운영체제 실습

# [Assignment3]


**Class** : **[B]**

**Professor** : **[최상호 교수님]**

**Student ID** : **[2021202089]**

**Name** : **[오나균]**

# Introduction

이번 Assignment 과제는 3-1, 3-2 , 3-3 으로 나뉜다. .

Assignment 3-1 은 2 차 과제에서 작성한 ftrace 시스템콜을 Process_tracer.c 를 작성하여, asmlinkage pid_t process_tracer(pid_t trace_task) 함수로 wrapping 하여 사용하면 된다. 그리고 Kernel ring buffer 에 다음의 목록을 포함하여 출력하면 된다. [ 현재 프로세스의 상태, Context switch 횟수 ,부모 프로세스 정보, 형제 프로세스 정보, 자식 프로세스 정보] 추가적으로 커널 소스코드를 수정하여 fork()를 호출한 횟수를 구현해야 한다.

Assignment 3-2 는 우선 생성한 프로세스/ 스레드의 수의 2 배만큼 i 번째 값을 정수형 양수로 기록하기 위해서 ./temp.txt 를 생성하여 이를 기록하는 numgen.c 를 작성한다. 다음으로 다중 프로세스를 생성하여, 최상단 프로세스마다 2 개의 숫자를 읽고, 각 프로세스는 두개의 숫자를 더한 후 부모 프로세스에게 값을 전달 한뒤, 최종적으로 나온 값 과 전체 프로그램 수행시간을 측정하는 fork.c 를 작성한다. 또한 이를 스레드로 수행하는 프로그램 또한 thread.c 로 작성한 뒤 Makefile 을 통해서, 결과에 대한 분석 내용을 작성하면 된다.

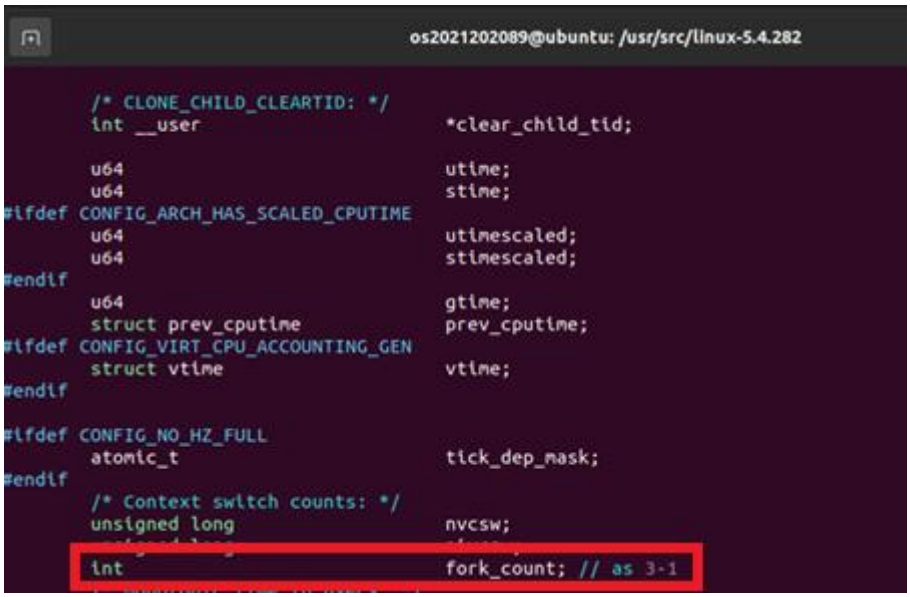
Assignment 3-3 은 도착 시간(Arrival Time) 과 실행 시간(Burst Time) 을 바탕으로, 여러 CPU 스케줄링 알고리즘을 직접 구현하고 그 결과를 비교해야 한다. 구현해야 하는 스케줄링 알고리즘은 FCFS(First Come First Served), SJF(Shortest Job First), SRTF(Shortest Remaining Time First), RR(Round Robin) 이다.
각 알고리즘은 동일한 입력 데이터에 대해서 Gantt Chart, Average Waiting Time, Average Turnaround Time, Average Response Time, CPU Utilization 값을 출력해야 하며, 알고리즘간 성능 차이를 수치적으로 비교 및 분석해야 한다.

# 결과화면

## 1. Assignment 3-1

1) task_strcut 구조체에 fork() 호출 횟수를 저장하는 변수 추가

```
os2021202089@ubuntu:/usr/src/linux-5.4.282$ sudo vi include/linux/sched.h
```




- /usr/src/linux-5.4.282/include/linux/sched.h 경로로 들어가서, int fork_count; 코드를 추가하여 fork 호출 횟수를 저장하는 변수를 추가합니다.

2) fork() 로 자식 프로세스를 생성할 때 해당 변수를 초기화 시킨다.

```
os2021202089@ubuntu:/usr/src/linux-5.4.282$ sudo vi init/init_task.c
```

```
.fork_count     = 0,
.thread_pid     = &init_struct_pid,
.thread_group   = LIST_HEAD_INIT(init_task.thread_group),
```

- init_task 는 정적으로 생성되는 커널 태스크이므로, init/init_task.c 내의 init_task 구조체 초기화 블록에서 fork_count 를 0 으로 명시적으로 초기화하였다. 이는 init 프로세스가 시스템 부팅 시 가장 먼저 존재하는 부모 프로세스이며, 기존에 복사되어 생성되는 프로세스들과 다르게 메모리에서 직접 선언되기 때문이다. 따라서 BSS 로 자동 초기화가 되더라도, 코드의 명확성과 생성 시점의 일관성을 위해 fork_count 값을 명시적으로 0 으로 설정하였다.

