // Write A name of Thread to figure out
61    // who wear this process.
62    p->thread_idx = th - pthread_table(pthread);
63    // The thread who wear process first is
64    // owner of process.
65    p->thread_id = MAINTH;
66    p->main_thread_idx = p->thread_idx;
67    p->stacksz = 1;
68    p->front = 0;
69    p->back = 0;
70    p->freed = 0;
71    p->parent_proc = 0;
72    p->parent_pthread = 0;
73    p->joiner_pthread = th;
74    memset(p->freed_sz, 0, sizeof(p->freed_sz));
75    release(&ptable.lock);
76    return p;
77 }

```

- 수정된 `allocproc()` 함수이다.
 - line 19, `allocthread()` 를 해준다.
 - line 32, `p->kstack` 을 새로 할당하지 않고, `allocthread()` 로 생성한 `pthread_t` 의 `kstack` 으로 대체한다.
- ⇒ Process와 같이 생성된 Thread가 Main Thread이기 때문이다.
- line 56:73까지 `proc` 변수들을 초기화한다.

allocthread()

```

1 static struct pthread_t*
2 allocthread(void) {
3     struct pthread_t* th;
4     char* sp;
5
6     for (th = pthread_table(pthread); th < &pthread_table(pthread)[NTHRD]; th++)
7         if (th->state == UNUSED)
8             goto found;
9
10    return 0;
11
12 found:
13
14    th->state = EMBRYO;
15
16    // Allocate kernel stack.
17    if ((th->kstack = kalloc()) == 0) {
18        th->state = UNUSED;
19        return 0;
20    }
21    sp = th->kstack + KSTACKSIZE;
22
23    // Leave room for trap frame.
24    sp -= sizeof * th->tf;
25    th->tf = (struct trapframe*)sp;
26
27    // Set up new context to start executing at forkret,
28    // which returns to trapret.
29    sp -= 4;
30    *(uint*)sp = (uint)trapret;
31
32    sp -= sizeof * th->context;
33    th->context = (struct context*)sp;
34    memset(th->context, 0, sizeof * th->context);
35    th->context->eip = (uint)forkret;
36    th->retval = 0; // set retval to zero;
37
38    return th;
39 }

```

- `allocproc()` 함수를 수정하여 Thread용으로 만든 함수이다.
- `allocproc()` 과 비슷하게, `pthread_table` 에서 빈 공간을 찾아, 할당하고, 초기화 해주는 역할을 담당한다.
- 새로운 Thread를 만들고, 그 pointer를 반환한다.
- Lock은 호출하는 함수에서 잡아주어야 한다.

2-4. userinit()

```

1 acquire(&ptable.lock);
2 p->state = RUNNABLE;
3
4 int ptable_idx = p - ptable.proc;
5 p->th_sz = p->sz;
6 p->parent_proc = p;
7 p->parent_pthread = &pthread_table(pthread[p->thread_idx]);
8 memset(p->freed_sz, 0, sizeof(p->freed_sz));
9
10 // Thread undress process.
11 mvProcToThread(ptable_idx, &pthread_table(pthread[p->thread_idx]));
12 // Dust off process.
13 clearProc(p);

```

- `initproc` 을 만들어주는 부분으로 xv6의 최초의 프로세스이다.
- Process를 생성하고 scheduling을 하기 위해서 `mvProcToThread()` 에 복사해서 Thread가 scheduling될 수 있도록 한다.

2-5. fork()

```
1 np->parent_proc = curproc;
2 np->parent_pthread = &(pthread_table(pthread[curproc->thread_idx]));
3 *np->tf = *curproc->tf;
4 np->sz = curproc->sz;
5
6 np->tf->eax = 0;
7
8 for (i = 0; i < NOFILE; i++) {
9     if (curproc->ofile[i])
10        np->ofile[i] = fileup(curproc->ofile[i]);
11 np->cwd = idup(curproc->cwd);
12
13 safestrcpy(np->name, curproc->name, sizeof(curproc->name));
14
15 pid = np->pid;
16
17 acquire(&ptable.lock);
18
19 // MAINTHREAD COPY
20 // pgdir copied
21 // name copied
22 // parent setted
23 // cwd copied
24 // ofile copied
25 np->memlimit = curproc->memlimit;
26 np->thread_assign = curproc->thread_assign;
27 np->stacksz = curproc->stacksz;
28 // pid setted
29 // thread_idx is special
30 // kstack copied
31 np->th_sz = curproc->th_sz;
32 np->state = RUNABLE;
33 np->chan = curproc->chan;
34 // tf copied.
35 // context setted
36 np->killed = curproc->killed;
37 np->thread_id = curproc->thread_id; // MAINTH
38 np->front = curproc->front;
39 np->back = curproc->back;
40 np->freed = curproc->freed;
41 memset(np->freed_sz, 0, sizeof(np->freed_sz));
42 for (int i = 0; i < np->freed; i++) {
43     np->freed_sz[(np->front + i) % NTHR] =
44         curproc->freed_sz[(curproc->front + i) % NTHR];
45 }
46
47 // Only fork() thread.
48 if (curproc->thread_id != MAINTH) {
49
50     np->thread_assign = 1;
51     np->thread_id = MAINTH;
52     np->joiner_pthread = &(pthread_table(pthread[curproc->thread_idx]));
53     mvProcToThread(np - ptable.proc, &pthread_table(pthread[np->thread_idx]));
54     clearProc(np);
55     release(&ptable.lock);
56     release(&pthread_table.lock);
57
58     return curproc->pid;
59 }
60 mvProcToThread(np - ptable.proc, &pthread_table(pthread[np->thread_idx]));
61 // Copy Threads.
62 struct pthread_t* th;
63 for (th = pthread_table.pthread; th < &pthread_table.pthread[NTHR]; th++) {
64     if (th->pid == curproc->pid) {
65         if (th->thread_id == MAINTH)
66             continue;
67         struct pthread_t* nth;
68         if ((nth = allocThread()) == null) {
69             np->killed = 1;
70             pid = -1;
71             break;
72         }
73         nth->th_sz = th->th_sz;
74         nth->pid = np->pid;
75         safestrcpy(nth->kstack, th->kstack, sizeof(th->kstack));
76         nth->state = th->state;
77         nth->chan = th->chan;
78         *nth->context = *th->context;
79         *nth->tf = *th->tf;
80         nth->ptable_idx = np - ptable.proc;
81         nth->thread_id = th->thread_id;
82         nth->joiner_pthread = &pthread_table(pthread[np->main_thread_idx];
83         nth->retval = th->retval;
84     }
85 }
86 release(&ptable.lock);
87 release(&pthread_table.lock);
```

- `fork()` 는 새로운 Process를 만들어서 완전히 새로운 virtual memory를 만드는 함수이다.

• Line 1:45, Process를 만들어서 기존의 Process 정보를 복사해서 넣어준다.

• Line 48:59, Main Thread가 아닌 Thread에서 `fork()` 를 실행한 경우이다.

이런 경우에, 해당 Thread의 내용만을 복사해주고, `fork` 된 Thread를 Main Thread로 만들어주고, Scheduling을 위해서 Thread로 값을 복사해준다.

• Line 60:87, Main Thread가 `fork()` 를 실행한 경우이다.

이런 경우에, Thread의 내용을 복사해 줄뿐만 아니라, Process의 모든 Thread를 복사해준다. 즉, Process의 모든 내용을 복사해준다.

2-6. exit()

```
1 acquire(&pthread_table.lock);
2 acquire(&ptable.lock);
3
4 wakeup1(curproc->parent_pthread);
5
6 // Pass abandoned children to init.
7 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
8     if (p->parent_proc == curproc) {
9         p->parent_proc = initproc;
10        p->parent_pthread = initproc->parent_pthread;
11        if (p->state == ZOMBIE)
12            wakeup1(initproc->parent_pthread);
13    }
14 }
15 curproc->state = ZOMBIE;
16
17 struct pthread_t* th;
18 for (th = pthread_table.pthread; th < &pthread_table.pthread[NTHR]; th++) {
19     if (th->pid == curproc->pid) {
20         th->state = THZOMBIE;
21     }
22 }
23
24 release(&ptable.lock);
25 // Jump into the scheduler, never to return.
26 sched();
27 panic("zombie exit");
```

- `exit()` 은 Process를 종료하는 함수이다.

• Thread를 명시적으로 종료하지 않았을 수도 있기 때문에, Process의 모든 Thread에 대해서 `THZOMBIE` 상태로 만들어주고, Process는 `ZOMBIE` 로 만들어준다.

◦ `THZOMBIE` : `ZOMBIE` 상태와 같지만, Thread가 `ZOMBIE` 임을 알려주는 상태이다.

◦ `exit()` 에서 다른 Thread를 정리해줄 수도 있었지만, 기존의 xv6에서는 `wait()` 에서 메모리를 정리해주기 때문에, 그렇게 하는 것이 원래의 구현과 비슷하다고 생각했다.

2-7. `wait()`

