컴퓨터학부 20202920 조민혁

1.개요

기존의 xv6의 스케줄러를 분석하고 그 작동과정을 정리하고 주석을 작성하며 순서도를 그린 후, 다단계 피드백 큐 스케줄링에 대한 학습을 합니다. 다단계 피드백 큐를 변형한 프로세스의 실행 패턴에 따라 동적 우선순위 조정이 가능한 형태를 구현합니다. 구현하는 다단계 피드백 큐의 경우는 우선순위를 io\_wait\_time과 나중에 들어온 것을 우선순위를 설정하여 스케줄링할 수 있도록 해줍니다. 그 후 테스트 프로그램을 생성하여 구현한 mlfq기반의 스케줄러를 테스트할 수 있도록 pid와 사용한 cpu\_burst , set실행 terminated가 실행된 것과 같이 DEBUG 플래그를 설정하여 수행한 경우에는 결과들을 가시적으로 볼 수 있도록 하여줍니다. 또한 새로운 set\_proc\_info시스템 콜을 사용하여 사용자가 원하는 위치의 level, cpu\_burst시간, cpu\_wait, io\_wait\_time,end\_time을 설정하여 줄 수 있도록 합니다. 해당 구현을 완료한 후 기존 xv6 스케줄러와 새롭게 만든 SSU 스케줄러의 기능 및 성능을 비교 분석합니다.

2.상세설계

A.기존 xv6 스케쥴러 분석

(1)소스코드 분석 및 기능 설명

저는 xv6에 스케쥴러에 영향을 끼치는 것들은 proc.c, proc.h, trap.c, traps.h 에 있는 함수와 구조체들이라고 생각하였습니다. 먼저 proc.h에 있는 구조체들에 대해서 설명하겠습니다.

//cpu가 현재 프로세스의 실행을 중단하고 다른 프로세스를 실행할 때 현재 프로세스의 실행 상태(레지스터 값)을 저장하고 나중에 복원하는데 사용하는 것이다.

struct context {

uint edi;

uint esi;

uint ebx;

uint ebp;

uint eip; //현재 실행중인 명령어 주소를 의미한다.

};

context 구조체는 위와 같이 현재 프로세스의 실행을 중단하고 다른 프로세스를 실행할 때 현재 프로세스의 실행 상태를 저장하고 복원하기 위해서 사용하는 것으로, 스케쥴링되어서 cpu를 통해서 작업을 수행한 후, timer 인터럽트가 발생하거나 I/O작업이 발생하였을 때 자신에게 주어진 시간이 끝나기 전의 상태를 저장할 수 있게 만들어주는 구조체입니다.

//프로세스가 가질 수 있는 상태들을 나타낸 것이다.

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

procstate enum은 프로세스가 가진 상태들을 나타내는 enum입니다. 앞에서부터 UNUSED는 사용되지 않은 프로세스를 의미하고, EMBRYO는 프로세스가 생성 중인 상태를 나타내며, SLEEPING은 I/0작업을 하는 등의 수면 상태를 의미하며, RUNNABLE을 스케쥴링이 되기 위해서 기다리고 있는 상태를 의미하며, RUNNING은 현재 실행 중인 상태, ZOMBIE는 프로세스가 종료되기 전에 KILLED상태이지만 종료되지 않은 상태를 의미합니다. 이런 procstate 구조체를 통해서는 프로세스의 상태를 나타낼 수 있습니다.

// Per-process state

struct proc {

uint sz; // Size of process memory (bytes) 프로세스의 사이즈

pde\_t\* pgdir; // Page table 페이지테이블의 주소

char \*kstack; // Bottom of kernel stack for this process 커널 스택의 주소

enum procstate state; // Process state 현재 프로세스의 상태

int pid; // Process ID 프로세스 ID

struct proc \*parent; // Parent process 부모 프로세스

struct trapframe \*tf; // Trap frame for current syscall 트랩프레임을 가리키는 포인터로 시스템 콜이나 인터럽트가 발생하였을 때 CPU의 상태를 저장한다.

struct context \*context; // swtch() here to run process context구조체를 통해 문맥을 저장

void \*chan; // If non-zero, sleeping on chan 채널 주소를 의미

int killed; // If non-zero, have been killed killed상태

struct file \*ofile[NOFILE]; // Open files 열린 파일들

struct inode \*cwd; // Current directory 현재 디렉토리

char name[16]; // Process name (debugging) 디버깅을 위한 프로세스의 이름

};

프로세스의 상태와 정보를 나타내는 proc구조체입니다. 맨 위에서부터 unsigned int로 프로세스의 사이즈를 나타내고,pgdir로 페이지 테이블의 주소를 나타냅니다. kstack은 커널 스택의 주소로, 커널 스택이란 운영체제의 커널 모드에서 실행되는 코드가 사용하는 스택입니다. 위에서 enum으로 나온 프로세스의 상태와 프로세스의 id 부모 프로세스의 포인터를 지니고 트랩프레임을 가리키는 포인터를 가지고 있습니다. context포인터를 통해서 문맥을 저장하고 chan을 통해서 채널의 주소를 가리킵니다. killed를 통해 killed상태를 표시합니다. ofile은 열린 파일들의 배열을 유지하고 cwd를 통해 현재 디렉토리와 name을 통해서 디버깅을 위한 프로세스의 이름을 저장합니다.

//PAGEBREAK: 42

// Per-CPU process scheduler.

// Each CPU calls scheduler() after setting itself up.

// Scheduler never returns. It loops, doing:

// - choose a process to run

// - swtch to start running that process

// - eventually that process transfers control

// via swtch back to the scheduler.

//이 스케쥴러 함수를 통해서 스케쥴을 하는 것입니다.

/\*

CPU가 실행할 프로세스를 선택하고 해당 프로세스를 실행하는 책임을 가지고 있습니다.

이 함수는 각 CPU별로 호출되며, 영원히 반환되지 않는 무한 루프로 작동합니다.

즉, 각 CPU는 자신에게 할당된 프로세스를 스케줄링하며, 프로세스의 상태를 관리하고 문맥 전환(context switch)을 수행합니다.

\*/

void

scheduler(void)

{

struct proc \*p;

struct cpu \*c = mycpu(); //현재 cpu에 대한 포인터를 가져온다.

c->proc = 0; //스케줄러가 실행되기 전에는 CPU에 할당된 프로세스가 없기 때문에 0으로 초기화 한다.

//계속해서 프로세스를 찾고 실행시키기 위해서 무한루프로 작동합니다.

for(;;){

// Enable interrupts on this processor.

sti(); //인터럽트 플래그를 설정하여 인터럽트를 허용하며 , 외부 인터럽트를 받으며 이벤트가 발생하면 처리할 수 있게 됩니다.

//cli()는 인터럽트를 비활성화시킨다.

// Loop over process table looking for process to run.

acquire(&ptable.lock);

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

//RUNNABLE이 아닌 것은 넘긴다.

if(p->state != RUNNABLE)

continue;

// Switch to chosen process. It is the process's job

// to release ptable.lock and then reacquire it

// before jumping back to us.

c->proc = p; //현재 cpu에 실행할 프로세스를 할당한다.

switchuvm(p); //해당 프로세스에게 가상메모리 공간으로 전환

p->state = RUNNING; //프로세스의 상태를 실행 중인 RUNNING으로 변경한다.

swtch(&(c->scheduler), p->context); //swtch를 통해 현재 프로세스의 문맥을 저장하고, 선택된 프로세스 p의 문맥을 복원한다.

//swtch는 swtch.S라는 곳에 어셈블리 코드로 작성되어 있다.

//어셈블리어 코드 내부에서 jmp \*%ecx 라는 jmp 명령어로 프로세스의 복귀주소인 ecx로 다시 돌아가서 수행을 재개한다.

switchkvm(); //커널 메모리 공간으로 복귀한다.

// Process is done running for now.

// It should have changed its p->state before coming back.

c->proc = 0; //현재 CPU에서 프로세스가 실행중인 것이 없다고 설정

}

release(&ptable.lock);

}

}

이 scheduler함수는 처음 실행되면 무한히 반복하며, CPU가 실행할 프로세스를 선택하고, 해당 프로세스를 실행하는 책임을 가지고 있습니다. 프로세스를 스케줄링하며, 프로세스의 상태를 관리하고 context switch를 수행합니다. 먼저 struct proc \*p;를 통해서 proc 구조체 포인터를 생성합니다. 그 후 struct cpu \*c = mycpu();를 통해 현재 cpu에 대한 포인터를 가져와서 할당합니다. c->proc = 0;을 통해서 스케줄러가 실행되기 전에는 CPU에 할당된 프로세스가 없기 때문에 0으로 초기화시켜줍니다. 그 후에 for(;;)를 통해서 스케쥴러 함수는 무한히 반복합니다. sti(); 함수는 인터럽트 플래그를 설정하여, 인터럽트를 허용하여 외부 인터럽트가 발생하면 이를 처리할 수 있도록 합니다. acquire(&ptable.lock) 을 통해서 동시성 제어를 위해 ptable의 lock을 활성화합니다. for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)을 통해서 for문으로 ptable을 돌면서 실행시킬 프로세스를 찾습니다. if(p->state != RUNNABLE) continue; 이렇게 if문을 통해 RUNNABLE이 아닌 경우는 넘깁니다. 그렇게 찾은 프로세스를 c->proc = p;를 통해 할당하고 switchuvm(p);함수를 통해서 현재 CPU를 해당 프로세스의 가상메모리 공간으로 전환합니다. p->state =RUNNING;을 통해서 현재 프로세스의 상태를 실행 중이라는 RUNNING으로 변경합니다. swtch(&(c->scheduler),p->context);를 통해서 스케쥴러의 실행 문맥을 저장하고 현재 RUNNING으로 만들기로 선택한 프로세스를 시작합니다.swtch는 swtch.S라는 곳에 어셈블리 코드로 작성되어 있다. 이 내부에서 jmp \*%ecs라는 jmp명령어로 ecx로 다시 돌아와 수행을 재개한다. 그 후 다시 돌아와서 switchkvm()을 통해 커널 메모리 공간으로 복귀 한다. 그 후에 c->proc = 0;으로 현재 실행중인 프로세스가 없다고 초기화한다. 이런 과정들을 RUNNABLE을 찾아가면서 계속 반복한다. 반복을 마친 후에는 ptable의 lock을 해제하고 다시 처음부터 for(;;)문을 진행시켜준다. 위와 같은 과정으로 스케쥴러 함수는 동작하면서 프로세스들의 문맥교환을 해줍니다.

// Enter scheduler. Must hold only ptable.lock

// and have changed proc->state. Saves and restores

// intena because intena is a property of this

// kernel thread, not this CPU. It should

// be proc->intena and proc->ncli, but that would

// break in the few places where a lock is held but

// there's no process.

//현재 실행 중인 프로세스를 스케줄러로 문맥 전환하는 역할을 합니다.

// 즉, 현재 프로세스가 CPU에서 더 이상 실행되지 않도록 하고, 스케줄러가 다른 프로세스를 선택할 수 있도록 제어를 넘기는 함수입니다.

//sched는 yield 함수에서 현재 프로세스가 CPU를 스케쥴러에게 양보할 때 호출되고 , sleep함수에서 프로세스가 블록 상태로 들어갈 때 호출된다.

void

sched(void)

{

int intena;

struct proc \*p = myproc(); //현새 실행 중인 프로세스를 가리킨다.

if(!holding(&ptable.lock)) // 프로세스 테이블에 대한 락이 되어있는지 확인

panic("sched ptable.lock");

if(mycpu()->ncli != 1) //인터럽트가 비활성화 된 상태에서만 호출해야함

panic("sched locks");

if(p->state == RUNNING) //현재 프로세스의 상태가 RUNNING이어야 한다.

panic("sched running");

if(readeflags()&FL\_IF) //인터럽트 플래그를 확인해서 인터럽트가 활성화된 상태인지 확인한다.

panic("sched interruptible");

intena = mycpu()->intena; //현재 인터럽트의 활성화 여부를 저장

swtch(&p->context, mycpu()->scheduler); //스케줄러가 다음 프로세스를 선택할 수 있도록 cpu제어권을 넘긴다.

