

# OS Project : 최종 발표

---

miniOS 설계 및 분석

4조

최서영 강성욱 박주은 백현우

# 0. Index

4조

---

1. OS 구현 환경 선정 및 주제선정
2. 역할분담
3. Scheduler 설계
  - a. FCFS
  - b. MLFQ
4. Scheduler 검증
  - a. Scheduler 별 검증
  - b. Scheduler 성능 비교
5. Thread 구현
6. Lesson learned

# 1. 역할분담

4조

팀원	최서영	박주은	강성욱	백현우
역할	Scheduler-MLFQ 구현	Scheduler-MLFQ 구현	Scheduler-FCFS 구현	Scheduler-FCFS 구현
모임일자	수행 결과			
04 월 17 일	OS 구현 계획 작성 및 주제 선정 : xv6 scheduler algorithm 구현			
04 월 24 일	Xv6 전체 동작 분석 및 기본으로 구현되어있는 RR 분석			
05 월 01 일	Systemcall -> ps() 작성	Systemcall -> memsize() 작성	Systemcall -> getread() 작성	Systemcall -> trace() 작성
05 월 06 일	MLFQ systemcall 작성 및 MLFQ 구현		FCFS 구현 완료	
05 월 13 일	중간 보고서 작성 및 MLFQ gaming shcheduler 구현		중간 보고서 작성 및 thread 구현 시도	
05 월 20 일	성능비교, queue 이동 code 작성		Thread 구현 및 검증	
06 월 03 일	프로젝트 최종 발표 준비 및 최종 보고서 작성			
06 월 10 일	결과 분석 및 최종 보고서 작성 & 발표자료 준비			

## 2. OS 구현 환경 설정 및 주제선정

4조

### ▷ OS 구현 환경 : xv6

- OS의 기본적인 구성이 구현됨 → 더욱 깊은 내용에 대한 설계 진행 가능
- Reference 다수 존재

### ▷ 주제선정

- Scheduler 구현 및 성능 분석
- Thread 구현 및 검증

# 3. Scheduler 설계

FCFS

---

## [FCFS Rule]

- 1) 먼저 생성된 process가 먼저 스케줄링 된다.
- 2) 스케줄링 된 process는 종료되기 전까지 switch-out 되지 않는다.
- 3) process가 스케줄링 된 이후 100 ticks가 지날때까지 종료 혹은 SLEEPING 상태로 전환되지 않으면 강제 종료한다.
- 4) 실행중인 process가 SLEEPING 상태로 전환되면 다음으로 생성된 process가 스케줄링된다.
- 5) SLEEPING 상태에서 깨어난 process가 먼저 생성된 process면 먼저 스케줄링된다.

# 3. Scheduler 설계


MLFQ

## [MLFQ Rule]

- 1) 스케줄링을 위한 5개의 큐가 존재하고, 우선순위가 높은 큐부터 TQ의 값을 {2,4,8,16,32}로 부여하였다.
- 2) 큐 내부의 우선순위는 존재하지 않으며, 각 큐에서 Time Quantum에 따라 Round Robin 스케줄링이 진행된다.
- 3) 새로운 프로세스가 시스템에 들어가면 가장 높은 우선순위를 부여한다.
- 4) 일정 시간 이상 하위 큐에 머무는 프로세스는 한 단계 위의 큐로 옮겨진다.

```
struct proc *queue[5][NPROC];  
...  
struct proc *p = queue[i][j];
```

[ 2차원 큐 배열 사용 예시 ]



```
for(int j=0; j <= q_tail[i]; j++)  
{  
    struct proc *p = queue[i][j];  
    int age = ticks - p->enter;  
    if(age > 30)  
    {  
        remove_proc_from_q(p, i);  
        add_proc_to_q(p, i-1);  
    }  
}
```

# 4. Scheduler 검증

FCFS

```
// Test 1
printf(1, "\nWithout sleep & yield\n");

p = create_child();
for(int i = 0; i < NUM_LOOP; i++){
    if (p == 0) printf(1, "process %d\n", getpid());
}

exit_child(p);
```