```
1 for (;;) {
2     // Scan through table looking for exited children.
3     havekids = 0;
4     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
5         if (p->parent_pthread != &(pthread_table(pthread[curproc->thread_idx])))
6             continue;
7         havekids = 1;
8         // Found one.
9         if (p->state == ZOMBIE) {
10            pid = p->pid;
11            // Find Threads.
12            // Release Threads memory that unexpectedly quited.
13            struct pthread_t* th;
14            for (th = pthread_table(pthread; th < &pthread_table(pthread[NTHRD]); th++) {
15                if (th->pid != p->pid) {
16                    continue;
17                }
18                th->th_sz = 0;
19                th->pid = 0;
20                kfree(th->kstack);
21                th->kstack = 0;
22                th->state = UNUSED;
23                th->chan = 0;
24                th->ptable_idx = 0;
25                th->thread_id = 0;
26                th->joiner_pthread = 0;
27                th->retval = 0;
28            }
29            freevm(p->pgdir);
30            p->name[0] = 0;
31            p->parent_pthread = 0;
32            p->parent_proc = 0;
33            p->joiner_pthread = 0;
34            p->memlimit = 0;
35            p->thread_assign = 0;
36            p->stacksz = 0;
37            p->pid = 0;
38            p->thread_idx = 0;
39            p->sz = 0;
40            p->state = UNUSED;
41            p->chan = 0;
42            p->killed = 0;
43            p->thread_id = 0;
44            p->front = 0;
45            p->back = 0;
46            p->freed = 0;
47            memset(p->freed_sz, 0, sizeof(p->freed_sz));
48            release(&ptable.lock);
49            return pid;
50        }
51    }
52    // No point waiting if we don't have any children.
53    if (!havekids || curproc->killed) {
54        release(&ptable.lock);
55        return -1;
56    }
57 }
58
59 // Wait for children to exit. (See wakeup call in proc_exit.)
60 sleep(&(pthread_table(pthread[curproc->thread_idx]), &ptable.lock); //DOC: wait-sleep
61 }
```

- `wait()` 은 자식 Process와 Thread를 정리해주는 함수이다.
- Line 14:28, 자식 Process의 Thread를 찾아서 정리하는 부분이다.
- Line 29:49, 자식 Process의 정보를 찾아서 정리하는 부분이다.
- Line 60, 현재 실행 중인 스레드를 sleep 시킨다.

2-8. `wakeup1()`

```
1 static void
2 wakeup1(void* chan)
3 {
4     struct pthread_t* th;
5     for (th = pthread_table(pthread; th < &pthread_table(pthread[NTHRD]); th++) {
6         if (th->state == SLEEPING && th->chan == chan)
7             th->state = RUNNABLE;
8     }
}
```

- 나의 구현에서는 Thread가 Process를 입고 작동하는 것이기 때문에, 모든 실체는 Thread이다. 그렇기 때문에 Process를 `RUNNABLE`로 만드는 것이 아니라, Thread를 `RUNNABLE`로 만들어 준다.

2-9. `kill()`

```
1 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
2     if (p->pid == pid) {
3         p->killed = 1;
4         // Wake process from sleep if necessary.
5         if (p->state == SLEEPING)
6             p->state = RUNNABLE;
7         for (th = pthread_table(pthread; th < &pthread_table(pthread[NTHRD]); th++) {
8             if (th->pid == pid) {
9                 th->state = THZOMBIE;
10                if (th->thread_id == MAINTH) {
11                    th->state = RUNNABLE;
12                }
13            }
14        }
15        release(&ptable.lock);
16        return 0;
17    }
18 }
19 }
```

- `killed` 를 1로 만들어주어 자신을 죽인다.
- Process와 Main Thread를 `RUNNABLE` 상태로 만들고, 그 외의 Thread는 모두 `THZOMBIE` 상태로 만들어 준다.

2-10. `setmemorylimit()`

```
1 int setmemorylimit(int pid, int limit) {
2     if (limit < 0) {
3         cprintf("Can not be memory limit negative : %d\n", limit);
4         return -1;
5     }
6     acquire(&ptable.lock);
7     struct proc* p;
8     for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
9         if (p->pid != pid) {
10            continue;
11        }
12        if (p->sz > (uint)limit) {
13            cprintf("memory limit %d is less than allocated memory size : %d\n", limit, p->sz);
14            release(&ptable.lock);
15            return -1;
16        }
17        p->memlimit = (uint)limit;
18        release(&ptable.lock);
19        return 0;
20    }
21    cprintf("There is no process with %d\n", pid);
22    release(&ptable.lock);
23    return -1;
24 }
25 }
```

- `cprintf` 문은 모두 `setmemorylimit` 을 제대로 실행하지 못한 경우에 대해서 출력을 표시해주는 부분이다.
- 제대로 실행되는 경우 process의 `limit` 을 설정해준다.

2-11. `thread_create()` & `pthread_fork()` & `pthread_exec()`

```

1 struct pthread_t* pthread_fork(void)
2 {
3
4     struct pthread_t* nth;
5     struct proc* curproc = myproc();
6
7     if ((nth = allocthread()) == 0) {
8         return null;
9     }
10    nth->pid = curproc->pid;
11    *nth->tf = *curproc->tf;
12    nth->thread_id = ++(curproc->thread_assign);
13    nth->phtable_idx = curproc - ptable.proc;
14    nth->tf->eax = 0;
15
16    return nth;
17 }
```

```

1 int pthread_exec(struct pthread_t* th, thread_t* thread, void* (*start_routine)(void*), void* arg)
2 {
3     uint sz, sp, ustack[1 + MAXARG + 3];
4     pde_t* pgdir;
5     struct proc* curproc = myproc();
6
7     uint stacksize = curproc->stacksz;
8     pgdir = curproc->pgdir;
9
10    // TODO : Stack Array Search
11    sz = curproc->sz;
12    if (curproc->freed) {
13        sz = (curproc->freed_sz[curproc->front]);
14    }
15
16    // Allocate two pages at the next page boundary.
17    // Make the first inaccessible. Use the second as the user stack.
18    sz = PGROUNDUP(sz);
19    if (curproc->memlimit != 0 && sz + (stacksize + 1) * PGSIZE > curproc->memlimit) {
20        cprintf("exceed memlim : %d\n", curproc->memlimit);
21        return -1;
22    }
23    if ((sz = allocuvvm(pgdir, sz, sz + (stacksize + 1) * PGSIZE)) == 0) {
24        // revert allocuvvm
25        return -1;
26    }
27    clearpte(pgdir, (char*)(sz - (stacksize + 1) * PGSIZE));
28    sp = sz;
29    ustack[0] = 0xffffffff;
30    sp -= 4;
31
32    ustack[1] = (uint)arg;
33    sp -= 4;
34
35    if (copyout(pgdir, sp, ustack, 2 * 4) < 0) {
36        return -1;
37    }
38
39    if (curproc->freed) {
40        // pop Q
41        curproc->freed_sz[curproc->front] = 0;
42        curproc->freed--;
43        curproc->front++;
44        curproc->front %= NTHRD;
45        curproc->sz = curproc->sz;
46    }
47    else {
48        curproc->sz = sz;
49    }
50    th->th_sz = sz;
51    curproc->pgdir = pgdir;
52    th->tf->eip = (uint)start_routine;
53    th->tf->esp = sp;
54    // The process doesn't have to return to the instruction
55    // after exec() when it gets the CPU next,
56    // but instead must start executing the new executable it
57    // just loaded from dist.
58    // exec() changes the return address in the trap frame
59    // to point to the entry address of the binary.
60    return 0;
61 }
```