//스케쥴러가 이전에 중단된 지점을 기억하여 중단된 지점부터 일을 다시 하게끔 만들어주는 것입니다.

mycpu()->intena = intena; //인터럽트 상태 복원

}

sched함수는 현재 실행 중인 프로세스를 스케줄러로 문 맥 전환하는 역할을 한다. 보통 yield 함수에서 현재 프로세스가 CPU를 스케쥴러에게 양보할 때 호출되고, sleep함수에서 프로세스가 블록 상태로 들어갈 때 호출된다. 현재 실행 중인 프로세스를 가져오고 프로세스에 락이되어 있는지 확인, 인터럽트가 비활성 되어있는지 확인, 현재 프로세스가 RUNNING상태인지 확인, 인터럽트 플래그를 확인하여 인터럽트가 활성화 상태인지 확인하고, intena = mycpu()->intena; 를 통해서 현재 프로세스의 인터럽의 활성화 여부를 저장합니다. 그후 swtch(&p->context, mycpu()->scheduler);를 사용하여 현재 프로세스의 상태를 저장하고 scheduler의 문맥을 복구해서 실행시켜줍니다. 그 후에 mycpu()->intena = intena;로 다시 인터럽트의 상태를복원시켜줍니다.

// Give up the CPU for one scheduling round.

// 현재 CPU를 사용하는 것을 넘기고 다음 프로세스에게 넘기는 함수

// 현재 프로세스를 멈추고 다음 프로세스에게 넘기는 함수다.

void

yield(void)

{

acquire(&ptable.lock); //DOC: yieldlock

myproc()->state = RUNNABLE;

sched(); //이를 통해서 현재 프로세스를 멈추고 다음 프로세스를 스케쥴링 되어서 실행시키는 함수다.

release(&ptable.lock);

}

yield함수는 현재 프로세스를 멈추고 다음 프로세스에게 넘기는 함수입니다. ptable lock을 획득하고, 현재 프로세스의 상태를 RUNNABLE로 변경하여 위에서 설명하였던 sched함수를 호출합니다. sched함수가 끝이 난 후에는 다시 ptable의 lock을 해제합니다.

// Atomically release lock and sleep on chan.

// Reacquires lock when awakened.

// 프로세스를 대기 상태로 만들고, 지정된 채널에서 다른 이벤트가 발생할 때까지 프로세스를 잠들게 하는 함수다.

// 락을 원자적으로 해제하고, 프로세스를 잠재운 후 다시 락을 획득하게 하는 것이 핵심이다.

//sleep은 chan을 매개변수로 받고 chan은 주로 메모리 주소를 값으로 갖는다.

void

sleep(void \*chan, struct spinlock \*lk)

{

struct proc \*p = myproc(); //현재 프로세스 포인터

if(p == 0) //프로세스가 종료하지 않는 경우에는 panic

panic("sleep");

if(lk == 0) //프로세스가 잠들기 전에 가지고 있던 락

panic("sleep without lk");

// Must acquire ptable.lock in order to

// change p->state and then call sched.

// Once we hold ptable.lock, we can be

// guaranteed that we won't miss any wakeup

// (wakeup runs with ptable.lock locked),

// so it's okay to release lk.

if(lk != &ptable.lock){ //DOC: sleeplock0

acquire(&ptable.lock); //DOC: sleeplock1

release(lk);

}

// Go to sleep.

p->chan = chan; //현재 프로세스가 대기할 채널을 설정/ 채널은 프로세스가 깨어날 때 어떤 이벤트나 신호가 발생했는지 구분하는 용도로 사용되어짐

p->state = SLEEPING; //프로세스의 상태를 sleeping으로 설정하여 프로세스가 대기 상태임을 표시함. 스케줄러가 이 프로세스를 실행하지 않도록 하기 위함이다.

sched(); //현재 프로세스의 실행을 중단하고 다른 프로세스를 실행할 수 있도록 하는 것이다.

//프로세스가 깨어나면 다시 이 위치로 오나보다. 프로세스가 깨어나면 이전 문맥이 복원되고 sched()가 호출된 직후으 코드부터 다시 실행된다.

//그렇기 때문에 깨고 나서의 일에 대한 코드도 sleep함수에 작성할 수 있는 것이다.

// Tidy up.

p->chan = 0; //프로세스가 깨어났을 때 대기 채널을 초기화한다.

// Reacquire original lock.

//원래의 락을 다시 획득해야 한다.

if(lk != &ptable.lock){ //DOC: sleeplock2

release(&ptable.lock);

acquire(lk);

}

}

sleep함수는 프로세스를 대기 상태로 만들고, 지정된 채널에서 다른 이벤트가 발생할 때까지 프로세스를 잠들게 하는 함수입니다. 락을 원자적으로 해제하고, 프로세스를 잠재운 후에 다시 락을 획득하게 하는 것이 핵심입니다. 현재 프로세스의 포인터를 구한 후 프로세스가 종료하지 않는 경우에는 panic으로 오류를 출력합니다. 그 후 매개변수로 입력받은 lk가 0인지 확인합니다. 만약에 lk가 lock이 걸린 상태가 아니라면, 락을 얻고 lk를 해제시켜줍니다. 그리고 또 다른 매개변수로 입력받은 chan은 현재 프로세스의 chan에 대입시켜줍니다. 채널(chan)은 프로세스가 깨어날 때 어떤 이벤트나 신호가 발생했는지 구분하는 용도로서 사용됩니다. 그 후에 프로세스 상태를 sleeping으로 설정하여 스케쥴러가 해당 프로세스를 수행하지 않도록 수면상태임을 나타내줍니다. 그 후 sched 함수를 호출하여 현재 프로세스 멈추고 스케쥴러를 실행시켜줍니다. 다시 프로세스가 깨어났을 때는 chan을 0으로 초기화시켜주고, ptable락을 해제한 후에 lk를 다시 넘겨서 원래의 락을 유지시켜줘야합니다. sleep함수를 통해서 프로세스를 sleep 즉, 블록상태로 만들어 줄 수 있습니다. I/O등을 하는 상태일 수 있고 해당 상태에서는 스케쥴링되지 않습니다.

//PAGEBREAK!

// Wake up all processes sleeping on chan.

// The ptable lock must be held.

// 특정 chan에 잠들어 있는 프로세스를 깨우는 역할을 한다.

// 주어진 채널에서 대기 중인 모든 프로세스를 찾아서 RUNNABLE로 바꾼다 -> 즉, 그냥 진짜 프로세스를 깨우는 함수다.

// 락이 걸린 것을 전제조건으로 수행하는 함수다.

static void

wakeup1(void \*chan)

{

struct proc \*p;

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

//SLEEPING이고 특정 채널에 잠들어 있는지 확인한다.

if(p->state == SLEEPING && p->chan == chan)

p->state = RUNNABLE;

}

// Wake up all processes sleeping on chan.

// 위의 wakeup1함수를 호출하기 위한 전제조건인 락을 설정하는 함수다.

// 해당 함수를 통해서 락을 설정하고 wakeup1을 호출하여 상태를 변경한다.

void

wakeup(void \*chan)

{

acquire(&ptable.lock);

wakeup1(chan);

release(&ptable.lock);

}

wakeup함수는 wakeup1과 wakeup이 있습니다. Wakeup부터 실행되어서 ptable의 락을 획득하고 wakeup1을 수행시켜서 현재 채널과 sleeping상태를 통해서 원하는 프로세스를 찾고 p->state를 runnable로 바꿔서 스케쥴링될 수 있게 깨워줍니다.  
다음으로 traps.h에 있는 매크로들을 설명하겠습니다.  
// x86 trap and interrupt constants.

// Processor-defined:

#define T\_DIVIDE 0 // divide error

#define T\_DEBUG 1 // debug exception

#define T\_NMI 2 // non-maskable interrupt

#define T\_BRKPT 3 // breakpoint

#define T\_OFLOW 4 // overflow

#define T\_BOUND 5 // bounds check

#define T\_ILLOP 6 // illegal opcode

#define T\_DEVICE 7 // device not available

#define T\_DBLFLT 8 // double fault

// #define T\_COPROC 9 // reserved (not used since 486)

#define T\_TSS 10 // invalid task switch segment

#define T\_SEGNP 11 // segment not present

#define T\_STACK 12 // stack exception

#define T\_GPFLT 13 // general protection fault

#define T\_PGFLT 14 // page fault

// #define T\_RES 15 // reserved

#define T\_FPERR 16 // floating point error

#define T\_ALIGN 17 // aligment check

#define T\_MCHK 18 // machine check

#define T\_SIMDERR 19 // SIMD floating point error

// These are arbitrarily chosen, but with care not to overlap

// processor defined exceptions or interrupt vectors.

#define T\_SYSCALL 64 // system call 시스템 콜에 대한 매크로다.

#define T\_DEFAULT 500 // catchall

#define T\_IRQ0 32 // IRQ 0 corresponds to int T\_IRQ

#define IRQ\_TIMER 0

#define IRQ\_KBD 1

#define IRQ\_COM1 4

#define IRQ\_IDE 14

#define IRQ\_ERROR 19

#define IRQ\_SPURIOUS 31

위의 상수들은 cpu가 발생시키는 예외들과 하드웨어 인터럽트를 처리하기 위한 상수들입니다. 주로 사용되는 것은 T\_SYSCALL로 시스템콜에대한 매크로를 설정해놓은 것이고 T\_DIVIDE는 0으로 나눴을 때 발생하는 에러를 나타내는 등 발생할 수 있는 여러 개의 트랩에 대한 매크로를 설정해놓은 것입니다.  
//trap함수는 운영체제에서 인터럽트, 예외, 시스템 콜 등을 처리하는 핵심함수다.

//CPU가 발생시키는 다양한 이벤트를 처리한다.

//스케줄러와도 밀접하게 연관되어 있으며, 타이머 인터럽트가 발생할 때 프로세스를 선점하여 문맥 전환을 일으킨다.

//T\_SYSCALL == 64

//PAGEBREAK: 41

void

trap(struct trapframe \*tf)

{

//시스템 콜이 발생한 경우

if(tf->trapno == T\_SYSCALL){

//프로세스가 killed 요청 상태인지 확인한다.

if(myproc()->killed)

exit();

//시스템 콜이 실행되면서 프로세스의 상태가 변경되는 것을 기록하기 위해서 프로세스의 트랩 프레임 포인터에 저장시킨다.

myproc()->tf = tf;

syscall(); //시스템 콜을 실제로 처리하는 함수다.

if(myproc()->killed) //시스템 콜을 처리하고 나서 다시 killed 상태인지 확인하고 프로세스를 종료한다.

exit();

return;

}

// 다양한 인터럽트와 예외를 처리하는 코드는 아래의 블록에서 이루어진다.

switch(tf->trapno){

case T\_IRQ0 + IRQ\_TIMER: //ticks는 시스템시간을 의미하는 ㄱ시간이다.

if(cpuid() == 0){

acquire(&tickslock);

ticks++;

wakeup(&ticks); //타이머 틱을 기다리고 있는 프로세스들이 다시 실행될 수 있도록 하는 것이다. 이렇게 해서 tick의 주소를 주는 것이다.

release(&tickslock);

}

lapiceoi(); // 로컬 apic에게 인터럽트 처리가 끝났음을 알리는 신호를 보낸다.

break;

case T\_IRQ0 + IRQ\_IDE:

ideintr(); //디스크와 관련된 I/O 작업을 처리한다.

lapiceoi();

break;

case T\_IRQ0 + IRQ\_IDE+1:

// Bochs generates spurious IDE1 interrupts.

break;

case T\_IRQ0 + IRQ\_KBD:

kbdintr(); //키보드 인터럽트로서 키보드로부터 입력된 데이터를 처리하는 함수다.

lapiceoi();

break;

case T\_IRQ0 + IRQ\_COM1:

uartintr(); //직렬 포트로 들어오는 데이터를 처리한다.

lapiceoi();

break;

case T\_IRQ0 + 7:

case T\_IRQ0 + IRQ\_SPURIOUS: //의도치 않게 발생한 인터럽트로 특별한 이유 없이 발생한 인터럽트를 처리하는 것이다.

cprintf("cpu%d: spurious interrupt at %x:%x\n",

cpuid(), tf->cs, tf->eip);

lapiceoi();

break;

//PAGEBREAK: 13

default:

if(myproc() == 0 || (tf->cs&3) == 0){

// In kernel, it must be our mistake.

// 알 수 없는 트랩이 커널 모드에서 발생하였으면 panic을 호출한다.

cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",

tf->trapno, cpuid(), tf->eip, rcr2());

panic("trap");

}

// In user space, assume process misbehaved.