Without sleep & yield

process 4
process 4
process 4
process 5
process 5
process 5
process 6
process 6
process 6
process 7
process 7
process 7
process 8
process 8
process 8

```
// Test 3
printf(1, "\nWith sleep\n");

p = create_child();

for (int i = 0; i < NUM_LOOP; i++) {
    if (p == 0) {
        printf(1, "process %d\n", getpid());
        printf(1, "process %d\n", getpid());
        printf(1, "process %d\n", getpid());
        printf(1, "\n");

        sleep(1);
    }
}

exit_child(p);
```

With sleep

process 14
process 14
process 14

process 15
process 15
process 15

process 15
process 14
process 14
process 14

process 16
process 16
process 16

process 14
process 14
process 14

process 17
process 17
process 17

process 15
process 15

process 18
process 18
process 18

process 16
process 16
process 16

process 17
process 17
process 17

process 15
process 15
process 15

process 18
process 18
process 18

process 16
process 16
process 16

process 18
process 18
process 18

process 17
process 17
process 17

process 18
------------

## 4. Scheduler 검증

### MLFQ

#### ▷ MLFQ queue 동작 검증

```
Process: PID 13 :9 Finished
```

PID	State	r_time	w_time	cur_q	q0	q1	q2	q3	q4
3	SLEEPING	3	0	1	2	0	0	0	0
4	RUNNABLE	628	1663	4	2	4	8	16	576
5	RUNNABLE	430	1860	4	2	4	8	16	384
6	RUNNABLE	298	1992	4	2	4	8	16	256
7	RUNNABLE	232	2058	4	2	4	8	16	192
8	RUNNABLE	166	2124	4	2	4	8	16	128
9	RUNNABLE	133	2156	4	2	4	8	16	96
10	RUNNABLE	101	2188	4	3	4	8	16	64
11	RUNNABLE	67	2221	4	2	4	8	16	32
12	RUNNABLE	34	2254	4	2	4	8	16	0
13	RUNNING	0	2288	0	0	0	0	0	0

```
[Terminate] --> Total Time for pid [13] is [2289]
```

- TQ 값을 부여한대로,  
{2, 4, 8, 16, 32} 값이 초과되면 다음 queue로 이동하여 동작함
- queue4에 처리하지 못한 프로세스가 쌓이며 starvation 발생  
⇒ Aging 기법을 추가적으로 구현하여 starvation 해결

```
Process with PID 5 is removed from Queue 4 at 744  
Process with PID 5 added to Queue 3 at 744
```

- 검증 시뮬레이션 파일은 반복문을 통해 높은 PID일수록 더 긴 sleep함수 동작을 함.

⇒ PID4는 가장 먼저 RUNNABLE 상태가 되며 CPU bound 임에도 불구하고 첫번째로 출력됨. sleep이 모두 동작한 PID13은 이후 첫 RUNNING 상태로 바뀌자마자 종료됨.

```
Process: PID 4 :0 Finished  
Process: PID 13 :9 Finished
```



## 4. Scheduler 검증

MLFQ

▷ SLEEPING 상태 아웃 이슈

PID	State	r_time	w_time
3	SLEEPING	1	1
4	RUNNABLE	3	205
5	RUNNABLE	1	207
6	RUNNABLE	1	207
7	RUNNABLE	1	207
8	SLEEPING	X 1	207
9	RUNNING	0	208
10	RUNNABLE	0	208
11	RUNNABLE	0	208
12	RUNNABLE	0	208
13	RUNNABLE	0	207

PID8 state : SLEEPING

PID	State	r_time	w_time
3	SLEEPING	1	1
4	RUNNABLE	3	206
5	RUNNABLE	1	208
6	RUNNABLE	1	208
7	RUNNABLE	1	208
8	RUNNABLE	X 1	208
9	SLEEPING	1	208
10	RUNNING	0	209
11	RUNNABLE	0	209
12	RUNNABLE	0	209
13	RUNNABLE	0	208