3) fork()가 호출될 때마다 해당 변수의 값을 1 씩 증가시킨다.

```
os2021202089@ubuntu:/usr/src/linux-5.4.282$ sudo vi kernel/fork.c
```

os2021202089@ubuntu: /usr/src/linux-5.4.282

```
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 *
 * args->exit_signal is expected to be checked for sanity by the caller.
 */
long _do_fork(struct kernel_clone_args *args)
{
        u64 clone_flags = args->flags;
        struct completion vfork;
        struct pid *pid;
        struct task_struct *p;
        int trace = 0;
        long nr;

        /*
         * Determine whether and which event to report to ptracer.  When
         * called from kernel_thread or CLONE_UNTRACED is explicitly
         * requested, no event is reported; otherwise, report if the event
         * for the type of forking is enabled.
         */
        if (!(clone_flags & CLONE_UNTRACED)) {
                if (clone_flags & CLONE_VFORK)
                        trace = PTRACE_EVENT_VFORK;
                else if (args->exit_signal != SIGCHLD)
                        trace = PTRACE_EVENT_CLONE;
                else
                        trace = PTRACE_EVENT_FORK;

                if (likely(!ptrace_event_enabled(current, trace)))
                        trace = 0;
        }

        p = copy_process(NULL, trace, NUMA_NO_NODE, args);
        add_latent_entropy();

        if (IS_ERR(p))
                return PTR_ERR(p);

        p->fork_count = 0; // Initalize fork_count
        current->fork_count++; // fork_count ++
```

- fork() 호출 시 부모 프로세스의 정보를 복사하여 자식 프로세스를 생성하는 과정에서,자식 프로세스는 생성 시점에 fork 횟수가 존재하지 않으므로 p->fork_count 를 0 으로 초기화한다. 반면, 자식 생성은 부모 프로세스에서 새로운 fork 발생을 의미하므로 current->fork_count 를 1 증가시 부모가 얼마나 많은 자식을 생성했는지 추적할 수 있도록 한다.

4 ) 수정된 커널 소스코드에 대해서, 커널 컴파일 후 재부팅

```
os2021202089@ubuntu:/usr/src/linux-5.4.282$ sudo make -j3
os2021202089@ubuntu:/usr/src/linux-5.4.282$ sudo make modules_install
os2021202089@ubuntu:/usr/src/linux-5.4.282$ sudo make install
os2021202089@ubuntu:/usr/src/linux-5.4.282$ sudo reboot
os2021202089@ubuntu:/usr/src/linux-5.4.282$ uname -r
5.4.282-os2021202089
```

5) Makefile 작성

```
os2021202089@ubuntu:~/Assignment3-1$ vi Makefile
```

os2021202089@ubuntu: ~/Assignment3-1

```
obj-m += Process_tracer.o # 커널 모듈로 컴파일할 오브젝트 파일

KDIR := /lib/modules/$(shell uname -r)/build # 현재 시스템의 커널 빌드 시스템 위치
PWD := $(shell pwd) # 현재 모듈을 컴파일하는 디렉토리 위치

# 빌드
all:
	$(MAKE) -C$(KDIR) M=$(PWD) modules
# 빌드 생성물 제거
clean:
	$(MAKE) -C$(KDIR) M=$(PWD) clean
```

5 ) Process_tracer.c 작성

```
os2021202089@ubuntu:~/Assignment3-1$ mkdir Assignment3-1
os2021202089@ubuntu:~/Assignment3-1$ vi Process_tracer.c
```

os2021202089@ubuntu: ~/Assignment3-1

```c
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/syscalls.h> // __SYSCALL_DEFINEX()
#include <linux/kallsyms.h> // kallsyms_lookup_name()
#include <linux/sched.h>
#include <asm/syscall_wrapper.h> /* __SYCALL_DEFINEx()  */
#include <asm/pgtable_types.h>
#include <asm/ptrace.h>

// sys_call_table pointer
unsigned long **sys_call_table;

// os_ftrace 포인터 저장
typedef asmlinkage long(*syscall_ptregs_t)(const struct pt_regs *);
static syscall_ptregs_t real_os_ftrace;


// sys_call_table 쓰기 on
void make_rw(void *addr)
{
        unsigned int level;
        pte_t *pte = lookup_address((unsigned long)addr,&level);

        if(!(pte->pte & _PAGE_RW))
                pte->pte |= _PAGE_RW;
}
```

**[모듈 로드 시]**

module_init() 함수에서 sys_call_table 주소를 획득하고,

sys_call_table[336] 항목을 os_ftrace_trampoline() 함수로 후킹한다.

**[실행 시]**

User Space -> syscall(336, pid)

-> System Call Table (336 번 엔트리)

-> os_ftrace_trampoline() <- **wrapping**

-> process_tracer(pid)

-> 커널 로그(dmesg)에 프로세스 정보 출력

**[모듈 제거 시]**

module_exit() 함수에서 sys_call_table[336]을 원래 함수로 복원한다.

6) 테스트 코드 작성 (336 번 시스템콜 호출)

```
os2021202089@ubuntu:~/Assignment3-1$ vi test.c
```

os2021202089@ubuntu: ~/Assignment3-1

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>

#ifndef __NR_ftrace
#define __NR_ftrace 336   // 네 커널에서의 os_ftrace 번호가 진짜 336인지 곧 검증
#endif

int main(int argc, char **argv){
    pid_t pid = (argc > 1) ? (pid_t)atoi(argv[1]) : getpid();
    long ret = syscall(__NR_ftrace, pid);
    if (ret == -1) {
        printf("syscall(%d,%d) failed: errno=%d (%m)\n", __NR_ftrace, pid, errno);
    } else {
        printf("syscall(%d,%d) success: ret=%ld\n", __NR_ftrace, pid, ret);
    }
    return 0;
}
~
~
```

7) 모듈 삽입

