컴퓨터학부 20192403 박상철

개요 : XV6을 통해 운영체제의 스케쥴러를 분석하여 스케쥴링의 동작 원리를 이해하고 문맥 교환 및 프로세스 lock, fork를 비롯한 운영체제 관련 지식 및 리눅스 함수들의 원리를 이해하며 기존 round\_robin 방식의 스케쥴러를 ULE 방식으로 직접 변환하여 다양한 스케쥴링 기법을 익히고 스케줄러를 설계하는 법을 학습하기 위한 과제입니다.

상세 구현

1. xv6 스케쥴러 분석

가장 바깥 범위부터 함수를 타고 들어가는 방식으로 분석글을 작성하겠습니다.

main.c의 mpmain에서 스케쥴러 함수가 호출되며 proc.c에 있는 scheduler() 함수가 실행되면

struct *proc* \*p;

struct *cpu* \*c = mycpu();

c->proc = 0;

현재 cpu 정보를 가져옵니다. 이후

for(;;){

// Enable interrupts on this processor.

sti();

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

acquire(&ptable.lock);

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

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;

for문으로 무한반복을 돌면서 ptable에 lock을 걸고 실행 가능한 프로세스를 탐색합니다.

switchuvm(p);

p->state = RUNNING;

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

다음으로 유저모드로 넘어와 문맥교환을 실행합니다.

switchkvm();

// Process is done running for now.

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

c->proc = 0;

}

release(&ptable.lock);

문맥교환 이후 다시 커널 모드로 넘어오고 프로세스 테이블에 걸린 lock을 해제합니다.

여기까지 스케쥴러 함수였고 아래는 trap.c의 time quantum 코드입니다. 1tick마다 yield 함수를 통해 cpu의 권한을 다음 프로세스에게 양보합니다.

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

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

yield();

전체 과정에 대해 복습하여 정리해보면 기존의 xv6의 스케쥴러는 1tick마다 현재 runnable한 프로세스를 ptable에서 가져와 반복하여 실행히는 방식의 Round Robin 방식의 스케쥴러임을 알 수 있었습니다. 문맥 교환영역에 대해서도 살펴보았습니다.

문맥교환 과정은 switch.S에 어셈블리 언어로 구현되어있습니다.

# Context switch

#

# void swtch(struct context \*\*old, struct context \*new);

#

# Save the current registers on the stack, creating

# a struct context, and save its address in \*old.

# Switch stacks to new and pop previously-saved registers.

.globl swtch

swtch:

movl 4(%esp), %eax

movl 8(%esp), %edx

# Save old callee-saved registers

pushl %ebp

pushl %ebx

pushl %esi

pushl %edi

# Switch stacks

movl %esp, (%eax)

