Operating System Project

최종 보고서

과목명	운영체제
교수명	**박재근**교수님
학과	전자정보공학부 it 융합전공
팀명	4 조
팀장	**최서영**(20211555)
팀원	**강성욱**(20192562), **박주은**(20212973), **백현우**(20192596)
제출일자	2024.06.12(수)

목차

Ⅴ. Thread 구현을 위한 xv6 동작 분석

- 분석 목적
- Various data structure
- Trap handling
- Process creation
- Paging
- Context switching

Ⅵ. Thread 구현

- Clone system call
  - 동작 설명
  - 구현
  - 동작 상세
- Join system call
  - 동작 설명
  - 구현
- User thread library
- Thread 구현 검증
- Thread stack challenge

Ⅶ. Project 결론

Ⅰ. 프로젝트 개요

- 주제 선정

본 프로젝트에서는 xv6 에서 CPU scheduler 알고리즘 설계를 구현하는 것을 목표로 한다. 기존 xv6 가 동작하는 Round Robin 스케쥴러 방식외에 FCFS 와 MLFQ 스케쥴러를 추가로 구현한다. 이들의 시뮬레이션을 진행하여 이론과 비교를 통한 스케쥴러의 동작을 검증하고, 세 가지 스케줄러의 성능 분석을 통해 프로젝트를 마무리 한다.

- OS 구현 환경 선정

xv6 는 MIT 에서 만든 교육용 운영체제이다. xv6 와 비슷한 운영체제로 v6 와 pintos 가 있는데, v6 는 xv6 의 초기버전이기 때문에 직관적인 코드 구성이 특징이지만 현재 운영체제와 다른 부분이 있어 프로젝트에 사용될 구현체로는 적합하지 않다고 생각했다. 또한 pintos 는 miniOS 와 비슷하게 코드만 동작 가능한 최소한의 설계로 되어 있어 운영체제 대부분의 기능을 직접 구현해야 한다는 문제점이 있었다.

따라서, 이번 프로젝트는 운영체제의 기본적인 부분들이 코드로 구현되어있는 xv6 구현체에 프로젝트를 진행하기로 결정하였다. 코드를 보며 운영체제의 동작을 공부하기 좋고, 이에 대한 textbook 또한 존재하기 때문에 운영체제를 처음 접하는 팀원들이 학습과 동시에 동작 구현을 진행하기에 적절할 것이다.

- 수행일지 및 역할분담

<table>
<tr><td>팀원</td><td>최서영</td><td>박주은</td><td>강성욱</td><td>백현우</td></tr>
<tr><td>역할</td><td>Scheduler-
MLFQ 구현</td><td>Scheduler-
MLFQ 구현</td><td>Scheduler-FCFS 구현</td><td>Scheduler-FCFS 구현</td></tr>
<tr><td>모임일자</td><td colspan="4">수행 결과</td></tr>
<tr><td>04 월 17 일</td><td colspan="4">OS 구현 계획 작성 및 주제 선정 : xv6 scheduler algorithm 구현</td></tr>
<tr><td>04 월 24 일</td><td colspan="4">Xv6 전체 동작 분석 및 기본으로 구현되어있는 RR 분석</td></tr>
<tr><td>05 월 01 일</td><td>Systemcall ->
ps() 작성</td><td>Systemcall ->
memsize() 작성</td><td>Systemcall ->
getread() 작성</td><td>Systemcall ->
trace() 작성</td></tr>
<tr><td>05 월 06 일</td><td colspan="2">MLFQ systemcall 작성 및
MLFQ 구현</td><td colspan="2">FCFS 구현 완료</td></tr>
<tr><td>05 월 13 일</td><td colspan="2">중간 보고서 작성 및
MLFQ gaming shcheduler 구현</td><td colspan="2">중간 보고서 작성 및
thread 구현 시도</td></tr>
<tr><td>05 월 20 일</td><td colspan="2">성능비교, queue 이동 code 작성</td><td colspan="2">Thread 구현 및 검증</td></tr>
<tr><td>06 월 03 일</td><td colspan="4">프로젝트 최종 발표 준비 및 최종 보고서 작성</td></tr>
<tr><td>06 월 10 일</td><td colspan="4">결과 분석 및 최종 보고서 작성 & 발표자료 준비</td></tr>
</table>

Ⅱ. xv6 동작 분석 및 system call 구현

- xv6 프로세스 동작 분석

xv6 single operating system 은 Unix kernel 의 기능을 최소화하여 만든 운영체제이다. 현대 대부분의 운영체제는 Unix 와 비슷한 interface 를 사용한다. 전체적인 동작을 확인하면서, scheduling 구현과 연관된 부분에 집중하여 분석을 진행하였다.

첫 번째 user process 는 `userinit` 함수를 통해 생성된다. 이 함수안에서는 프로세스 생성을 위해 `allocproc`을 호출하는데, 프로세스 테이블에서 UNUSED 상태인 struct proc 을 할당한다. 여기서 프로세스 테이블은 초기에 일정 크기의 메모리를 할당받아서 많은 프로세스들을 미리 생성해 놓는 프로세스 풀의 역할을 한다. 그리고 프로세스 테이블에 있는 프로세스들의 초기 상태는 UNUSED 가 된다. 즉, 사용이 가능한 프로세스를 의미한다. allocproc 은 프로세스 테이블에서 사용 가능한 새로운 프로세스를 찾으면, 초기화를 진행한다. 초기화는 UNUSED 를 EMBRYO 상태로 바꾸고, PID 를 할당한다. 그리고 프로세스 고유의 스택 영역을 위해서 커널 스택의 일부를 할당한다. 만약, 메모리 할당이 실패하면 프로세스의 상태를 `UNUSED`로 바꾸고 실패했다는 의미로 0 을 리턴한다.

하나의 프로세스마다 user stack 과 kernel stack 이 존재한다. 프로세스가 user mode 로 실행중일 때는 user stack 만을 사용하고 kernel stack 은 empty 이다. 반면 프로세스가 kernel mode 일 user stack 은 empty 가 아니다. 즉 커널은 유저 프로세스가 실행중이다가 H/W 자원을 필요로 하여 잠시 들리는 영역이라고 생각할 수 있다. 주된 작업은 user area 에서 이루어지고 필요할 때마다 kernel area 에서 작업하는 것이다. 프로세스가 kernel mode 로 진입하면 user mode 의 상태를 kernel stack 에 저장한 후 kernel stack 을 사용한다. 다시 user mode 로 복귀할 때 kernel stack 에 저장되어 있던 user mode 의 상태를 그대로 이용하면 된다. xv6`에서 새로운 프로세스가 생성되면, 그 프로세스는 무조건 유저 영역까지 가게된다. 그 이유는 `forkret`과 `trapret` 함수 때문이다.

- system call 구현 및 검증

새로운 시스템 콜을 구현하기 위해 기존의 시스템 콜이 어떻게 구현되어 있는지 분석하였다. xv6 에서는 세 가지 종류의 시스템 콜 방식을 정리할 수 있었다. 첫 번째는 인자가 없는 시스템 호출로, 이는 sysproc.c 의 uptime() 함수가 그 예시이다. 두 번째는 문자열 및 정수 인자가 있는 시스템 호출로, sysfile.c 의 open() 함수가 이에 해당한다. 세 번째는 구조체 인자가 있는 시스템 호출로, 이는 fstat() 함수에서 볼 수 있다.

시스템 콜 정의는 user.h 파일에서 이루어지며, 관련 파일들은 다음과 같다. usys.s 파일은 시스템 콜 리스트를 정의하고, syscall.h 파일은 시스템 콜 번호를 매핑한다. syscall.c 파일은 시스템 콜 인수를

구분 분석하는 함수와 실제 시스템 콜 구현에 대한 포인터를 포함한다. sysproc.c 파일은 프로세스 관련 시스템 콜을 구현하고, 프로세스 정보를 추적하기 위해 struct proc 구조를 정의한다. 마지막으로 proc.c 파일은 프로세스 간의 스케줄링 및 컨텍스트 스위칭을 수행하는 함수를 포함한다.

xv6 의 커널에 대한 이해를 마친 후, 팀원별로 ps(), memsize(), trace(), getread() 시스템 콜을 구현해 보았다. 다음은 memsize() 시스템 콜의 구현 및 검증 결과이다. memsize()는 호출한 프로세스의 메모리 사용량을 출력하는 시스템 콜이다.

1) user.h -> int memsize(void)를 추가하여 system call 정의

```
21 int dup(int);
22 int getpid(void);
23 char* sbrk(int);
24 int sleep(int);
25 int uptime(void);
26 // added system calls
27 int memsize(void); // print memory used
```

2) usys.s -> memsize()추가하여 system call list 에 추가

```
29 SYSCALL(sbrk)
30 SYSCALL(sleep)
31 SYSCALL(uptime)
32 //added SYSCALL
33 SYSCALL(memsize)
```

3) syscall.h -> SYS_memsize 22 추가하여 번호 매핑

```
20 #define SYS_link   19
21 #define SYS_mkdir  20
22 #define SYS_close  21
23 //added define SYS
24 #define SYS_memsize      22
```

4) syscall.c -> memsize() 구현에 대한 pointer

```
131 [SYS_unlink]  sys_unlink,
132 [SYS_link]    sys_link,
133 [SYS_mkdir]   sys_mkdir,
134 [SYS_close]   sys_close,
135 [SYS_memsize] sys_memsize,
```

```
105 extern int sys_wait(void);
106 extern int sys_write(void);
107 extern int sys_uptime(void);
108 // added extern int
109 extern int sys_memsize(void);
```

5) sysproc.c -> memsize() 구현

```
93 // return memory size - added syscall
94 int
95 sys_memsize(void)
96 {
97   uint size;
98   size = myproc() -> sz;
99   return size;
100 }
```

6) memsize test shell 구현

```c
#include "types.h"
#include "stat.h"
#include "user.h"

#define SIZE 2048

int main(){
        int msize = memsize();
        printf(1, "The process is using %dB\n", msize);

        char *tmp = (char *)malloc(SIZE * sizeof(char));

        printf(1, "Allocating more memory\n");
        msize = memsize();
        printf(1, "The process is using %dB\n", msize);

        free(tmp);
        printf(1, "Freeing memory\n");
        msize = memsize();
        printf(1, "The process is using %dB\n", msize);

        exit();
}
```