```

1 int thread_create(pthread_t* thread, void* (*start_routine)(void*), void* arg) {
2     acquire(&pthread_table.lock);
3     struct proc* curproc = myproc();
4     // Allocate Stack for thread Start.
5     struct pthread_t* new_thread = pthread_fork();
6     if (new_thread == null) {
7         release(&pthread_table.lock);
8         return -1;
9     }
10    // Allocate Stack for thread End.
11    acquire(&ptable.lock);
12    if (pthread_exec(new_thread, thread, start_routine, arg) == -1) {
13        goto bad;
14    }
15
16    // Setting Thread Variable
17    new_thread->pid = curproc->pid;
18    // kstack DONE in pthread_fork
19    new_thread->state = RUNNABLE;
20    // new_thread->chan is not need to be setted.
21    new_thread->tf->eax = 0;
22    new_thread->tf->eip = (uint)(start_routine);
23    // new_thread->context is already setted.
24    // new_thread->killed is not need to be setted.
25    new_thread->phtable_idx = curproc - ptable.proc;
26
27    new_thread->joiner_pthread = &pthread_table.pthread[curproc->thread_idx];
28
29    *thread = new_thread->thread_id;
30    release(&ptable.lock);
31    release(&pthread_table.lock);
32
33    return 0;
34 bad:
35    // Unfork thread
36    new_thread->state = UNUSED;
37    release(&ptable.lock);
38    release(&pthread_table.lock);
39    return -1;
40 }
```

- `pthread_fork()`

`fork()` 함수와 비슷하며, thread를 생성해주고 기본정보를 설정해준다.

- `pthread_exec()`

`exec2()` 함수와 비슷하며, thread의 새로운 실행 정보를 설정해준다.

Line 11:14, memory 공간을 할당할 시작점을 설정한다. 만약 중간에 비어있는 공간이 있다면, 그 공간을 선택해주어서 공간을 재활용한다.

Line 18:27, `sz`에 대해서 공간을 할당할 수 있는지 확인해보고, 메모리제한이 걸리면 할당하지 않고, 할당할 수 있으면 `pgdir`에 할당한다. 이 때, 확보해야하는 stack page의 수는 process의 stack page의 수와 같다.

Line 28:37, 시작하는 함수에 전달할 인자와 fake return 을 설정해준다.

Line 39:46, free page를 사용했다면, Q에서 그 공간을 제거해준다.

Line 50:53, thread의 시작 지점을 설정해준다.

2-12. `thread_exit()`

```

1 void thread_exit(void* retval) {
2     struct proc* curproc = myproc();
3     acquire(&pthread_table.lock);
4     if (curproc->thread_id == MAINTH) {
5         kill(curproc->pid);
6     }
7     else {
8         pthread_table.pthread[curproc->thread_idx].retval = retval;
9         curproc->state = THZOMBIE;
10    }
11    acquire(&ptable.lock);
12    release(&pthread_table.lock);
13    wakeup1(curproc->joiner_pthread);
14
15    sched();
16    panic("thread zombie exit");
17 }
```

- Thread를 종료시키고, return값을 받아주는 함수이다.

- 만약 `thread_exit` 을 호출한 주체가 Main Thread라면, 전체 Process를 종료하기 위해서 `kill` 을 호출한다.

이는 일반적이지 않은 경우로, Process의 종료는 `exit()` 을 호출해야 한다.

- 만약 Sub Thread가 호출한 경우, return 값을 `retval`에 담아주고, `state` 를 `THZOMBIE` 로 바꾸어 준다.

- 그리고 자신을 join해줄, `joiner_pthread` 를 깨워준다.

2-13. exec.c

2-13-1. exec()

```
1 // Clean up other sibling threads.
2 deleteThreads(curproc);
3 memset(curproc->freed_sz, 0, sizeof(curproc->freed_sz));
4 curproc->front = 0;
5 curproc->back = 0;
6 curproc->freed = 0;
7
8 // Commit to the user image.
9 oldpgdir = curproc->pgdir;
10 curproc->pgdir = pgdir;
11 curproc->sz = sz;
12 curproc->th_sz = sz; // thread_sz
13 curproc->memlimit = 0; // default memlimit
14 curproc->thread_assign = 1; // default thread_assign
15 curproc->stacksz = 1; // default stacksz
16 curproc->thread_id = MAINTH;
17 curproc->main_thread_idx = curproc->thread_idx;
18 // The process doesn't have to return to the instruction
19 // after exec() when it gets the CPU next,
20 // but instead must start executing the new executable it
21 // just loaded from dist.
22 // exec() changes the return address in the trap frame
23 // to point to the entry address of the binary.
24 curproc->tf->eip = elf.entry;
25 curproc->tf->esp = sp;
```

- 기존의 exec()에서 수정된 부분이다.
 - 다른 Thread들을 정리해주는 deleteThreads()함수가 추가되었다.
 - Line 3:25, 새로운 Process를 올렸기 때문에 이것에 대해서 값을 초기화 준다.

2-13-2. exec2()

```

1 int exec2(char* path, char** argv, int stacksize) {
2     char* sz = last;
3     int i, off;
4     uint argc, sz, sp, ustack[3 + MAXARG + 1];
5     struct elfhdr* elf;
6     struct inode* ip;
7     struct proghdr* ph;
8     pde_t* pgdir, * oldpgdir;
9     int locked = 0;
10    struct curproc* curproc = myproc();
11    if (stacksize < 1 || stacksize>100) {
12        fprintf(stderr, "execve Fail : stacksize should be in range of [1,100]\n");
13        return -1;
14    }
15    begin_op();
16
17    if ((ip = namei(path)) == 0) {
18        end_op();
19        return -1;
20    }
21    ilock(ip);
22    pgdir = 0;
23
24    // curproc must be scheduled.
25    curproc->state = RUNNING;
26
27    // Check ELF header
28    if (readelf(p, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
29        goto bad;
30    if (elf.magic != ELF_MAGIC)
31        goto bad;
32
33    if ((pgdir = setupkvm()) == 0)
34        goto bad;
35
36    // Load program into memory.
37    sz = 0;
38    for (i = 0, off = elf.phoff; i < elf.phnum; i++, off += sizeof(ph)) {
39        if (!loadfile(ip, (char*)ph, off, sizeof(ph)) != sizeof(ph))
40            goto bad;
41        if (ph.type != ELF_PROG_LOAD)
42            continue;
43        if (ph.memsz < ph.filesz)
44            goto bad;
45        if (ph.vaddr + ph.memsz < ph.vaddr)
46            goto bad;
47        if ((sz + allocum(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
48            goto bad;
49        if (ph.vaddr % PGSIZE != 0)
50            goto bad;
51        if (loadmem(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
52            goto bad;
53    }
54    unlockput(ip);
55    end_op();
56    lock();
57    locked = 1;
58    ip = 0;
59
60    // Allocate two pages at the next page boundary.
61    // sz is the first inaccessible. Use the second as the user stack.
62    sz = PGROUNDOFF(sz);
63    if ((sz + allocum(pgdir, sz, sz + (stacksize + 1) * PGSIZE)) == 0)
64        goto bad;
65    clearpage(pgdir, (char*)(sz - (stacksize + 1) * PGSIZE));
66    sp = sz;
67
68    // Push argument strings, prepare rest of stack in ustack.
69    for (argc = 0; argv[argc]; argc++)
70        if (argc > MAXARG)
71            goto bad;
72    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
73    if (copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
74        goto bad;
75    ustack[3 + argc] = sp;
76
77    // Save program name for debugging.
78    for (last = s = path; *s++;)
79        if (*s == '/')
80            last = s + 1;
81    safestrcpy(curproc->name, last, sizeof(curproc->name));
82
83    // Clean up other sibling threads.
84    if (curproc->parent == curproc)
85        memset(curproc->freed, 0, sizeof(curproc->freed));
86    curproc->front = 0;
87    curproc->back = 0;
88    curproc->freed = 0;
89
90    // Commit to the user image.
91    oldpgdir = curproc->pgdir;
92    curproc->pgdir = pgdir;
93    curproc->sz = sz; // PROC_SZ
94    curproc->th_sz = sz; // THREAD_SZ
95    curproc->tf->spip = elf_entry; // main
96    curproc->tf->esp = sp;
97    curproc->meminit = 0; // newly exec
98    curproc->thread_assign = 1; // default thread_assign
99    curproc->stacksz = stacksize; // default stacksz
100   curproc->thread_id = MAINTH;
101   curproc->main_thread_idx = curproc->thread_idx;
102   unlock();
103   switchin(curproc);
104   freevad(alppgdir);
105
106   return 0;
107
108 bad:
109    if (pgdir)
110        freevad(pgdir);
111    if (ip)
112        unlockput(ip);
113    end_op();
114    }
115    if (locked)
116        unlock();
117    return -1;
118 }
```

- 새롭게 추가된 system call인 `exec2()` 이다.
 - `stacksize` 만큼의 stack page와 1개의 guard page를 할당한다.
 - `exec()` 함수와 큰 틀은 다른 것이 없고, `stacksize` 만큼 할당하는 부분만 다르다.

2-13-3. `deleteThread()` & `deleteThreads()`

```

1 // delete Thread.
2 void deleteThread(struct pthread_t* th) {
3     th->th_sz = 0;
4     th->pid = 0;
5     kfree(th->kstack);
6     th->kstack = 0;
7     th->state = UNUSED;
8     th->chan = 0;
9     th->phtable_idx = 0;
10    th->thread_id = 0;
11    th->joiner_pthread = 0;
12    th->retval = 0;
13 }
14
15 // delete Threads left only one.
16 void deleteThreads(struct proc* p) {
17     struct pthread_t* th;
18     for (th = pthead_table.pthead; th < &pthead_table.pthead[NTHR0]; th++) {
19         // Find the threads that is in process group except who calls exec().
20         if (th->pid != p->pid || th->thread_id == p->thread_id)
21             continue;
22         for (struct proc* child = ptable.proc; child < &ptable.proc[NPROC]; child++) {
23             if (child->parent_pthread != th)
24                 continue;
25             // adopt other child process of other sibling threads
26             child->parent_pthread = &(pthead_table.pthead[p->thread_idx]);
27         }
28         deleteThread(th);
29     }
30     p->joiner_pthread = &(pthead_table.pthead[p->thread_idx]);
31 }

```

- `deleteThreads()` 함수는 sibling Thread를 찾으면서 thread를 정리해준다.
 - 살아남은 Thread를 자신을 join할 수 있는 thread로 만들어준다

2-13-4. lock() & unlock()

- exec.c에서는 lock을 따로 잡아줄 수가 없기 때문에 만든 함수이다.