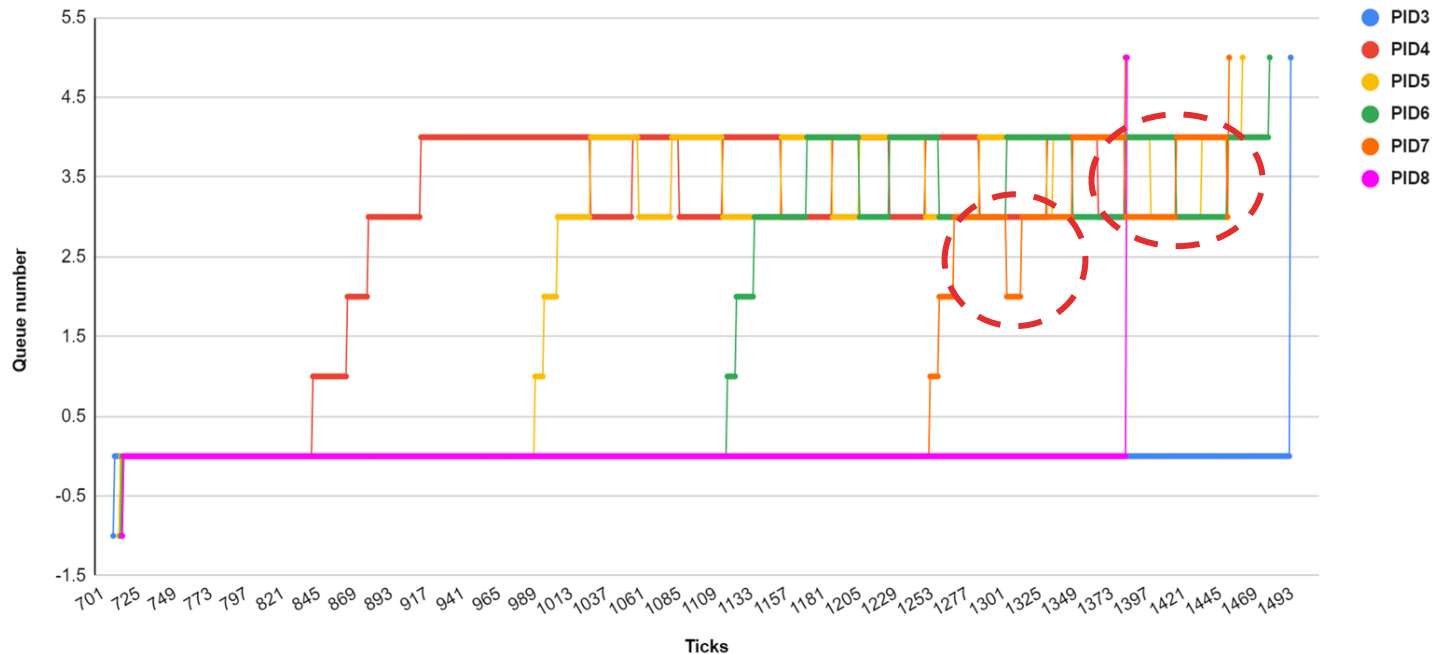
PID8 state : RUNNING

sleep(200) 시간 조건 만족하지 않고  
RUNNABLE로 상태 변환

# 4. Scheduler 검증

MLFQ

[ fork()를 통해 생성한 프로세스의 PID 별 Queue Timeline ]



- 프로세스들의 Queue 이동 및 Aging에 의한 큐 상승 관찰 가능

## 4. Scheduler 검증

성능비교

Flag	Run time	Wait time	IO time	Total
RR	2	13	3141	3156
FCFS	2	2	5503	5507
*MLFQ	2	1	2865	2868