7) 실행 결과

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ memsize
The process is using 12288B
Allocating more memory
The process is using 45056B
Freeing memory
The process is using 45056B
$
```

위 과정을 바탕으로 systemcall 을 구현하기 위한 과정과 동작을 분석하였으며, 실제로 system call 을 추가하고 검증을 완료하였다.

Ⅲ. CPU scheduler 설계

- FCFS 설계

FCFS(First-Come, First-Served) 스케줄링은 먼저 생성된 프로세스가 먼저 스케줄링되는 방식으로 설계되며, 구현에 있어 다음과 같은 원칙을 적용했다.
먼저 생성(fork())된 process 가 먼저 스케줄링되며 스케줄링된 process 는 종료되기 전까지 switch-out 되지 않는다. 그러나, 만약 스케줄링된 process 가 스케줄링된 이후 100ticks 이 지나도 종료되지 않거나 SLEEPING 상태로 전환되지 않는다면, 강제로 종료된다. 실행 중인 process 가 SLEEPING 상태로 전환되면, 다음으로 생성된 프로세스가 스케줄되며, SLEEPING 상태에서 깨어난 프로세스는 만약 먼저 생성된 프로세스라면 다시 우선적으로 스케줄링된다. 이를 통해 FCFS 방식의 순서를 유지하면서도 효율적인 스케줄링을 보장한다. 이러한 규칙들을 통해 FCFS 스케줄링은 단순하면서도 공정한 프로세스 관리 방식을 제공한다. 프로세스는 생성된 순서대로 처리되며, 각 프로세스는 충분한 실행 시간을 보장받게 된다.

1. 프로세스 생성 순서 구별

proc.c 의 allocproc() 함수에서 p->pid = nextpid++를 통해 프로세스가 생성될 때마다 pid 가 증가하도록 구현하였다. 이를 통해 pid 값을 기준으로 프로세스의 생성 순서를 확인할 수 있다.

```
324  scheduler(void)
325  {
326    struct proc *p;
327    struct proc *tmp; //FCFS
328    struct cpu *c = mycpu();
329    c->proc = 0;
330
331    for(;;){
332      // Enable interrupts on this processor.
333      sti();
334
335      // Loop over process table looking for process to run.
336      acquire(&ptable.lock);
337      p = 0;
338      int pid = 99947;
339      for(tmp = ptable.proc; tmp < &ptable.proc[NPROC]; tmp++){
340        if(tmp->state != RUNNABLE)
341          continue;
342        if(tmp->pid < pid && tmp->pid > 0){
343          pid = tmp->pid;
344          p = tmp;
345        }
346      }
347
348      if(p){
349        c->proc = p;
350        switchuvm(p);
351        p->state = RUNNING;
352
353        swtch(&(c->scheduler), p->context);
354        switchkvm();
355        c->proc = 0;
356      }
357      release(&ptable.lock);
358      }
```

Scheduler( )에서 ptable 에서 pid 값이 작은 프로세스부터 CPU 를 점유할 프로세스로 설정한다. if (p > 0) 조건을 통해 ptable 의 빈 프로세스를 걸러내어 유효한 프로세스만을 고려한다. 이를 통해 FCFS 방식의 순서를 유지하면서 공정하게 프로세스를 스케줄링할 수 있다.

2. CPU 점유 및 타임아웃

기존 xv6 에서는 1 틱마다 timer_interrupt 를 통해 trap() 함수가 호출된다. trapframe 이 timer_interrupt 인 경우, yield()를 통해 context switching 이 이루어진다. 이러한 메커니즘을 활용하여, trap.c 파일에서 틱이 100ticks 를 넘는 조건을 추가하여 프로세스를 종료하는 로직을 삽입하였다. 이를 통해 프로세스가 일정 시간 동안 CPU 를 점유하는 것을 제한하고, 시스템의 응답성을 유지할 수 있다.

3. Runnable 상태로 변경된 프로세스를 스케줄러에 알리기

기존 xv6 에서는 timer_interrupt 마다 wakeup() 함수를 호출하여 SLEEPING 상태의 프로세스를 RUNNABLE 상태로 변경한다. 이를 확장하여 wakeup_ps 변수를 도입하고, wakeup()이 발생할 때마다 해당 변수를 증가시킨다.

만약 깨어난 프로세스가 존재하여 wakeup_ps 가 1 이상이면, trap() 함수에서 trapframe 이 timer_interrupt 인 경우의 조건문을 변경하여 wakeup_ps 를 다시 0 으로 초기화하고 yield() 함수를 호출한다. 이를 통해 가장 먼저 생성된 프로세스가 CPU 를 할당받을 수 있도록 한다.

```
//FCFS
if(myproc() && myproc()->state == RUNNING &&
   tf->trapno == T_IRQ0+IRQ_TIMER && (myproc()->ticks > TICKS_LIMIT || wakeup_ps > 0)){
    if(myproc()->ticks > TICKS_LIMIT){
      kill(myproc()->pid);
    }
    else{
      wakeup_ps = 0;
      yield();
    }
   }
```

위와 같은 설계를 통해 FCFS 스케줄링 정책을 효율적으로 구현하였다.

■ FCFS Challenge

1. proc 구조체에서 ticks 변수사용

Timeout 을 확인하기 위한 Proceess 의 ticks 변수 사용 시 첫 설계에서 전역변수로 ticks 를 설정하였다. Process 가 실행하다가 timeout 혹은 system call 로 인해 종료 되면 process_ticks 의 변수를 다시 0 으로 초기화 한 뒤 다음 scheduling 이 되는 process 의 시간을 측정하는 방법을 사용 했었고 이는 sleep 의 호출이 없는 경우에 정상 동작했다. 이후 wakeup 에 대한 내용의 이해와 구현을 진행하며 sleep 호출을 진행했을 때 process 가 CPU 를 무한히 점유하는 것을 확인하였다. 그 이유는 sleep 시 process_ticks 가 0 으로 초기화 되여 process 가 실제 사용한 시간을 저장하지 못했기 때문이라고 생각하였다. 따라서 전역변수 선언이 아닌 proc 구조체안에 ticks 라는 변수를 추가하였다. 이를 통해 sleep 으로 switching 이 되더라도 그 process 가 사용한 ticks 를 알 수 있어 100ticks 가 넘어가는 timeout 에 대해 구현할 수 있었다.

2. myproc() 함수 사용

매 타임 인터럽트마다 프로세스의 ticks 를 증가시키기 위해 myproc()->ticks++로 설계하였다. 그러나 이러한 설계는 컴파일 시 부팅이 불가능했다. 분석 결과, myproc()이 NULL, 즉 가리키는 프로세스가 없는 경우에 에러가 발생하는 것이 원인이었다. 따라서 if (myproc())라는 조건문을 사용하여 에러를 처리하였다. 이를 통해 프로세스가 존재할 때만 ticks 를 증가시켜 안정적으로 시스템이 동작하도록 하였다.

- MLFQ 설계

MLFQ 는 여러 개의 ready queue 로 구성되어 있으며, 각 큐마다 고유의 스케줄링을 진행한다. 프로그램에는 고정된 우선순위가 부여되지 않으며, CPU bound 와 I/O bound 특성에 따라 동적으로 우선순위가 부여된다. 프로세스가 하단 큐로 내려갈수록 Time Quantum 이 길어지도록 설계되어, CPU 사용량이 많은 프로세스가 할당받은 시간을 초과하면 아래 큐로 이동해 더 많은 CPU 사용 시간을 할당받을 수 있게 된다. 이는 프로그램의 특성을 미리 알 수 없기 때문에, 프로세스가 하단 큐로 내려가면 CPU 를 점유했을 때 한 번에 실행되는 시간이 길어지도록 하여 CPU bound 특성을 가진 프로세스를 식별할 수 있도록 설계된 것이다. 또한, 처음 들어오는 프로그램은 가장 우선순위가 높은 큐에 배치된다. 이를 통해 response time 을 최소화하고 turnaround time 을 최적화할 수 있다. 첫 번째 큐에서 프로세스가 완료되지 않으면, 아래 큐로 이동하여 대기하게 된다. 이와 같은 MLFQ 알고리즘의 동작을 바탕으로 우리 팀의 설계 MLFQ Rule 을 아래와 같이 작성했다.

[MLFQ 설계 rule]

1) 스케줄링을 위한 5 개의 큐가 존재하고, 우선순위가 높은 큐부터 낮은 큐까지 {2,4,8,16,32}의 TQ 를 갖는다.

2) 큐 내부의 우선순위는 존재하지 않으며, 각 큐에서 Time Quantum 에 따라 Round Robin 스케줄링에 진행된다.

3) 새로운 프로세스가 시스템에 들어가면 가장 높은 우선순위를 부여한다.

4) 일정 시간 이상 하위 큐에 머무는 프로세스는 한 단계 위의 큐로 옮겨진다.

1 번부터 3 번은 MLFQ 의 기본 동작에 대한 정의를 나타낸다. 중간 평가까지 기본적인 Rule 을 수행하는 MLFQ 를 구현하였고, starvation 문제를 해결하기 위한 방안은 그 이후 고려되었다.

기존 xv6 상의 Round Robin 알고리즘과는 다르게 여러 개의 큐가 존재하고, 생성된 프로세스들이 해당 큐 내부에서 이동하는 것을 쉽게 볼 수 있어야한다. 따라서, 대기하는 하나의 프로세스가 queue 라는 2 차원 배열 내부에 위치하게 설계했으며, 프로세스 생성 시 여러 변수를 담은 proc 구조체가 포인터로 해당 배열의 원소를 받으면 그 자리에 위치하는 프로세스 객체에 대한 변수들을 접근할 수 있는 구조를 구성하였다.

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

[사용 예시]

이러한 구조체로 사용되는 proc 내부에는 해당 프로세스의 정보를 포함하고 있는 다양한 변수들이 존재한다. 이들은 proc.h 파일 내부에 서술되어있으며, 프로세스 별 개인정보로 생성된다. MLFQ

스케줄링을 위해 iotime, rtime, ctime, etime 과 같은 프로세스 생성과 동작 소요시간, 대기 시간을 나타내는 변수들을 설정하였다. 설정된 변수들은 proc.c 파일이 동작하며 alloc 함수 내부에서 초기화를 진행한다.