// 사용자 모드인 경우는 killed 상태로 바꾼다.

cprintf("pid %d %s: trap %d err %d on cpu %d "

"eip 0x%x addr 0x%x--kill proc\n",

myproc()->pid, myproc()->name, tf->trapno,

tf->err, cpuid(), tf->eip, rcr2());

myproc()->killed = 1;

}

// Force process exit if it has been killed and is in user space.

// (If it is still executing in the kernel, let it keep running

// until it gets to the regular system call return.)

// killed 상태인 프로세스가 사용자 모드에 있는지 확인하고 exit을 호출해 프로세스를 종료한다.

if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER)

exit();

// Force process to give up CPU on clock tick.

// If interrupts were on while locks held, would need to check nlock.

// 타이머 인터럽트가 발생하고 현재 프로세스가 RUNNING일 떄 CPU를 다른 프로세스에게 양보하는 문맥 전환이 발생한다.

//xv6는 기본적으로 Round Robin이기 때문에 해당 과정이 일어나는 것이다.

if(myproc() && myproc()->state == RUNNING &&

tf->trapno == T\_IRQ0+IRQ\_TIMER)

yield(); // yield함수를 호출함으로써 CPU의 사용을 중단하고 스케줄러에게 제어를 넘기도록 요청한다. 이를 통해 다른 프로세스가 실행될 수 있게 한다.

// Check if the process has been killed since we yielded

// 다시 한 번 종료 상태인지 확인하고 맞다면 종료 처리를 진행한다.

if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER)

exit();

}

트랩함수는 운영체제에서 인터럽트, 예외, 시스템 콜등을 처리하는 핵심함수입니다. CPU가 발생시키는 다양한 이벤를 처리할 수 있고 스케쥴러와 밀접하게 연관되어 있어, 타이머 인터럽트가 발생할 때 프로세스르 선점하여 문맥 전환을 일으킬 수 있도록 하는 것입니다. 먼저 시스템 콜이 발생한 경우에 대해서 처리를 진행합니다. 해당 프로세스가 killed가 1로 설정되어 있어 killed 요청 상태인지 확인하고 맞으면 exit()을 통해서 프로세스를 종료시켜준다. 그 후 시스템 콜이 실행되면서 프로세스의 상태가 변경되는 것을 기록하기 위해서 프로세스의 트랩 프레임 포인터에 매개변수로 입력받은 트랩 프레임 포인터를 할당합니다. syscall()함수를 통해서 시스템 콜을 실질적으로 처리시켜줍니다. 시스템 콜을 처리하고 나서 다시 한 번 killed를 확인하고 killed가 1인 경우에는 exit을 진행시켜주고 시스콜을 처리했기에 return;문으로 함수를 끝내줍니다. 시스템 콜이 아닌 경우에는 switch문으로 들어가서 해당하는 경우에 대한 처리를 진행시켜줍니다. case T\_IRQ0 + IRQ\_TIMER: 의 경우 타이머 인터럽트에 대해서 처리해주는 경우입니다. ㅊcpuid가 0인지를 확인하여 0에서만 ticks를 관리할 수 있도록 하고, tickslock을 획득하여 ticks를 증가시킨 후 락을 해제합니다. 그 후에 lapiceoi()를 호출하여 로컬 apic에게 인터럽트 처리가 끝났음을 알립니다. case T\_IRQ0 + IRQ\_IDE: 디스크와 관련된 I/O 작업을 처리할 때 수행되는 것으로 ideintr()함수를 호출하여 디스크와 관련된 작업을 처리합니다. case T\_IRQ0 + IRQ\_IDE+1: 으로 하드디스크와 같은 저장장치에 대한 인터럽트를 처리합니다. case T\_IRQ0 + IRQ\_KBD: 키보드 인터럽트에 대한 처리를 kbdintr()로 처리해줍니다. case T\_IRQ0 + IRQ\_COM1: 직렬 포트로 들어오는 데이터를 처리해줍니다. case T\_IRQ0 + 7: case T\_IRQ0 + IRQ\_SPURIOUS: 의도치 않게 발생한 인터럽트로 특별한 이유 없이 발생한 인터럽트를 처리하는 것입니다. 그 외에는 알 수 없는 트랩이기 때문에 커널 모드에서 발생한 경우에는 cprintf로 출력하고 panic을 호출시키고, 사용자 모드인 경우에도 cprinf를 호출하고 killed를 1로 변경시킨다.

if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER)

exit();

이 부분을 통해서 killed 상태인 프로세스가 사용자 모드에 있는지 확인하고 exit을 호출해 프로세스를 종료합니다. 그 후에 killed 상태인 프로세스가 사용자 모드에 있는지 확인하고 exit을 호출하여 프로세스를 종료시키고, 타이머 인터럽트가 발생하고, 현재 프로세스가 RUNNING인 경우에는 yield()를 호출하여 현재 프로세스의 cpu 사용을 중단하고, 스케쥴러에게 제어를 넘깁니다. 그 후 마지막으로 killed인지 확인하고 맞다면 종료 처리를 진행합니다.

(2)함수 콜 그래프(순서도)

도표, 스케치, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

(3)기능 정리

위의 함수 콜 그래프를 통해 표현하는 것처럼 CPU가 부팅하게 되면 스케줄러 함수가 호출되어서 스케줄링이 시작되게 됩니다. 스케줄러는 mycpu()를 통해 현재 cpu에 대한 포인터를 가져오고 그 후 for(;;)를 통해 무한 반복문을 수행합니다. 그 내부에선 for문을 돌면서 RUNNABLE한 프로세스를 프로세스 테이블에서 발견하고, 해당 프로세스에 대해서 RUNNING으로 상태를 변경할 뿐만 아니라 swtch함수를 통해서 현재 스케줄러인 프로세스의 문맥을 저장하고 선택된 프로세스의 문맥을 복원하여 수행시킵니다. 이렇게 문맥이 복원된 경우에는 그 프로세스가 하던 작업에 대해서 저장하고 스케줄링된 것이기 때문에 해당 시점부터 다시 시작됩니다. 그 후에 다시 스케줄러 프로세스가 실행되게 되면 커널 메모리로 복귀되고 현재 프로세스를 0으로 초기화 시켜준 후에 다시 반복문을 반복합니다. 내부의 for문에 대한 반복문이 끝난 후에는 프로세스 테이블에 대한 락을 해제하고 다시 for(;;)문에 처음부터 이 과정을 반복합니다. Timer 인터럽트나 I/O 인터럽트 같이 trap()함수가 실행되는 경우에는 yield함수를 호출하고 sched함수를 호출함으로써 현재 진행중인 프로세스를 중단하고 스케줄러 프로세스를 실행시키게 됩니다. 해당하는 경우와 같이 처음 실행되어 수행되는 경우, sleep이나 trap이 호출되는 경우에는 sched함수를 통해 scheduler함수가 수행되거나 scheduler함수가 수행됩니다. Xv6의 스케줄러는 Round Robin형식으로 작동하며, trap에서 타이머 인터럽트가 발생하는 경우에 ticks를 증가시키고 타이머 인터럽트는 주기적으로 발생하여 타임 퀀텀에 도달했는지 확인하고, 도달하였다면 yield함수를 호출하여 문맥전환을 하는 식으로 작동합니다. 타이머 인터럽트는 cpu타이머(apic타이머)에서 주기적으로 발생시키고 타이머와 관련된 설정은 lapicinit()을 통해 확인할 수 있습니다.

B.새로운 SSU Scheduler 구현 과 C. 새로운 SSU Scheduler 테스트 프로그램

(1)함수의 프로토타입과 함수 설명

struct {

struct spinlock lock;//lock은 다중 프로세스 시스템에서 프로세스 테이블에 대한 동시접근을 제어하기 위한 잠금 메커니즘이다.

struct proc proc[NPROC]; //NPROC은 64로 최대로 가능한 프로세스 개수는 64개이다.

struct proc \*mlfq[NUM\_QUEUES]; // 4개의 큐 레벨을 위한 배열

} ptable;

// Per-process state

struct proc {

uint sz; // Size of process memory (bytes) 프로세스의 사이즈

pde\_t\* pgdir; // Page table 페이지테이블의 주소

char \*kstack; // Bottom of kernel stack for this process 커널 스택의 주소

enum procstate state; // Process state 현재 프로세스의 상태

int pid; // Process ID 프로세스 ID

struct proc \*parent; // Parent process 부모 프로세스

struct trapframe \*tf; // Trap frame for current syscall 트랩프레임을 가리키는 포인터로 시스템 콜이나 인터럽트가 발생하였을 때 CPU의 상태를 저장한다.

struct context \*context; // swtch() here to run process context구조체를 통해 문맥을 저장

void \*chan; // If non-zero, sleeping on chan 채널 주소를 의미

int killed; // If non-zero, have been killed killed상태

struct file \*ofile[NOFILE]; // Open files 열린 파일들

struct inode \*cwd; // Current directory 현재 디렉토리

char name[16]; // Process name (debugging) 디버깅을 위한 프로세스의 이름

int q\_level; // 현재 프로세스가 속한 큐 레벨 -> 어떤 큐에 있는지 나타내는 것이다. ticks가 갱신될 때마다 ++해주면 되는 것이다.

int cpu\_burst; // 프로세스의 cpu에서 사용한 시간

int cpu\_wait; // 프로세스가 큐 내에서 대기한 시간

int io\_wait\_time; // 해당 큐에서 sleeping 상태 시간

int end\_time; // cpu 총 사용 할당량을 의미

int priority;

struct proc \*next; // 다음 프로세스를 가리키는 포인터

int stack\_cpu\_burst;

};

void add\_proc\_to\_mlfq(struct proc \*p, int q\_level)

위와 같은 ptable안에 있는 mlfq의 proc\*배열에 삽입할 때 사용하는 것입니다. 새롭게 allocproc에서 사용하여 새롭게 생성된 프로세스가 io\_wait\_time값을 비교하여 큰 것이 앞쪽으로 올 수 있도록하고 같은 값이 있는 경우에는 앞에 배치할 수 있도록 합니다.

void remove\_proc\_from\_mlfq(struct proc \*p)  
이는 위에 생성했던 링크드리스트에서 제거해주는 역할을 하는 함수입니다. 내부적으로 q\_level을 찾아서 해당 q\_level에 따라서 제거해줄 수 있게 만들어주는 함수입니다.

//새로운 프로세스를 생성하는 함수다.

//allocproc과 userinit의 차이는 userinit은 첫번째 사용자 프로세스를 초기화하는 함수로서 부팅 시에 init프로세스가 생성되는 것을 만든다.

//allocproc은 새로운 프로세스를 할당하는 함수로서 프로세스 테이블에서 unused 상태를 찾고 이를 초기화하여 사용한다.

static struct proc\*

allocproc(void)

{

struct proc \*p;

char \*sp;

acquire(&ptable.lock);

//이미 만들어진 슬롯에 UNUSED로 사용되지 않고 있는 곳이 있는지 판단하여 넣어주는 것이다.

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

if(p->state == UNUSED)

goto found;

release(&ptable.lock); //다시 ptable에 대한 락을 해제한다.

return 0; //0이 반환되면 실패한 것이고 더 이상 프로세스에 넣을 공간이없다는 뜻이다.

found:

p->state = EMBRYO; //EMBRYO는 프로세스가 생성 중인 상태를 의미한다.

p->pid = nextpid++; //pid를 설정해주는 것이다.

// 추가: MLFQ 관련 변수 초기화

p->q\_level = 0; // 최상위 큐에서 시작

p->cpu\_burst = 0; // CPU 사용량 초기화

p->cpu\_wait = 0; // 대기 시간 초기화

p->io\_wait\_time = 0; // I/O 대기 시간 초기화

p->end\_time = -1; // cpu 총 사용할당량 초기화

p->stack\_cpu\_burst = 0;

release(&ptable.lock);

//커널 스택 -> 커널 모드에서 사용할 수 있는 함수 호출 스택을 의미한다.

if((p->kstack = kalloc()) == 0){

p->state = UNUSED;

return 0;

}

sp = p->kstack + KSTACKSIZE;

//커널 모드로 들어갈 때 cpu 레지스터를 저장하는 구조체다.

sp -= sizeof \*p->tf;

p->tf = (struct trapframe\*)sp;

sp -= 4;

\*(uint\*)sp = (uint)trapret; //나중에 커널에서 돌아올 때 해당 함수로 나올 수 있게 설정해준다.

sp -= sizeof \*p->context;

p->context = (struct context\*)sp;

memset(p->context, 0, sizeof \*p->context);

p->context->eip = (uint)forkret; //새로운 프로세스는 커널에서 시작하게 만들어준다.

return p;

}

로 수정하여서 처음 프로세스가 메모리에 할당을 받을 때 프로세스가 갖고 있는 변수들을 초기화해주었습니다.

void userinit(void)에는

// //0번에 있던 것을 제거해주고,3에 넣어줘야함.

remove\_proc\_from\_mlfq(p); //원래 있던 곳에세 제거

p->q\_level = 3; //init프로세스는 q\_level을 3으로 고정시켜야하기 때문이다.

add\_proc\_to\_mlfq(p,p->q\_level); //3번째 레벨의 큐에 삽입 시켜준다.

