HW7. Dynamic Process Allocator 구현하기

학번: 20170404
이름: 한종수
제출일: 2021. 5. 18

1.과제 개요

1.1.본 과제에서는 Slab Allocator 를 이용하여 동적으로 프로세스를 할당하고 해제하는 코드를 완성한다.

2.소스코드 (수정한 코드를 캡처하고 간단히 설명함)

2.1.Traverse 관련

2.1.1.아래 함수들의 기존에 for를 이용하여 프로세스 배열을 순차 탐색 하는 부분을 바꾸었다. //todo 바로 아래의 for문을 수정하였다. //todo를 보면 for에서 p가 dummy의 다음인 ptable.proc->next를 받는다. 해당 주소가 0인지 아닌지 검사하고, dummy인지 검사 후에 다음 proc으로 넘어가며 순회하도록 했다. 한 바퀴를 돌았음은 p가 dummy를 가리킴을 확인하여 알 수 있다. 나머지 동작은 기존 함수와 동일하다. 아래의 Traverse 관련 함수들에 모두 적용되었다.

2.1.2.exit()

```
void exit(void)
{
    struct proc *curproc = myproc();
    struct proc *p;
    int fd;

    if (curproc == initproc)
        panic("init exiting");

    // Close all open files.
    for (fd = 0; fd < NOFILE; fd++)
    {
        if (curproc->ofile[fd])
        {
            fileclose(curproc->ofile[fd]);
            curproc->ofile[fd] = 0;
        }
    }

    begin_op();
    iput(curproc->cwd);
    end_op();
    curproc->cwd = 0;

    acquire(&ptable.lock);

    /*************** todo ***************/

    // Parent might be sleeping in wait().
    wakeup1(curproc->parent);

    // Pass abandoned children to init.
    for (p = ptable.proc->next; (p != ptable.proc) && (p != 0); p = p->next)
    {
        if (p->parent == curproc)
        {
            p->parent = initproc;
            if (p->state == ZOMBIE)
                wakeup1(initproc);
        }
    }

    // Jump into the scheduler, never to return.
    curproc->state = ZOMBIE;
    sched();
    panic("zombie exit");
}
```

2.1.3.scheduler()

```
void scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for (;;)
    {
        // Enable interrupts on this processor.
        sti();

        if (ptable.proc->next != 0)
        {
            // Loop over process table looking for process to run.
            acquire(&ptable.lock);
            /*************** todo ***************/
            for (p = ptable.proc->next; (p != ptable.proc) && (p != 0); p =
            p->next)
            {
                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;
                switchuvm(p);
                p->state = RUNNING;

                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);
        }
    }
}
```

2.1.4.wakeup1()

```
//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
    struct proc *p;

    /*************** todo ***************/
    for (p = ptable.proc->next; (p != ptable.proc) && (p != 0); p = p->next)
        if (p->state == SLEEPING && p->chan == chan)
            p->state = RUNNABLE;
}
```

2.1.5.kill()

```
// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
int kill(int pid)
{
    struct proc *p;

    acquire(&ptable.lock);
    for (p = ptable.proc->next; (p != ptable.proc) && (p != 0); p = p->next)
    {
        if (p->pid == pid)
        {
            p->killed = 1;
            // Wake process from sleep if necessary.
            if (p->state == SLEEPING)
                p->state = RUNNABLE;
            release(&ptable.lock);
            return 0;
        }
    }
    release(&ptable.lock);
    return -1;
}
```

2.1.6.procdump()

```
//PAGEBREAK: 36
// Print a process listing to console.  For debugging.
// Runs when user types ^P on console.
// No lock to avoid wedging a stuck machine further.
void procdump(void)
{
    static char *states[] = {
        [UNUSED] "unused",
        [EMBRYO] "embryo",
        [SLEEPING] "sleep ",
        [RUNNABLE] "runble",
        [RUNNING] "run   ",
        [ZOMBIE] "zombie"};
    int i;
    struct proc *p;
    char *state;
    uint pc[10];

    for (p = ptable.proc->next; (p != ptable.proc) && (p != 0); p = p->next)
    {
        if (p->state == UNUSED)
            continue;
        if (p->state >= 0 && p->state < NELEM(states) && states[p->state])
            state = states[p->state];
        else
            state = "???";
        cprintf("%d %s %s %p", p->pid, state, p->name);
        //cprintf("%d %s %s", p->pid, state, p->name);
        if (p->state == SLEEPING)
        {
            getcallerpcs((uint *)p->context->ebp + 2, pc);
            for (i = 0; i < 10 && pc[i] != 0; i++)
                cprintf(" %p", pc[i]);
        }
        cprintf("\n");
    }
}
```