```
int iotime;                 // sleep time
int rtime;                  // total run time
int ctime;                  // creation time
int etime;                  // end time
int waitshh;
int priority;
int num_run;
int qticks[5];
int queue;
int curr_ticks;
int change_q;
int enter;
```

[proc.h 파일에 struct proc 코드내부 변수 선언]

MLFQ 의 전체적인 스케줄링은 scheduler() 함수에 작성되었다. xv6 상에서 부팅 후 첫 프로세스가 시작되어 scheduler()에 들어가면 반복문 동작을 통해 scheduler 를 빠져나오지 못하고 동작이 계속 반복된다. scheduler 함수에 들어가면 CPU 할당을 위한 프로세스 선택 전, aging 동작을 할 수 있도록 우선적으로 배치하였다. starvation 은 CPU bound 프로그램이 큐에서 주어진 TQ 를 모두 소진하며 가장 우선순위가 낮은 큐에 위치하여 CPU 를 할당받지 못하고 있는 상황을 뜻한다. 해당 경우에서는 다른 큐에 프로세스가 존재한다면 절대 RUNNING 상태가 될 수 없으므로, RUNNABLE 상태로 무한정 다른 프로세스의 완료를 기다린다. 이러한 문제를 해결하기 위한 방법으로 우리 팀은 MLFQ 에 aging 을 추가적으로 구현하기로 결정했다. age 는 현재 ticks 에서 프로세스가 현재 큐에 들어간 시간의 차를 구해 나타내며 30 이라는 임계값을 넘어가는 경우, 현재 큐 위치(i)에서 삭제 후 앞선 우선순위 큐(i-1)에 추가한다. 위 반복문은 0 번 큐를 제외한 4 개의 큐에서 진행된다.

```
for(int i=1; i < 5; i++)
        {
            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);
                }
            }
        }
```

[scheduler() 내 aging 기법 구현]

aging 이후 CPU 할당을 위한 프로세스 탐색이 진행된다. 가장 높은 우선순위의 큐부터 출력 가능한 프로세스가 존재하는지 탐색하고 프로세스를 proc pointer p 를 통해 지목한다. 지목한 프로세스가 비어있지 않고, 상태가 RUNNABLE 임을 확인하면 동작 시간 변수를 증가시키고 state 를 RUNNING 으로 변경 후, CPU 를 할당하여 동작한다. 이후, 프로세스의 상태가 RUNNABLE 이라면 할당 받은 TQ 를 모두

사용하여 다음 큐로 이동해야 하는, 즉, 큐의 이동을 나타내는 change_q 가 1 임을 확인하고 큐를 다음 우선순위 큐로 이동시킨다. flag 가 0 이라면 ticks 값 초기화만 진행하면 된다. 이때 change_q 는 현재 실행 중인 프로세스에서 타이머 인터럽트 발생 시 프로세스의 실행 기간 ticks 와 해당 큐의 TQ 조건을 비교하여 큐의 이동이 필요함을 알리는 flag 역할을 한다. 아래 코드는 최종적으로 CPU 할당 시에만 변경되는 값들을 초기화 시켜주고, 다시 큐에 대기상태로 머무르도록 한다. ticks 변수를 사용하기 위해서는 시스템 호출 트랩 번호를 확인하고 타이머 인터럽트를 처리하는 과정 코드내부에서 ticking 이 진행되어야 한다. 이는 trap.c 파일에 위치하며 cs 에 위치한 ticks 변수 값 증가 동작을 한다. 이외에 ticking 이 필요한 iotime, rtime 은 ticking() 함수로 선언하여 같은 구문 안에 위치한다.

```
if(p!=0 && p->state == RUNNABLE)
        {
            if(p->change_q == 1)
            {
                p->change_q = 0;
                p->curr_ticks = 0;
                if(p->queue != 4){
                p->queue+=1;
                }
            }
            else p->curr_ticks = 0;
            add_proc_to_q(p,p->queue);
        }
```

[RUNNING 상태가 끝난 프로세스의 큐 이동 진행]

각 큐의 TQ 는 2, 4, 8, 16, 32 로, 프로세스의 큐 간 이동을 관찰하기 용이하도록 설정하였다. 제시한 Rule 에 맞추어 큐의 우선순위가 낮아질수록 긴 TQ 를 가지게 된다. 위 함수들 이외에도 큐에 프로세스를 추가하는 add_proc_to_q 함수, 큐에서 프로세스를 삭제하는 remove_proc_to_qg 함수, 실행 시간을 카운트하는 incr_curr_ticks 함수 등 다양한 함수가 요구되었다. 또한 waitx, ps, getps 의 추가적인 system call 을 추가하여 동작 분석의 제한을 받지 않도록 했다.

-MLFQ Challenge

1) 코드적 오류

MLFQ 구현을 위해 xv6 에서 다양한 파일에서 변수 추가와 함수 생성이 필수적이었다. 여러 파일을 수정하며 잘못된 선언으로 인한 코드 오류, lappicid 0 : panic(복구 불가 오류 발생)등이 발생했었다. 가장 잦게 발생한 문제는 파일 간 변수의 중복 선언에 대한 문제였다. '변수가 선언되지 않음'이 표시되어 해당 파일에 변수를 추가하면 중복 선언으로 인한 오류가 발생했다. 이는 xv6 를 수정할 때 proc.c 에서 변수를 종합적으로 선언하고 trap.c 에서는 proc.c 에서 선언한 함수를 사용하는 것으로 통일하여 자주 발생하는 코드 오류를 해결할 수 있었다.

2) SLEEPING state 아웃 이슈

fork() 함수를 통해 자식 프로세스 10 개를 생성하고 SLEEP 상태와 CPU 사용 비율을 자식 프로세스 별로 다르게 설정하여 시뮬레이션의 동작 경과를 관찰하였다. 시뮬레이션 코드 일부분에 getps() 함수를 위치시켜 일정한 주기를 갖고 출력되는 화면은 아니나, 코드 실행 과정에서 PID 별 프로세스의 state, run time, waiting time 등을 출력할 수 있도록 설정하였다. 해당 문제는 모든 프로세스가 sleep()함수 동작을 진행하고 wakeup 이 된 시점에서 대기 순서대로 CPU 를 할당받고 다시 sleep() 동작을 하는 시점에서 할 때 발생한다. 아래 사진에서는 PID6 이 CPU 를 할당받아 RUNNING 상태임을 볼 수 있고, 다음 출력화면에서 SLEEPING 상태이기 때문에 잘 동작하고 있음을 알 수 있었으나, 연속해서 다음 출력화면에서 바로 RUNNABLE 상태가 되며 sleep(200) 시뮬레이션 시간 조건을 만족하지 않고 일찍 빠져나오는 모습을 확인했다. 결과적으로 프로세스들은 sleep(200)만큼 SLEEPING 이 아닌 RUNNABLE 상태로 대기하는 것으로 확인할 수 있었다.

PID	State	r_time	w_time	c_time	PID	State	r_time	w_time	c_time
3	SLEEPING	1	1	561	3	SLEEPING	1	1	561
4	RUNNABLE	3	205	562	4	RUNNABLE	3	206	562
5	RUNNABLE	1	207	562	5	RUNNABLE	1	208	562
6	RUNNABLE	1	207	562	6	RUNNABLE	1	208	562
7	RUNNABLE	1	207	562	7	RUNNABLE	1	208	562
8	SLEEPING	1	207	562	8	RUNNABLE	1	208	562
9	RUNNING	0	208	562	9	SLEEPING	1	208	562
10	RUNNABLE	0	208	562	10	RUNNING	0	209	562
11	RUNNABLE	0	208	562	11	RUNNABLE	0	209	562
12	RUNNABLE	0	208	562	12	RUNNABLE	0	209	562
13	RUNNABLE	0	207	563	13	RUNNABLE	0	208	563

위 문제에 대하여 SLEEP 이 계속 wakeup 되고 유지되지 못하는 것, 더 나아가 이로인한 입출력 시간이 체크되지 않은 점들을 오류 발생 원인으로 지목하였고, FCFS 및 RR 알고리즘을 동일 환경에서 실행했을 때 SLEEPING 상태가 문제없이 유지되었다는 점을 바탕으로 MLFQ 의 큐 스케줄링 구현 시 오류를 발생 구간이 존재한다고 추측하였다. 따라서 trap.c 에 구현되어 동작 시간과 입출력 시간을 구현하는 MLFQ 의 ticking() 함수에 대해서 디버깅하였고, RR 과 FCFS 별개의 큐에 배열하여 프로세스를 지정하는 방식에서 오류가 났을 수도 있다고 가정하여 해당 부분도 함께 디버깅을 진행하였다.

하지만, xv6 상에서 MLFQ 와 SLEEP 의 정확한 오류가 나는 코드를 발견할 수 없었으며 이 부분은 차후 더 깊은 분석을 통해 해결해야할 문제로 남게되었다. 프로젝트를 진행하며 이와 같은 결과를 발견하게 되었는데, 그럼에도 RUNNING 상태의 CPU 는 정상적으로 동작한다는 점, 대기와 같은 문제를 제외하고 봤을 때 큐의 이동이 명확하고 aging 동작을 관측할 수 있다는 점을 들어, 최종 결말 부분까지 해당 알고리즘의 결과도 함께 분석하기로 결정하였다.

IV. 검증

- FCFS 시뮬레이션 결과 및 설명

[테스트 환경 설정]

Create_child() 함수: fork() 이후 자식 프로세스에서 sleep(10)을 호출하도록 설정하였다.

Exit_child() 함수: 자식 프로세스를 종료시킨다.

<Test 1> 순차 실행

기본적으로 프로세스를 순차적으로 실행해본 결과, FCFS 스케줄링 정책에 따라 프로세스가 생성 순서대로 정확하게 실행되었다. 이 실험을 통해 구현된 스케줄러가 기본적인 FCFS 정책을 충실히 따르고 있음을 확인할 수 있었다.

```
// 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 2> CPU 점유를 포기하는 yield() 포함 경우

yield() 함수 호출이 포함된 경우에도 먼저 생성된 프로세스를 찾아 스케줄링한다. yield()로 인해 CPU 점유를 포기하더라도, FCFS 스케줄링 정책에 따라 프로세스가 순차적으로 실행됨을 확인할 수 있다. 이는 yield() 호출이 있어도 FCFS 정책이 유지됨을 보여준다.

```
//Test 2
printf(1, "\nWith yield\n");

p = create_child();

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

exit_child(p);
```

```
With yield
process 9
process 9
process 9
process 10
process 10
process 10
process 11
process 11
process 11
process 12
process 12
process 12
process 13
process 13
process 13
```

<Test 3> sleep()과 yield()가 포함된 경우

sleep() 상태에 들어가도 순차적으로 진행되지 않고, wakeup 이후 바로 다음 프로세스로 스케줄링되는것을 확인할 수 있다.

```c
// 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 14
process 14
process 14

process 14
process 14
process 14

process 15
process 15

process 16
process 16
process 16

process 15
process 15
process 15

process 16
process 16
process 16

process 17
process 17
process 17
```

```
process 15
process 15
process 15

process 16
process 16
process 16

process 17
process 17
process 17

process 18
process 18
process 18

process 17
process 17
process 17

process 18
process 18
process 18

process 18
process 18
process 18
```

[그림] Test3 code, 결과 1, 결과 2

결과 1 과 결과 2 에서, sleep()과 yield()가 포함된 상황에서도 각 프로세스가 순차적으로 진행되지 않고 wakeup 이후 다음 프로세스로 스케줄링되는 현상을 확인할 수 있다. 이는 sleep() 호출로 인해 프로세스가 SLEEPING 상태로 전환되면서 발생한 현상이며, 이 경우, SLEEPING 상태에서 깨어난 프로세스는 FCFS 순서를 따르지 않고, 바로 실행 상태로 전환된다.