movl %edx, %esp

# Load new callee-saved registers

popl %edi

popl %esi

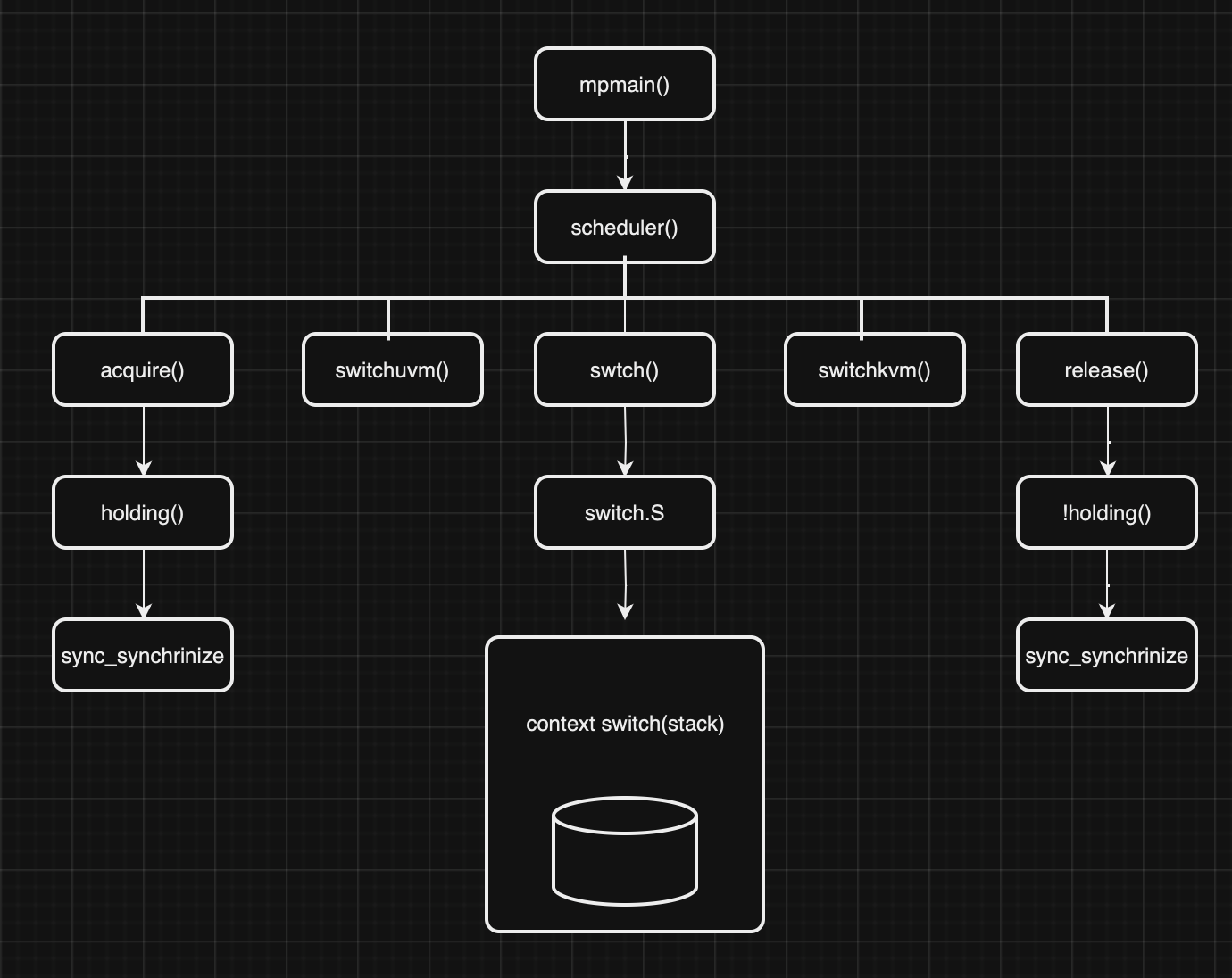
popl %ebx

popl %ebp

ret

어셈블리 언어에 대해서는 자세히 알지 못하였지만 movl을 통해 문맥에 대한 두 포인터 레지스터에 로드하여 push를 통해 스택에 해당 레지스터들을 저장합니다. 이후 movl을 통해 현재 프로세스의 포인터를 esp에 저장하고 새로운 스택으로 전환한 후 pop을 통해 레지스터에서 스택을 복원하는 방식이라고 판단하였습니다.

전체 스케쥴링 흐름도



2. 스케쥴러 구현

핵심 기능은 trap.c와 proc.c에 존재합니다. 먼저 기존 스케쥴러는 1tick마다 time quantum이 발생하기 때문에 time quantum을 30tick으로 수정했습니다. 각 프로세스의 cpu 사용량이 30의 배수가 될때마다 yield 함수를 호출하여 30tick마다 프로세스 전환이 이루어 지도록 코드를 수정하였습니다.

*uint* flowticks = 0;

if(flowticks!=0 && flowticks%60 == 0) {

//cprintf("Rescheduling\n");

reSchedule();

}

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

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

if(myproc()->cpu\_used % 30 == 0 && myproc()->cpu\_used > 0) {

//30ticks print.

cprintf("PID : %d, PRIORITY : %d, PROC\_TICK: %d ticks, total\_cpu\_usage: %d ticks (%d)\n", myproc()->pid, myproc()->priority, myproc()->proc\_tick, myproc()->cpu\_used, 1);

myproc()->proc\_tick = 0;

yield();

}

}

proc.c의 함수들로 넘어오면 먼저 25개의 배열포인터로 링크드리스트를 관리하기 위해 아래 구조체들을 정의하였습니다.