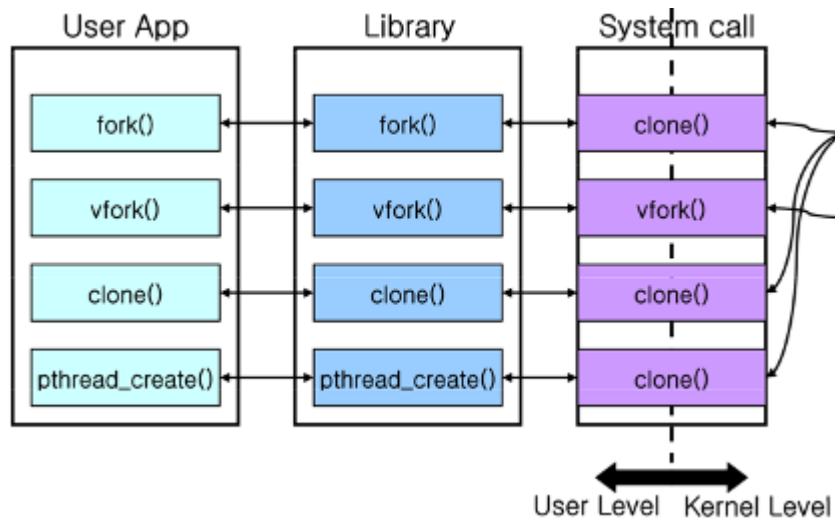
# 5. Thread 구현

Thread concept

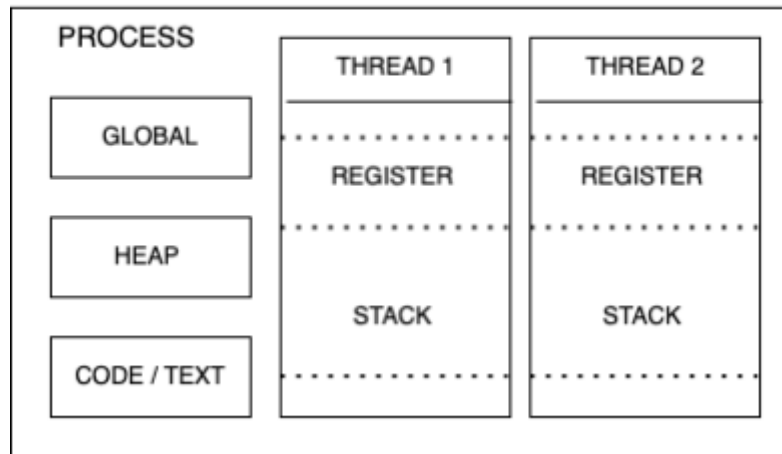
## ▷ Thread concept

linux clone system call 모방

process의 memory space 공유



## Shared virtual address space