2.1.7.ps()

```
void ps(void)
{
    static char *states[] = {
        [UNUSED] "unused",
        [EMBRYO] "embryo",
        [SLEEPING] "sleep ",
        [RUNNABLE] "runble",
        [RUNNING] "run   ",
        [ZOMBIE] "zombie"};
    struct proc *p;
    char *state, *name;

    acquire(&ptable.lock);

    /***************** todo ****************/
    for (p = ptable.proc->next; (p != ptable.proc) && (p != 0); p = p->next)
    {
        if (p->state >= 0 && p->state < NELEM(states) && states[p->state])
            state = states[p->state];
        else
            state = "???";

        if (p->state == UNUSED)
            name = "unknown";
        else
            name = p->name;

        cprintf("%d %s %s %p\n", p->pid, state, name, p);
    }

    release(&ptable.lock);

    // print slab cache
    slabdump();

    return;
}
```

2.2.Allocate 관련

2.2.1.allocproc()

```
//PAGEBREAK: 32
// Look in the process table for an UNUSED proc.
// If found, change state to EMBRYO and initialize
// state required to run in the kernel.
// Otherwise return 0.
static struct proc *
allocproc(void)
{
    struct proc *p;
    char *sp;

    acquire(&ptable.lock);

    /*************** todo ***************/
    p = (struct proc *)kmalloc(sizeof(struct proc));
    if (p == 0)
    {
        release(&ptable.lock);
        return 0;
    }
    // allocated
    memset(p, 0, sizeof(struct proc));
    p->state = EMBRYO;
    p->pid = nextpid++;
    p->prev = 0;
    p->next = 0;

    release(&ptable.lock);

    // Allocate kernel stack.
    if ((p->kstack = kalloc()) == 0)
    {
        p->state = UNUSED;
        return 0;
    }
    sp = p->kstack + KSTACKSIZE;

    // Leave room for trap frame.
    sp -= sizeof *p->tf;
    p->tf = (struct trapframe *)sp;

    // Set up new context to start executing at forkret,
    // which returns to trapret.
    sp -= 4;
    *(uint *)sp = (uint)trapret;

    sp -= sizeof *p->context;
    p->context = (struct context *)sp;
    memset(p->context, 0, sizeof *p->context);
    p->context->eip = (uint)forkret;

    return p;
}
```

2.2.1.1.//todo 아래를 보면 구현내용이 있다. 배열로 되어있을 때는 할당을 배열에서 빈공간을 찾아서 했고 중간 중간 쓰지 않는 배열공간으로 인해 공간낭비가 있었다. 지금은 kmalloc에서 struct proc에 알맞은 공간을 찾아서 동적으로 사용하는 만큼만 할당해준다. 프로세스 할당시 kmalloc으로 원하는 크기를 인자로 넘겨주면 해당하는 주소를 준다. 배열과 달리 따로 초기화 되어있지 않기때문에 받아서 메모리 공간을 0으로 초기화 시켜 준다. 이후 구조체의 값을 설정해준다.

2.2.2.fork()

```
// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int fork(void)
{
    int i, pid;
    struct proc *np;
    struct proc *curproc = myproc();

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

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

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

    pid = np->pid;

    acquire(&ptable.lock);

    /*************** todo ***************/
    np->next = ptable.proc->next;
    np->prev = ptable.proc;
    ptable.proc->next->prev = np;
    ptable.proc->next = np;
    np->state = RUNNABLE;

    release(&ptable.lock);

    return pid;
}
```

2.2.2.1.//todo부분을 보면 allocproc에서 새 프로세스 구조체에 대한공간을 받았고 내부에 커널스택을 넣어주는 등 초기화 작업을 마쳤다. 나머지 작업은 이 프로세스를 현재 실행중인 프로세스들의 더블링크드리스트인 ptable에 연결시켜주어야한다. 위치는 dummy의 바로 뒤에 위치하도록한다. 먼저 새로운 프로세스의 next를 dummy의 next로 연결시키고, prev를 dummy로 연결시킨다. 그리고 dummy->next의 프로세스의 prev를 새로운 프로세스로 설정해주고 dummy의 next를 새 프로세스로 연결시킨다. 이후 상태를 실행가능한상태로 변경 시켜주면 새 프로세스가 생성되어 프로세스 목록에 잘 들어가게 되고 실행 가능상태에서 cpu 할당을 기다린다.