// 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)

int priority;

int proc\_tick;

int cpu\_used;

int priority\_tick;

int start\_tick;

int limit\_tick;

};

struct *p\_node* {

struct *proc* \*node;

struct *p\_node* \*next;

};

struct *listManager* {

struct *p\_node* \*head;

struct *p\_node* \*tail;

};

전역 변수로 runQueue를 선언하여 관리하였으며 각각 가장 우선순위가 높은 Node를 추출하는 getFirstPriorityNode 함수와 tail을 이용해 runQueue의 마지막 노드에 삽입을 해주는 insertNode함수 스케쥴링 이후 맞지 않는 인덱스의 runQueue를 재스케쥴 해주는 reSchedule함수를 정의하였습니다.

struct *listManager* \*runQueue[25];

struct *p\_node* \*getFirstPriorityNode() {

int min\_pri = 100, flag=0;

struct *p\_node* \*ret=0;

for(int i=0; i<25; i++) {

if(runQueue[i] == 0) continue;

struct *p\_node* \*curNode = runQueue[i]->head;

while(curNode != 0) {

if((curNode->node->cpu\_used < curNode->node->limit\_tick) && curNode->node->priority\_tick == 0 && curNode->node->state == RUNNABLE) {

if(min\_pri > curNode->node->priority) {

flag = 1;

min\_pri = curNode->node->priority;

ret = curNode;

}

}

curNode = curNode->next;

}

if(flag) break;

}

return ret;

}

void insertProc(struct *p\_node* \**new\_node*) {

if (*new\_node*->node->priority > 99) *new\_node*->node->priority = 99;

int idx = *new\_node*->node->priority / 4;

// 삽입을 tail에서 수행

*new\_node*->next = 0; // 다음 노드를 없음으로 설정

if (runQueue[idx] == 0) { // 현재 runQueue가 비어있을 때

runQueue[idx] = (struct *listManager* \*)kalloc();

runQueue[idx]->head = *new\_node*;

runQueue[idx]->tail = *new\_node*; // 새로운 노드가 유일한 노드이므로 head와 tail 모두 새로운 노드를 가리킴

return;

}

// tail이 새로운 노드를 가리키게 하고, 새로운 노드의 이전 노드를 tail로 설정

runQueue[idx]->tail->next = *new\_node*;

runQueue[idx]->tail = *new\_node*;

}

void reSchedule() {

for(int i=0; i<25; i++) {

if(runQueue[i] == 0) continue;

struct *p\_node* \*curNode = runQueue[i]->head;

struct *p\_node* \*prevNode = runQueue[i]->head;

while(curNode != 0) {

if(curNode->node->priority\_tick != 0) {

curNode->node->priority += curNode->node->priority\_tick / 10;

if (curNode == runQueue[i]->head)

runQueue[i]->head = curNode->next; // 만약 현재 노드가 head 노드라면 head를 다음 노드로 변경

else

prevNode->next = curNode->next; // 현재 노드가 head가 아니라면 이전 노드의 next를 현재 노드의 next로 변경

if(curNode->next == 0) runQueue[i]->tail = prevNode;

if(runQueue[i]->head == 0) runQueue[i]=0;

curNode->next = 0;

curNode->node->proc\_tick = 0;

curNode->node->priority\_tick = 0;

insertProc(curNode);

}

prevNode = curNode;

curNode = curNode->next;

}

}

}

이후 idle의 priority값 정의 및 프로세스 변수의 초기화는 allocproc 시스템 함수에서 아래처럼 정의했으며

if(p->pid < 3) p->priority = 99;

else p->priority = 0;

p->proc\_tick = 0;

p->cpu\_used = 0;

p->priority\_tick = 0;

p->start\_tick = 0;

p->limit\_tick = 0;

가장 메인인 scheduler() 함수에서 runQueue의 요소 중 priority\_tick이 0이며 수행 될 수 있는 가장 우선순위가 높은 프로세스를 가져와 문맥교환 이전에 출력하고 문맥교환을 하는 과정을 반복합니다.

void

scheduler(void)

{

struct *proc* \*p;

struct *cpu* \*c = mycpu();

c->proc = 0;

int priority\_ticks = 0;

for(;;){

// Enable interrupts on this processor.

sti();

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

acquire(&ptable.lock);

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

if(p->state != RUNNABLE)

continue;

struct *p\_node* \*minProc = getFirstPriorityNode();

if(minProc != 0) {

p = minProc->node;

}

// 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;

switchuvm(p);

p->state = RUNNING;

cprintf("PID : %d, PRIORITY : %d, PROC\_TICK: %d ticks, total\_cpu\_usage: %d ticks (%d)\n",p->pid, p->priority, p->proc\_tick, p->cpu\_used, 2);

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

switchkvm();

// Process is done running for now.