```
1 void lock(void) {
2     acquire(&pthread_table.lock);
3 }
4 void unlock(void) {
5     release(&pthread_table.lock);
6 }
```

2-14. procInfo()

- Process의 정보를 출력해주는 함수이다.
- Pmanager의 list 명령어로 실행된다.

```
1 void procInfo(void) {
2     acquire(&ptable.lock);
3     int i;
4     struct proc *p;
5     int proc_idx[NPROC];
6     int num_proc = 0;
7
8     for (i = 0; i < NPROC; i++) { // save ptable index that contains proc.
9         if (ptable.proc[i].pid != 0 && (ptable.proc[i].state == RUNNABLE || ptable.proc[i].state == RUNNING || ptable.proc[i].state == SLEEPING)) {
10             proc_idx[num_proc] = i;
11         }
12     }
13
14     for (i = 0; i < num_proc; i++) {
15         for (int j = i+1; j < num_proc; j++) {
16             if ((proc_idx[i] <= proc_idx[j]) && (j).pid > ptable.proc[proc_idx[j]].pid) {
17                 temp = (proc_idx[i] - i) + proc_idx[j];
18                 proc_idx[j] = temp;
19                 proc_idx[i] = temp;
20             }
21         }
22     }
23     name: 18          pid: 12      stacksz: 12      pg: 12      memory: 12
24     CpuInfo[.....name.....].....pid.....|.....stack pages.....|.....current memory.....|.....memory limit.....|\n";
25     CpuInfo[.....name.....].....pid.....|.....stack pages.....|.....current memory.....|.....memory limit.....|\n";
26     CpuInfo[.....name.....].....pid.....|.....stack pages.....|.....current memory.....|.....memory limit.....|\n";
27
28     int len;
29     for (i = 0; i < num_proc; i++) {
30         p = &ptable.proc[proc_idx[i]]));
31         CpuInfo(" ");
32         char name[55];
33         strcpy(name, p->name, 55);
34         len = 0;
35         CpuInfo("%s", name);
36         for (int j = strlen(name) + 1; j < 18; j++) {
37             CpuInfo(" ");
38         }
39         CpuInfo(" ");
40         CpuInfo(" ");
41         int pid = p->pid;
42         if (pid == 0) {
43             pid = i;
44         }
45         len = 0;
46         while (pid) {
47             pid /= 10;
48             len++;
49         }
50         CpuInfo("%d", p->pid);
51         for (int j = len + 1; j < 12; j++) {
52             CpuInfo(" ");
53         }
54         CpuInfo(" ");
55         CpuInfo(" ");
56         uint stacksz = p->stacksz;
57         while (stacksz) {
58             stacksz /= 10;
59             len++;
60         }
61         CpuInfo("%d", p->stacksz);
62         for (int j = len + 1; j < 15; j++) {
63             CpuInfo(" ");
64         }
65         CpuInfo(" ");
66         CpuInfo(" ");
67         CpuInfo(" ");
68         uint sz = p->sz;
69         len = 0;
70         while (sz) {
71             sz /= 10;
72             len++;
73         }
74         CpuInfo("%d", sz);
75         for (int j = len + 1; j < 18; j++) {
76             CpuInfo(" ");
77         }
78         CpuInfo(" ");
79         CpuInfo(" ");
80         uint memlimit = p->memlimit;
81         if (memlimit == 0) {
82             memlimit = 1;
83         }
84         len = 0;
85         while (memlimit) {
86             memlimit /= 10;
87             len++;
88         }
89         CpuInfo("%d", p->memlimit);
90         for (int j = len + 1; j < 16; j++) {
91             CpuInfo(" ");
92         }
93         CpuInfo(" ");
94         CpuInfo(" ");
95         CpuInfo(" ");
96         CpuInfo(" ");
97         CpuInfo(" ");
98     }
99     CpuInfo(" |.....|.....|.....|.....|.....|.....|.....|\n");
100    CpuInfo(" |.....|.....|.....|.....|.....|.....|.....|\n");
101    CpuInfo(" |.....|.....|.....|.....|.....|.....|.....|\n");
102    release(&ptable.lock);
103 }
```

2-15. System calls

[syscall.c]

```
1 extern int sys_exec2(void);
2 extern int sys_thread_create(void);
3 extern int sys_thread_exit(void);
4 extern int sys_thread_join(void);
5 extern int sys_procInfo(void);
```

```
1 [SYS_setmemorylimit] sys_setmemorylimit,
2 [SYS_exec2] sys_exec2,
3 [SYS_thread_create] sys_thread_create,
4 [SYS_thread_exit] sys_thread_exit,
5 [SYS_thread_join] sys_thread_join,
6 [SYS_procInfo] sys_procInfo,
```

[syscall.h]

```
1 #define SYS_setmemorylimit 22
2 #define SYS_exec2 23
3 #define SYS_thread_create 24
4 #define SYS_thread_exit 25
5 #define SYS_thread_join 26
6 #define SYS_procInfo 27
```

[sysproc.c]

```
1 int sys_setmemorylimit(void) {
2     int pid, limit;
3     if (argint(0, &pid) < 0 || argint(1, &limit) < 0) {
4         return -1;
5     }
6     return setmemorylimit(pid, limit);
7 }
8
9 int sys_thread_create(void) {
10    int thread, start_routine, arg;
11    if (argint(0, &thread) < 0 || argint(1, &start_routine) < 0 || argint(2, &arg) < 0) {
12        return -1;
13    }
14    return thread_create((thread_t*)thread, (void*)start_routine, (void*)arg);
15 }
16
17 int sys_thread_exit(void) {
18    int retval;
19    if (argint(0, &retval) < 0) {
20        return -1;
21    }
22    thread_exit((void*)retval);
23    return 0;
24 }
25
26 int sys_thread_join(void) {
27    int thread, retval;
28    if (argint(0, &thread) < 0 || argint(1, &retval) < 0) {
29        return -1;
30    }
31    return thread_join((thread_t*)thread, (void**)retval);
32 }
33
34 int sys_procInfo(void) {
35    procInfo();
36    return 0;
37 }
```

[sysfile.c]

```
1 int sys_exec2(void) {
2     char* path, * argv[MAXARG];
3     int i;
4     uint uargv, uarg;
5     int stacksz;
6     if (argstr(0, &path) < 0 || argint(1, (int*)&uargv) < 0 || argint(2, &stacksz) < 0) {
7         return -1;
8     }
9     memset(argv, 0, sizeof(argv));
10    for (i = 0; i++ < uargv) {
11        if (i >= NELEM(argv))
12            return -1;
13        if (fetchInt(uargv + 4 * i, (int*)&uarg) < 0)
14            return -1;
15        if (uarg == 0) {
16            argv[i] = 0;
17            break;
18        }
19        if (fetchStr(uarg, &argv[i]) < 0)
20            return -1;
21    }
22    return exec2(path, argv, stacksz);
23 }
```

[user.h]

```
1 int exec2(char*, char**, int);
2 int setmemorylimit(int, int);
3 int thread_create(thread_t*, void* (*)(void*), void* );
4 int thread_exit(void* );
5 int thread_join(thread_t, void** );
6 int procInfo(void);
```

[usys.S]

```
1 SYSCALL(setmemorylimit)
2 SYSCALL(exec2)
3 SYSCALL(thread_create)
4 SYSCALL(thread_exit)
5 SYSCALL(thread_join)
6 SYSCALL(procInfo)
```

- `setmemorylimit()`, `exec2()`, `thread_create()`, `thread_exit()`, `thread_join()`, `procInfo()` 를 system call로 제공하기 위해서 추가한 내용이다.
- `user.h`에 정의된 system call 함수가 호출되면, `usys.S`에서 해당 함수를 system call로 바꾸어 전달한다.
- 각각의 system call의 번호는 `syscall.h`에 정의되어 있고, 이에 따라서 `syscall.c`에 선언되어 있는 wrapper function을 실행하게 된다.
- `sysproc.c`에 세부적인 wrapper function의 정의가 되어 있다.
- 다른 system call과 같이 인자를 받아서 실제 함수를 호출하는 형식으로 되어 있다.

2-16. Definitions

[types.h]