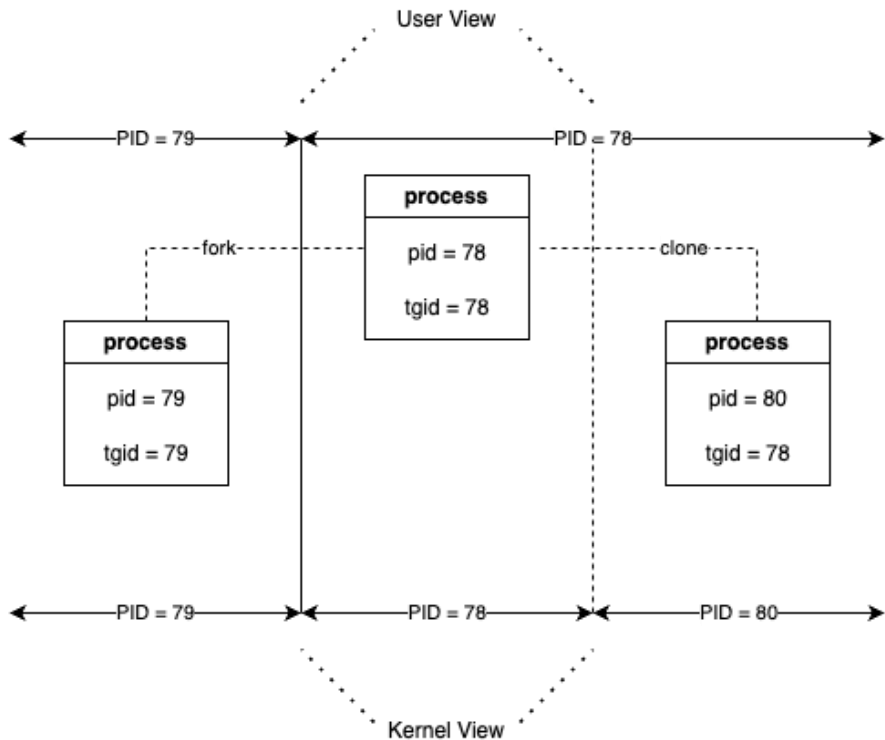


# 5. Thread 구현

## Clone system call

### ▷ Clone system call flag

- CLONE\_FS
- CLONE\_FILES
- CLONE\_THREAD
- CLONE\_VM

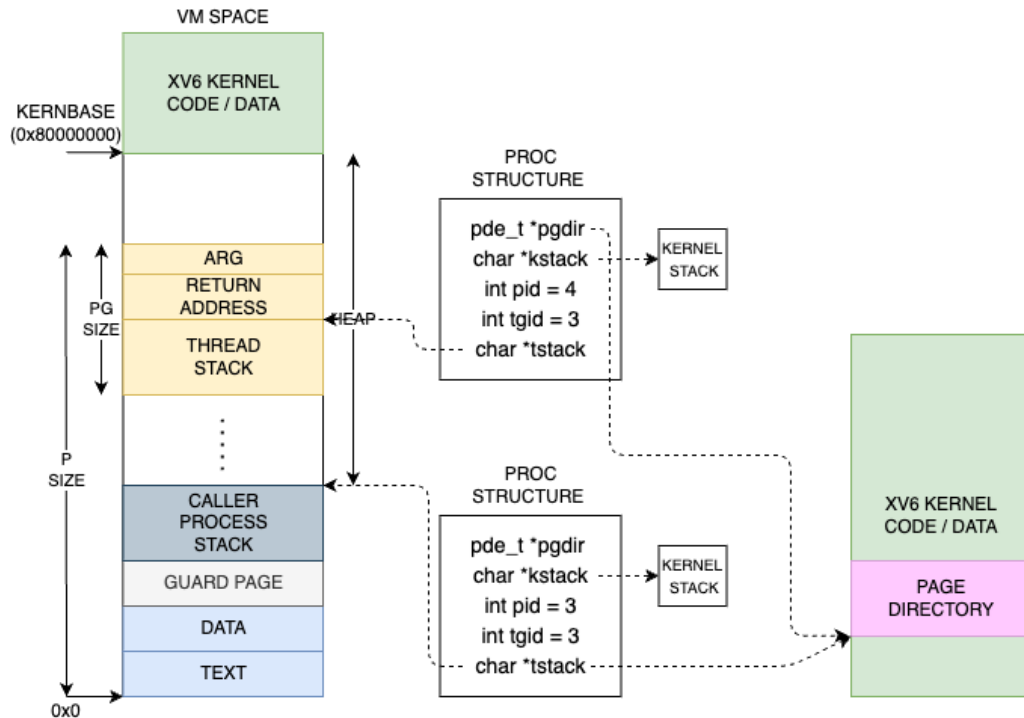


[ CLONE\_THREAD ]