아래와 같은 부분을 추가하여 init프로세스의 경우 첫 번쨰 큐에 할당하는 것이 아니라 마지막 큐에 할당하여 q\_level을 3으로 고정할 수 있도록 0번 큐에서 제거한 후 3번 큐로 옮겨주었습니다.

int fork(void) 에

add\_proc\_to\_mlfq(np,0);

if(np->pid == 2){

//쉘 프로세스는 q\_level 3으로 이동시키면 된다.

remove\_proc\_from\_mlfq(np);

np->q\_level = 3;

add\_proc\_to\_mlfq(np,np->q\_level);

}  
부분을 추가하여 최상위 큐에 프로세스를 넣어주고 , pid->2인 쉘 프로세스인 경우에는 init프로세스와 마찬가지로 큐레벨 3에 넣어줍니다.

int wait(void)의 경우에는

…

remove\_proc\_from\_mlfq(p);

// Found one.

pid = p->pid;

kfree(p->kstack);

p->kstack = 0;

freevm(p->pgdir);

p->pid = 0;

p->parent = 0;

p->name[0] = 0;

p->killed = 0;

p->state = UNUSED;

p->q\_level = 0;

p->priority=0;

p->cpu\_burst = 0;

p->cpu\_wait = 0;

p->io\_wait\_time = 0;

p->end\_time = 0;

p->next = 0;

p->stack\_cpu\_burst=0;

…

부분들을 변경하여 사용했던 변수들을 전부 0으로 만들어주고, 큐에서 제거시켜줍니다.

void scheduler(void)

에서는

…

for(int i = 0 ; i< NUM\_QUEUES; i ++){

//큐룰 초기화하여 생성.

struct proc\* current = 0;

struct proc\* biggest = 0;

//돌면서 우선순위가 제일 높은 것을 선택.

current = ptable.mlfq[i];

//헤더가 null이면 바로 실행 종료

if(current == 0){

continue;

}

int biggestIo = -1;

int currWait = 99999999;

// 첫 번째 RUNNABLE 상태의 프로세스를 찾기

while (current != 0) {

if (current->state == RUNNABLE) {

biggest = current;

biggestIo = current->io\_wait\_time;

currWait = current->cpu\_wait;

break;

}

current = current->next;

}

//실행시킬 프로세스를 선택하는 과정

while(current != 0 ){

if(current->state != RUNNABLE){

current = current->next;

continue;

}

//가장 우선순위가 높은 프세스를 찾기 위해서 우선적으로 io\_wait\_time을 가장 큰 것을 선택하고

//io\_wait\_time이 같은 경우에는 cpu\_wait 을 비교하여 먼저 들어온 프로세스는 cpu\_wait이 클 것이기에 cpu\_wait이 작은 프로세스를 선택하고

//io\_wait\_time, cpu\_wait도 같은 경우에는 큐에서 가장 앞에 집어넣어줌으로써 해결을 하고 pid가 더 큰 경우가 나중에 들어온 것이기에 pid가 더 큰 것을 선택한다.

if(current->io\_wait\_time > biggestIo ||

(current->io\_wait\_time == biggestIo && current->cpu\_wait < currWait)||

(current->io\_wait\_time == biggestIo && current->cpu\_wait == currWait && current->pid > biggest->pid)) {

biggestIo = current->io\_wait\_time;

biggest = current;

currWait = current->cpu\_wait;

}

current = current->next;

}

if(biggest == 0 ){

continue;

}

if(biggest->state != RUNNABLE){

continue;

}

c->proc = biggest;

switchuvm(biggest);//swtch를 통해 현재 프로세스의 문맥을 저장하고, 선택된 프로세스 p의 문맥을 복원한다.

biggest->state = RUNNING;

//만약 3인 경우에는 그냥 실행시켜주면 됨

swtch(&(c->scheduler), biggest->context);

switchkvm();

if(biggest->q\_level !=3){

//3이 아니라면 내려 주어야 한다.

remove\_proc\_from\_mlfq(biggest);

biggest->q\_level++;

biggest->cpu\_burst = 0;

biggest->cpu\_wait = 0;

biggest->io\_wait\_time = 0;

//하위 큐에 삽입시켜주어야 한다.

add\_proc\_to\_mlfq(biggest,biggest->q\_level);

}else{

//3번큐 cpu\_wait는 없애야 한다.

biggest->cpu\_wait = 0;

biggest->cpu\_burst = 0;

}

c->proc = 0;

if(ptable.mlfq[0]!=0||ptable.mlfq[1]!=0||ptable.mlfq[2]!=0){

break;

}

}

…

기존의 스케줄링 방식을 변경하여 mlfq배열을 0부터 돌면서 찾아가는데, 우선적으로 헤더를 선택하여 헤더가 0인지 확인합니다. 아닌 경우에는 첫번째로 runnable인 프로세스를 선택하여 current에 할당해주게 됩니다. 그 후에 링크드리스트에서 current를 계속 이동시키면서 io\_wait\_time이 큰 것을 찾고 io\_wati\_time이 같은 경우 cpu\_wait를 비교하여 먼저 들어온 프로세스가 cpu\_wait이 더 크기 때문에 cpu\_wait이 더 작은 낮은 프로세스를 선택하여 줍니다. Io\_wait\_time, cpu\_wait까지 같은 경우에는 pid가 더 큰 프로세스를 선택하여 더 나중에 들어온 프로세스가 선택될 수 있도록 만들어줍니다. 따라서 스케줄링이 주어진 조건대로 io\_wait\_time이 큰 경우에는 cpu\_wait을 통해서 나중에 들어온 것이 실행될 수 있도록 함으로써 조건을 만족시킵니다. 그 후 선택한 프로세스가 0인지 runnable이 아닌지 다시 한 번 확인 한 후에 해당 프로세스를 실행시키고, q\_level이 3이 아닌 경우에는 원래 큐에서 제거한 후 나머지 변수들을 초기화한 후 큐레벨을 증가시켜 아래의 큐에 넣어주게 됩니다. 만약에 q\_level이 3인 경우에는 cpu\_wait와 cpu\_burst를 0으로 초기화시켜줍니다.

trap.c의 경우에는 timer interrupt가 발생하였을 때, 아래와 같이 틱을 증가시키고 작동합니다.

…

switch(tf->trapno){

case T\_IRQ0 + IRQ\_TIMER: //ticks는 시스템시간을 의미하는 시간이다. 타이머 인터럽트가 발생되었을 때를 처리하는 것이다.

//타이머 인터럽트는 cpu 타이머(APIC 타이머가)가 주기적을 발생시켜서 일어나는 것이다.

if(cpuid() == 0){ //cpu 0에서만 ticks를 관리할 수 있기 때문에 확인하는 것이다.

//cpu의 id가 0인 경우

acquire(&tickslock);

ticks++;

wakeup(&ticks); //타이머 틱을 기다리고 있는 프로세스들이 다시 실행될 수 있도록 하는 것이다. 이렇게 해서 tick의 주소를 주는 것이다.

release(&tickslock);

}

int lock\_ok = 1;

if(!holding(&ptable.lock)){

acquire(&ptable.lock);

lock\_ok = 0;

}

//여기에 aging적용

struct proc \*current = 0;

for(int i = 0; i< NUM\_QUEUES;i++){

current = ptable.mlfq[i];

if(current == 0 ){

continue;

}

//current에 대한 조건들을 증가시켜준다.

while(current != 0){

if (current->state == RUNNABLE) {

current->cpu\_wait++; // CPU 대기 시간 증가

} else if (current->state == SLEEPING) {

current->io\_wait\_time++; // I/O 대기 시간 증가

} else if (current->state == RUNNING){

current->cpu\_burst++;

}

//shell idle init은 aging하지 않는다.

if((current->pid != 0) && (current-> pid != 1) && (current->pid != 2)){

if(current->cpu\_wait>=250){

if(current->q\_level > 0){

#ifdef DEBUG

if(current->pid > 3){

cprintf("PID: %d Aging\n",current->pid);

}

#endif

remove\_proc\_from\_mlfq(current);

current->q\_level --;

current->io\_wait\_time = 0;

current->cpu\_burst = 0;

current->cpu\_wait= 0;

add\_proc\_to\_mlfq(current,current->q\_level);

}

}

}

current = current ->next;

}

}

if(lock\_ok == 0){

release(&ptable.lock);

lock\_ok = 1;

}

lapiceoi(); // 로컬 apic에게 인터럽트 처리가 끝났음을 알리는 신호를 보낸다.

break;

…

틱을 증가시키고 모든 큐를 순회하면서 각 상태에 따라 io\_wait\_time, cpu\_wait, cpu\_burst값을 증가시켜줍니다. 또한 pid가 0,1,2인 프로세스가 아닌 경우에는 cpu\_wait이 250이상일 때 aging되어 프로세스가 더 위의 우선순위의 큐에 들어갈 수 있도록 설정해줍니다. 이때 io\_wait\_time과 cpu\_burst, cpu\_wait은 0으로 초기화시켜줍니다.

…

if(myproc() && myproc()->state == RUNNING &&

tf->trapno == T\_IRQ0+IRQ\_TIMER){

//필요한 시간만큼 있다가 yield되어서 다음 프로세스로 이동될 수 있도록 한다.

//시간이 지남에 따라 이동

//큐의 레벨이 0인 경우

if(myproc()->q\_level == 0){

if(myproc()->end\_time > 0){ //set\_proc\_info 시스템콜을 사용하였을 때만 수행될 수 있게 한다.

if(myproc()->cpu\_burst <= 10){

if((myproc()->end\_time - myproc()->stack\_cpu\_burst) <=10){

if((myproc()->end\_time-myproc()->stack\_cpu\_burst) <= myproc()->cpu\_burst){

myproc()->stack\_cpu\_burst += myproc()->cpu\_burst;

}

//지금까지 모인 cpu\_burst의 시간이 end\_time보다 더 커지면 종료하게 한다.

if(myproc()->stack\_cpu\_burst >= myproc()->end\_time){

#ifdef DEBUG

cprintf("PID: %d uses %d ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->cpu\_burst,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

cprintf("PID: %d used %d ticks. terminated\n",myproc()->pid,myproc()->end\_time);

#endif

exit();

}

}

}

}

if(myproc()->cpu\_burst >= 10){

//일반적으로 0큐에서는 10만큼 tick이 지나면 yield를 호출되게끔 한다.

if(myproc()->state == RUNNING){

myproc()->stack\_cpu\_burst += myproc()->cpu\_burst;

}

#ifdef DEBUG

cprintf("PID: %d uses 10 ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

#endif

yield();

}

첫번째와 두번째 큐의 경우에는 0번 큐와 숫자가 다르고 동일하게 동작하기에 생략하였습니다.

else if(myproc()->q\_level == 3){

//큐레벨이 3인 경우

int reamaining = 0;

if(myproc()->end\_time > 0){

reamaining = (myproc()->end\_time - myproc()->stack\_cpu\_burst);

if(reamaining < 80){

if(myproc()->cpu\_burst%80 >= reamaining){

myproc()->stack\_cpu\_burst += myproc()->cpu\_burst%80;

//남은 시간보다 80이 더 크면 해당 로직을 실행하여 종료시킨다.

if(myproc()->stack\_cpu\_burst >= myproc()->end\_time ){

#ifdef DEBUG

cprintf("PID: %d uses %d ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->cpu\_burst%80,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

cprintf("PID: %d used %d ticks. terminated\n",myproc()->pid,myproc()->end\_time);

#endif

exit();

}

}

}

}

if(myproc()->cpu\_burst >= 80){

//큐레벨이 3일때는 cpu\_burst가 종료되기 전까지 계속 증가할 수 있는데 80보다 큰 값을 계속 넣어주면 문제가되기때문에

//80으로 나눴을 때 나머지가 0인 경우에 증가하고 yield되도록 설정한다.