```
os2021202089@ubuntu:~/Assignment3-1$ sudo insmod Process_tracer.ko
```

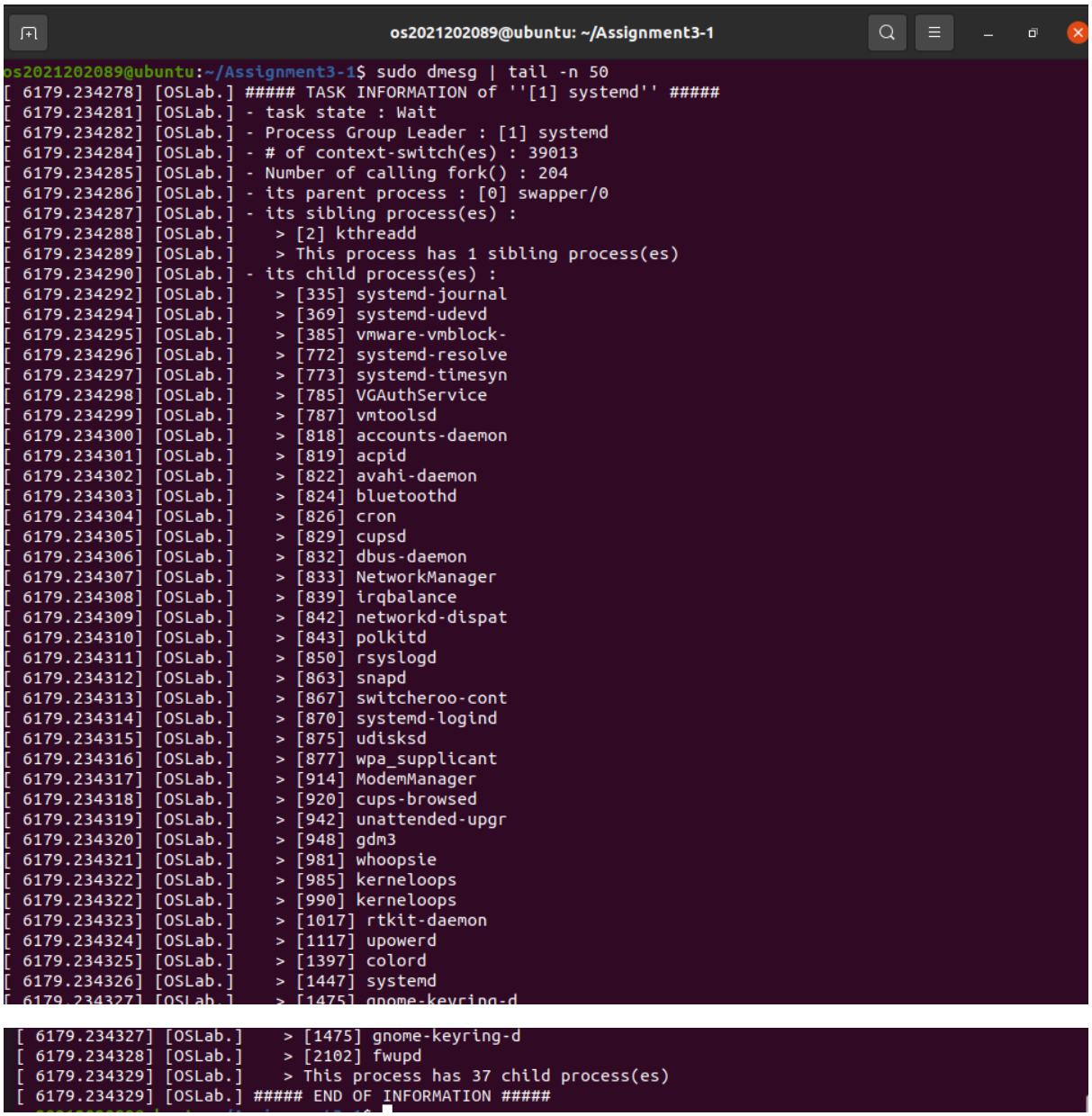
sudo insmod Process_tracer.ko 명령어를 사용하여, 커널에 모듈을 로드한다.

8) 테스트 프로그램 컴파일 및 실행

```
os2021202089@ubuntu:~/Assignment3-1$ gcc test.c -o test
os2021202089@ubuntu:~/Assignment3-1$ ./test 1
syscall(336,1) success: ret=1
```

다음으로 gcc test.c -o test 명령어를 통해서, 테스트 코드를 컴파일하고, ./test 1 명령어를 통해서 PID 1 의 정보를 커널 버퍼에 기록한다.

9) 커널 로그 출력 확인




커널 로그를 확인해보면 PID를 바탕으로 , (1)프로세스 이름, (2) 현재 프로세스의 상태, (3)프로세스 그룹 정보, (4) 해당 프로세스를 실행하기 위해 수행된 context switch 횟수, (5)fork()를 호출한 횟수, (6)부모 프로세스 정보 , (7)형제자매 프로세스 정보, (8)자식 프로세스 정보가 과제 조건에 맞게 출력되는 것을 확인할 수 있습니다.

## 2. Assignment 3-2

1) numgen.c 작성

```
os2021202089@ubuntu:~/as3-2$ vi numgen.c
```

os2021202089@ubuntu: ~/as3-2

```c
#include <stdio.h>
#include <stdlib.h>

#ifndef MAX_PROCESSES
#define MAX_PROCESSES 8
#endif

int main()
{
        const int N = MAX_PROCESSES * 2;
        FILE* f = fopen("./temp.txt","w"); // open  temp.txt

        // except
        if(!f){
                perror("fopen");
                return 1;
        }

        // 생성할 프로세스 수의 2배만큼 i번째 값을 기록
        for(int i = 1; i<=N;i++)
                fprintf(f,"%d\n",i);

        fclose(f); // close temp.txt

        return 0;

}
```

- 특정 파일 ./temp.txt 를 생성하고, 파일에서 i 번째 값을 integer 형 양수으로 생성할 프로세스 수의 2 배만큼(MAX_PROCESSES) 기록하는 nungen.c 코드를 작성한다.

2) fork.c , thread.c 작성

```
os2021202089@ubuntu:~/as3-2$ vi fork.c
```

os2021202089@ubuntu: ~/as3-2

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // fork(), exit()
#include <sys/wait.h> // waitpid()
#include <time.h>    // clock_gettime()

// 프로세스 개수 기본값 설정
#ifndef MAX_PROCESSES
#define MAX_PROCESSES 8
#endif

static int read_input(int* out)
{
        // 입력 파일 열기
```

```
os2021202089@ubuntu:~/as3-2$ vi thread.c
```

os2021202089@ubuntu: ~/as3-2

```c
#include <stdio.h>
#include <stdlib.h>
#  de <pthread.h>
#  de <time.h>    // clock_gettime()

// 프로세스 개수 기본값 설정
#ifndef MAX_PROCESSES
#define MAX_PROCESSES 8
#endif

typedef struct{
        int a,b;
}Task;
```

Files

Step 1. MAX_PROCESSES 만큼 프로세스 또는 스레드를 생성

( MAX_PROCESSES = 8, 64 )

Step 2. 최상단 프로세스/스레드마다 2 개의 숫자를 읽음.