# 5. Thread 구현

Clone system call

▷ Clone system call 동작 시 memory 내 변화



## 5. Thread 구현

Join system call

---

### ▷ join system call

- thread를 찾아서 실행이 완료될 때까지 부모 process sleep
- zombie thread 발견 시 자원 할당 해제

# 5. Thread 구현

User thread library

---

## ▷ thread\_create()

clone system call을 내부적으로 호출하는 user library 함수

사용자가 직접 stack 할당 x

## ▷ thread\_join()

join system call을 내부적으로 호출하는 user library 함수

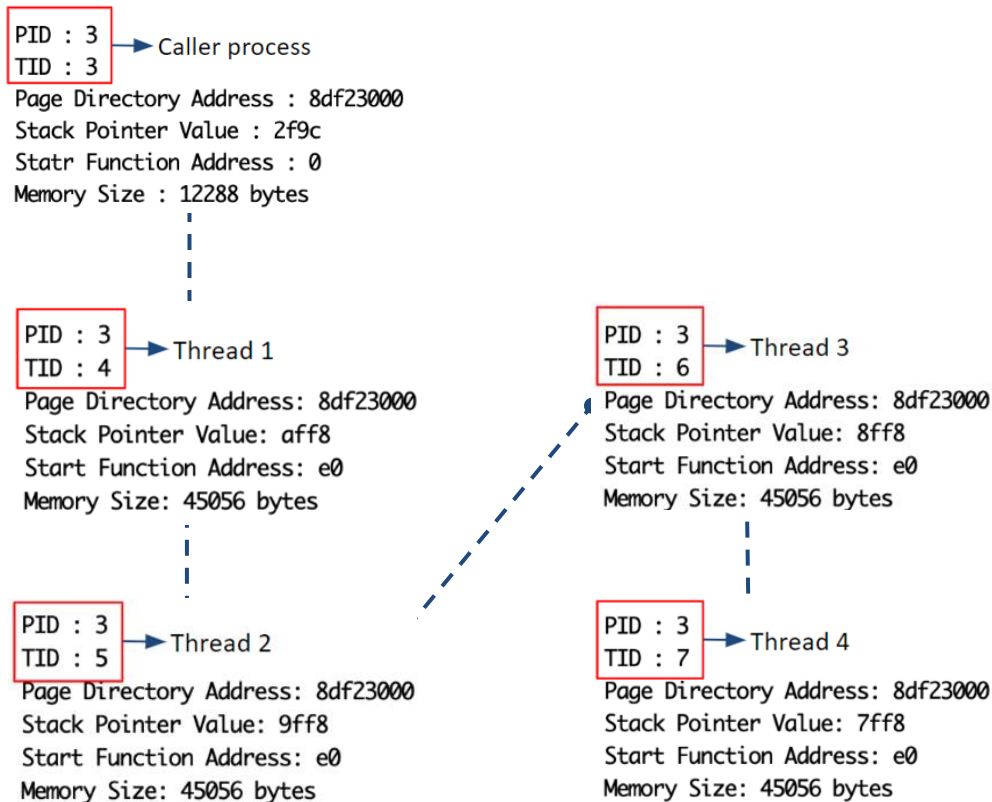


# 5. Thread 구현

구현 결과

▷ PID → Thread Group ID로 전부 동일

▷ TID → Thread ID로 고유 값 존재



# 5. Thread 구현

구현 결과

## ▷ Page Directory Address

→ 0x8df23000

## ▷ Stack Pointer

→ PAGE SIZE 만큼의 변화

PID : 3

TID : 3

Page Directory Address : 8df23000