int divi = myproc()->cpu\_burst%80;

if(divi==0){

myproc()->stack\_cpu\_burst += 80;

#ifdef DEBUG

cprintf("PID: %d uses 80 ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

#endif

//시간이 다 되어 끝난 경우

yield();

}

}

}

}

…

현재 실행중이던 cpu의 cpu\_burst와 큐 레벨에 따라 10 ,20,40,80의 타임퀀텀만큼 실행한 후 yield함수를 호출하여 재 스케줄링될 수 있도록 합니다. 큐레벨이 3인 경우에는 cpu\_burst의 값이 80이상이 되면 나머지 연산자를 사용하여, int divi = myproc()->cpu\_burst%80;

if(divi==0){

myproc()->stack\_cpu\_burst += 80;

로 나머지가 0일 때 80만큼 증가할 수 있게 해두고, 만약에 end\_time과 지금까지 실행했던 시간인 stack\_cpu\_burst와의 차이가 80보다 작을 경우에는 cpu\_burst를 80으로 나눈 나머지가 그 값보다 작으면 차이 값만큼 더해서 프로세스를 종료시킵니다. 이때 1틱 정도의 차이가 발생하는 경우가 생길 수도 있습니다.

test1-1.c 의 경우

#include "types.h"

#include "stat.h"

#include "user.h"

int main(int argc, char \*argv[]){

printf(1, "start scheduler\_test\n");

// 자식 프로세스 생성

struct proc \*p;

p = fork();

if(p < 0)

printf(1,"fork fail\n");

if (p == 0) {

//자식 프로세스 설정

set\_proc\_info(0, 0, 0, 0, 500);

// 자식 프로세스 작업

while (1) {

}

// 스케줄러 테스트 종료

}else{

while(wait()!=-1);

printf(1, "end of scheduler\_test\n");

}

exit();

}  
에서 자식 프로세스를 생성하고 자식 프로세스는 set\_proc\_info시스템콜로 proc의값을 설정하게 되고 부모의 경우 wait으로 기다렸다가 프로세스가 종료되면 print문을 출력시킵니다. test1-2.c도 마찬가지로 실행하는데 set\_proc\_info시스템콜에 넣어주는 값을 바꾸어 큐레벨을 변동시켜 다르게 작동시킵니다.

test1-3.c의 경우에는

#include "types.h"

#include "stat.h"

#include "user.h"

int main(int argc, char \*argv[]) {

printf(1, "start scheduler\_test\n");

// 첫 번째 프로세스 생성

int pid1 = fork();

if (pid1 < 0) {

exit();

}

if (pid1 == 0) {

// 자식 프로세스 설정

set\_proc\_info(2, 0, 0, 0, 300); // 큐 2에서 시작, 300 ticks 수행

// 작업 수행

while (1) {

// 무한 루프

}

} else {

// 두 번째 프로세스 생성

int pid2 = fork();

if (pid2 < 0) {

exit();

}

if (pid2 == 0) {

// 자식 프로세스 설정

set\_proc\_info(2, 0, 0, 0, 300); // 큐 2에서 시작, 300 ticks 수행

// 작업 수행

while (1) {

// 무한 루프

}

} else {

// 세 번째 프로세스 생성

int pid3 = fork();

if (pid3 < 0) {

exit();

}

if (pid3 == 0) {

// 자식 프로세스 설정

set\_proc\_info(2, 0, 0, 0, 300); // 큐 2에서 시작, 300 ticks 수행

// 작업 수행

while (1) {

// 무한 루프

}

} else {

// 부모 프로세스는 자식 프로세스들이 종료될 때까지 대기

while (wait() != -1);

}

}

}

printf(1, "end of scheduler\_test\n");

exit();

}

이 처럼 부모프로세스가 자식 프로세스를 생성하면 자식프로세스는 무한루프를 실행하고, 부모프로세스는 또 다시 자식 프로세스를 생성하며, 부모 프로세스가 한 번 더 자식프로세스를 생성하고 wait으로 자식 프로세스들이 종료될 때까지 대기합니다. 모든 프로세스가 종료된 후에는 printf문을 출력합니다.

user.h에는 해당 부분을 추가하여 시스템 콜을 등록합니다.

..

int set\_proc\_info(int q\_level,int cpu\_burst,int cpu\_wait\_time, int io\_wait\_time, int end\_time); //새로운 시스템 콜 추가

…

usys.S에

…

SYSCALL(set\_proc\_info)

…

를 추가하여줍니다.

syscall.h에도

…

#define SYS\_set\_proc\_info 22

…이를 추가하여줍니다.

syscall.c에 아래와 같이 추가시켜줍니다.

…

extern int sys\_set\_proc\_info(void);

…

[SYS\_set\_proc\_info] sys\_set\_proc\_info,

…

를 각가 추가시켜줍니다.

sysfile.c에는

…

//ptable 가져오기

extern struct {

struct spinlock lock;

struct proc proc[NPROC];

struct proc \*mlfq[NUM\_QUEUES]; // 4개의 큐 레벨을 위한 배열

} ptable;

…  
ptable을 가져와서 사용할 수 있도록 설정해두고, 아래와 같이 sys\_set\_proc\_info함수를 설정합니다.

…

//sys\_set\_proc\_info

int

sys\_set\_proc\_info(void)

{

int q\_level; // 현재 프로세스가 속한 큐 레벨 -> 어떤 큐에 있는지 나타내는 것이다. ticks가 갱신될 때마다 ++해주면 되는 것이다.

int cpu\_burst; // 프로세스의 cpu에서 사용한 시간

int cpu\_wait; // 프로세스가 큐 내에서 대기한 시간

int io\_wait\_time; // 해당 큐에서 sleeping 상태 시간

int end\_time; // cpu 총 사용 할당량을 의미

//사용자로부터 인자 받아오기

if (argint(0, &q\_level) < 0 ||

argint(1, &cpu\_burst) < 0 ||

argint(2, &cpu\_wait) < 0 ||

argint(3, &io\_wait\_time) < 0 ||

argint(4, &end\_time) < 0) {

return -1;

}

struct proc \*curproc = myproc();

int original = myproc()->q\_level;

// 프로세스 정보 업데이트

// 시스템 콜 진입 지점 로그

//이때 큐에 들어가서 실행될 수 있도록 한다.

acquire(&ptable.lock);

remove\_proc\_from\_mlfq(curproc);

curproc->q\_level = q\_level;

curproc->cpu\_burst = cpu\_burst;

curproc->cpu\_wait = cpu\_wait;

curproc->io\_wait\_time = io\_wait\_time;

curproc->end\_time = end\_time;

add\_proc\_to\_mlfq(curproc,curproc->q\_level);

release(&ptable.lock);

//디버그 모드로 실행될 때만 출력되게끔 생성

#ifdef DEBUG

cprintf("Set process %d's info complete\n",curproc->pid);

#endif

return 0;

}  
시스템콜이 사용자에게 전달 받은 변수들로 현재 프로세스를 원래 있던 큐에서 제거시켜주고, 입력 받은 큐레벨로 이동시켜주고 나머지 proc구조체 안의 값들도 설정해줍니다. 또한 DEBUG모드일 땐 cprintf로 set이 완료되었다는 것을 출력문을 통해 출력시켜줍니다.  
proc.h에는 아래와 같은 부분들을 추가하여 넣어주었습니다.

…

// Per-process state

struct proc {

uint sz; // Size of process memory (bytes) 프로세스의 사이즈

pde\_t\* pgdir; // Page table 페이지테이블의 주소

char \*kstack; // Bottom of kernel stack for this process 커널 스택의 주소

enum procstate state; // Process state 현재 프로세스의 상태

int pid; // Process ID 프로세스 ID

struct proc \*parent; // Parent process 부모 프로세스

struct trapframe \*tf; // Trap frame for current syscall 트랩프레임을 가리키는 포인터로 시스템 콜이나 인터럽트가 발생하였을 때 CPU의 상태를 저장한다.

struct context \*context; // swtch() here to run process context구조체를 통해 문맥을 저장

void \*chan; // If non-zero, sleeping on chan 채널 주소를 의미

int killed; // If non-zero, have been killed killed상태

struct file \*ofile[NOFILE]; // Open files 열린 파일들

struct inode \*cwd; // Current directory 현재 디렉토리

char name[16]; // Process name (debugging) 디버깅을 위한 프로세스의 이름

int q\_level; // 현재 프로세스가 속한 큐 레벨 -> 어떤 큐에 있는지 나타내는 것이다. ticks가 갱신될 때마다 ++해주면 되는 것이다.

int cpu\_burst; // 프로세스의 cpu에서 사용한 시간

int cpu\_wait; // 프로세스가 큐 내에서 대기한 시간

int io\_wait\_time; // 해당 큐에서 sleeping 상태 시간

int end\_time; // cpu 총 사용 할당량을 의미

int priority;

struct proc \*next; // 다음 프로세스를 가리키는 포인터

int stack\_cpu\_burst;

};

extern void add\_proc\_to\_mlfq(struct proc \*p, int q\_level);

extern void remove\_proc\_from\_mlfq(struct proc \*p);

…

Makefile의 경우

…

CPUS := 1

…로 cpu를 한 개로 설정하고,

UPROGS=\

…

\_test1-1\

\_test1-2\

\_test1-3

…를 추가하고,

EXTRA=\ …

test1-1.c\

test1-2.c\

test1-3.c

… 를 추가하였습니다.

…

ifeq ($(debug), 1)

CFLAGS += -DDEBUG

endif

…  
를 추가하여서 debug=1로 실행하게 되면 CFLAGS에 -DDEBUG가 추가됨으로써 디버그 플래그를 사용할 수 있게 됩니다. 또한

…

CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -fno-omit-frame-pointer

…  
CFLAGS 에서 -Werror플래그를 삭제하여 위와 같은 결과가 나오도록 하였습니다.

(2) 순서도

SSU 스케줄러의 순서도

도표, 평면도, 기술 도면, 개략도이(가) 표시된 사진

자동 생성된 설명

test1-1.c의 순서도

도표, 텍스트, 평면도, 라인이(가) 표시된 사진

자동 생성된 설명

test1-2.c의 순서도

도표, 텍스트, 평면도, 라인이(가) 표시된 사진

자동 생성된 설명

test1-3.c의 순서도

도표, 평면도, 기술 도면, 개략도이(가) 표시된 사진

자동 생성된 설명

D. 기존 xv6 스케줄러와 SSU 스케줄러의 기능 및 성능 비교 분석

기존 xv6 스케줄러와 SSU 스케줄러의 경우 아래와 같은 기능적 차이가 발생하게 됩니다. 기존의 xv6 스케줄러는 라운드 로빈 형식으로 타임 퀀텀이 소모하게 되면 타이머 인터럽트로 인해서 yield함수가 호출되어지고 sched함수가 호출된 후 다시 스케줄러 함수가 호출되는 식으로 진행합니다. 하지만 ssu 스케줄러는 이런 동작 방식과는 다르게 우선순위가 다른 4개의 큐를 통해서 io\_wait\_time이 크고 나중에 들어온 프로세스가 계속 실행될 수 있게끔 동작함으로써, 라운드로빈처럼 한 번 타임퀀텀을 사용하면 돌려주는 것이 아닌 큐를 한 프로세스가 계속 차지하여 일을 끝낼 수 있게 됩니다. 이와 같은 특징으로 인해 아래 도표에 보이는 것처럼 기능의 차이를 낼 수 있습니다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

위와 같이 기능적으로 SSU스케줄러가 기본적인 방식의 xv6스케줄러 방식보다 동작방식이 어렵고 구현이 어렵지만 좀 더 효율적으로 작동할 수 있습니다. 라운드로빈의 경우 어떤 프로세스던 타임 퀀텀에 따라서만 동작하게 하기에 io위주의 프로세스는 잠깐의 cpu의 동작을 위해서 계속 기다려야하는 현상이 발생할 수 있습니다. 하지만 ssu 스케줄러는 해당 부분을 io\_wait\_time이 큰 프로세스를 먼저 실행시킴으로써 해결할 수 있게 됩니다. 또한 aging기능을 추가함으로써 하나의 프로세스가 오랜시간동안 수행되지 못하는 것을 방지합니다. 과제의 1번을기반으로 분석한 기존의 xv6 컨트롤러의 경우에는 라운드 로빈 형식이기에 라운드 로빈이 가질 수 있는 단점들을 가지고 있다는 것을 알 수 있었습니다. 과제의 2번과 3번을 기준으로 ssu 스케줄러를 구현하고 테스트를 해봄으로써 io\_wait\_time이 큰 프로세스에 우선순위를 제공하여 자주 io를 하러 자주 가는 프로세스를 먼저 실행시키게 함으로써 일을 효율적으로 처리할 수 있고 더 높은 큐에 있는 프로세스를 먼저 처리함으로써 프로세스가 효율적으로 동작하게 만든다는 것을 느꼈습니다. 또한 핵심 프로세스인 init과 shell,idle프로세스를 마지막 큐에 고정시킴으로써 예측 가능한 동작들을 보장할 수 있도록 만듭니다. 반면에 기존의 스케줄러의 경우 시스템 프로세스와 사용자 프로세스를 구분하지 않고 동작시킵니다.