Step 3. 각 프로세스/스레드는 두 개의 숫자를 더한 후, 부모 프로세스/스레드에게 값을 전달 ( fork à exit() 사용)

Step 4. 최종적으로 나온 값, 전체 프로그램 수행시간 측정

다음의 작업을 다중 프로세스/ 스레드로 수행하는 fork.c , thread.c 를 작성합니다.

3) Makefile 작성

```
CC = gcc
CFLAGS = -O2 -Wall -Wextra
THREAD_FLAGS = -pthread

# 기본 프로세스 개수 = 8
MAX ?=8

# 실행 파일 이름
TARGETS = numgen fork thread

# 8개버전
all: $(TARGETS)

numgen: numgen.c
	$(CC) $(CFLAGS) -DMAX_PROCESSES=$(MAX) -o $@ $<

fork: fork.c
	$(CC) $(CFLAGS) -DMAX_PROCESSES=$(MAX) -o $@ $<

thread: thread.c
	$(CC) $(CFLAGS) $(THREAD_FLAGS) -DMAX_PROCESSES=$(MAX) -o $@ $<


# 64개 버전
all64:
	$(MAKE) clean
	$(MAKE) all MAX=64

# 실행 도우미
run:
	./numgen
	./fork
	./thread

# 캐시 및 파일 정리
clean:
	rm -f $(TARGETS) temp.txt
~
```

- Makefile 은 MAX_PROCESSES 값을 매크로 형태로 프로그램에 전달하여 프로세스/스레드 수를 쉽게 조정할 수 있도록 구성되었다.
  make 명령을 통해 기본(8 개) 버전 빌드가 가능하고, make all64 명령을 통해 64 개 버전 실행 파일을 자동으로 생성할 수 있다. 또한, run 및 clean 규칙을 통해 실행 및 파일 정리를 손쉽게 수행할 수 있도록 구현되었다.

**MAX_PROCESSES = 8 (프로세스/스레드 수)**

매 실험전 캐시 및 버퍼를 비우는 명령어를 수행하여, 영향을 받지 않게끔 합니다.

프로그램 빌드 및 temp.txt 파일 생성

```
os2021202089@ubuntu:~/as3-2$ make clean && make
rm -f numgen fork thread temp.txt
gcc -O2 -Wall -Wextra -DMAX_PROCESSES=8 -o numgen numgen.c
gcc -O2 -Wall -Wextra -DMAX_PROCESSES=8 -o fork fork.c
gcc -O2 -Wall -Wextra -pthread -DMAX_PROCESSES=8 -o thread thread.c
os2021202089@ubuntu:~/as3-2$ ./numgen
```

./fork

```
os2021202089@ubuntu:~/as3-2$ rm -rf tmp*
os2021202089@ubuntu:~/as3-2$ sync
os2021202089@ubuntu:~/as3-2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2021202089@ubuntu:~/as3-2$ ./fork
value of fork : 136
0.001647
```

./thread

```
os2021202089@ubuntu:~/as3-2$ rm -rf tmp*
os2021202089@ubuntu:~/as3-2$ sync
os2021202089@ubuntu:~/as3-2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2021202089@ubuntu:~/as3-2$ ./thread
value of thread : 136
0.001151
```

**정확성**

- ⇨ fork 방식과 thread 방식 모두 1 + 2 + ... + (2×MAX) = n(n+1)/2, MAX=8
- ⇨ n=16 -> 16×17/2 = 136 으로 알맞게 나오는 것을 확인 할 수 있다.

성능(경과시간)

- ⇨ fork = 0.001647s , thread = 0.001151s 로 스레드가 약 1.4 배 빠르다.

**MAX_PROCESSES = 64 (프로세스/스레드 수)**

매 실험전 캐시 및 버퍼를 비우는 명령어를 수행하여, 영향을 받지 않게끔 합니다.

```
os2021202089@ubuntu:~/as3-2$ make clean && make all64
rm -f numgen fork thread temp.txt
make clean
make[1]: Entering directory '/home/os2021202089/as3-2'
rm -f numgen fork thread temp.txt
make[1]: Leaving directory '/home/os2021202089/as3-2'
make all MAX=64
make[1]: Entering directory '/home/os2021202089/as3-2'
gcc -O2 -Wall -Wextra -DMAX_PROCESSES=64 -o numgen numgen.c
gcc -O2 -Wall -Wextra -DMAX_PROCESSES=64 -o fork fork.c
gcc -O2 -Wall -Wextra -pthread -DMAX_PROCESSES=64 -o thread thread.c
make[1]: Leaving directory '/home/os2021202089/as3-2'
os2021202089@ubuntu:~/as3-2$ ./numgen
```

```
os2021202089@ubuntu:~/as3-2$ rm -rf tmp*
os2021202089@ubuntu:~/as3-2$ sync
os2021202089@ubuntu:~/as3-2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2021202089@ubuntu:~/as3-2$ ./fork
value of fork : 8256
0.026298
```

```
os2021202089@ubuntu:~/as3-2$ rm -rf tmp*
os2021202089@ubuntu:~/as3-2$ sync
os2021202089@ubuntu:~/as3-2$ echo 3 | sudo tee /proc/sys/vm/drop_caches
3
os2021202089@ubuntu:~/as3-2$ ./thread
value of thread : 8256
0.017331
```

⇨ 두 방식 모두 temp.txt 의 1 ~ 128 을 모두 더한 값이므로, 8256 으로 동일하다. 이를 통해 각 프로세스/스레드가 올바르게 분할 계산하고 값을 전달했음을 알 수 있다.

⇨ 스레드 방식(thread) 의 수행 시간이 fork 방식보다 약 1.5 배 빠르다. 이는 fork() 는 각 자식 프로세스를 생성할 때 부모의 메모리 공간을 복사해야 하므로, 프로세스 생성 오버헤드가 크다.