```
1 typedef uint thread_t;
```

[param.h]

```
1 #define NTHRD      256 // maximum number of threads
2 #define MAINTH      1 // dictate main thread number.
```

[defs.h]

```
1 // exec.c
2 int          exec2(char*, char**, int);
3
4 // proc.c
5 int          setmemorylimit(int, int);
6 void         mvProcToThread(int, struct pthread_t*);
7 void         clearProc(struct proc* );
8 void         deleteThread(struct pthread_t* );
9 void         deleteThreads(struct proc* );
10 void        mvThreadToProc(int, struct proc* );
11 struct pthread_t* pthread_fork(void);
12 int          pthread_exec(struct pthread_t*, thread_t*, void* (*)(void*), void* );
13 int          thread_create(thread_t*, void* (void*), void* );
14 void         thread_exit(void* );
15 int          thread_join(thread_t, void** );
16 void        procInfo(void);
17 void        lock(void);
18 void        unlock(void);
```

- `proc.c`와 `exec.c`에 새로 정의된 함수들을 선언했다.

2-17. Pmanager

```
1 void strSplit(char* str, char* buf, int* idx) {
2     int st = 0;
3     for (int i = *idx; i < nbuf;i++) {
4         if (buf[i] == ' ' || buf[i] == '\n') {
5             *idx = i + 1;
6             str[*idx] = '\0';
7             break;
8         }
9         str[*idx] = buf[i];
10    }
11 }
12
13 int main(int argc, char* argv[]) {
14     printf(2, "Hello Pmanager!\n");
15     while (1) {
16         printf(2, "> ");
17
18         char buf[nbuf];
19         char* argv[MAXARGS];
20
21         gets(buf, nbuf); // read line
22
23         // Parsing cmd
24         char cmd[10];
25         int idx = 0;
26         char cpid[20];
27         char cstacksz[20];
28         char cmemlim[20];
29         char cpath[55];
30         int pid;
31         int memlim;
32         int stacksz;
33         strSplit(cmd, buf, &idx);
34
35         if (!strcmp(cmd, "list")) { // cmd == list
36             // Listing info of all the processes executing.
37             // Info : process name, pid, # of stack pages,
38             // size of allocated memory, memory limitation.
39             // Info must includes thread info.
40             // Thread's info should not be printed separately.
41             // It is up to me that define any system call to get infos of process.
42             procInfo();
43         }
44         else if (!strcmp(cmd, "kill")) { // cmd == kill
45             // Kill the process with pid.
46             // Use "kill" syscall.
47             // Print weather it is success or not.
48             strSplit(cpid, buf, &idx);
49
50             printf(2, "KILL Process : %s\n", cpid);
51             pid = atoi(cpid);
52             if (kill(pid, 0) {
53                 printf(2, "Kill Process : %d failed\n", pid);
54             }
55             else {
56                 printf(2, "kill Process : %d succeed\n", pid);
57             }
58         }
59         else if (!strcmp(cmd, "execute")) { // cmd == execute
60             // Execute program that is located in <path>,
61             // with stack pages which is amount of <stacksize>.
62             // A argument parsed to program is 0-th parameter, <path>.
63             // pmanager is still executing, does not wait about terminating program executed.
64             // Do not print any message when it succeed, otherwise print.
65             if (fork1() == 0) {
66                 if (fork1() == 0) {
67                     strSplit(cpath, buf, &idx);
68                     strSplit(cstacksz, buf, &idx);
69                     argv[0] = cpath;
70                     stacksz = atoi(cstacksz);
71                     exec2(argv[0], argv, stacksz);
72                 }
73                 else {
74                     exit();
75                 }
76                 printf(2, "EXECUTE Process in %s failed\n", cpath);
77                 exit();
78             }
79             wait();
80         }
81         else if (!strcmp(cmd, "memlim")) { // cmd == memlimit
82             // Set memory limit of process with <pid> to <limit>.
83             // <limit> is integer greater or equal to 0.
84             // If it is 0, there is no limit.
85             // Otherwise, it has limit with same amount of <limit>.
86             // Process's memory should be considered thread's memory.
87             // Print weather it is success or not.
88
89             // setmemomylimit syscall
90             strSplit(cpid, buf, &idx);
91             strSplit(cmemlim, buf, &idx);
92             pid = atoi(cpid);
93             memlim = atoi(cmemlim);
94             if (setmemomylimit(pid, memlim) < 0) { // failure
95                 printf(2, "set process %d's memory limit : %d failed\n", pid, memlim);
96             }
97             else {
98                 printf(2, "set process %d's memory limit : %d succeed\n", pid, memlim);
99             }
100        }
101        else if (!strcmp(cmd, "exit")) { // cmd == kill
102            // Terminate pmanager.
103            printf(2, "Goodbye Pmanager!\n");
104            exit();
105        }
106        else {
107            printf(1, "Wrong Command : %s\n", cmd);
108        }
109    }
110    exit();
111 }
```

- `Pmanager` 함수의 구현이다.
- `strSplit()`은 들어오는 명령어를 파싱하기 위한 함수이다.
- 실행하면 “Hello Pmanager!”가 출력되고 “>”를 통해서 명령어를 입력 받을 수 있음을 알려준다.
- “list”를 입력하면 `procInfo()`를 실행하여, process의 정보를 보여준다.
- “kill {pid}”를 입력하면 {pid}를 가진 process를 `kill` 한다.
 - 성공과 실패 여부를 출력한다.
- “execute {path} {stacksize}”를 입력하면, {path}의 프로그램을 {stacksize}만큼의 stack page를 할당해서 실행한다.
 - 실행에 실패한 경우에 한해서만 출력한다.
 - 백그라운드 실행을 위해서 2번의 `fork()`를 이용했다.
 - 프로그램이 종료되면, xv6의 구조상 orphan process를 initproc에 옮겨주기 때문에 “zombie process!”가 출력된다.
- “memlim {pid} {limit}”을 통해서 {pid}를 가진 process의 최대 메모리 사용량을 조절할 수 있다.
 - 성공 여부를 출력한다.
- “exit”를 입력하면 “Goodbye pamanger!”를 출력하고 `pmanager`를 종료한다.
- 이 외의 명령어가 들어온 경우 잘못된 명령어임을 알려준다.

3. Result

3-1. Test 1 - Default

[Test 1의 소스 코드]

```
1 void* thread_sbdk(void* arg)
2 {
3     int val = (int)arg;
4     printf(1, "Thread %d start\n", val);
5     int i, j;
6
7     if (val == 0) {
8         ptr = (int*)malloc(65536);
9         sleep(100);
10        free(ptr);
11        ptr = 0;
12    } else {
13        while (ptr == 0)
14            sleep(1);
15        for (i = 0; i < 16384; i++)
16            ptr[i] = val;
17    }
18
19    while (ptr != 0)
20        sleep(1);
21
22    for (i = 0; i < 2000; i++) {
23        int* p = (int*)malloc(65536);
24        for (j = 0; j < 16384; j++)
25            p[j] = val;
26        for (j = 0; j < 16384; j++) {
27            if (p[j] != val) {
28                printf(1, "Thread %d found %d\n", val, p[j]);
29                failed();
30            }
31        }
32        free(p);
33    }
34
35    thread_exit(arg);
36    return 0;
37 }
38
39 void create_all(int n, void* (*entry)(void*))
40 {
41     int i;
42     for (i = 0; i < n; i++) {
43         if (thread_create(&thread[i], entry, (void*)i) != 0) {
44             printf(1, "Error creating thread %d\n", i);
45             failed();
46         }
47     }
48 }
49
50 void join_all(int n)
51 {
52     int i, retval;
53     for (i = 0; i < n; i++) {
54         if (thread_join(thread[i], (void**)&retval) != 0) {
55             printf(1, "Error joining thread %d\n", i);
56             failed();
57         }
58         if (retval != expected[i]) {
59             printf(1, "Thread %d returned %d, but expected %d\n", i, retval, expected[i]);
60             failed();
61         }
62     }
63 }
64 }
```

[Test 1의 소스 코드]