<Test 4> 100 ticks 가 넘어가는 경우

일정 시간이 경과한 후, 100 ticks 를 초과하는 프로세스는 강제 종료된다. 이는 FCFS 스케줄링 정책에서 프로세스가 CPU 를 독점하는 것을 방지하고, 시스템의 응답성을 유지하기 위한 타임아웃 메커니즘이 정상적으로 작동함을 보여준다.

```c
// Test 4
printf(1, "\nInfinite loop\n");

p = create_child();

if (p == 0) while (1);

exit_child(p);
printf(1, "ok\n");

exit();
```

```
Infinite loop
ok
```

- MLFQ 시뮬레이션 결과 및 설명

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ tester_ps
```

설계한 MLFQ scheduler 를 검증하기 위해 tester..c code 를 작성하여 동작을 관찰하였다. 10 개의 process 를 fork() syscall 을 통해 생성하였고, 각 자식 process 의 생성 순서에 따라 조건문을 제시하여 CPU 와 I/O 사용 시간을 할당해주었다. 각 process 가 CPU 시간을 사용하는 동안 우선 순위가 낮아지며, I/O 시간을 사용하는 동안 대기 상태로 전환되도록 코드를 작성하였다.

다음은 동작 결과이다.

```
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]
```

5 개의 우선순위 대기열(queue0, queue1, queue2, queue3, queue4)을 사용하여 MLFQ 스케줄러를 구축하였고, queue0 은 가장 높은 priority 를 가지며 queue4 는 가장 낮은 priority 를 갖는다. 프로세스가 해당 시간 분할을 모두 사용하면 다음(낮은) priority 수준으로 다운그레이드되어 queue 를 차례로 내려가는 것을 확인하였다.

process 가 처음 생성되면 가장 높은 우선순위(queue0)에 대기열의 끝에 배치한다. queue 의 priority 대로 timer tick(TQ)를 각각 {2, 4, 8, 16, 32}으로 할당하였고 timer tick 이 발생하면 현재 CPU 를 사용하고 있던 process 가 timer tick 의 CPU 를 모두 사용한 것으로 간주한다. process 가 다른 priority queue 로 이동할 때마다 해당 process 는 해당 대기열의 끝에 배치되어야 한다. 우선 순위가 가장 낮은 queue3 는 라운드 로빈 방식으로 스케줄링된다. 이는 우선 순위가 낮은 프로세스들 간에 CPU 사용 시간을 공평하게 할당하기 위한 방식이다.

```
Process: PID 4 :0 Finished
PID   State      r_time  w_time  cur_q   q0      q1      q2      q3      q4
3   SLEEPING     3       2       1       2       0       0       0       0
4   RUNNING      699     2362    4       2       4       8       16      645
5   RUNNABLE     496     2564    4       2       4       8       16      448
6   RUNNABLE     397     2663    4       2       4       8       16      352
7   RUNNABLE     331     2729    4       2       4       8       16      288
8   RUNNABLE     265     2795    4       2       4       8       16      224
9   RUNNABLE     232     2827    4       2       4       8       16      192
10   RUNNABLE    200     2859    4       3       4       8       16      160
[Terminate] --> Total Time for pid [4] is [3062]
```

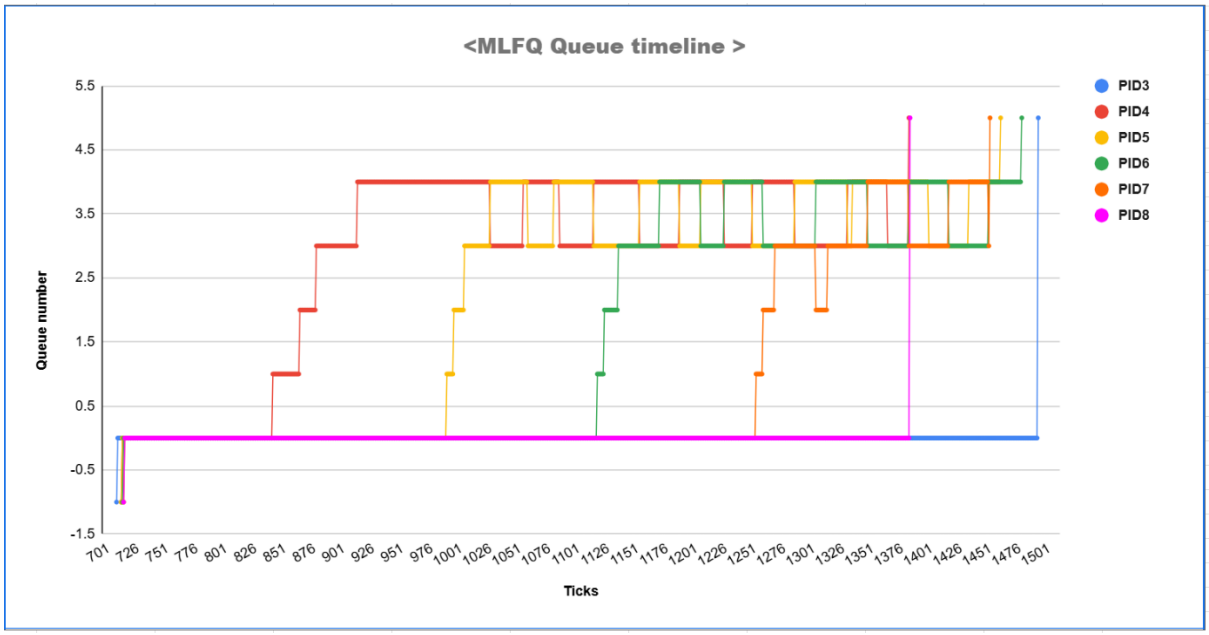
PID4 와 같이 이른 시기에 생성된 process 는 주로 CPU 바운드 작업을 부여받으며, Scheduler 에 의한 우선 순위 지정으로 인해, 높은 CPU 작업을 부여받았음에도 불구하고 빠르게 완료되는 것을 확인하였다. 후순위 process 일수록, I/O 할당을 많이 해주었는데, I/O 작업은 CPU 작업에 비해 상대적으로 짧기 때문에, 이에 따라 작업 권한을 한 번 부여받으면 바로 완료하는 것을 확인하였다.

```
Scheduling tester with PID 4 from Queue 4 with current tick 0 at tick 711
Process with PID 6 added to Queue 0 at 712
Process with PID 7 added to Queue 0 at 712
Process with PID 8 added to Queue 0 at 712
Process with PID 4 added to Queue 4 at 744
Process with PID 5 is removed from Queue 4 at 744
Process with PID 5 added to Queue 3 at 744
Process with PID 6 is removed from Queue 0 at 744
```