⇨ 이와 달리 스레드 방식에서 pthread() 는 동일한 주소 공간을 공유하며, 스택만 별도로 할당되므로 문맥 전환(Context Switch) 비용이 작고 메모리 효율이 높음.

⇨ 또한 프로세스/스레드 수가 증가함에 따라 전체 수행 시간이 증가하였는데, 이는 CPU 코어 수를 초과한 스케줄링 및 문맥 전환 비용이 증가했기 때문이다.

따라서 병렬 연산에서는 스레드 기반 구현이 더 효율적임을 확인할 수 있다.

| MAX_PROCESS | fork<br>실행시간 | Thread<br>실행시간 | 성능<br>(fork/thread) | 최종적으로<br>나온 값 |
|---|---|---|---|---|
| 8 | 0.001647s | 0.001151s | 약 1.4 | 136 |
| 64 | 0.026298s | 0.017331s | 약 1.5 | 8256 |

**결과 분석**

**a. 최종적으로 나온 값**

MAX_PROCESSES = 8 인 경우 입력으로 사용된 정수의 범위가 1 부터 16 까지이므로, 각 프로세스/스레드가 두 개씩 값을 더한 결과를 모두 합산하면 최종 결과는 136 이 된다. MAX_PROCESSES = 64 인 경우 입력 범위가 1 부터 128 까지로 확장되며 동일한 방식으로 합산될 때 최종 결과는 8256 이 된다.

**b. 스레드 방식의 우위**

측정 결과, 스레드 방식이 프로세스 방식 대비 약 1.4~1.5 배 더 빠른 성능을 보였습니다. 즉, 스레드를 사용했을 때 총 실행 시간이 프로세스보다 짧게 측정되었습니다.

이러한 성능 차이는 주로 프로세스 생성 overhead 와 context switch 비용에서 기인합니다.

process: 새 프로세스는 독립적인 주소 공간과 커널 자원(페이지 테이블, 파일 디스크립터 등)을 생성하거나 복제해야 하므로, 생성 과정(fork 시간)에 높은 비용이 발생합니다.

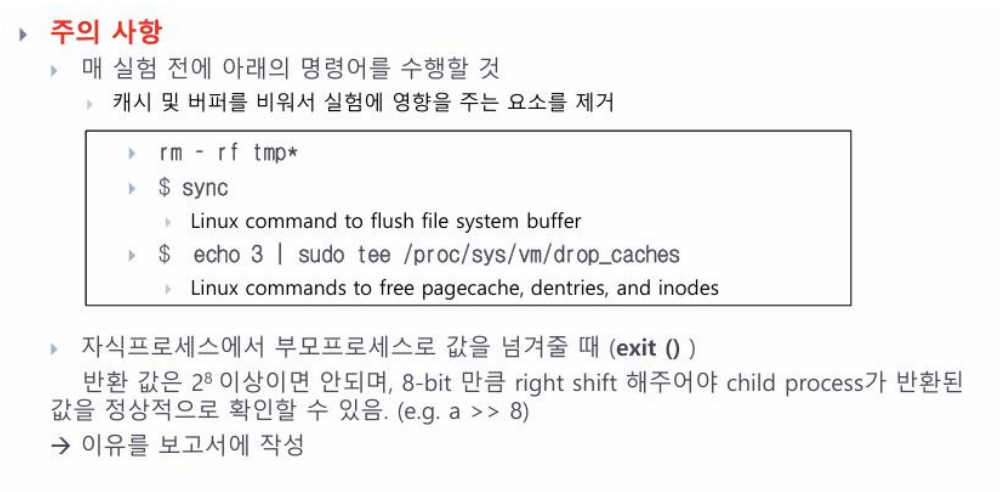
thread: 스레드는 주소 공간을 공유하며 최소한의 실행 상태(스택, 레지스터)만 생성하므로, 프로세스 생성에 비해 오버헤드가 훨씬 가볍습니다.

### c. 일관된 스레드의 우위

MAX_PROCESSES 가 8 에서 64 로 증가했음에도 불구하고, 두 방식 간의 성능 비율(1.4 배 ,1.5 배)은 유사하게 유지됩니다. 이는  동시 실행 개수(N)가 증가함에 따라, 두 방식 모두 오버헤드 역시 선형적으로 증가합니다. 프로세스 방식의 높은 오버헤드가 N 배로 증가하고, 스레드 방식의 낮은 오버헤드가 N 배로 증가하면서, 두 오버헤드의 비율 자체는 N 값과 무관하게 안정적으로 유지됩니다.

### d. 결론

결론적으로, 스레드가 프로세스 대비 약 1.4~1.5 배로 더 빠른 것은 OS 자원 생성 및 전환 비용 차이 때문이며, N 의 변화에도 이 비율이 유지되는 것은 생성/스케줄링 오버헤드가 전체 시간에 지배적이기 때문입니다.




### 실험 전 캐시 초기화

프로그램 실행 시간은 page cache, directory entry cache, inode 캐시등의 영향을 받는다. 따라서 이를 제거하기 위해 다음의 3 개 명령어를 수행해야한다.
rm -rf tmp*, sync, echo 3 | sudo tee /proc/sys/vm/drop_caches 명령을 수행하였다.
이를 통해 I/O 캐시가 초기화된 동일한 환경에서 각 실행을 비교할 수 있다.