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

c->proc = 0;

}

release(&ptable.lock);

}

}

마지막으로

switch(*tf*->trapno){

case T\_IRQ0 + IRQ\_TIMER:

해당영역에 아래처럼 코드를 작성하여 limit\_tick 이상 프로세스를 점유한 프로세스를 kill하였습니다.

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

if(myproc()->cpu\_used >= myproc()->limit\_tick && myproc()->pid>2) {

cprintf("PID : %d, PRIORITY : %d, PROC\_TICK: %d ticks, total\_cpu\_usage: %d ticks (%d)\n", myproc()->pid, myproc()->priority, myproc()->proc\_tick, myproc()->cpu\_used, 3);

cprintf("PID: %d, terminated \n", myproc()->pid);

//KILL PROCESS

kill(myproc() -> pid);

}

}

3. scheduler\_test.c 및 set\_sche\_info() 시스템 콜 추가

위의 과정을 테스트하기 위해 먼저 set\_sche\_info 시스템 콜을 구현 했으며 두 인자를 받아 초기 우선순위 값과 프로세스 수명을 넣어준 후 insertProc 함수로 RunQueue에 프로세스를 넣어줍니다.

int sys\_set\_sche\_info(void) {

int priority,limit\_tick;

if(argint(0, &priority) < 0)

return -1;

if(argint(1, &limit\_tick) < 0)

return -1;

struct *proc* \*curProc = myproc();

curProc->priority = priority; curProc->limit\_tick = limit\_tick;

struct *p\_node* \*temp = (struct *p\_node* \*)kalloc();

temp->node = curProc; temp->next = 0;

insertProc(temp);

}

이후 scheduler\_test.c는 아래와 같이 PNUM만큼 반복하여 자식 프로세스를 생성하고 pid가 0일때(자식프로세스일 때) set\_shce\_info 함수를 호출하여 프로세스를 정보를 넣어줍니다. 이후 부모 프로세스는 wait 함수를 통해 자식이 좀비가 되는 것을 막아줍니다.

4. 디버깅 기능 구현

디버깅 기능은 명세에 있는것과 같이 Makefile에 아래처럼 넣어주었으며 아래 1,2,3을 각각 time quantum, 문맥교환 직전, 프로세스 킬을 수행하는 곳에 넣어 디버깅을 하였습니다.

ifeq ($(debug), 1)

CFLAGS += -DDEBUG

endif

cprintf("PID : %d, PRIORITY : %d, PROC\_TICK: %d ticks, total\_cpu\_usage: %d ticks (%d)\n", myproc()->pid, myproc()->priority, myproc()->proc\_tick, myproc()->cpu\_used, 1);

cprintf("PID : %d, PRIORITY : %d, PROC\_TICK: %d ticks, total\_cpu\_usage: %d ticks (%d)\n",p->pid, p->priority, p->proc\_tick, p->cpu\_used, 2);

cprintf("PID : %d, PRIORITY : %d, PROC\_TICK: %d ticks, total\_cpu\_usage: %d ticks (%d)\n", myproc()->pid, myproc()->priority, myproc()->proc\_tick, myproc()->cpu\_used, 3);

결과

텍스트, 스크린샷, 패턴, 패브릭이(가) 표시된 사진

자동 생성된 설명

(첨부된 결과에 Rescheduling은 재스케쥴링이 실행되는 시점을 출력하여 결과의 값이 옳은지를 보기 편하게 하기 위해 같이 출력하였습니다. 제출된 소스 코드에는 주석 처리하여 제출하겠습니다.)

소스코드

상세 설계 과정에서 코드와 글을 같이 작성하는 것이 구현과정을 가장 잘 설명 할 수 있는 방법이라 생각이 되어 작성한 모든 코드를 상세 설계에 글과 같이 첨부하였습니다.