2.2.3.userinit()

```
//PAGEBREAK: 32
// Set up first user process.
void userinit(void)
{
    struct proc *p;
    extern char _binary_initcode_start[], _binary_initcode_size[];

    p = allocproc();

    initproc = p;
    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("/");

    // this assignment to p->state lets other cores
    // run this process. the acquire forces the above
    // writes to be visible, and the lock is also needed
    // because the assignment might not be atomic.
    acquire(&ptable.lock);

    /***************** todo ****************/
    ptable.proc->next = ptable.proc->prev = p;
    p->next = p->prev = ptable.proc;
    p->state = RUNNABLE;

    release(&ptable.lock);
}
```

2.2.3.1.첫 user의 프로세스를 만드는 곳이다. init은 dummy의 바로 뒤에 위치시킨다. 그리고 직후에는 프로세스가 두개 밖에 존재하지 않기 때문에 dummy 의 next, prev를 init과 연결시키고, init의 next, prev를 dummy와 연결시킨다. 이 원형 이중 연결리스트로 프로세스 리스트를 구성할 수 있다. 그리고 프로세스를 실행 가능한 상태로 두어 cpu할당을 받을 수 있게 한다.

2.3.DeAllocate & Traverse 관련

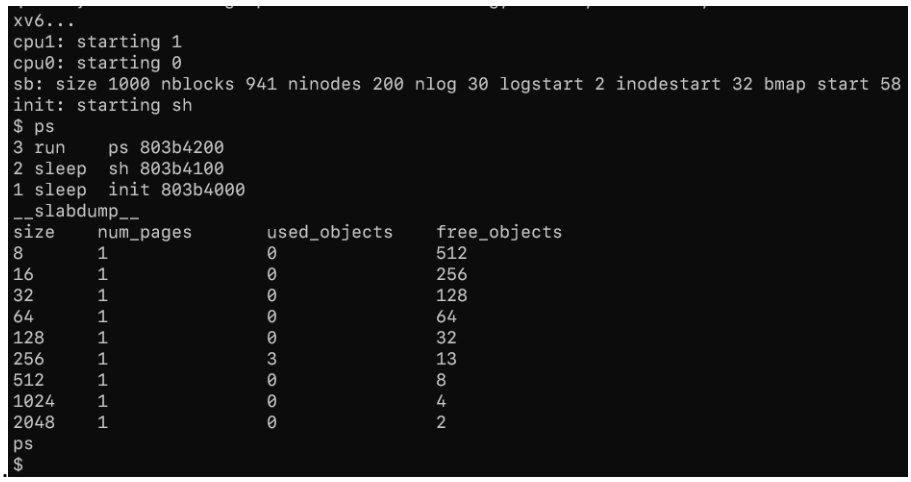
2.3.1.wait()

```
// 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;
    int havekids, pid;
    struct proc *curproc = myproc();

    acquire(&ptable.lock);
    for (;;)
    {
        // Scan through table looking for exited children.
        havekids = 0;
        /*************** todo ***************/

        for (p = ptable.proc->next; (p != ptable.proc) && (p != 0); p = p->next)
        {
            if (p->parent != curproc)
                continue;
            havekids = 1;

            if (p->state == ZOMBIE)
            {
                // Found one.
                pid = p->pid;
                p->next->prev = p->prev;
                p->prev->next = p->next;
                p->prev = 0;
                p->next = 0;

                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;

                kmfree((char *)p, sizeof(struct proc));

                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
    }
}
```

2.3.1.1.//todo를 보면 일단 wait은 Traverse 수정과 내부 구현 수정이 모두 필요하다. Traverse 수정은 앞서 말한 방식으로 수정된다. wait함수는 부모가 자식 프로세스의 자원을 회수 하는 것으로 이 함수에서 자식프로세스에게 할당된 자원을 할당 해제 해야한다. 새로 구현된 할당 해제는 프로세스 리스트에서 이 프로세스를 제거하는 것과 slaballocator에게 주소를 넘겨주어 해당 주소가 다시 사용가능 함을 알려준다. 프로세스 리스트에서 이 프로세스를 제거하려면 이전 프로세스의 다음을 현재 프로세스의 다음으로 연결시켜주고, 다음 프로세스의 이전을 현재프로세스의 이전과 연결 시켜주고 현재 프로세스의 next, prev를 0으로 초기화 해주면된다. 그리고 프로세스 구조체의 할당해제는 kmfree를 호출해 주면 된다.

3.결과 (테스트 실행결과를 캡처)

3.1.


xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ ps
3 run   ps 803b4200
2 sleep sh 803b4100
1 sleep init 803b4000
__slabdump__
size   num_pages   used_objects   free_objects
8   1   0   512
16   1   0   256
32   1   0   128
64   1   0   64
128   1   0   32
256   1   3   13
512   1   0   8
1024   1   0   4
2048   1   0   2
ps
$