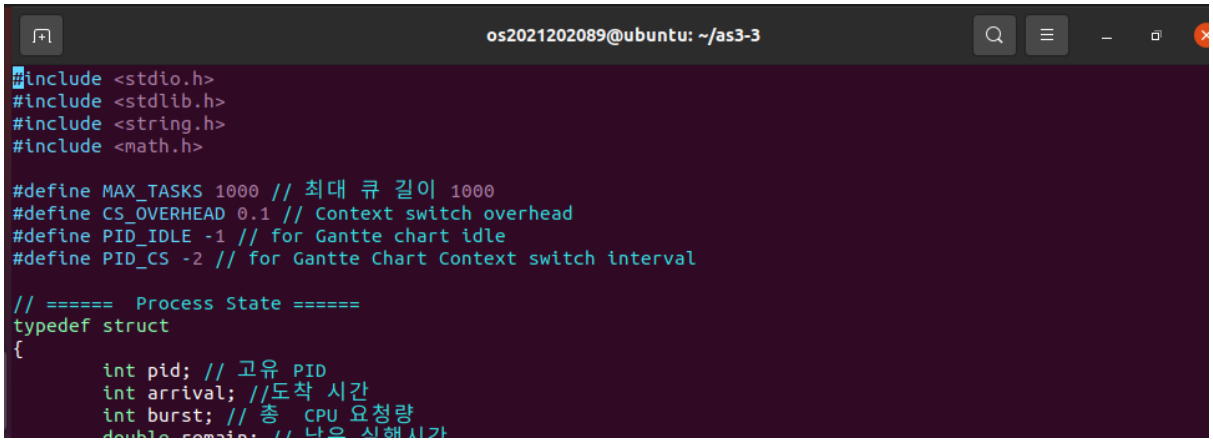
**exit() 값의 8 비트 제한**

자식 프로세스의 exit() 반환 값은 커널 내부적으로 8 비트로 저장된다.
따라서 256 이상의 값은 상위 비트가 손실되며, (value & 0xFF) 또는 (value >> 8) 처리를 통해 올바른 값을 전달해야 한다. 이는 리눅스의 wait() 시스템 콜이 하위 8 비트만을 WEXITSTATUS(status)로 반환하기 때문이다.

## 3. Assignment 3-3

1) cpu_scheduler.c 작성

```
os2021202089@ubuntu:~/as3-3$ vi cpu_scheduler.c
```



2) Makeifle 작성

os2021202089@ubuntu: ~/as3-3

```
CC = gcc
CFLAGS = -Wall -O2 -std=c11
LDLIBS = -lm              # ⬅math 라이브러리 추가

TARGET = cpu_scheduler
SRC = cpu_scheduler.c

all: $(TARGET)


$(TARGET): $(SRC)
        $(CC) $(CFLAGS) -o $@ $(SRC) $(LDLIBS)

clean:
        rm -f $(TARGET)

~
```

3) make

```
os2021202089@ubuntu:~/as3-3$ make clean && make
rm -f cpu_scheduler
gcc -Wall -O2 -std=c11 -o cpu_scheduler cpu_scheduler.c -lm
```

## Test case 01

```
os2021202089@ubuntu:~/as3-3$ cat input.1
1 0 10
2 0 9
3 3 5
4 7 4
5 10 6
6 10 7
os2021202089@ubuntu:~/as3-3$  ./cpu_scheduler input.1 FCFS
Gantt Chart:
| P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P3 | P3 |
P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 | P6 |
Average Waiting time =  14.42
Average Turnaround Time = 21.25
Average Response time = 14.42
CPU utilization = 98.80 %
os2021202089@ubuntu:~/as3-3$ ./cpu_scheduler input.1 SJF
Gantt Chart:
| P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P2 | P4 | P4 | P4 | P4 | P3 | P3 | P3 | P3 | P3 | P5 | P5 | P5 |
P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 | P6 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 |
Average Waiting time =  11.08
Average Turnaround Time = 17.92
Average Response time = 11.08
CPU utilization = 98.80 %
os2021202089@ubuntu:~/as3-3$  ./cpu_scheduler input.1 RR 2
Gantt Chart:
| P1 | P1 | P2 | P2 | P1 | P1 | P3 | P3 | P2 | P2 | P1 | P1 | P4 | P4 | P3 | P3 | P5 | P5 | P6 | P6 | P2 |
P2 | P1 | P1 | P4 | P4 | P3 | P5 | P5 | P6 | P6 | P2 | P2 | P1 | P1 | P5 | P5 | P6 | P6 | P2 | P6 |
Average Waiting time =  24.18
Average Turnaround Time = 31.02
Average Response time = 4.45
CPU utilization = 95.13 %
os2021202089@ubuntu:~/as3-3$  ./cpu_scheduler input.1 SRTF
Gantt Chart:
| P2 | P2 | P2 | P3 | P3 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P2 | P2 | P2 | P2 | P2 | P2 | P5 | P5 | P5 |
P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 | P6 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 |
Average Waiting time =  10.85
Average Turnaround Time = 17.68
Average Response time = 9.30
CPU utilization = 98.56 %
```

### Test case 02

```
os2021202089@ubuntu:~/as3-3$ cat input.2
1 0 1
2 0 2
3 0 3
4 0 4
5 0 5
6 0 6
os2021202089@ubuntu:~/as3-3$ ./cpu_scheduler input.2 FCFS
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 |
Average Waiting time =  6.08
Average Turnaround Time = 9.58
Average Response time = 6.08
CPU utilization = 97.67 %
os2021202089@ubuntu:~/as3-3$ ./cpu_scheduler input.2 SJF
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 |
Average Waiting time =  6.08
Average Turnaround Time = 9.58
Average Response time = 6.08
CPU utilization = 97.67 %
os2021202089@ubuntu:~/as3-3$ ./cpu_scheduler input.2 RR 2
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P4 | P4 | P5 | P5 | P6 | P6 | P3 | P4 | P4 | P5 | P5 | P6 | P6 | P5 | P6 | P6 |
Average Waiting time =  8.75
Average Turnaround Time = 12.25
Average Response time = 4.42
CPU utilization = 95.02 %
os2021202089@ubuntu:~/as3-3$  ./cpu_scheduler input.2 SRTF
Gantt Chart:
| P1 | P2 | P2 | P3 | P3 | P3 | P4 | P4 | P4 | P4 | P5 | P5 | P5 | P5 | P5 | P6 | P6 | P6 | P6 | P6 | P6 |
Average Waiting time =  6.08
Average Turnaround Time = 9.58
Average Response time = 6.08
CPU utilization = 97.67 %
```