```
1 void* thread_basic(void* arg)
2 {
3     int val = (int)arg;
4     printf(1, "Thread %d start\n", val);
5     if (val == 1) {
6         sleep(200);
7         status = 1;
8     }
9     printf(1, "Thread %d end\n", val);
10    thread_exit(arg);
11    return 0;
12 }
13
14 void* thread_fork(void* arg)
15 {
16     int val = (int)arg;
17     int pid;
18
19     printf(1, "Thread %d start\n", val);
20     pid = fork();
21     if (pid < 0) {
22         printf(1, "Fork error on thread %d\n", val);
23         failed();
24     }
25
26     if (pid == 0) {
27         printf(1, "Child of thread %d start\n", val);
28         sleep(100);
29         status = 3;
30         printf(1, "Child of thread %d end\n", val);
31         exit();
32     }
33     else {
34         status = 2;
35         if (wait() == -1) {
36             printf(1, "Thread %d lost their child\n", val);
37             failed();
38         }
39     }
40     printf(1, "Thread %d end\n", val);
41     thread_exit(arg);
42     return 0;
43 }
```

[Test 1의 실행 결과]

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

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

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

All tests passed!
$
```

[분석]

- Test1은 `thread_basic` 을 실행하는 test로, 제대로 thread가 생성 및 삭제가 되는지를 테스트한다.
- Test2는 `thread_fork` 를 실행하는 test로, 총 5개의 thread를 생성해서 각각의 Thread를 `fork` 한다.
`fork` 된 Thread는 모두 child process가 된다.
해당 프로세스는 `exit` 으로 종료되고, thread는 각각 `thread_exit` 으로 종료된다.
이는 `fork`, `exit`, `thread_exit`, `wait` 으로 thread가 잘 정리되는지를 test한다.
- Test3은 `thread_sbdk` 를 실행하는 test로, 각각의 thread가 `sbrk` 를 호출할 때, 잘 할당이 되는지를 확인하는 테스트이다.
⇒ 모두 정상적으로 실행 및 종료된다.

3-2. Test 2 - Exit

[Test 2의 소스 코드]

```
1 void* thread_main(void* arg)
2 {
3     int val = (int)arg;
4     printf(1, "Thread %d start\n", val);
5     if (arg == 0) {
6         sleep(100);
7         printf(1, "Exiting...\n");
8         exit();
9     }
10    else {
11        sleep(200);
12    }
13
14    printf(1, "This code shouldn't be executed!!\n");
15    exit();
16    return 0;
17 }
18
19 thread_t thread[NUM_THREAD];
20
21 int main(int argc, char* argv[])
22 {
23     int i;
24     printf(1, "Thread exit test start\n");
25     for (i = 0; i < NUM_THREAD; i++) {
26         thread_create(&thread[i], thread_main, (void*)i);
27     }
28     sleep(200);
29     printf(1, "This code shouldn't be executed!!\n");
30     exit();
31 }
32
```

[Test 2의 실행 결과]

```
$ testThreadExit
Thread exit test start
Thread 0 start
Thread 1 stThread 2 start
Thread 3 start
Thread 4 start
art
Exiting...
$
```

[분석]

- Thread를 생성하고, `thread_exit` 이 아닌 `exit` 을 호출했을 때, Process가 종료되는지를 test한다.
- “Exiting...”이 1 번만 출력되었기 때문에 정상적으로 종료되었음을 알 수 있다.

3-3. Test 3 - Exec

[Test 3의 소스 코드]

```
1 void* thread_main(void* arg)
2 {
3     int val = (int)arg;
4     printf(1, "Thread %d start\n", val);
5     if (arg == 0) {
6         sleep(100);
7         char* pname = "/testThreadHello";
8         char* args[2] = { pname, 0 };
9         printf(1, "Executing...\n");
10        exec(pname, args);
11    }
12    else {
13        sleep(200);
14    }
15
16    printf(1, "This code shouldn't be executed!!\n");
17    exit();
18    return 0;
19 }
20
21 thread_t thread[NUM_THREAD];
22
23 int main(int argc, char* argv[])
24 {
25     int i;
26     printf(1, "Thread exec test start\n");
27     for (i = 0; i < NUM_THREAD; i++) {
28         thread_create(&thread[i], thread_main, (void*)i);
29     }
30     sleep(200);
31     printf(1, "This code shouldn't be executed!!\n");
32     exit();
33 }
```

[Test 3의 소스 코드]

```
1 int main(int argc, char* argv[])
2 {
3     printf(1, "Hello, thread!\n");
4     exit();
5 }
```

[Test 3의 실행 결과]

[분석]

- Thread가 `exec()` 을 실행했을 때, 정상적으로 작동하지를 나타내는 test이다.
⇒ 5개의 Thread이지만, `exec()` 후 하나의 “Hello, thread!”만 출력되었다.
⇒ 정상적으로 다른 Thread가 정리되었다.

```
$ testThreadExec
Thread exec test start
Thread 0 start
Thread 1 start
Thread 2 start
Thread 3 startThread 4 start

Executing...
Hello, thread!
$
```

3-4. Test 4 - Kill

[Test 4의 소스 코드]

```
1 void* thread1(void* arg)
2 {
3     sleep(200);
4     printf(1, "This code shouldn't be executed!!\n");
5     exit();
6     return 0;
7 }
8
9 void* thread2(void* arg)
10 {
11     int val = (int)arg;
12     sleep(100);
13     if (val != 0) {
14         printf(1, "Killing process %d\n", val);
15         kill(val);
16     }
17     printf(1, "This code should be executed 5 times.\n");
18     thread_exit(0);
19     return 0;
20 }
21
22 thread_t t1[NUM_THREAD], t2[NUM_THREAD];
23
24 int main(int argc, char* argv[])
25 {
26     int i, retval;
27     int pid;
28
29     printf(1, "Thread kill test start\n");
30     pid = fork();
31     if (pid < 0) {
32         printf(1, "Fork failed!!\n");
33         exit();
34     }
35     else if (pid == 0) { //child
36         for (i = 0; i < NUM_THREAD; i++)
37             thread_create(&t1[i], thread1, (void*)i);
38         sleep(300);
39         printf(1, "This code shouldn't be executed!!\n");
40         exit();
41     }
42     else { // parent
43         for (i = 0; i < NUM_THREAD; i++) {
44             if (i == 0)
45                 thread_create(&t2[i], thread2, (void*)pid);
46             else
47                 thread_create(&t2[i], thread2, (void*)0);
48         }
49         for (i = 0; i < NUM_THREAD; i++)
50             thread_join(t2[i], (void**)&retval);
51
52         while (wait() != -1)
53             ;
54
55         printf(1, "Kill test finished\n");
56         exit();
57     }
58 }
```

[Test 4의 실행 결과]

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

[분석]

- Thread에서 `kill` 을 호출했을 때, 정상적으로 Process가 종료되는지 확인하는 test이다.

⇒ 정상적으로 작동했다.

3-5. Test 5 - 종합 TEST (Test code from 정형수 교수님 OS)

[Test 5의 소스 코드]

```
1 int (*testfunc[NTEST])(void) = {  
2     racingtest,  
3     basictest,  
4     jointest1,  
5     jointest2,  
6     stresstest,  
7     exittest1,  
8     exittest2,  
9     forktest,  
10    exectest,  
11    sbrktest,  
12    killtest,  
13    pipetest,  
14    sleepertest,  
15    // stridetest,  
16};
```

[Test 5의 실행 결과]

```
4. stresstest start
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
24000
25000
26000
27000
28000
29000
30000
31000
32000
33000
34000
35000

stresstest exit
4. stresstest finish
```

[Test 5의 실행 결과]

```
5. exittest1 start
threadthread_exit ...
thread_exit ...
exittest1 exit
5. exittest1 finish
6. exittest2 start
6. exittest2 finish
7. forktest start
parent
parent
parent
parent
pareparent
parent
nt
parent
parent
child
parent
forktest exit
7. forktest finish
```

```
8. execetest start
echo is executed!
8. execetest finish
9. sbrktest start
sbrktest exit
9. sbrktest finish
10. killtest start
10. killtest finish
11. pipetest start
pipetest exit
11. pipetest finish
12. sleepertest start
sleepertest exit
12. sleepertest finish
$
```

[분석]

- **racingtest**

Thread가 `thread_create`로 생성되고, `thread_exit`으로 종료되고, `thread_join`으로 return 값을 잘 받아주는지를 test한다.

- **basictest**

Thread가 `thread_create`로 생성되고, `thread_exit`으로 종료되고, `thread_join`으로 return 값을 잘 받아주는지를 test한다.

- **jointest1**

Thread가 생성된 뒤에 `thread_join`을 통해서 thread가 기다리는 경우에 `thread_exit`이 정상적으로 작동하고, return 값을 잘 받아오는지 test한다.

- **jointest2**

Thread가 생성된 뒤에 `thread_exit`이 먼저 실행되었을 때, `thread_join`을 통해서 return 값을 잘 받아오는지 test한다.

- **stresstest**

Thread의 생성과 삭제를 반복하는 test이다.

- **exittest1**

Thread가 종료되지 않았을 때, process가 종료되는지를 test한다.

- **exittest2**

Thread가 `exit()`을 호출하는 test이다.

- **forktest**

Thread가 `fork()`을 실행해서 자식 process를 생성하는 test이다.