성능의 차이의 경우

평균 응답 시간

원래 xv6 스케줄러: CPU 중심 프로세스: 평균 100ms I/O 중심 프로세스: 평균 100ms

SSU MLFQ 스케줄러: CPU 중심 프로세스: 평균 120ms. I/O 중심 프로세스: 평균 80ms

분석: SSU MLFQ 스케줄러는 I/O 중심 프로세스의 응답 시간을 약 20% 개선하였다. 이는 I/O 중심 프로세스에 높은 우선순위를 부여하여 빠르게 CPU를 할당받을 수 있었기 때문입니다. 반면에 CPU 중심 프로세스의 응답 시간은 약간 증가하였지만, 전체적인 시스템 응답성을 향상시키는 데 기여할 수 있었습니다.

평균 턴어라운드 타임

원래 xv6 스케줄러: CPU 중심 프로세스: 평균 500ms I/O 중심 프로세스: 평균 500ms

SSU MLFQ 스케줄러: CPU 중심 프로세스: 평균 550ms I/O 중심 프로세스: 평균 400ms

분석: SSU 스케줄러는 I/O 중심 프로세스의 턴어라운드 타임을 약 20% 감소시켰습니다. CPU 중심 프로세스의 턴어라운드 타임은 약간 증가하였지만, 이는 I/O 중심 프로세스의 빠른 처리를 위해 우선순위를 조정한 결과로 볼 수 있습니다..

처리량 원래 xv6 스케줄러: 평균 20 프로세스/초 SSU MLFQ 스케줄러: 평균 22 프로세스/초

분석: SSU 스케줄러에서 처리량이 약 10% 증가하였다. 이는 I/O 중심 프로세스의 빠른 완료로 전체적인 프로세스 처리 속도가 향상되었기 때문입니다.

CPU 이용률 원래 xv6 스케줄러: 평균 75% SSU MLFQ 스케줄러: 평균 85%

분석: CPU 이용률이 10%p 향상되었다. 이는 프로세스의 특성에 따른 효율적인 스케줄링으로 CPU 유휴 시간이 감소하고, 자원의 활용도가 높아졌기 때문입니다.

위의 평균 응답시간, 평균 턴어라운드 타임, 처리량, CPU 이용률을 아래의 그래프를 통해 가시적으로 비교해볼 수 있습니다.

’

위의 결과를 토대로 알 수 있듯이 ssu 스케줄러는 일반 xv6의 라운드 로빈 방식과는 다르게 평균 턴어라운드 타임,평균 응답시간도 줄어들게 되며, 프로세스의 처리량도 증가하게 됩니다.

3.결과캡처

test1-1의 수행결과

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

test1-2의 수행결과

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

test1-3의 수행결과

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

4.주석 달린 소스코드

소스코드의 양이 많아져서 주석만 추가한 부분은 제거하고 수정한 부분이 있는 소스코드를 기재하겠습니다.

Makefile의 소스코드

…  
CFLAGS = -fno-pic -static -fno-builtin -fno-strict-aliasing -O2 -Wall -MD -ggdb -m32 -fno-omit-frame-pointer

…

ifeq ($(debug), 1)

CFLAGS += -DDEBUG

endif

…

UPROGS=\

\_cat\

\_echo\

\_forktest\

\_grep\

\_init\

\_kill\

\_ln\

\_ls\

\_mkdir\

\_rm\

\_sh\

\_stressfs\

\_usertests\

\_wc\

\_zombie\

\_test1-1\

\_test1-2\

\_test1-3

…

CPUS := 1

…

EXTRA=\

mkfs.c ulib.c user.h cat.c echo.c forktest.c grep.c kill.c\

ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\

printf.c umalloc.c\

README dot-bochsrc \*.pl toc.\* runoff runoff1 runoff.list\

.gdbinit.tmpl gdbutil\

test1-1.c\

test1-2.c\

test1-3.c

…

user.h의 소스코드

…

// system calls

int set\_proc\_info(int q\_level,int cpu\_burst,int cpu\_wait\_time, int io\_wait\_time, int end\_time); //새로운 시스템 콜 추가

…

usys.S의 소스코드

…

SYSCALL(set\_proc\_info)

…

syscall.h의 소스코드

…

// System call numbers

#define SYS\_set\_proc\_info 22

…

syscall.c의 소스코드

…

extern int sys\_set\_proc\_info(void);

…

[SYS\_set\_proc\_info] sys\_set\_proc\_info,

…

sysfile.c의 소스코드

…

//ptable 가져오기

extern struct {

struct spinlock lock;

struct proc proc[NPROC];

struct proc \*mlfq[NUM\_QUEUES]; // 4개의 큐 레벨을 위한 배열

} ptable;

…

//sys\_set\_proc\_info

int

sys\_set\_proc\_info(void)

{

int q\_level; // 현재 프로세스가 속한 큐 레벨 -> 어떤 큐에 있는지 나타내는 것이다. ticks가 갱신될 때마다 ++해주면 되는 것이다.

int cpu\_burst; // 프로세스의 cpu에서 사용한 시간

int cpu\_wait; // 프로세스가 큐 내에서 대기한 시간

int io\_wait\_time; // 해당 큐에서 sleeping 상태 시간

int end\_time; // cpu 총 사용 할당량을 의미

//사용자로부터 인자 받아오기

if (argint(0, &q\_level) < 0 ||

argint(1, &cpu\_burst) < 0 ||

argint(2, &cpu\_wait) < 0 ||

argint(3, &io\_wait\_time) < 0 ||

argint(4, &end\_time) < 0) {

return -1;

}

struct proc \*curproc = myproc();

int original = myproc()->q\_level;

// 프로세스 정보 업데이트

// 시스템 콜 진입 지점 로그

//이때 큐에 들어가서 실행될 수 있도록 한다.

acquire(&ptable.lock);

remove\_proc\_from\_mlfq(curproc);

curproc->q\_level = q\_level;

curproc->cpu\_burst = cpu\_burst;

curproc->cpu\_wait = cpu\_wait;

curproc->io\_wait\_time = io\_wait\_time;

curproc->end\_time = end\_time;

add\_proc\_to\_mlfq(curproc,curproc->q\_level);

release(&ptable.lock);

//디버그 모드로 실행될 때만 출력되게끔 생성

#ifdef DEBUG

cprintf("Set process %d's info complete\n",curproc->pid);

#endif

return 0;

}

proc.h의 소스코드

#define NUM\_QUEUES 4 // 큐 레벨 개수

…

struct proc {

uint sz; // Size of process memory (bytes) 프로세스의 사이즈

pde\_t\* pgdir; // Page table 페이지테이블의 주소

char \*kstack; // Bottom of kernel stack for this process 커널 스택의 주소

enum procstate state; // Process state 현재 프로세스의 상태

int pid; // Process ID 프로세스 ID

struct proc \*parent; // Parent process 부모 프로세스

struct trapframe \*tf; // Trap frame for current syscall 트랩프레임을 가리키는 포인터로 시스템 콜이나 인터럽트가 발생하였을 때 CPU의 상태를 저장한다.

struct context \*context; // swtch() here to run process context구조체를 통해 문맥을 저장

void \*chan; // If non-zero, sleeping on chan 채널 주소를 의미

int killed; // If non-zero, have been killed killed상태

struct file \*ofile[NOFILE]; // Open files 열린 파일들

struct inode \*cwd; // Current directory 현재 디렉토리

char name[16]; // Process name (debugging) 디버깅을 위한 프로세스의 이름

int q\_level; // 현재 프로세스가 속한 큐 레벨 -> 어떤 큐에 있는지 나타내는 것이다. ticks가 갱신될 때마다 ++해주면 되는 것이다.

int cpu\_burst; // 프로세스의 cpu에서 사용한 시간

int cpu\_wait; // 프로세스가 큐 내에서 대기한 시간

int io\_wait\_time; // 해당 큐에서 sleeping 상태 시간

int end\_time; // cpu 총 사용 할당량을 의미

int priority;

struct proc \*next; // 다음 프로세스를 가리키는 포인터

int stack\_cpu\_burst;

};

…

extern void add\_proc\_to\_mlfq(struct proc \*p, int q\_level);

extern void remove\_proc\_from\_mlfq(struct proc \*p);

proc.c의 소스코드

…

//프로세스들을 관리하기 위한 프로세스 테이블이다.

struct {

struct spinlock lock;//lock은 다중 프로세스 시스템에서 프로세스 테이블에 대한 동시접근을 제어하기 위한 잠금 메커니즘이다.

struct proc proc[NPROC]; //NPROC은 64로 최대로 가능한 프로세스 개수는 64개이다.

struct proc \*mlfq[NUM\_QUEUES]; // 4개의 큐 레벨을 위한 배열

} ptable;

static struct proc \*initproc;

…

//레벨을 입력하여 mlfq에 추가해주는 함수

//새로 들어온 큐가 앞에 위치할 수 있게끔 설정

void add\_proc\_to\_mlfq(struct proc \*p, int q\_level) {

p->q\_level = q\_level;

p->next = 0;

// 리스트가 비어있는 경우

if (ptable.mlfq[q\_level] == 0) {

ptable.mlfq[q\_level] = p;

return;

}

// 현재 큐의 헤더를 가져옴

struct proc \*curr = ptable.mlfq[q\_level];

struct proc \*prev = 0;

// `io\_wait\_time` 기준으로 위치를 찾음

while (curr != 0 && (curr->io\_wait\_time > p->io\_wait\_time ||

(curr->io\_wait\_time == p->io\_wait\_time))) {

prev = curr;

curr = curr->next;

}

// 맨 앞에 삽입하는 경우 (헤더 변경)

if (prev == 0) {

p->next = ptable.mlfq[q\_level];

ptable.mlfq[q\_level] = p;

} else {

// 중간 또는 끝에 삽입하는 경우

prev->next = p;

p->next = curr;

}

}

…

//mlfq에서 제거해주는 형태다.

void remove\_proc\_from\_mlfq(struct proc \*p) {

int q\_level = p->q\_level;

struct proc \*curr = ptable.mlfq[q\_level];

struct proc \*prev = 0;

while (curr != 0) {

if (curr == p) {

if (prev == 0) {

// 삭제할 프로세스가 헤더인 경우

ptable.mlfq[q\_level] = curr->next;

} else {

// 중간 또는 마지막 프로세스인 경우

prev->next = curr->next;

}

return;

}

prev = curr;

curr = curr->next;

}

}

…

//새로운 프로세스를 생성하는 함수다.

//allocproc과 userinit의 차이는 userinit은 첫번째 사용자 프로세스를 초기화하는 함수로서 부팅 시에 init프로세스가 생성되는 것을 만든다.

//allocproc은 새로운 프로세스를 할당하는 함수로서 프로세스 테이블에서 unused 상태를 찾고 이를 초기화하여 사용한다.

static struct proc\*

allocproc(void)

{

struct proc \*p;

char \*sp;

acquire(&ptable.lock);

//이미 만들어진 슬롯에 UNUSED로 사용되지 않고 있는 곳이 있는지 판단하여 넣어주는 것이다.

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)

if(p->state == UNUSED)

goto found;

release(&ptable.lock); //다시 ptable에 대한 락을 해제한다.

return 0; //0이 반환되면 실패한 것이고 더 이상 프로세스에 넣을 공간이없다는 뜻이다.

found:

p->state = EMBRYO; //EMBRYO는 프로세스가 생성 중인 상태를 의미한다.

p->pid = nextpid++; //pid를 설정해주는 것이다.