### Test case 03

```
os2021202089@ubuntu:~/as3-3$ cat input.3
1 0 6
2 1 5
3 2 4
4 3 3
5 5 2
6 6 1
os2021202089@ubuntu:~/as3-3$ ./cpu_scheduler input.3 FCFS
Gantt Chart:
| P1 | P1 | P1 | P1 | P1 | P1 | P2 | P2 | P2 | P2 | P2 | P3 | P3 | P3 | P3 | P4 | P4 | P4 | P5 | P5 | P6 |
Average Waiting time =  9.08
Average Turnaround Time = 12.58
Average Response time = 9.08
CPU utilization = 97.67 %
os2021202089@ubuntu:~/as3-3$ ./cpu_scheduler input.3 SJF
Gantt Chart:
| P1 | P1 | P1 | P1 | P1 | P1 | P6 | P5 | P5 | P4 | P4 | P4 | P3 | P3 | P3 | P3 | P2 | P2 | P2 | P2 | P2 |
Average Waiting time =  5.75
Average Turnaround Time = 9.25
Average Response time = 5.75
CPU utilization = 97.67 %
os2021202089@ubuntu:~/as3-3$ ./cpu_scheduler input.3 RR 2
Gantt Chart:
| P1 | P1 | P2 | P2 | P3 | P3 | P1 | P1 | P4 | P4 | P2 | P2 | P5 | P5 | P6 | P3 | P3 | P1 | P1 | P4 | P2 |
Average Waiting time =  12.18
Average Turnaround Time = 15.68
Average Response time = 4.17
CPU utilization = 95.02 %
os2021202089@ubuntu:~/as3-3$  ./cpu_scheduler input.3 SRTF
Gantt Chart:
| P1 | P1 | P1 | P1 | P1 | P1 | P6 | P5 | P5 | P4 | P4 | P4 | P3 | P3 | P3 | P3 | P2 | P2 | P2 | P2 | P2 |
Average Waiting time =  5.75
Average Turnaround Time = 9.25
Average Response time = 5.75
CPU utilization = 97.67 %
```

## 각 알고리즘별 성능 분석

**Test case 01)**

| | Average Waiting time | Average Turnaround time | Average Response time | CPU Utilization |
|---|---|---|---|---|
| FCFS | 14.42 ms | 21.25 ms | 14.42 ms | 98.80 % |
| SJF | 11.08 ms | 17.92 ms | 11.08 ms | 98.80 % |
| SRTF | 10.85 ms | 17.68. ms | 9.30 ms | 98.56 % |
| RR | 24.18 ms | 31.02 ms | 4.45 ms | 95.13 % |

**Test case 02)**

| | Average Waiting time | Average Turnaround time | Average Response time | CPU Utilization |
|---|---|---|---|---|
| FCFS | 6.08 ms | 9.58 ms | 6.08 ms | 97.67 % |
| SJF | 6.08 ms | 9.58 ms | 6.08 ms | 97.67 % |
| SRTF | 6.08 ms | 9.58 ms | 6.08 ms | 97.67 % |
| RR | 8.75 ms | 12.25 ms | 4.42 ms | 95.02 % |

**Test case 03)**

| | Average Waiting time | Average Turnaround time | Average Response time | CPU Utilization |
|---|---|---|---|---|
| FCFS | 9.08 ms | 12.58 ms | 9.08 ms | 97.67 % |
| SJF | 5.75 ms | 9.25 ms | 5.75 ms | 97.67 % |
| SRTF | 5.75 ms | 9.25 ms | 5.75 ms | 97.67 % |
| RR | 12.18 ms | 15.68 ms | 4.17 ms | 95.02 % |

**본 실험에서는 동일한 프로세스 집합(input1 ,input2 ,input3)에 대해 FCFS, SJF, RR, SRTF 스케줄링 알고리즘을 적용하고, 각 알고리즘의 평균 대기 시간(Waiting Time), 평균 반환 시간(Turnaround Time),평균 응답 시간(Response Time), CPU 활용도를 비교하였다.**

**Test case 01 분석**

- SRTF 가 가장 낮은 대기 시간과 반환 시간, 응답 시간을 보여 가장 효율적이다.
- RR 은 응답 시간은 가장 빠르지만, Time Quantum 으로 인해 Context Switching 오버헤드가 발생하여 , 대기 시간과 반환 시간이 크게 증가하였다.
- FCFS 는 작업 순서에 의존하기 때문에 Convoy(호위) 효과로 인해 성능이 떨어진다.
- SJF 는 비선점 방식이지만 평균 대기 시간과 반환 시간이 FCFS 와 대비해서, 효율적이다. ( Shortest Job First 특성)

**Test case 02 분석**

- FCFS, SJF, SRTF 가 모두 동일한 결과를 보이고 있는데, input.2 의 입력값에 대해서
  이는 모든 프로세스의 도착 시간이 0 으로 동일하고, Burst Time 또한 단조적으로 증가하는 형태라서, Ready Queue 에 처음부터 모든 프로세스가 동시에 존재한다, , 이에 따라 SJF,SRTF 가 FCFS 와 동일한 순서로 실행되었기 때문에 동일한 결과를 보인다.
- RR 은 응답 시간은 짧지만, Context Switching 으로 인해 Average Turnadround time 이 증가하였다.
- 그리고 Test case02 를 통해서, 프로세스 실행 시간이 균일하면 FCFS 도 충분히 효율적으로 동작할 수 있다는 점을 알 수 있다.

**Test case 03 분석**

Test Case 03 에서는 프로세스마다 Burst Time(실행 시간)의 편차가 크기 때문에, 스케줄링 알고리즘에 따라 성능 차이가 뚜렷하게 나타난다.

- 먼저 FCFS(First-Come First-Served) 은 도착 순서대로 실행을 진행하므로, 긴 실행 시간을 가진 프로세스가 먼저 실행될 경우 후순위 프로세스들이 장시간 대기해야 한다. 따라서 평균 대기 시간(Waiting Time)과 반환 시간(Turnaround Time)이 상대적으로 크게 나타난다.