- **exectest**

Thread가 `exec()`을 실행했을 때, 다른 Thread들이 잘 정리되는지 test한다.

- **sbrktest**

Thread가 `sbrk`을 호출하는 test이다.

- **killtest**

Thread가 Process를 kill하는 test이다.

- **pipetest**

Thread 상태에서 pipe가 잘 작동하는지 확인하는 test이다.

- **sleeptest**

Thread가 `sleep`을 호출할 때, 정상적으로 작동하는지 확인하는 test이다.

⇒ 모든 test에 이상없이 통과했다.

3-5. Test Pmanager

[Pmanager의 실행 결과]

```
$ pmanager
Hello Pmanager!
>
```

[Pmanager의 실행 결과]

```
> execute usertests 100
> usertests starting
mp tests ok
createdelete test
createdelete ok
testunlink test
list
-----Process Information-----
|---name---|---pid---|---stack pages---|---current memory---|---memory limit---|
| init | 1 | 1 | 12288 | 0 |
| sh | 2 | 1 | 16384 | 0 |
| pmanager | 3 | 1 | 16384 | 0 |
| usertests | 5 | -100 | 16384 | 40960 |
-----Linkunlink ok
-----create test
```

```
> list
-----Process Information-----
|---name---|---pid---|---stack pages---|---current memory---|---memory limit---|
| init | 1 | 1 | 12288 | 0 |
| sh | 2 | 1 | 16384 | 0 |
| pmanager | 3 | 1 | 16384 | 0 |
-----
```

```
> memlim 3 200000
set process 3's memory limit : 200000 succeed
> list
-----Process Information-----
|---name---|---pid---|---stack pages---|---current memory---|---memory limit---|
| init | 1 | 1 | 12288 | 0 |
| sh | 2 | 1 | 16384 | 0 |
| pmanager | 3 | 1 | 16384 | 200000 |
-----memlim 3 323000
set process 3's memory limit : 323000 succeed
> memlim 2 100000
memlim 2 100000
memlim 2 100000 is less than allocated memory size : 16384
set process 2's memory limit : 16384 failed
> list
-----Process Information-----
|---name---|---pid---|---stack pages---|---current memory---|---memory limit---|
| init | 1 | 1 | 12288 | 0 |
| sh | 2 | 1 | 16384 | 0 |
| pmanager | 3 | 1 | 16384 | 323000 |
-----
```

```
> list
-----Process Information-----
|---name---|---pid---|---stack pages---|---current memory---|---memory limit---|
| init | 1 | 1 | 12288 | 0 |
| sh | 2 | 1 | 16384 | 0 |
| pmanager | 3 | 1 | 16384 | 323000 |
-----
```

[분석]

- shell에서 pmanager를 실행했더니 정상적으로 동작했다.

- “list”를 입력해서 process의 정보를 볼 수 있었다.

- “memlim {pid} {limit}”을 입력해서 memory의 제한을 설정했다.

- 성공, 실패여부를 출력했다.

- “kill {pid}”를 통해서 process를 종료했다.

- “execute {path} {stacksize}”를 통해서 process를 실행했다.

- 백그라운드 실행이 되기 때문에 도중에 “list”를 실행할 수 있었다.

- 잘못된 명령어를 입력하면 실행할 수 없음을 알려준다.

- “exit”를 입력하면 pmanager가 종료된다.

4. Trouble shooting

4-1. Design for Thread

Thread의 Design을 어떻게 할 것인가는 이 과제의 대주제이자, 세부적인 구현을 하기 위해서는 필수적으로 고민해야 하는 것이었다.

두 가지의 Design을 놓고 고민을 했는데, 하나는 process를 thread처럼 이용하는 것과 다른 하나는 thread와 process를 구분하는 것이었다.

4-1-1. Thread as process

- 장점

- 다른 자료구조를 만들지 않아도 된다.
- 함수의 logic을 크게 바꾸지 않아도 된다.

- 단점

- linked list를 구현해야 하기 때문에 pointer 연산이 필요하다.

4-1-2. Thread separated from process

- 장점

- pointer연산이 필요없다.
- 객체 지향 언어처럼 코딩하기 쉽다.

- 단점

- 다른 자료구조를 만들어야 한다.
- 함수의 logic이 세부적으로 조정이 필요하다.

4-1-3. 결론

Thread와 process를 분리한 디자인을 선택했고 새로운 자료구조인 `pthread_t`를 만들어서 진행했다.

다 구현한 뒤인 지금 생각해보면, 확실히 pointer연산만 해주면 되는 `proc` 자료구조를 thread처럼 이용하는 방법이 코딩측면에서 더 나아보인다.

그렇지만, linked list 구현을 어려워하기도 하고, `proc`을 thread의 자료구조로 사용한다는 디자인이 좋아보이지 않았다.

또한 각 부분을 모듈화해서 구현하거나, logic을 직관적으로 이해하기 힘들다고 생각했다.

그래서, thread를 위한 자료구조를 만들고, 그 thread가 process라는 옷을 입고 작업을 하는 디자인을 선택했다.

4-2. Process & Thread Group

구현을 제일 어렵게 했던, 부분으로 가장 애를 먹고, 이해하기 힘들었던 부분이다.

Thread에 대해서, Process에 대해서 제대로 이해하지 못했기 때문에 생긴 일이었다.

아래와 같은 질문들에 대해서 정확히 파악하고, 구현을 했어야 하는데, 그렇지 못했기 때문에 구현 방향이 지속적으로 바뀌면서 결국에는 깔끔하지 못한 구현이 되어버렸다.

1. Sub Thread가 fork한 것은 Thread인가 Process인가?
2. Main Thread가 fork한 것은 Thread인가 Process인가?
3. Sub Thread가 fork한 것은 누구의 것인가?
4. Main Thread가 fork한 것은 누구의 것인가?
5. Sub Thread가 thread_create한 것은 누구의 thread인가?
6. Main Thread가 thread_create한 것은 누구의 thread인가?
7. Sub Thread는 누가 메모리 정리를 해주는가?
8. Main Thread는 누가 메모리 정리를 해주는가?

4-2-1. `fork()`

`fork` 란, 새로운 process를 만들어주어, xv6 기준으로 완전히 새로운 virtual memory를 만드는 행위이다.

즉, `fork`를 하게 되면, 비록 `fork` 된 process가 부모 프로세스의 복사라고 하여도, 완전히 구분된 process라는 것이다.

그렇기 때문에, 1번 2번 질문에 대한 대답은 모두 Process이다.

단, 복사하는 내용에서 차이가 존재하는데, Sub Thread가 `fork` 하는 경우에는, Sub Thread의 내용만 복사된다. 반면에 Main Thread가 `fork` 하는 경우에는, Process 전체가 복사된다.

또한 이 상황에서 3,4번 질문으로 이어지게 되는데, `fork` 된 Process는 분명히 현재 Process의 Child이다. 그렇지만, 나의 디자인에서는 Process와 Thread가 구분되기 때문에, 어떤 Thread의 Child인지, 다르게 말하면, 이 Process의 parent가 어떤 Thread인지 구분해야 한다라는 문제가 발생한다.

따라서, 3,4번 질문의 답 역시 해당 Process를 `fork` 한 Thread의 것이고, 이를 구분해주기 위해서 Process에 `parent_thread`라는 변수를 추가했다.

4-2-2. `thread_create()`

`thread_create()` 란, 새로운 thread를 만들어주는 것이다.

Thread과 제이기 때문에 6번의 질문에 대한 답은 너무나도 당연하게, Main Thread의 것이다. 그러나 나는 처음에는 Thread를 Process에만 종속되는 존재라고 생각했다. 그렇기 때문에, Sub Thread가 `thread_create` 해도 그것은 Process에 종속되는(Main Thread에 종속되는) 것으로 생각했다. 이 부분에서 잘못 생각했기 때문에, `thread_join`에서도 잘못된 구현을 했었다.

잘못 생각한 이유는, 내 구현은 Process와 Thread가 나누어져있고, Thread가 Process를 입기 때문에, 당연히 Main Thread 아래에 있다고 생각했기 때문이다.