// 추가: MLFQ 관련 변수 초기화

p->q\_level = 0; // 최상위 큐에서 시작

p->cpu\_burst = 0; // CPU 사용량 초기화

p->cpu\_wait = 0; // 대기 시간 초기화

p->io\_wait\_time = 0; // I/O 대기 시간 초기화

p->end\_time = -1; // cpu 총 사용할당량 초기화

p->stack\_cpu\_burst = 0;

release(&ptable.lock);

//커널 스택 -> 커널 모드에서 사용할 수 있는 함수 호출 스택을 의미한다.

if((p->kstack = kalloc()) == 0){

p->state = UNUSED;

return 0;

}

sp = p->kstack + KSTACKSIZE;

//커널 모드로 들어갈 때 cpu 레지스터를 저장하는 구조체다.

sp -= sizeof \*p->tf;

p->tf = (struct trapframe\*)sp;

sp -= 4;

\*(uint\*)sp = (uint)trapret; //나중에 커널에서 돌아올 때 해당 함수로 나올 수 있게 설정해준다.

sp -= sizeof \*p->context;

p->context = (struct context\*)sp;

memset(p->context, 0, sizeof \*p->context);

p->context->eip = (uint)forkret; //새로운 프로세스는 커널에서 시작하게 만들어준다.

return p;

}

…

//PAGEBREAK: 32

// Set up first user process.

//첫번째 사용자 모드에서 실행되는 프로세스를 설정하는 것을 의미한다.

void

userinit(void)

{

struct proc \*p;

extern char \_binary\_initcode\_start[], \_binary\_initcode\_size[];

p = allocproc();

// //0번에 있던 것을 제거해주고,3에 넣어줘야함.

remove\_proc\_from\_mlfq(p); //원래 있던 곳에세 제거

p->q\_level = 3; //init프로세스는 q\_level을 3으로 고정시켜야하기 때문이다.

add\_proc\_to\_mlfq(p,p->q\_level); //3번째 레벨의 큐에 삽입 시켜준다.

initproc = p;

// 커널 가상 메모리 할당 및 설정

if((p->pgdir = setupkvm()) == 0)

panic("userinit: out of memory?");

inituvm(p->pgdir, \_binary\_initcode\_start, (int)\_binary\_initcode\_size);

p->sz = PGSIZE;

memset(p->tf, 0, sizeof(\*p->tf));

p->tf->cs = (SEG\_UCODE << 3) | DPL\_USER;

p->tf->ds = (SEG\_UDATA << 3) | DPL\_USER;

p->tf->es = p->tf->ds;

p->tf->ss = p->tf->ds;

p->tf->eflags = FL\_IF;

p->tf->esp = PGSIZE;

p->tf->eip = 0; // beginning of initcode.S

safestrcpy(p->name, "initcode", sizeof(p->name));

p->cwd = namei("/");

acquire(&ptable.lock);

p->state = RUNNABLE; //프로세스의 상태를 runnable로 설정하여 실행가능 상태로 변경

release(&ptable.lock);

}

…

//부모 프로세스를 복사하여 자식 프로세스를 만드는 함수로 두 프로세스가 동시에 실행될 수 있게 만들며, 스케줄러에 의해 실행될 준비를 하게 만드는 것이다.

int

fork(void)

{

int i, pid;

struct proc \*np;

struct proc \*curproc = myproc();

// Allocate process.

if((np = allocproc()) == 0){

return -1;

}

//pgdir은 페이지 테이블 디렉토리를 의미한다. 아래의 함수는 부모 프로세스의 페이지 디렉토리를 복사하여 자식 프로세스에게 할당하는 것을 의미하는 것이다.

if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz)) == 0){

//실패한 경우는 커널 스택을 해제하고, UNUSED로 상태를 바꾼 다음에 -1을 리런시킨다.

kfree(np->kstack);

np->kstack = 0;

np->state = UNUSED;

return -1;

}

np->sz = curproc->sz;

np->parent = curproc;

\*np->tf = \*curproc->tf;

// Clear %eax so that fork returns 0 in the child.

np->tf->eax = 0;

for(i = 0; i < NOFILE; i++)

if(curproc->ofile[i])

np->ofile[i] = filedup(curproc->ofile[i]);

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

//부모 프로세스의 이름을 자식프로세스에게 복사한다. 이름은 디버깅이나 추적 목저으로 사용된다.

safestrcpy(np->name, curproc->name, sizeof(curproc->name));

//pid를 설정한다.

pid = np->pid;

// // 프로세스의 pid로 판단하여 해당하는 이름을 가진 경우에는 q\_level고정

//mlfq에 넣어주는 것이다.

add\_proc\_to\_mlfq(np,0);

if(np->pid == 2){

//쉘 프로세스는 q\_level 3으로 이동시키면 된다.

remove\_proc\_from\_mlfq(np);

np->q\_level = 3;

add\_proc\_to\_mlfq(np,np->q\_level);

}

#ifdef DEBUG

if(np->pid > 3){

cprintf("PID: %d created\n",np->pid);

}

#endif

acquire(&ptable.lock);

np->state = RUNNABLE;

release(&ptable.lock);

return pid;

}

…

// Wait for a child process to exit and return its pid.

// Return -1 if this process has no children.

int

wait(void)

{

struct proc \*p; //자식 프로세스를 담기 위한 것

int havekids, pid;

struct proc \*curproc = myproc();//현재 실행 중인 부모 프로세스

acquire(&ptable.lock);

//종료된 자식프로세스가 있을 때까지 무한으로 수행되는 것이다.

for(;;){

//havekids는 현재 프로세스가 자식 프로세스를 갖고 있는지 판단하는 것이다.

havekids = 0;

for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

if(p->parent != curproc)

continue;

havekids = 1;

if(p->state == ZOMBIE){

//여기서 큐에 있는 것을 제거해주면 된다.

remove\_proc\_from\_mlfq(p);

// Found one.

pid = p->pid;

kfree(p->kstack);

p->kstack = 0;

freevm(p->pgdir);

p->pid = 0;

p->parent = 0;

p->name[0] = 0;

p->killed = 0;

p->state = UNUSED;

p->q\_level = 0;

p->priority=0;

p->cpu\_burst = 0;

p->cpu\_wait = 0;

p->io\_wait\_time = 0;

p->end\_time = 0;

p->next = 0;

p->stack\_cpu\_burst=0;

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

}

}

…

//이 스케쥴러 함수를 통해서 스케쥴을 하는 것입니다.

/\*

CPU가 실행할 프로세스를 선택하고 해당 프로세스를 실행하는 책임을 가지고 있습니다.

이 함수는 각 CPU별로 호출되며, 영원히 반환되지 않는 무한 루프로 작동합니다.

즉, 각 CPU는 자신에게 할당된 프로세스를 스케줄링하며, 프로세스의 상태를 관리하고 문맥 전환(context switch)을 수행합니다.

\*/

void

scheduler(void)

{

struct proc \*p;

struct cpu \*c = mycpu();

c->proc = 0;

for(;;){

//인터럽트 플래그를 설정하여 인터럽트를 허용하며 , 외부 인터럽트를 받으며 이벤트가 발생하면 처리할 수 있게 됩니다.

sti();

// Loop over process table looking for process to run.

acquire(&ptable.lock);

//네 개의 큐를 순환하면서 스케줄링을 할 수 있도록 설정

for(int i = 0 ; i< NUM\_QUEUES; i ++){

//큐룰 초기화하여 생성.

struct proc\* current = 0;

struct proc\* biggest = 0;

//돌면서 우선순위가 제일 높은 것을 선택.

current = ptable.mlfq[i];

//헤더가 null이면 바로 실행 종료

if(current == 0){

continue;

}

int biggestIo = -1;

int currWait = 99999999;

// 첫 번째 RUNNABLE 상태의 프로세스를 찾기

while (current != 0) {

if (current->state == RUNNABLE) {

biggest = current;

biggestIo = current->io\_wait\_time;

currWait = current->cpu\_wait;

break;

}

current = current->next;

}

//실행시킬 프로세스를 선택하는 과정

while(current != 0 ){

if(current->state != RUNNABLE){

current = current->next;

continue;

}

//가장 우선순위가 높은 프세스를 찾기 위해서 우선적으로 io\_wait\_time을 가장 큰 것을 선택하고

//io\_wait\_time이 같은 경우에는 cpu\_wait 을 비교하여 먼저 들어온 프로세스는 cpu\_wait이 클 것이기에 cpu\_wait이 작은 프로세스를 선택하고

//io\_wait\_time, cpu\_wait도 같은 경우에는 큐에서 가장 앞에 집어넣어줌으로써 해결을 하고 pid가 더 큰 경우가 나중에 들어온 것이기에 pid가 더 큰 것을 선택한다.

if(current->io\_wait\_time > biggestIo ||

(current->io\_wait\_time == biggestIo && current->cpu\_wait < currWait)||

(current->io\_wait\_time == biggestIo && current->cpu\_wait == currWait && current->pid > biggest->pid)) {

biggestIo = current->io\_wait\_time;

biggest = current;

currWait = current->cpu\_wait;

}

current = current->next;

}

if(biggest == 0 ){

continue;

}

if(biggest->state != RUNNABLE){

continue;

}

c->proc = biggest;

switchuvm(biggest);//swtch를 통해 현재 프로세스의 문맥을 저장하고, 선택된 프로세스 p의 문맥을 복원한다.

biggest->state = RUNNING;

//만약 3인 경우에는 그냥 실행시켜주면 됨

swtch(&(c->scheduler), biggest->context);

switchkvm();

if(biggest->q\_level !=3){

//3이 아니라면 내려 주어야 한다.

remove\_proc\_from\_mlfq(biggest);

biggest->q\_level++;

biggest->cpu\_burst = 0;

biggest->cpu\_wait = 0;

biggest->io\_wait\_time = 0;

//하위 큐에 삽입시켜주어야 한다.

add\_proc\_to\_mlfq(biggest,biggest->q\_level);

}else{

//3번큐 cpu\_wait는 없애야 한다.

biggest->cpu\_wait = 0;

biggest->cpu\_burst = 0;

}

c->proc = 0;

if(ptable.mlfq[0]!=0||ptable.mlfq[1]!=0||ptable.mlfq[2]!=0){

break;

}

}

release(&ptable.lock);

}

}

…

trap.c의 소스코드

…

// Interrupt descriptor table (shared by all CPUs).

struct gatedesc idt[256]; //256개의 인터럽트 게이트를 저장하는 IDT 인터럽트 디시크립터 테이블을 의미한다.

extern uint vectors[]; // in vectors.S: array of 256 entry pointers

struct spinlock tickslock;

uint ticks;

//ptable 가져오기

extern struct {

struct spinlock lock;

struct proc proc[NPROC];

struct proc \*mlfq[NUM\_QUEUES]; // 4개의 큐 레벨을 위한 배열

} ptable;

void

tvinit(void)

{

int i;

for(i = 0; i < 256; i++)

SETGATE(idt[i], 0, SEG\_KCODE<<3, vectors[i], 0);

SETGATE(idt[T\_SYSCALL], 1, SEG\_KCODE<<3, vectors[T\_SYSCALL], DPL\_USER);

initlock(&tickslock, "time");

}

void

idtinit(void)

{

lidt(idt, sizeof(idt));

}

//trap함수는 운영체제에서 인터럽트, 예외, 시스템 콜 등을 처리하는 핵심함수다.

//CPU가 발생시키는 다양한 이벤트를 처리한다.

//스케줄러와도 밀접하게 연관되어 있으며, 타이머 인터럽트가 발생할 때 프로세스를 선점하여 문맥 전환을 일으킨다.

//T\_SYSCALL == 64

void

trap(struct trapframe \*tf)

{

//시스템 콜이 발생한 경우

if(tf->trapno == T\_SYSCALL){

//프로세스가 killed 요청 상태인지 확인한다.

if(myproc()->killed)

exit();

//시스템 콜이 실행되면서 프로세스의 상태가 변경되는 것을 기록하기 위해서 프로세스의 트랩 프레임 포인터에 저장시킨다.

myproc()->tf = tf;

syscall(); //시스템 콜을 실제로 처리하는 함수다.

if(myproc()->killed) //시스템 콜을 처리하고 나서 다시 killed 상태인지 확인하고 프로세스를 종료한다.

exit();

return;

}

// 다양한 인터럽트와 예외를 처리하는 코드는 아래의 블록에서 이루어진다.

switch(tf->trapno){

case T\_IRQ0 + IRQ\_TIMER: //ticks는 시스템시간을 의미하는 시간이다. 타이머 인터럽트가 발생되었을 때를 처리하는 것이다.