- SJF/ SRTF 에 대해서, SJF 는 현재 시점에 도착해 있는 프로세스 중 Burst time 이 가장 짧은 프로세스 를 선택하고, SRTF 는 선점 방식으로 실행 도중이라도 새로 들어온 프로세스가 실행 시간이 더 짧으면 선점한다. 그런데 input.3 에서는 실행중인 프로세스의 남은 시간보다 더 짧은 Burst Time 을 가진 프로세스가 도착할 수가 없어서 (도착 시간과 Burst 의 대칭적인

증가/감소 구조)로 인해 SRTF 가 선점하는 상황이 나오지 않는다. 따라서 두 알고리즘 모두 동일한 순서를 따르게 되고, 결과도 동일하게 나온다.

- 한편, RR(Round Robin) 은 Time Quantum 단위로 CPU 를 순환 배분하기 때문에, 응답 시간(Response Time)은 가장 짧게 유지되지만, Context Switching 이 빈번하게 발생한다. 이로 인해 대기 시간과 반환 시간은 다른 알고리즘보다 가장 비효율적인 결과를 보인다

**종합 결론 )**

본 실험을 통해 프로세스 스케줄링 알고리즘은 작업 도착 시점과 실행 시간(Burst Time)의 분포에 따라 성능 차이가 명확하게 나타난다는 것을 확인할 수 있었다.

- SRTF 는 작업 실행 시간을 동적으로 고려하고 필요 시 선점을 수행하기 때문에,
  대기 시간·반환 시간·응답 시간 측면에서 가장 효율적인 알고리즘임을 확인하였다.

- SJF 는 선점이 없다는 점에서 SRTF 보다는 유연성이 떨어지지만,
  Burst Time 편차가 존재하는 작업 집합(input1,input3)에서 FCFS 대비 성능 향상 효과가 확실하게 나타났다.

- FCFS 는 구현이 단순하지만, 실행 시간이 긴 작업이 먼저 도착하는 경우
  Convoy Effect(호위 효과) 로 인해 전체 성능이 악화될 수 있다.

- RR 스케줄링은 응답 시간 측면에서 가장 우수하지만,
  Time Quantum 단위의 잦은 Context Switching 오버헤드로 인해 대기 시간과 반환 시간이 증가하는 단점이 있다.

또한 입력 데이터가 동시에 도착하거나 실행 시간이 균일한 경우 (input2 경우),
SJF / SRTF / FCFS 가 동일한 스케줄링 결과를 나타낼 수 있음을 Test Case 02 를 통해 확인하였다.
반면, Burst Time 에 차이가 존재하는 경우, SJF/SRTF 는 FCFS 대비 확실한 성능 이점을 보였다.

따라서, 스케줄링 알고리즘의 선택은 프로세스들의 도착 시점과 실행 시간 분포에 따라 달라져야 하며, CPU 집중형 작업이 다양한 환경에서는 SRTF 가 가장 효율적이고,
응답 속도가 중요한 시스템에서는 RR 이 적합하다는 점을 확인할 수 있다.

# 고찰

이번 Assignment 3 을 진행하면서, 단순한 사용자 레벨 프로그램이 아니라 운영체제 내부의 동작 방식을 직접 다룬다는 것이 얼마나 복잡하고 미세한 요소에 의해 좌우되는지 알 수 있었다.

특히 3-1 에서 fork_count 를 커널에 추가하고 system call wrapping 을 구현할 때, sys_call_table 수정으로 인한 커널 패닉(Kernel Panic) 을 여러 번 겪었는데, 또 수정하고 커널 컴파일을 해야되는데, 이때 커널 컴파일이 너무 오래 걸려서, 상당히 머리가 아팠던 것 같다. 이 경험을 통해 커널 공간의 수정은 단 한 줄의 실수도 허용되지 않는다는 것을 배웠다. 또한 pt_regs 기반으로 system call wrapper 를 구성하면서 사용자 레벨에서 전달되는 인자가 커널 내부에서는 어떻게 처리되는지를 명확히 이해할 수 있었다.

3-2 에서는 프로세스 생성과 스레드 생성의 비용 차이를 실험적으로 비교해보면서, 문서로만 보던 스레드가 상대적으로 가볍다는 개념이 실제 실행 시간에서도 명확히 드러난다는 것을 확인했다. 특히 캐시 및 파일시스템 버퍼가 측정 결과에 영향을 준다는 점에서, 정확한 실험 환경을 통제하는 것이 실험 분석에서 매우 중요함을 배울 수 있었다.

3-3 에서는 스케쥴러 알고리즘 FCFS,SJF,SRTF,RR 을 C 언어를 통해서, 구현해야 됐는데, 확실히 이론을 구체적인 코드로 옮기는 과정은 상당히 복잡한 과정인 것같다. 그래도 이를 통해서, 스케줄러 알고리즘의 특징과 성능 차이를 수치적으로 비교해보면서, 이론 시간에 이해했던 알고리즘들을 실제 실행 결과로 연결해 볼수  있었다. 동일한 입력에서도 도착 시간과 Burst Time 분포에 따라 성능 차이가 드러나는 과정은, 스케줄링 알고리즘은 정답이 아니라 상황에 따른 적합성의 문제라는 점을 알게 해주었다..

종합적으로, 이번 과제를 통해 운영체제는 단순히 이론으로만 이해해서는 안 되고, 실제 구현과 실험을 통해 동작 원리를 정확히 이해해야 한다는 것을 배웠다. 더불어, 작은 수정이라도 시스템 전체에 영향을 미치기 때문에, 논리적인 설계와 세심한 디버깅 능력이 중요한 것을

# Reference


https://man7.org/linux/man-pages/man2/wait.2.html

: WEXITSTATUS(status))는 status 의 lower 8bits 를 반환한다는 것이 명시되어있다. (As3-2)

https://man7.org/linux/man-pages/man3/exit.3.html

: exit()의 종료코드가 0 ~255 즉 8bit 라고 서술되어있음. (As3-2)

https://www.kernel.org/doc/Documentation/sysctl/vm.txt

: " Writing 1 to /proc/sys/vm/drop_caches will clear pagecache.
Writing 2 will clear dentries and inodes.
Writing 3 will clear all caches.
This is intended for testing, benchmarking or memory debugging.
It does not affect dirty page " (As3-2)

https://man7.org/linux/man-pages/man2/sync.2.html

: "sync() flushes the filesystem buffers to disk. " (As3-2)