```
PID   State      r_time  w_time  cur_q   q0      q1      q2      q3      q4
3   SLEEPING     2       1       0       2       0       0       0       0
4   RUNNABLE     561     1563    2       2       24      112     192     192
5   RUNNABLE     416     1708    1       2       16      96      144     128
```

starvation 문제는 우선순위가 낮은 대기열에 있는 프로세스가 지속적으로 CPU 시간을 할당받지 못하는 상황을 의미한다. 이를 해결하기 위해 aging 기법을 추가하였다. aging 기법은 오랜 시간 대기한 프로세스를 더 높은 우선순위 대기열로 이동시켜주는 방식으로 동작한다. 구체적으로, 각 대기열(queue0 를 제외한 queue1 부터 queue4 까지)에 대해 모든 프로세스를 순회하면서, 각 프로세스의 대기 시간을 계산한다. 만약 특정 프로세스가 해당 대기열에 30 ticks 이상 머물렀다면, 그 프로세스는 한 단계 높은 우선순위 대기열로 이동한다. 이를 통해 낮은 우선순위 대기열에 오래 머무른 프로세스가 더 높은 우선순위 대기열로 이동하여 CPU 자원을 할당받을 수 있도록 구현하였다. 이와 같은 aging 기법을 적용함으로써 MLFQ 스케줄러는 특정 프로세스를 무한히 낮은 우선순위 대기열에 가둬두지 않고, 모든 프로세스가 공평하게 CPU 자원을 사용할 수 있도록 보장한다.

< fork()를 통해 생성한 프로세스의 PID 별 Queue Timeline >



MLFQ 스케줄러의 큐 내부 동작과 aging 효과를 명확히 보이기 위해 graph 를 통한 시각화를 도입하였다. 이는 scheduler() 함수 내부에 add_pro_to_q 와 remove_pro_to_q 와 같은 큐 이동이 일어나는 함수에 print 문을 포함시켜 PID 별로 해당 ticks 일 때 큐 변화를 알 수 있었다. 그래프의 세로 축은 Queue number 로, -1 은 초기화 상태를 의미하고 5 는 프로세스가 종료됨을 의미한다. 이를 바탕으로 각 대기열에 쌓이는 프로세스의 상태와 이동 과정을 그래프로 나타내었다.

부모 프로세스인 PID3 가 0 번째 큐에서 SLEEPING 상태를 유지하다 마지막 자식 프로세스가 종료되었을 때, 짧게 RUNNING 후 바로 종료되는 것을 확인하였다. 또한 queue2~queue4 에서 대부분의 프로세스가 aging 이 발생하여 앞선 우선순위 큐로 이동하는 것을 확인할 수 있다. 결과적으로 스케줄러의 동작 방식을 직관적으로 이해할 수 있었고 프로세스의 우선순위 변화, aging 에 의한 우선순위 상승이 잘 일어나는것을 확인하였다.

- Scheduler 별 성능 비교

XV6 에 기본 내장된 RR(Round Robin) 스케줄러와, 앞서 구현한 FCFS(First-Come, First-Served) 및 *MLFQ(Multilevel Feedback Queue) 스케줄러의 성능을 비교하기 위해, tester.c 의 getps() 함수를 사용하여 각 스케줄러의 r_time, w_time, s_time 을 분석을 진행하였다. 그 결과는 다음과 같다

flag	run time	wait time	io time	Total
RR	2	13	3141	3156
FCFS	2	2	5503	5507
***MLFQ**	2	1	2865	2868

마감 시간을 기준으로 스케줄링 방식을 비교해본 결과, FCFS > RR > *MLFQ 의 순으로 나타났다. 각 스케줄링 방식의 특성에 따라 프로세스의 마감 시간이 다르게 나타났으며, 이는 스케줄링 전략의 설계 원리에 기인한다.

FCFS 방식에서는 프로세스가 도착한 순서에 따라 실행되므로, CPU 시간이 긴 프로세스가 먼저 도착하면 뒤에 도착한 다른 프로세스들은 CPU 를 사용하기 위해 오랫동안 대기해야 한다. 긴 실행 시간이 필요한 프로세스가 선두에 위치할 경우, 후속 프로세스들은 모두 지연되어 평균 마감 시간이 증가하게 되고, 이러한 이유로 FCFS 는 가장 긴 마감 시간을 갖게 된 것을 볼 수 있다.

RR 은 일정한 시간 간격으로 process 를 교체하는 선점형 스케줄링으로, 프로세스가 타임 퀀텀을 다 쓰면 CPU 를 반환해야 한다. RR 은 프로세스에 일정한 타임 퀀텀을 부여하여 공평하게 CPU 시간을 분배하므로, 마감 시간이 FCFS 보다 짧고, 프로세스가 오랫동안 대기하는 현상을 방지한다.

MLFQ 방식은 프로세스의 특성을 동적으로 반영하여 스케줄링하므로, 마감 시간이 가장 짧다. CPU 바운드 프로세스는 낮은 우선순위 큐로 이동해 긴 시간 동안 CPU 를 사용하고, I/O 바운드 프로세스는 높은 우선순위 큐에서 빠르게 처리된다. 다양한 워크로드에 대해 효과적으로 대응할 수 있으며, I/O 바운드 프로세스가 높은 우선순위를 유지하여 빠르게 처리되는 덕분에 전체 마감 시간이 최소화된다.

*(앞서 언급한 바와 같이, MLFQ 는 SLEEPING 과 관련한 이슈가 있었다. 따라서 최종 성능 분석에 포함시키는것은 타당하지 않으나, queue 의 이동이 나타나고 aging 과 같은 동작이 가능하다는 점을 들어 전체 시간 비교대신 동작 양상을 확인하고자 포함시키게 되었다.)

이와 같이 각 스케줄링 방식의 특성을 고려한 성능 비교 결과, 마감 시간을 기준으로 스케줄링 방식을 정렬하였을때 FCFS > RR > *MLFQ 의 순임을 확인할 수 있었다.

Ⅴ. Thread 구현을 위한 xv6 동작 분석

- 분석 목적

본 Chapter 에서는 thread 구현을 위한 xv6 동작 분석을 진행하였다. Thread 는 Process 와 다른 개념이지만 비슷한 점도 있기 때문에 xv6 에서는 Process 수행 시 어떤 동작이 이루어 지는 지 살펴보았다. 그리고 분석한 내용들을 이용해서 Thread 를 구현하기 위해 수정해야 하는 Code 와 추가할 Code 들을 Chapter-VI에서 소개할 것이다.

- Various data structure

(1) Per – CPU state

CPU 마다 state 를 저장하는 data structure 이다. 현재 실행 중인 Process 에 대한 정보, Context (register 값들) 등을 포함하고 있다.

```
struct cpu {
    uchar apicid;                // Local APIC ID
    struct context* scheduler;   // swtch() here to enter scheduler
    struct taskstate ts;         // Used by x86 to find stack for interrupt
    struct segdesc gdt[NSEGS];   // x86 global descriptor table
    volatile uint started;       // Has the CPU started?
    int ncli;                    // Depth of pushcli nesting.
    int intena;                  // Were interrupts enabled before pushcli?
    struct proc* proc;           // The process running on this cpu or null
};
```

전부 중요한 정보들 이지만 일부만 살펴보면 다음과 같다.

- struct context* scheduler -> cpu 가 현재 실행 중인 process 의 context
- struct proc* proc -> cpu 가 현재 실행 중인 process 의 정보

(2) Per – Process state

Process 별 state 를 저장하는 data structure 이다. Process 의 memory management, synchronization, trap/interrupt 그리고 file system 등을 사용하기 위한 정보들이 저장되어 있다. 좀 더 구체적으로 보면 memory management 에 필요한 memory size, page table 그리고 kernel stack 의 주소 값을 가지고 있고 trap/interrupt 에 필요한 trap frame, context switch 시 사용되는 context(register 값들) 정보를 가지고 있다.

```
// Per-process state
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)
};
```

(3) Context

Context switch 동작 시 사용되는 주요한 register 값들을 담고 있다.

```
struct context {
    uint edi; // Destination Index
    uint esi; // Source Index
    uint ebx; // Base Index
    uint ebp; // Stack Base Pointer
    uint eip; // Instruction Pointer
};
```

(4) Trap frame

User mode 에서 Kernel mode 로 전환할 때 다시 User mode 에서 실행하기 위한 정보들을 저장하기 위해 사용된다. 구성 요소를 간단하게 살펴보면 크게 4 가지로 구분할 수 있다.

(1) Register

General purpose register 값들을 저장

(2) Segment register 및 trap number

Segment descriptor 와 trap number 저장

(3) x86 hardware define

hardware 에 자동으로 push 되는 정보들 저장

(4) Mode 전환

특히 user -> kernel 로의 전환이 발생할 때 추가로 생성되는 정보 저장

```
struct trapframe {
    // registers as pushed by pusha
    uint edi;
    uint esi;
    uint ebp;
    uint oesp;      // useless & ignored
    uint ebx;
    uint edx;
    uint ecx;
    uint eax;

    // rest of trap frame
    ushort gs;
    ushort padding1;
    ushort fs;
    ushort padding2;
    ushort es;
    ushort padding3;
    ushort ds;
    ushort padding4;
    uint trapno;

    // below here defined by x86 hardware
    uint err;
    uint eip;
    ushort cs;
    ushort padding5;
    uint eflags;