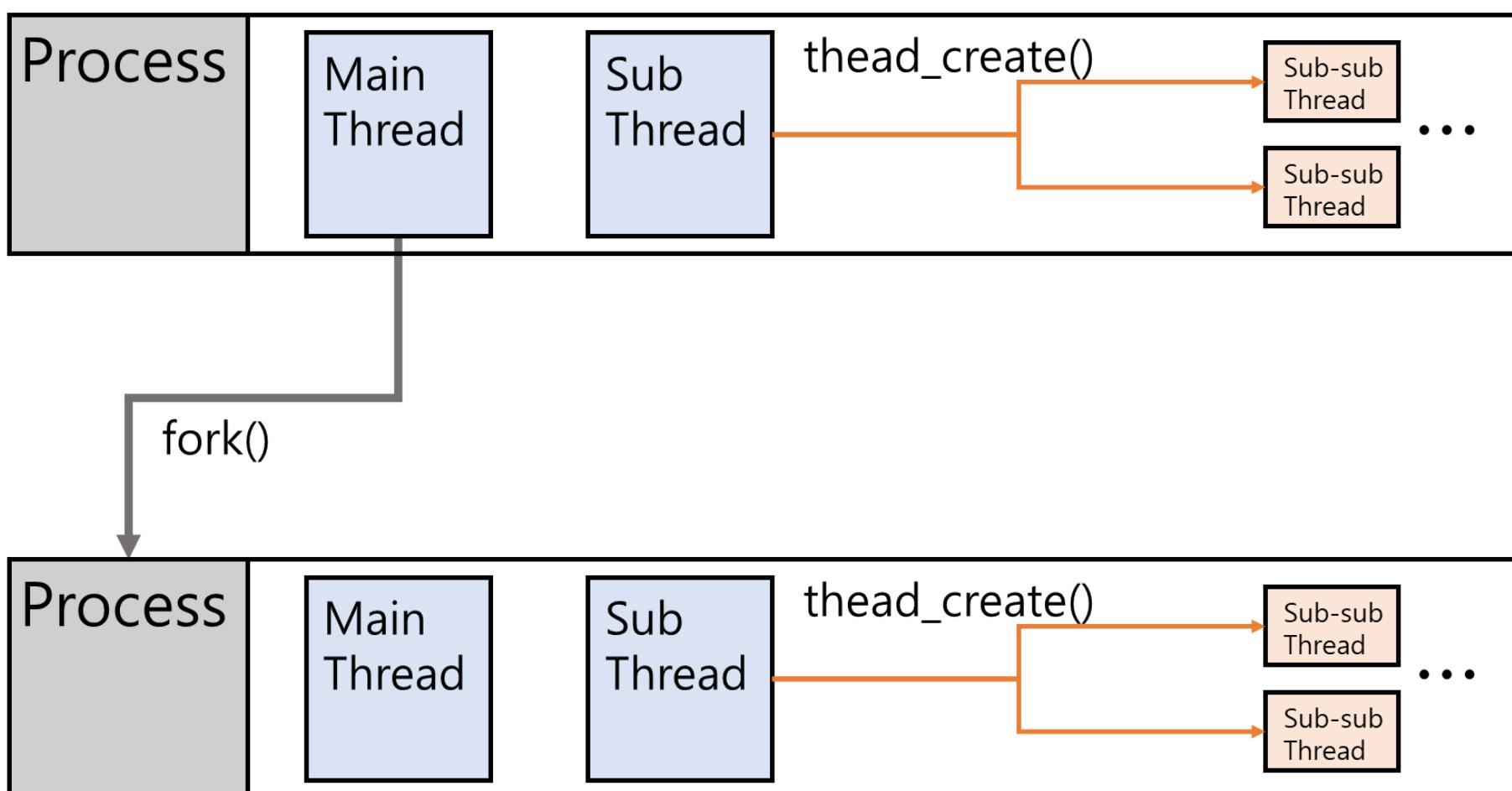
그러나 Sub Thread가 thread를 만들어 그 return 값을 이용해서 다른 작업을 할 수도 있었기 때문에, 타당하지 않은 구현이었고 무엇보다 Main Thread가 만들지 않은 Thread를 Main이 정리해줘야한다는 이상한 구조가 만들어진다.

그렇기 때문에 5번 질문에 대한 올바른 답변은 Sub Thread이다.

이렇게 된다면, 7,8번의 질문도 자연스럽게 해결된다. 왜냐하면, Child는 Parent가 정리하는 구조이기 때문에, 각각 만들어준 Thread가 정리해주면 된다.

4-2-3. Group

결론적으로 나는 다음과 같은 방식으로 Process와 Thread가 묶인다는 것을 제대로 이해하지 못하고 구현했기 때문에 어려움을 겪었다.



4-3. Free Page & Heap

종합 테스트의 Stress Test를 겪고 나니, Free Page를 관리해야 한다는 생각이 들었다.

처음에는 Page 단위로 관리하기 위해서 총 4GB의 메모리를 4KB로 나누어 배열로 만들어서 index를 1개의 page를 나타내는 방식으로 구현을 하려고 했으나, 너무 큰 숫자여서 xv6에서 컴파일이 되지 않았다.

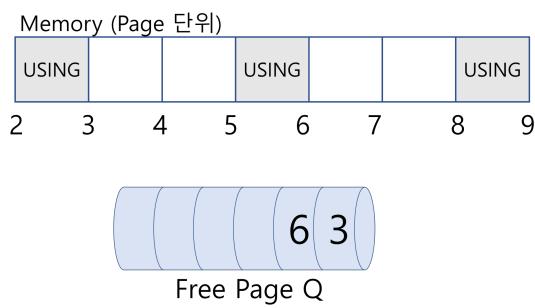
그래서 어떻게 구현을 해야 나의 옷입히기 디자인과 비슷한 색으로 구현할 수 있을까 고민했다.

포인터로 비어있는 공간을 가리키도록 하는 방법은 나의 디자인과 완전히 반대되는 방법이었기 때문에, 다른 방법이 필요했고, 결국에는 배열을 이용한 Circular Q를 통해서 free page를 관리했다. Circular Q에 stack이 free 될 때마다 하나씩 push하고, 새로 할당할 때는 이것을 이용한 뒤에 하나씩 pop하는 방법을 사용했다.

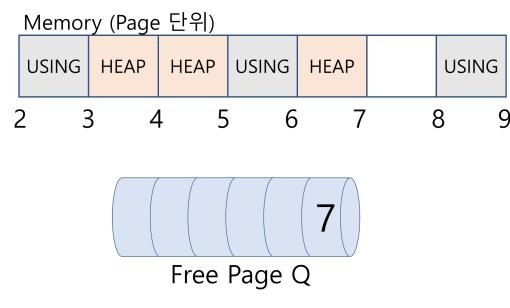
이렇게 Free Page를 관리하고 나니, Heap 영역도 사실 비어있는 공간에 할당할 수 있기 때문에, 이를 이용해서 할 수 있지 않을까라는 생각이 들었다.

그런데 이 부분은 구현이 너무 어려웠다. 왜냐하면, Q에 있는 빈 공간은 무작위적으로 들어가 있다. 다시 말하면, Q에 있는 순서는 virtual memory의 크기순서로 정렬되어 있지 않다.

이것은 사실 heap만을 할당할 때에는 문제가 되지 않는다. 왜냐하면, 떨어져있는 공간이라도, heap 영역이기 때문에 할당자체는 할 수 있기 때문이다. 문제는 이 위에 stack을 할당해야하기 때문에 발생한다. 예를 들어 다음과 같은 상황을 생각해보자.



[그림 1]



[그림 2]

[그림1]과 같은 상황에서 sbrk로 3개의 Page 만큼의 크기할당이 들어오게 되면 [그림2]와 같은 상황으로 바뀐다.

이 때, Q에 있는 빈 공간이 stack을 할당하기에 알맞지 않다는 문제가 발생한다.

더구나, 이 상황에서 8번 PAGE를 사용중인 Memory가 해제된다면, 7번 8번은 연속된 공간이기 때문에 하나로 합쳐져야한다.

이러한 문제를 효율적으로 해결하기 위한 방법을 찾지 못했다.

5. Review

과제를 다하고 리팩토링을 하는 과정에서, 코드가 너무 누더기처럼, 흔히 말하는 spaghetti code처럼 구현했다는 생각이 든다.

처음에 완전히 틀을 잡고 한 MLFQ 과제와는 다르게, 실수한 부분들이나 잘못 생각한 부분들이 있다보니 계속 구현방향이 바뀔 수 밖에 없었고, 그 과정에서 끼워맞추는 코딩을 하는 바람에 코드의 질이 많이 낮아졌다.

굳이 정의하지 않아도 되는 함수를 정의하기도 하고, coupling은 쓸데없이 높고, cohesion은 낮은 코딩이 완성되었다. 또한 계속 반복해서 초기화를 하거나, 쓸데없이 초기화를 하는 등의 코딩을 했다.

주석이나 commit도 이전과는 다르게 세세하지 않게 쓰는 모습을 발견했다.

과제가 어렵기는 했지만, 코딩하는 방법을 제대로 지키지 못한 것이 아쉽다.

그래서 리팩토링을 하기 보다는 그대로 놔두기로 했다.

나중에라도 다시 과제를 리뷰하면서 찾아볼 때, 이런식으로 구현을 했었구나, 지금은 어떻게 다른가를 복기해보기 위한 용도로 쓰기 위해서다.

그리고 과제가 너무 어려워서 올었다.