Stack Pointer Value : 2f9c

Start Function Address : 0

Memory Size : 12288 bytes

PID : 3

TID : 4

Page Directory Address: 8df23000

Stack Pointer Value: aff8

Start Function Address: e0

Memory Size: 45056 bytes

PID : 3

TID : 5

Page Directory Address: 8df23000

Stack Pointer Value: 9ff8

Start Function Address: e0

Memory Size: 45056 bytes

PID : 3

TID : 6

Page Directory Address: 8df23000

Stack Pointer Value: 8ff8

Start Function Address: e0

Memory Size: 45056 bytes

PID : 3

TID : 7

Page Directory Address: 8df23000

Stack Pointer Value: 7ff8

Start Function Address: e0

Memory Size: 45056 bytes

PG SIZE 4096

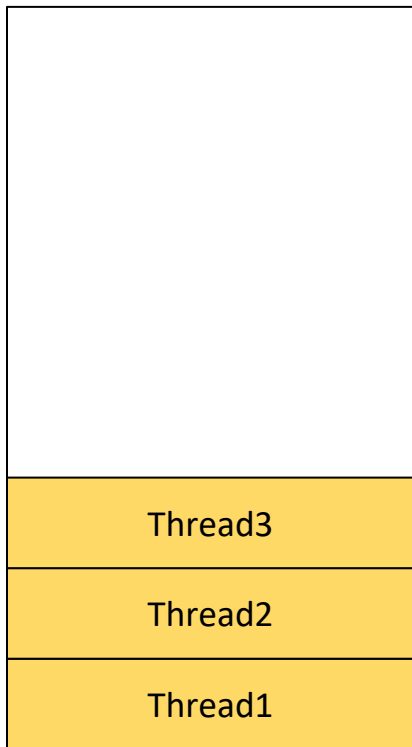
PG SIZE 4096

PG SIZE 4096