    // below here only when crossing rings, such as from user to kernel
    uint esp;
    ushort ss;
    ushort padding6;
};
```

- Trap handling

Trap 은 system call, interrupt 그리고 exception 등을 일컫는다. Xv6 에서는 trap event 가 발생하면 perl script 인 vectors.pl 에 의해 vectors.S 가 생성된다. Vectors.S 에서는 모든 trap 정보가 저장된 array 를 현재 kernel stack 에 push 하고 trap handler(trapasm.S)를 호출한다. 호출된 trapasm.S 에서는 새로운 trap frame 을 build 해서 trap event number(어떤 trap 인 지)와 user mode 에서의 register 값들을 저장한다. 그리고 build 된 trap frame 을 trap.c 에 정의된 trap()함수를 호출하면서 인자로 전달한다. Trap() 함수에서는 trap frame 에 전달된 trap event number 를 이용해서 정해진 SW routine 을 수행한다. 예를 들어 trap 이 system call 이라면 현재 실행 중인 process 의 tf pointer 에 trap frame 을 저장한 뒤 syscall() 함수를 통해 system call 을 수행한다. 수행이 완료되면 proc 에 저장된 trap frame 을 다시 load 해 오면서 kernel mode 에서 user mode 로 전환 후 기존 process 를 계속해서 수행한다.

- Process creation

(1) 초기 user process 생성

앞서 소개 했지만 조금 더 자세하게 살펴보면 다음과 같다. Kernel 의 main() 함수가 실행되면 userinit() 함수가 실행되면서 첫 번째 user process 가 생성된다. Userinit()함수는 내부에서 allocproc() 함수를 호출해서 PTABLE 에서 UNUSED 상태인 proc 을 할당받는다. 그 다음 여러 변수 값들을 setting 한다. 그 중 eip register 값을 강제로 0 으로 저장하는 동작을 통해 initcode.S 를 실행한다. Initcode.S 에서 exec() 함수를 실행해서 init.c 를 실행하고 init.c 가 실행되면 console 에서 사용자의 입력을 받을 수 있는 상태가 된다.

(2) allocproc() 함수

process 생성 시에 allocproc() 함수를 통해 cpu 별 할당되어 있는 ptable 내에서 UNUSED state 인 proc 을 탐색한다. 발견할 경우 pid 할당 및 kernel stack memory 를 할당한다. Kernel stack 을 할당하지 못할 경우 proc 의 state 를 다시 UNUSED 로 변경해 준다. Kernel stack 을 할당 받으면 trap frame 과 context size 만큼의 memory 를 미리 할당 해 둔다. 그리고 context -> eip register 값을 forkret() 함수의 주소 값으로 초기화 해서 process 생성 이후 처음 실행되는 함수가 forkret()이 되도록 설정하고 forkret() 함수 다음에 trapret() 함수가 실행되도록 주소 값을 추가로 할당한다.

◆ forkret() -> fork return, child process 의 생성을 끝냈으니 ptable 에 걸린 lock 을 해제

◆ trapret() -> trap return, kernel mode 에서 작업을 끝냈으니 user mode 로 돌아가겠다는 의미(assembly 어로 구현), process 의 trap frame 에 저장된 값들을 register 로 load

(3) fork() 함수

위에서 살펴본 내용을 통해 fork 함수의 동작을 이해해 보았다. Fork() system call 이 호출되면 trap handling 과정을 거쳐서 kernel mode 에 진입한다. 그 후 sys_fork()를 통해 proc.c 에 정의된 fork routine 이 실행된다. Fork routine 이 실행되면 allocproc() 함수를 이용해서 ptable 에서 UNUSED proc 을 찾아서 할당하고 copyuvm() method 를 이용해서 부모의 process state 를 복사한다. 즉, virtual memory address space 를 복사해서 할당 받는다.

(4) Memory layout

A. VP & PF

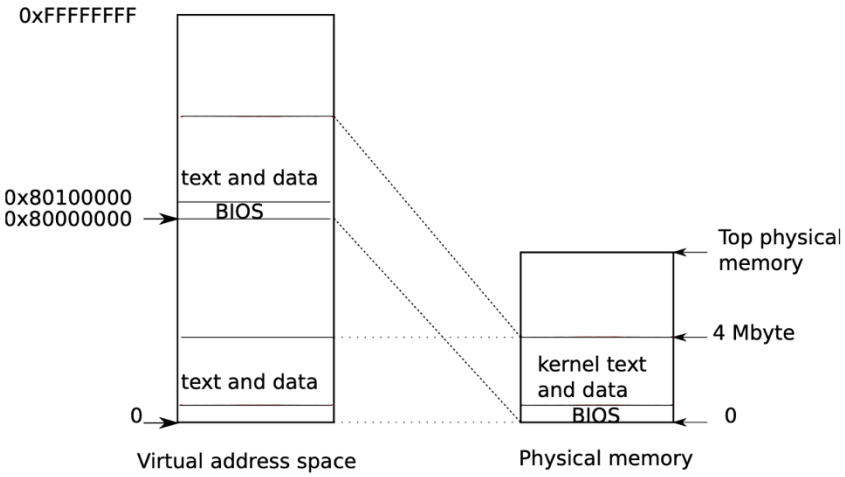


Figure 1-2. Layout of a virtual address space

Xv6 는 모든 process 의 virtual address space 에 physical memory 상에 존재하는 kernel 영역을 포함한다. 따라서 각각의 process 는 자신의 virtual address 를 통해 kernel 영역에 접근을 하게 되고 이에 따라 동기화 동작 등을 통해 protection 을 수행한다.

B. Kernel stack

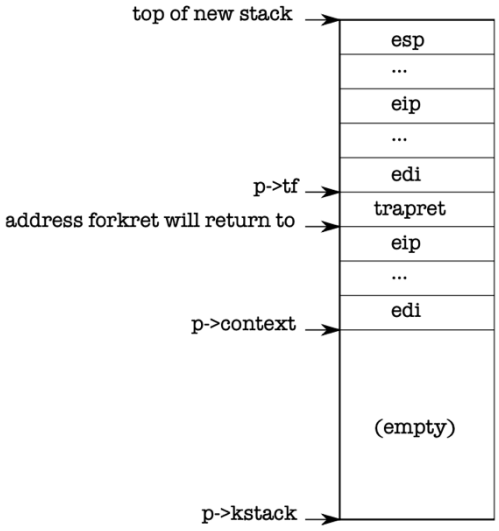


Figure 1-3. A new kernel stack.

Kernel stack 은 kernel mode 에서 함수를 수행할 때 필요하고 또한 여러 register 값 정보를 저장하기 위해 필요하다. Process 에 할당되는 kernel stack 을 살펴보면 user mode 일 때의 정보인 trap frame 영역이 할당되어 있고 process 실행 정보인 context 영역이 할당되어 있다. 이를 통해 context switch 동작이 왜 kernel mode 에서 수행되는 지 이해할 수 있다.

- Paging

Paging 에 대해서는 간단하게 살펴 보도록 한다. Process 가 생성되면 kernel memory 영역을 포함한 user virtual address space 가 생성되고 이를 physical memory 영역과 mapping 을 해주는 page table 이 생성된다.

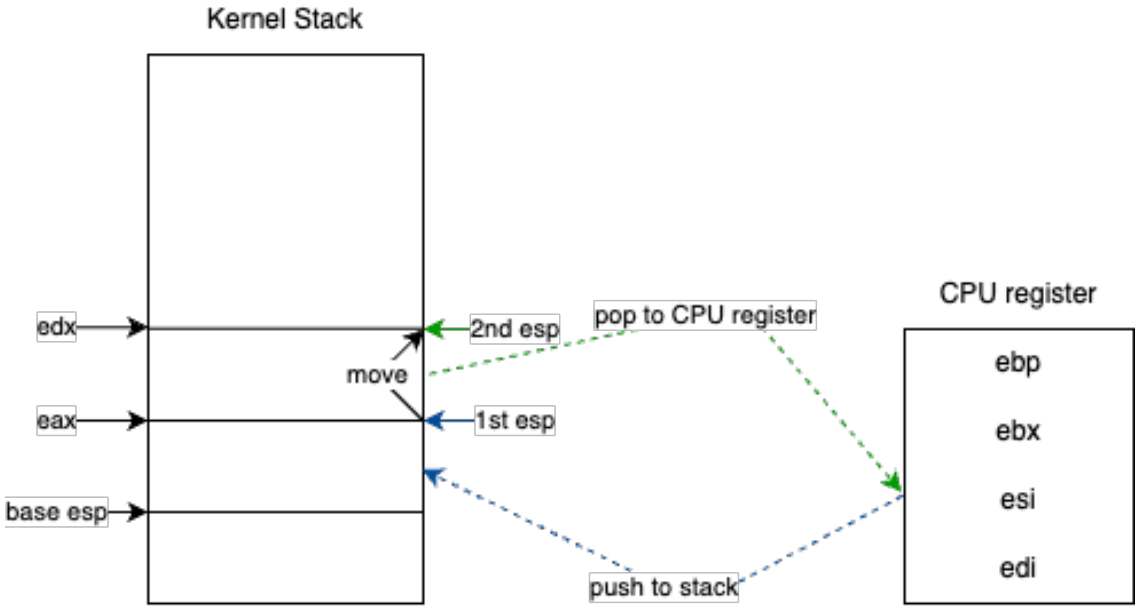
- Context switching

CPU 별 scheduler 함수가 존재한다. Context switch 가 발생할 경우 trap handling 을 통해 kernel mode 로 진입하고 sched() 함수 호출을 통해 실행 중인 process 와 scheduler routine 과의 switch 가 먼저 일어난다. Sched() 함수가 호출되어서 context switch 가 발생하는 경우는 3 가지가 존재한다.

- **Process terminated**
- **Process sleep**
- **Process yield (timer interrupt)**

실제로 동작이 수행되면 old process -> scheduler -> new process 순으로 context switch 가 일어나면서 새로운 process 가 수행된다.

swtch.S assembly 어 함수의 동작 (register 값을 바꾸는 동작)

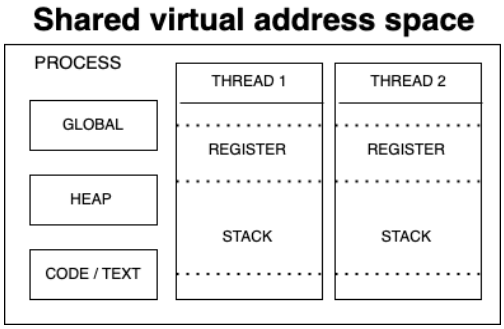


또한 Process 별로 page table 이 할당되므로 process 가 교체됨에 따라 Hardware 에서 사용되는 page table 도 변경해 주어야 한다. Scheduler() 함수에서 switchuvm() 함수를 사용해서 page table 을 교체해 준다.

VI. Thread 구현

- Thread concept

Thread 는 동일한 memory space 를 지니는 특징을 가지고 있으며 개념적인 부분을 그림으로 도식했을 때 다음과 같다.

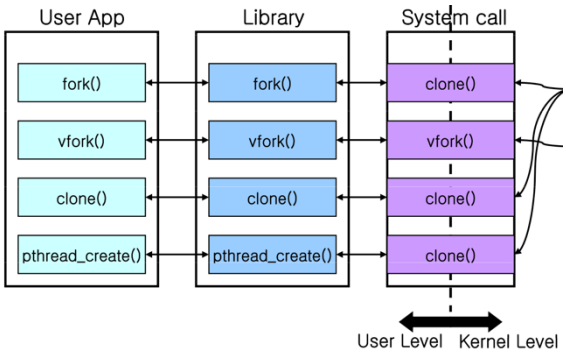


전역 변수, code 등의 data 들은 공유해서 사용하고 함수 호출 및 지역 변수를 저장할 stack 공간과 실행에 필요한 register set 들은 thread 별로 할당 받아서 사용한다.

- Clone system call

■ 동작 설명

Linux 에서는 process, thread 의 구분이 따로 존재하지 않고 task 라는 명칭으로 부른다. 다만 process 를 생성할 때 사용하는 fork(), thread 를 생성할 때 사용하는 clone() 함수들의 동작이 다르다. 또한 fork(), clone() 함수들은 호출되면 결론적으로는 clone system call 을 호출하는 동작을 수행하는데 flag 값이 다르다는 특징이 존재한다. 다음 사진을 통해 실제 linux 상에서 fork(), clone() user 입장에서 호출되면 내부적으로는 동일하게 clone system call 을 호출하는 사실을 확인할 수 있다.



이러한 clone system call 은 flag 값을 통해 어떤 동작을 수행할 지 구분한다. 간단하게 thread 구현 시 필요한 flag 들을 살펴보면 다음과 같다. 아래 설명은 Linux manual page 에서 발췌한 것이다.

◆ CLONE_FILES : If CLONE_FILES is set, the calling process and the child process share the same file descriptor table.

◆ CLONE_FS : "If CLONE_FS is set, the caller and the child process share the same filesystem information.

◆ CLONE_TRHEAD : If CLONE_THREAD is set, the child is placed in the same thread group as the calling process.

◆ CLONE_VM : If CLONE_VM is set, the calling process and the child process run in the same memory space.

요약하자면 thread 를 구현하기 위해 clone system call 을 호출할 때 위 flag 를 사용해야 하며 해당 flag 사용 시 file descriptor table, file system information 그리고 memory space 를 공유하며 clone system call 을 호출한 process 와 동일한 thread group 으로 묶이게 된다.

■ 구현

Linux 에서는 process 와 thread 간 구분이 존재하지 않기 때문에 xv6 상에서 구현할 때 proc 을 할당받아서 일반 process 처럼 생성을 하였다.

<thread 할당>

```
// Allocate process.
if((np = allocproc()) == 0)
}
    return -1;
}
```

Code 상에서도 allocproc() 함수를 통해 일반 process 처럼 생성되는 모습을 볼 수 있다. Allocproc() 함수 내부에서 kernel stack 을 할당하므로 one to one threading model 을 사용하고자 하는 목적과도 부합하는 것을 알 수 있다.

Clone system call 구현 시 flag 를 사용하는 방식으로 구현하고자 했으나 thread 구현 목적에 비해 너무 많은 부분을 고려해야 된다는 판단이 들어 flag 를 사용했다는 가정하에 fork 와 다른 함수로 clone 을 구현하였다. 그리고 각 flag 를 사용했을 시 어떤 동작이 일어나는 지 분석하기 위해 fork 함수와의 차이점을 flag 역할에 맞게 구분해 보았다. 다만 CLONE_FILES, CLONE_FS 는 fork 와 clone 모두 사용하는 flag 이므로 동작 차이에 대해서는 기술하지 않고 어떤 동작인 지만 기술하였다.

(1) CLONE_FILES

<clone 구현 부>

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

CLONE_FILES flag 가 set 되면 caller process 의 file descriptor 를 공유한다. Filedup() 함수는 인자로

받은 file 의 참조 카운트를 증가 시켜 해당 파일을 참조하는 task 가 증가했다는 것을 저장하고 인자로 받은 파일 포인터를 반환해서 새로운 task 가 동일한 파일을 공유할 수 있도록 해준다.

(2) CLONE_FS

<clone 구현 부>