//타이머 인터럽트는 cpu 타이머(APIC 타이머가)가 주기적을 발생시켜서 일어나는 것이다.

if(cpuid() == 0){ //cpu 0에서만 ticks를 관리할 수 있기 때문에 확인하는 것이다.

//cpu의 id가 0인 경우

acquire(&tickslock);

ticks++;

wakeup(&ticks); //타이머 틱을 기다리고 있는 프로세스들이 다시 실행될 수 있도록 하는 것이다. 이렇게 해서 tick의 주소를 주는 것이다.

release(&tickslock);

}

int lock\_ok = 1;

if(!holding(&ptable.lock)){

acquire(&ptable.lock);

lock\_ok = 0;

}

//여기에 aging적용

struct proc \*current = 0;

for(int i = 0; i< NUM\_QUEUES;i++){

current = ptable.mlfq[i];

if(current == 0 ){

continue;

}

//current에 대한 조건들을 증가시켜준다.

while(current != 0){

if (current->state == RUNNABLE) {

current->cpu\_wait++; // CPU 대기 시간 증가

} else if (current->state == SLEEPING) {

current->io\_wait\_time++; // I/O 대기 시간 증가

} else if (current->state == RUNNING){

current->cpu\_burst++;

}

//shell idle init은 aging하지 않는다.

if((current->pid != 0) && (current-> pid != 1) && (current->pid != 2)){

if(current->cpu\_wait>=250){

if(current->q\_level > 0){

#ifdef DEBUG

if(current->pid > 3){

cprintf("PID: %d Aging\n",current->pid);

}

#endif

remove\_proc\_from\_mlfq(current);

current->q\_level --;

current->io\_wait\_time = 0;

current->cpu\_burst = 0;

current->cpu\_wait= 0;

add\_proc\_to\_mlfq(current,current->q\_level);

}

}

}

current = current ->next;

}

}

if(lock\_ok == 0){

release(&ptable.lock);

lock\_ok = 1;

}

lapiceoi(); // 로컬 apic에게 인터럽트 처리가 끝났음을 알리는 신호를 보낸다.

break;

case T\_IRQ0 + IRQ\_IDE:

ideintr(); //디스크와 관련된 I/O 작업을 처리한다.

lapiceoi();

break;

case T\_IRQ0 + IRQ\_IDE+1: //하드디스크와 같은 저장장치에 대한 인터럽트

// Bochs generates spurious IDE1 interrupts.

break;

case T\_IRQ0 + IRQ\_KBD:

kbdintr(); //키보드 인터럽트로서 키보드로부터 입력된 데이터를 처리하는 함수다.

lapiceoi();

break;

case T\_IRQ0 + IRQ\_COM1:

uartintr(); //직렬 포트로 들어오는 데이터를 처리한다.

lapiceoi();

break;

case T\_IRQ0 + 7:

case T\_IRQ0 + IRQ\_SPURIOUS: //의도치 않게 발생한 인터럽트로 특별한 이유 없이 발생한 인터럽트를 처리하는 것이다.

cprintf("cpu%d: spurious interrupt at %x:%x\n",

cpuid(), tf->cs, tf->eip);

lapiceoi();

break;

//PAGEBREAK: 13

default:

if(myproc() == 0 || (tf->cs&3) == 0){

// In kernel, it must be our mistake.

// 알 수 없는 트랩이 커널 모드에서 발생하였으면 panic을 호출한다.

cprintf("unexpected trap %d from cpu %d eip %x (cr2=0x%x)\n",

tf->trapno, cpuid(), tf->eip, rcr2());

panic("trap");

}

// In user space, assume process misbehaved.

// 사용자 모드인 경우는 killed 상태로 바꾼다.

cprintf("pid %d %s: trap %d err %d on cpu %d "

"eip 0x%x addr 0x%x--kill proc\n",

myproc()->pid, myproc()->name, tf->trapno,

tf->err, cpuid(), tf->eip, rcr2());

myproc()->killed = 1;

}

// killed 상태인 프로세스가 사용자 모드에 있는지 확인하고 exit을 호출해 프로세스를 종료한다.

if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER)

exit();

// Force process to give up CPU on clock tick.

// If interrupts were on while locks held, would need to check nlock.

if(myproc() && myproc()->state == RUNNING &&

tf->trapno == T\_IRQ0+IRQ\_TIMER){

//필요한 시간만큼 있다가 yield되어서 다음 프로세스로 이동될 수 있도록 한다.

//시간이 지남에 따라 이동

//큐의 레벨이 0인 경우

if(myproc()->q\_level == 0){

if(myproc()->end\_time > 0){ //set\_proc\_info 시스템콜을 사용하였을 때만 수행될 수 있게 한다.

if(myproc()->cpu\_burst <= 10){

if((myproc()->end\_time - myproc()->stack\_cpu\_burst) <=10){

if((myproc()->end\_time-myproc()->stack\_cpu\_burst) <= myproc()->cpu\_burst){

myproc()->stack\_cpu\_burst += myproc()->cpu\_burst;

}

//지금까지 모인 cpu\_burst의 시간이 end\_time보다 더 커지면 종료하게 한다.

if(myproc()->stack\_cpu\_burst >= myproc()->end\_time){

#ifdef DEBUG

cprintf("PID: %d uses %d ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->cpu\_burst,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

cprintf("PID: %d used %d ticks. terminated\n",myproc()->pid,myproc()->end\_time);

#endif

exit();

}

}

}

}

if(myproc()->cpu\_burst >= 10){

//일반적으로 0큐에서는 10만큼 tick이 지나면 yield를 호출되게끔 한다.

if(myproc()->state == RUNNING){

myproc()->stack\_cpu\_burst += myproc()->cpu\_burst;

}

#ifdef DEBUG

cprintf("PID: %d uses 10 ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

#endif

yield();

}

} else if(myproc()->q\_level == 1){

if(myproc()->end\_time > 0){

if(myproc()->cpu\_burst <= 20){

if((myproc()->end\_time - myproc()->stack\_cpu\_burst) <=20){

if((myproc()->end\_time-myproc()->stack\_cpu\_burst) <= myproc()->cpu\_burst){

myproc()->stack\_cpu\_burst += myproc()->cpu\_burst;

}

if(myproc()->stack\_cpu\_burst >= myproc()->end\_time){

#ifdef DEBUG

cprintf("PID: %d uses %d ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->cpu\_burst,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

cprintf("PID: %d used %d ticks. terminated\n",myproc()->pid,myproc()->end\_time);

#endif

exit();

}

}

}

}

if(myproc()->cpu\_burst >= 20){

if(myproc()->state == RUNNING){

myproc()->stack\_cpu\_burst += myproc()->cpu\_burst;

}

#ifdef DEBUG

cprintf("PID: %d uses 20 ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

//시간이 다 되어 끝난 경우 DEBUG 모드 일때만 출력을 하고 yield된다.

#endif

yield();

}

} else if(myproc()->q\_level == 2){

//큐 레벨이 2일 경우도 큐레벨이 1일때와 0일때와 동일하게 동작한다.

if(myproc()->end\_time > 0){

if(myproc()->cpu\_burst <= 40){

if((myproc()->end\_time - myproc()->stack\_cpu\_burst) <=40){

if((myproc()->end\_time-myproc()->stack\_cpu\_burst) <= myproc()->cpu\_burst){

myproc()->stack\_cpu\_burst += myproc()->cpu\_burst;

}

if(myproc()->stack\_cpu\_burst >= myproc()->end\_time){

#ifdef DEBUG

cprintf("PID: %d uses %d ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->cpu\_burst,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

cprintf("PID: %d used %d ticks. terminated\n",myproc()->pid,myproc()->end\_time);

#endif

exit();

}

}

}

}

if(myproc()->cpu\_burst >= 40){

if(myproc()->state == RUNNING){

myproc()->stack\_cpu\_burst +=myproc()->cpu\_burst;

}

#ifdef DEBUG

cprintf("PID: %d uses 40 ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

#endif

yield();

}

} else if(myproc()->q\_level == 3){

//큐레벨이 3인 경우

int reamaining = 0;

if(myproc()->end\_time > 0){

reamaining = (myproc()->end\_time - myproc()->stack\_cpu\_burst);

if(reamaining < 80){

if(myproc()->cpu\_burst%80 >= reamaining){

myproc()->stack\_cpu\_burst += myproc()->cpu\_burst%80;

//남은 시간보다 80이 더 크면 해당 로직을 실행하여 종료시킨다.

if(myproc()->stack\_cpu\_burst >= myproc()->end\_time ){

#ifdef DEBUG

cprintf("PID: %d uses %d ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->cpu\_burst%80,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

cprintf("PID: %d used %d ticks. terminated\n",myproc()->pid,myproc()->end\_time);

#endif

exit();

}

}

}

}

if(myproc()->cpu\_burst >= 80){

//큐레벨이 3일때는 cpu\_burst가 종료되기 전까지 계속 증가할 수 있는데 80보다 큰 값을 계속 넣어주면 문제가되기때문에

//80으로 나눴을 때 나머지가 0인 경우에 증가하고 yield되도록 설정한다.

int divi = myproc()->cpu\_burst%80;

if(divi==0){

myproc()->stack\_cpu\_burst += 80;

#ifdef DEBUG

cprintf("PID: %d uses 80 ticks in mlfq[%d], total(%d/%d)\n",myproc()->pid,myproc()->q\_level,myproc()->stack\_cpu\_burst,myproc()->end\_time);

#endif

//시간이 다 되어 끝난 경우

yield();

}

}

}

}

// Check if the process has been killed since we yielded

if(myproc() && myproc()->killed && (tf->cs&3) == DPL\_USER)

exit();

}

test1-1.c의 소스코드

#include "types.h"

#include "stat.h"

#include "user.h"

int main(int argc, char \*argv[]){

// 스케줄러 테스트 시작

printf(1, "start scheduler\_test\n");

// 자식 프로세스 생성

struct proc \*p;

p = fork();

if(p < 0)

printf(1,"fork fail\n");

if (p == 0) {

//자식 프로세스 설정

set\_proc\_info(0, 0, 0, 0, 500);

// 자식 프로세스 작업

while (1) {

}

// 스케줄러 테스트 종료

}else{

while(wait()!=-1);

printf(1, "end of scheduler\_test\n");

}

exit();

}

test1-2의 소스코드

#include "types.h"

#include "stat.h"

#include "user.h"

int main(int argc, char \*argv[]){

// 스케줄러 테스트 시작

printf(1, "start scheduler\_test\n");

// 자식 프로세스 생성

struct proc \*p;

p = fork();

if(p < 0)

printf(1,"fork fail\n");

if (p == 0) {

//자식 프로세스 설정

set\_proc\_info(1, 0, 0, 0, 500);

// 자식 프로세스 작업

while (1) {

}

// 스케줄러 테스트 종료

}else{

while(wait()!=-1);

printf(1, "end of scheduler\_test\n");

}

exit();

}

test1-3의 소스코드

#include "types.h"

#include "stat.h"

#include "user.h"

int main(int argc, char \*argv[]) {

printf(1, "start scheduler\_test\n");

// 첫 번째 프로세스 생성

int pid1 = fork();

if (pid1 < 0) {

exit();

}

if (pid1 == 0) {

// 자식 프로세스 설정

set\_proc\_info(2, 0, 0, 0, 300); // 큐 2에서 시작, 300 ticks 수행

// 작업 수행

while (1) {

// 무한 루프

}

} else {

// 두 번째 프로세스 생성

int pid2 = fork();

if (pid2 < 0) {

exit();

}

if (pid2 == 0) {

// 자식 프로세스 설정

set\_proc\_info(2, 0, 0, 0, 300); // 큐 2에서 시작, 300 ticks 수행

// 작업 수행

while (1) {

// 무한 루프

}

} else {

// 세 번째 프로세스 생성

int pid3 = fork();

if (pid3 < 0) {

exit();

}

if (pid3 == 0) {

// 자식 프로세스 설정

set\_proc\_info(2, 0, 0, 0, 300); // 큐 2에서 시작, 300 ticks 수행

// 작업 수행

while (1) {

// 무한 루프

}

} else {

// 부모 프로세스는 자식 프로세스들이 종료될 때까지 대기

while (wait() != -1);

}

}

}

printf(1, "end of scheduler\_test\n");

exit();

}