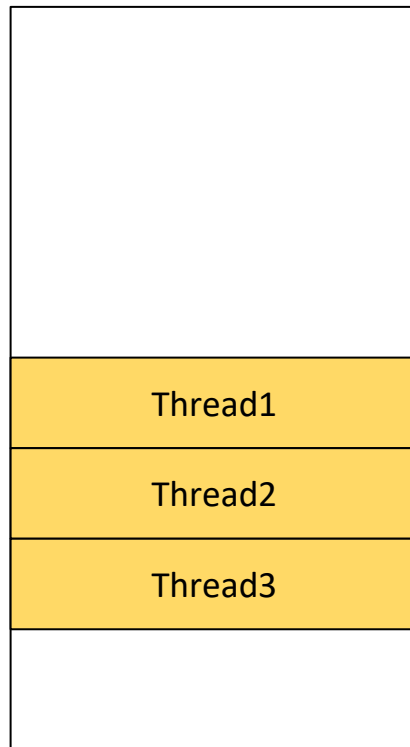
# 5. Thread 구현

malloc challenge

HEAP



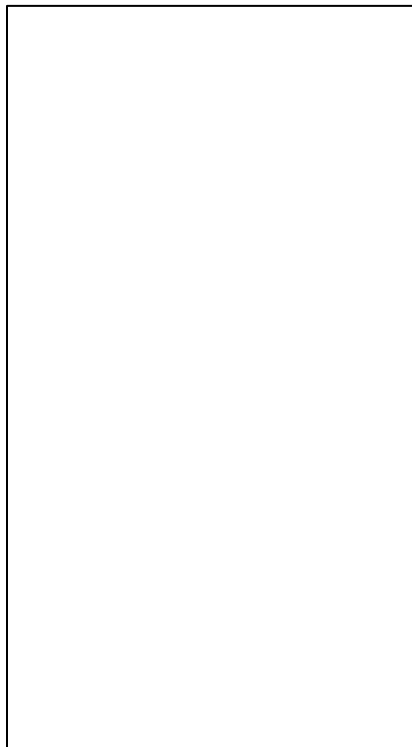
HEAP



# 5. Thread 구현

malloc challenge

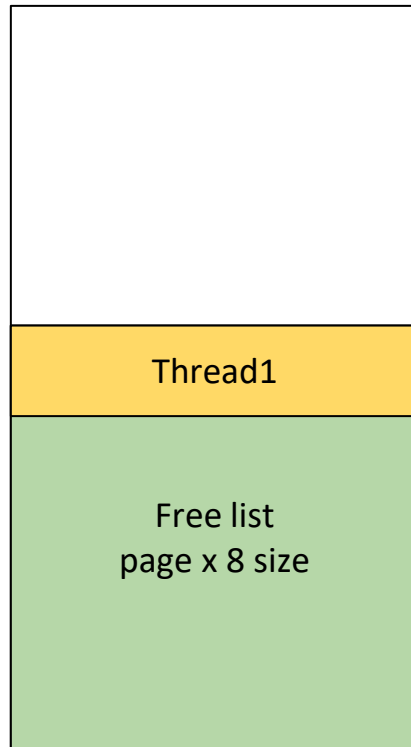
HEAP



malloc first call



HEAP



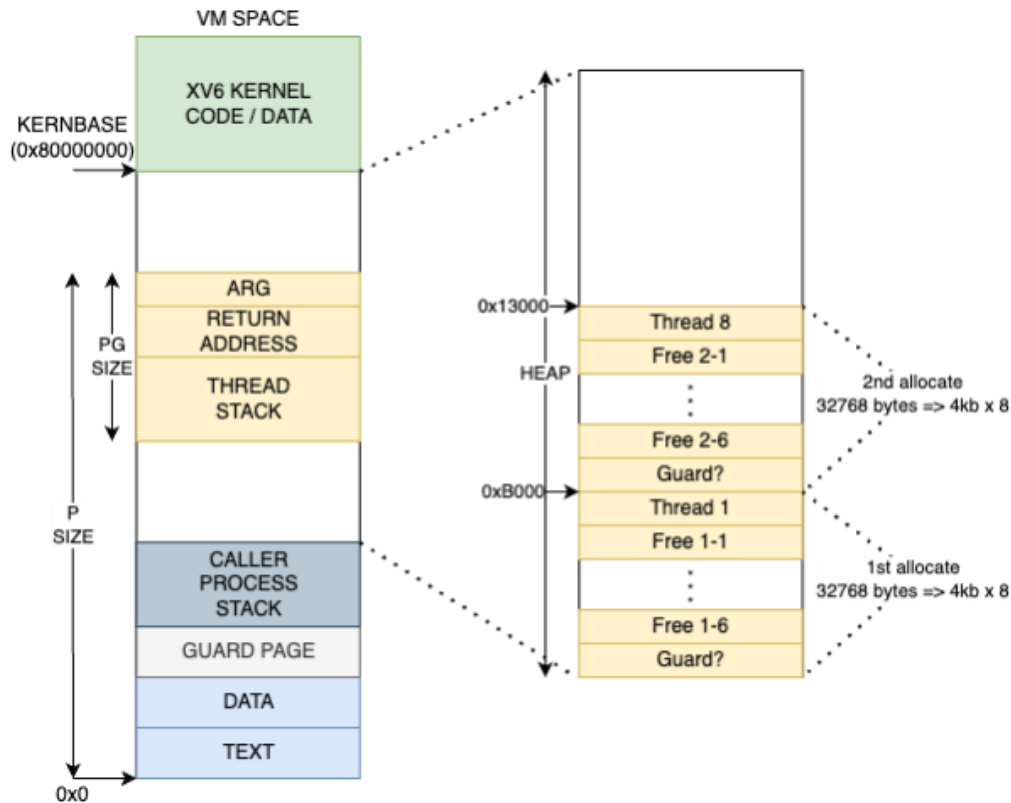
# 5. Thread 구현

malloc challenge

```
current size : 12288
32768 nsize
32768byte size up -> sz+n = 45056
PID : 3
TID : 4
```

memory  
추가 할당

```
current size : 45056
32768 nsize
32768byte size up -> sz+n = 77824
PID : 3
TID : 11
```



## 6. Lesson Learned

4조

---

- CPU의 프로세스 스케줄링 메커니즘과 커널에서 동작 방식을 이해
- 동기화를 위해 필수적인 lock 동작과 API 내부에서의 OS의 동작에 대해 탐구

# OS Project : 최종 발표

---

miniOS 설계 및 분석

4조

최서영 강성욱 박주은 백현우