```
np->cwd = idup(curproc->cwd);
```

CLONE_FS flag 가 set 되면 caller process 의 file system 을 공유한다. 이는 inode 를 공유한다는 말인데 caller process 에서 사용중인 directory 들을 복사해서 사용한다는 의미이다. File system 을 공유할 때와 마찬가지로 idup() 함수를 통해 caller process 가 사용중인 inode 를 입력으로 받아 참조 카운트를 증가시키고 해당 inode 포인터를 그대로 반환해준다.

(3) CLONE_THREAD

<fork 구현 부>

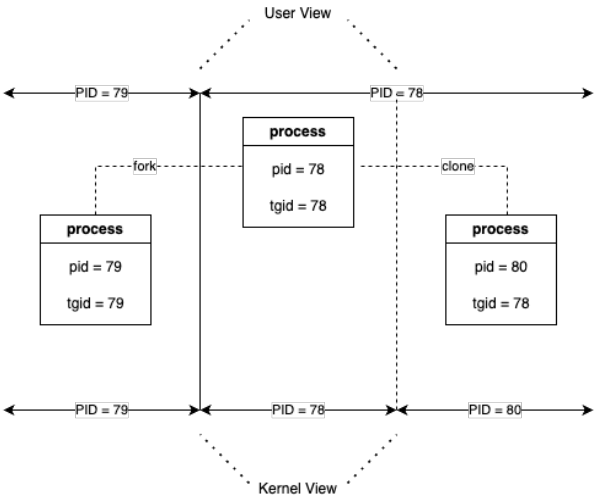
```
p->tgid = nexttgid++; /* Define tgid for process(thread) */
```

<clone 구현 부>

```
np->tgid = curproc->tgid; /* Same tgid with parent => thread */
```

CLONE_THREAD flag 는 단순히 tgid 라는 개념을 사용해서 구현하였다. 사용자 입장에서 thread 는 부모 process 와 동일한 pid(tgid) 값을 가지고 kernel 입장에서는 다른 pid 값을 가진다. 즉, linux 관점으로 해석해본다면 thread 는 caller process 와 동일한 task 에 해당하고 child process 는 다른 task 라는 의미가 된다. 위의 code 를 통해 확인해봤을 때 fork 함수 실행 시 pid 값을 설정할 때와 마찬가지로 tgid 값도 caller process 와 다른 값을 가지도록 하고 있다. 반면에 clone 함수에서는 새로 생성한 task(thread)의 tgid 를 caller process 와 동일하게 할당함으로써 동일한 thread group 에 속한다는 것을 보여주고 있다.

아래 사진은 각 task 들의 user / kernel view 를 보여주고 있다.



Kernel 입장에서는 thread 와 process 는 전부 다른 task 로 인식이 되어서 동작하지만 user 입장에서는 동일한 pid 값을 가진 task 로 인식이 되는 부분을 확인할 수 있다.

(4) CLONE_VM

<fork 구현 부>

```
// Copy process state from proc.
  if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
```

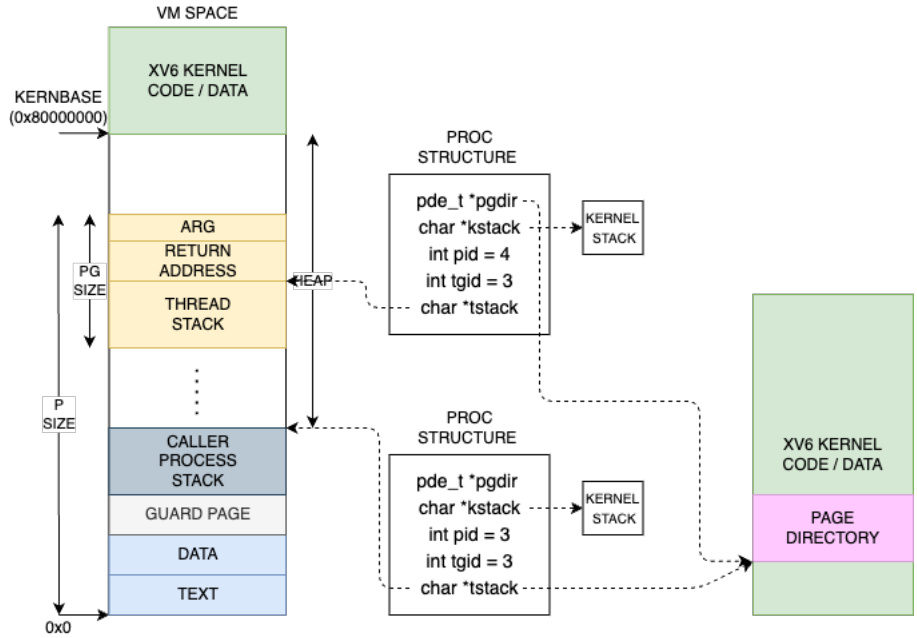
<clone 구현 부>

```
// from thread.c
void *stack = malloc(PGSIZE);

// set page directory same with parent
np->pgdir = curproc->pgdir;

// Stack pointer is at the bottom, bring it up; push return
// address and arg
*(uint*)(stack + PGSIZE - 1 * sizeof(void *)) = (uint)arg;
*(uint*)(stack + PGSIZE - 2 * sizeof(void *)) = 0xFFFFFFFF;

// Set esp (stack pointer register) and ebp (stack base register)
// eip (instruction pointer register)
np->tf->esp = (uint)stack + PGSIZE - 2 * sizeof(void*);
np->tf->ebp = np->tf->esp;
np->tf->eip = (uint) function;

// Set thread stack
np->tstack = stack;
```

CLONE_VM flag 가 set 되면 virtual address space 를 공유하는 동작이 수행된다. Fork() 함수에서 생성된 process 의 page directory 는 copyuvm() 함수를 통해 caller process 의 vm 을 복사해서 새로운 memory 공간으로 할당받는 것을 확인할 수 있다. 하지만 clone() 함수에서는 caller process 와 별도로 vm 을 할당 받는 것이 아닌 caller process 내부에 memory 에 stack 을 할당 해서 사용한다. 따라서 위의 clone 구현 부를 살펴보면 malloc() 함수를 통해 page size 만큼의 stack 을 caller process 의 heap 영역에 할당 받고 page directory 도 caller process 와 동일하게 사용하는 것을 확인할 수 있다. Stack 할당 이후 인자와 0xFFFFFFFF 값을 setting 해주고 trap frame 의 register 값들을 thread 가 수행될 수 있도록 변경해준다. Esp register 의 경우 stack pointer 를 의미하고 thread 가 할당 받은 stack 을 가리키도록 한다. 그리고 eip register 의 경우 instruction pointer 를 의미하고 thread 생성 이후 실행할 함수의 주소 값을 저장한다.

■ **동작 상세**

다음은 clone system call 이 호출될 경우 virtual memory address space 에서 어떤 변화가 일어나는지 그림으로 도식해본 결과이다.



Thread 가 생성되면 user virtual memory address space 의 heap 영역에서 malloc() 함수를 통해 PAGE_SIZE 에 해당하는 stack 영역을 할당 받는다. 그 이후 실행할 함수의 인자를 stack 최상단에 저장해 준 뒤 비정상적인 실행이 발생했을 경우 return 할 address 를 저장해주고 stack pointer 에는 argument, address 를 제외한 주소를 가리키도록 해준다. Stack 영역을 할당할 때 이루어 지는 동작을 조금 더 자세히 살펴보면 malloc()함수를 이용해서 stack 영역을 먼저 heap 에서 할당한다. 다만 여기서 malloc()으로 할당할 수 있는 free list 가 존재하지 않을 경우 memory 를 먼저 free list 로 만들어주고 growproc()함수를 이용해서 추가할 memory 크기 만큼을 process 가 사용할 memory 크기로 추가해준다. 즉, process 가 사용 중인 memory size 가 더 커지는 것이고 이를 반영해서 switchuvm()함수가 실행됨에 따라 page table 이 추가된 memory 를 mapping 하면서 동작이 수행된다.

- Join system call

■ **동작 설명**

Join system call 이 수행되면 process table 에 lock 을 걸고 입력으로 받은 tid 값을 이용해서 process table 내에 해당 tid 를 가진 thread 가 존재하는 지 찾고 존재할 경우 실행이 완료될 때까지 현재 process 를 sleep 상태로 둔다. 그리고 만약에 zombie thread 를 발견한다면 할당된 모든 자원을 삭제하고 process table 에서도 해당 thread 를 지운다.

■ 구현

```
acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for exited children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->parent != curproc || p->pgdir != curproc->pgdir || p-
>pid != tid)
        continue;
      havekids = 1;
      if(p->state == ZOMBIE){
        // Found one.
        pid = p->pid;
        kfree(p->kstack);
        p->kstack = 0;
        *stack = p->tstack;
        p->tstack = 0;
        p->pid = 0;
        p->parent = 0;
        p->name[0] = 0;
        p->killed = 0;
        p->state = UNUSED;

        release(&ptable.lock);
        return pid;
      }
    }

    // No point waiting if we don't have any children.
    if(!havekids || curproc->killed){
      release(&ptable.lock);
      return -1;
    }

    // Wait for children to exit.  (See wakeup1 call in proc_exit.)
    sleep(curproc, &ptable.lock);  //DOC: wait-sleep
  }
```

무한 loop 를 돌면서 caller process 에서 생성된 thread 가 존재하는 지 확인하고 만약 존재한다면 sleep 을 통해 발견한 thread 의 수행이 종료될 때까지 기다린다. 그리고 중간에 ZOMBIE state 를 발견하면 해당 thread 에 할당된 모든 자원을 반환하고 ZOMBIE state 였던 thread 의 pid 를 돌려주는 동작이 수행된다.

- User thread library

사용자가 직접 clone, join system call 을 호출하지 않고 wrapper function 을 정의해서 내부적으로 호출되도록 하였다.

(1) thread_create()

```
int thread_create(void (*function) (void *), void *arg)
{
    void *stack = malloc(PGSIZE);

    if (stack == 0)
        return -1;
    if ((uint)stack % PGSIZE != 0)
        stack += PGSIZE - ((uint)stack % PGSIZE);

    return clone(function, arg, stack);
}
```

Malloc() 함수를 사용해서 PAGESIZE 만큼의 영역을 할당한 뒤 clone() 함수의 인자로 주소 값을 보내준다. 따라서 clone() 함수를 호출할 때 사용자가 직접 memory 공간에 할당하는 동작을 수행하지 않아도 된다.

(2) thread_join()

```
int thread_join(int tid)
{
    int retval;
    void *stack;
    retval = join(tid, &stack);
    free(stack);
    return retval;
}
```

Join()함수도 마찬가지로 thread_join() wrapper 를 정의해서 내부적으로 join system call 이 호출되도록 하였다.

- Thread 구현 검증

마지막으로 thread 구현 검증을 진행하였다. Thread 가 생성되었을 때 정보들을 출력해보고 동작이 잘 되는 지 검증하였다. Thread 생성 시 process 와의 차이점을 확인하기 위한 정보들은 다음과 같다.

■ Page directory address

■ Stack pointer

■ Function address

■ Memory size

결과는 다음과 같다.

```
PID : 3
TID : 3
Page Directory Address : 8df23000
Stack Pointer Value : 2f9c
Statr 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
```

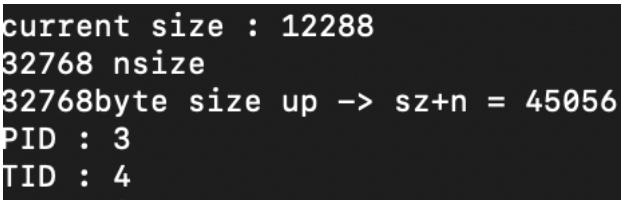
```
[Terminate] --> Total Time for pid [4] is [1]
[Terminate] --> Total Time for pid [5] is [0]
[Terminate] --> Total Time for pid [6] is [0]
[Terminate] --> Total Time for pid [7] is [0]
[0] = 0
[1] = 0
[2] = 0
[3] = 0
[4] = 0
[5] = 5
[6] = 5
[7] = 5
[8] = 5
[9] = 5
[10] = 10
[11] = 10
[12] = 10
[13] = 10
[14] = 10
[15] = 15
[16] = 15
[17] = 15
[18] = 15
[19] = 15
[Terminate] --> Total Time for pid [3] is [2]
```

왼쪽의 결과는 caller process -> thread 1 -> thread 2 -> ... -> thread 4 까지 생성되었을때의 정보들을 출력한 것이다. 예상한대로 process 와 thread 는 동일한 page directory 를 공유하는 것을 확인할 수 있고 thread group id 가 모두 동일한 것을 확인할 수 있다.

- Thread stack challenge

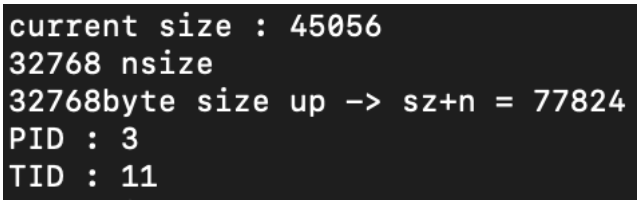
앞서 thread 를 위한 stack 영역을 할당할 때 동작이 예상과 다른 부분이 존재하였다. 기존에 user virtual memory 에서 사용하는 heap 영역에 stack memory 를 할당해서 사용하기 때문에 thread 가 추가될 수록 주소 값이 증가될 줄 알았지만 반대로 감소하는 양상을 보여주었다. 이 부분에 궁금한 점이 생겨 malloc()으로 heap 영역에 memory 할당 시 어떤 동작이 수행되는 지 분석해보았다. Malloc() 함수가 호출되면 기본적으로 free list 에서 사용할 수 있는 memory 를 찾아서 할당하게 된다. 하지만 malloc()이 처음 호출될 때 free list 는 초기화가 되고 free list 에서 할당 받을 수 없기 때문에 morecore()함수를 이용해서 system call 을 통해 free list 에 추가하는 동작을 수행한다. 이때 예상한 동작으로는 thread 가 생성될 때 마다 page 를 할당 할 것이라고 생각했지만 처음에 page 를 여러 개 할당해 둔 뒤 두 번째 thread 가 생성될 때부터 미리 생성한 page 를 할당하는 방식을 사용하였다. 코드를 이용해서 thread stack 을 처음 할당할 때 page 8 개에 해당하는 크기(32768 bytes)만큼 할당이 되는 모습을 확인할 수 있었다.

아래 사진은 첫 번째 thread(tid = 4)를 생성할 시 heap 영역에 할당되는 memory size 를 보여주고 있다.



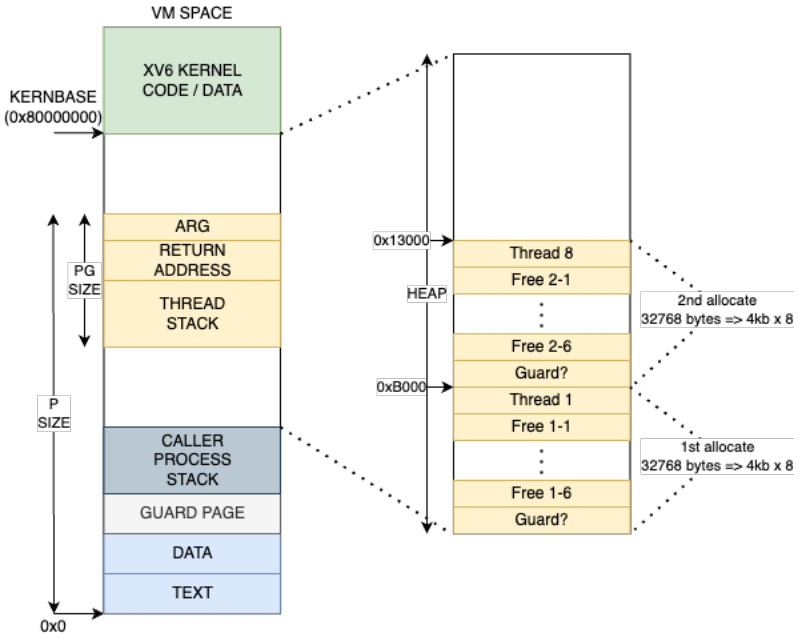
그리고 추가적으로 할당된 page 를 다 사용한 뒤 thread 를 생성했을 때 위의 동작이 반복된다고 예상하였고 실제로 확인해본 결과 32768 bytes 크기만큼의 메모리를 할당하는 모습을 확인할 수 있었다.

아래 사진은 여덟 번째 thread(tid = 11)을 생성할 시 32768 bytes 만큼의 memory 가 추가적으로 할당되는 모습을 보여주고 있다.



다만 page 8 개 만큼의 memory 를 할당한 뒤 가장 아래 존재하는 page 를 사용하지 않고 넘어가는 모습을 확인할 수 있었는데 이는 추가적으로 학습해야 할 부분이라고 생각한다. 또한 xv6 에는 guard page 라는 개념이 존재하기 때문에 가장 마지막 page 를 guard page 로 사용한다고 추측해볼 수 있다.

최종적으로 위의 동작을 그림으로 도식했을 때 다음과 같다.



User virtual memory space 에서 heap 영역에서 malloc()이 수행되면서 thread stack 이 할당되고 malloc()을 처음 수행할 때 여러 개의 page 를 미리 할당해 둔 뒤 다음 thread 를 생성할 때 할당해주는 방식을 사용하고 있다.

Ⅶ. Project 결론

본 프로젝트를 통해 xv6 운영체제의 기본 구조와 동작 원리를 깊이 있게 이해하고, 실제로 시스템 콜, CPU 스케줄러, 그리고 스레드를 구현해보며 운영체제의 핵심 개념들을 직접 다뤄볼 수 있었다. 특히 FCFS 와 MLFQ 스케줄러의 설계 및 구현 과정을 통해 다양한 스케줄링 알고리즘의 차이점을 명확히 이해할 수 있었다.

FCFS 스케줄러는 구현이 상대적으로 간단하고 이해하기 쉬운 반면, 모든 프로세스를 생성 순서대로 처리하기 때문에 특정 프로세스가 과도한 대기 시간을 가질 수 있다는 단점을 확인할 수 있었다. 이를 보완하기 위해 타임아웃을 적용하여 CPU 점유 시간을 제한함으로써 시스템의 응답성을 높였다.

MLFQ 스케줄러는 우선순위 큐와 다양한 타임 퀀텀을 통해 CPU-bound 와 I/O-bound 프로세스를 효과적으로 구분하고 스케줄링하는 방식이다. 이를 통해 시스템의 자원을 보다 효율적으로 관리할 수 있었다. 하지만 구현 과정에서 큐 관리와 프로세스 상태 전환의 복잡성으로 인해 다양한 디버깅 과정을 거쳐야 했다. 특히 SLEEPING 상태에서 RUNNABLE 상태로 전환되는 과정에서 발생한 문제는 추후 더 깊은 분석이 필요하다.

스레드 구현 과정에서는 프로세스와 스레드의 차이점을 명확히 이해하고, 스레드 라이브러리를 설계 및 구현하였다. 스레드는 프로세스 내에서 독립적으로 실행되는 여러 실행 흐름을 가능하게 하며, 이를 통해 멀티스레딩을 구현하여 프로그램의 효율성을 높일 수 있었다. 특히 스레드 생성 시 memory 관점에서 어떤 동작이 수행되는 지 확인할 수 있었으며 프로세스와의 차이점을 알 수 있었다.

이번 프로젝트에서는 다양한 스케줄링 알고리즘을 실험하고 비교하면서, CPU 의 프로세스 스케줄링 메커니즘과 커널의 전반적인 작동 방식을 이해하게 되었다. 또한, 동기화를 위해 필수적으로 필요한 lock 동작과 더불어 API 내부 운영체제의 동작 방식에 대해 깊게 고민하고 탐구해볼 수 있었다. 이러한 내용은 앞으로 더욱 복잡한 운영체제 기능을 구현하고 최적화할 수 있는 역량을 키우는 데 큰 도움이 될 것이다. xv6 라는 교육용 운영체제를 통해 기본기를 다지고, 이를 토대로 실제 운영체제 개발에 필요한 다양한 기술들을 습득해 나갈 계획이다.