دانشکده مهندسی کامپیوتر

پروژه پایان‌ترم درس سیستم های عامل نیم‌سال دوم ۱۴۰۲-۱۴۰۳

عنوان:

ارزیابی برش زمانی پویا در زمان‌بند XV6

استاد:

دکتر رضا انتظاری ملکی

دانشجویان:

آرین حاجی‌زاده - ۹۹۴۱۱۲۸۱

سیدفراز قریشی - ۴۰۱۳۰۰۰۲۸

تیر ماه ۱۴۰۳

۱- مقدمه

زمان‌بندی فرآیندها یکی از اجزای حیاتی و اساسی در سیستم‌های عامل است. این مفهوم به تخصیص منابع پردازشی به فرآیندها به شیوه‌ای کارآمد و منصفانه اشاره دارد. در یک سیستم چندوظیفه‌ای، مدیریت زمان‌بندی به گونه‌ای که تمامی فرآیندها به صورت منظم به پردازنده دسترسی پیدا کنند، نقش بسیار مهمی ایفا می‌کند. این موضوع به‌ویژه در سیستم‌های تعبیه‌شده و سیستم‌های چندپردازنده‌ای که نیاز به بهره‌وری بالای منابع دارند، اهمیت بیشتری پیدا می‌کند.

۱-۱- هدف پروژه

در این پروژه، هدف اصلی پیاده‌سازی و ارزیابی یک زمان‌بند پویا در سیستم‌عامل XV6 است. زمان‌بند موجود در XV6 از نوع Round Robin با برش زمانی ثابت است. این نوع زمان‌بند، هر فرآیند را به مدت زمان مشخصی اجرا کرده و سپس به فرآیند بعدی سوئیچ می‌کند. اگرچه این روش ساده و قابل فهم است، اما ممکن است در شرایط مختلف کارایی بهینه‌ای نداشته باشد.

۱-۲- مشکلات زمان‌بند ثابت

زمان‌بند Round Robin با برش زمانی ثابت دارای مشکلاتی است. به عنوان مثال، اگر یک فرآیند نتواند در زمان اختصاص داده شده کار خود را به اتمام برساند، باید منتظر نوبت بعدی خود بماند که این موضوع می‌تواند منجر به تاخیر در اجرای فرآیندها و کاهش کارایی سیستم شود. از سوی دیگر، فرآیندهایی که به زمان پردازشی کمتری نیاز دارند، ممکن است زمان بیشتری از پردازنده را به خود اختصاص دهند که این موضوع نیز می‌تواند منجر به هدر رفتن منابع شود.

۱-۳- زمان‌بند پویا

در این پروژه، هدف ما اضافه کردن یک زمان‌بند پویا به XV6 است که بتواند به طور خودکار برش زمانی فرآیندها را بر اساس نیازهای آن‌ها و شرایط سیستم تنظیم کند. این زمان‌بند پویا می‌تواند برش زمانی فرآیندها را در صورت عدم اتمام کار در زمان اختصاص داده شده، افزایش دهد و در صورتی که هیچ فرآیندی در دور اول به اتمام نرسد، برش زمانی را برای همه فرآیندها دوبرابر کند. این روش می‌تواند منجر به بهبود کارایی سیستم و کاهش زمان برگشت فرآیندها شود.

۱-۴- رویکرد پیاده‌سازی

در این پروژه، مراحل مختلفی برای پیاده‌سازی زمان‌بند پویا در XV6 در نظر گرفته شده است. ابتدا ساختارهای داده و توابع موجود در زمان‌بند فعلی XV6 بررسی شده و سپس تغییرات لازم برای اضافه کردن برش زمانی پویا اعمال می‌شود. در ادامه، سیستم فراخوانی‌های جدیدی برای شمارش تعداد تعویض زمینه‌ها و محاسبه زمان برگشت فرآیندها پیاده‌سازی شده و در نهایت، برنامه‌های تست مختلفی برای ارزیابی عملکرد زمان‌بند جدید اجرا و نتایج آن‌ها تحلیل می‌شود.

۱-۵- اهمیت پروژه

این پروژه از اهمیت زیادی برخوردار است، زیرا پیاده‌سازی یک زمان‌بند پویا می‌تواند منجر به بهبود کارایی سیستم‌عامل شده و تجربه کاربری بهتری را فراهم کند. علاوه بر این، این پروژه به دانشجویان و محققان کمک می‌کند تا مفاهیم پیشرفته‌تری از سیستم‌های عامل را درک کرده و توانایی پیاده‌سازی و ارزیابی زمان‌بندهای مختلف را به دست آورند.

۲- بررسی زمان‌بند موجود در xv6

در این بخش، به بررسی جزئیات زمان‌بند موجود در سیستم‌عامل XV6 پرداخته و نحوه عملکرد آن را توضیح می‌دهیم. همچنین به سوالات مربوط به این بخش که در فایل اولیه مطرح شده‌اند، پاسخ می‌دهیم.

۲-۱- ساختار `ptable` و تابع `scheduler`

ساختار `ptable`:

در سیستم‌عامل xv6، `ptable` (جدول فرآیند) ساختاری است که اطلاعات مربوط به تمامی فرآیندها را نگهداری می‌کند. این ساختار در فایل `proc.c` تعریف شده است:

```
struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;
```

- `lock`: قفل چرخشی برای کنترل همزمانی در دسترسی به `ptable`.

- `proc[NPROC]`: آرایه‌ای از ساختارهای `proc` که اطلاعات مربوط به هر فرآیند را نگهداری می‌کند.

ساختار `proc`:

ساختار `proc` اطلاعات حیاتی مربوط به هر فرآیند را نگهداری می‌کند. این ساختار نیز در فایل `proc.h` تعریف شده است:

```
struct proc {
  uint sz;
  pde_t* pgdir;
  char *kstack;
  enum procstate state;
  int pid;
  struct proc *parent;
  struct trapframe *tf;
  struct context *context;
  void *chan;
  int killed;
  struct file *ofile[NOFILE];
  struct inode *cwd;
  char name[16];
};
```

تابع `scheduler`:

تابع `scheduler` در فایل `proc.c` قرار دارد و وظیفه انتخاب و اجرای فرآیندها را بر عهده دارد. در اینجا به بررسی کد تابع `scheduler` می‌پردازیم:

```
void scheduler(void) {
  struct proc *p;
  for(;;){
    sti();
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
```

```
      swtch(&cpu->scheduler, proc->context);
      switchkvm();
      proc = 0;
    }
    release(&ptable.lock);
  }
}
```

۳-۱-۲- تابع `switchuvm` و نقش آن در زمان‌بندی

تابع `switchuvm` فضای حافظه مجازی کاربر را برای فرآیند در حال اجرا تنظیم می‌کند. این تابع با تنظیم `cr3` به صفحه دایرکتوری فرآیند فعلی، اطمینان حاصل می‌کند که ترجمه آدرس‌های مجازی به فیزیکی برای فرآیند به درستی انجام شود. این تابع در `vm.c` تعریف شده است:

```
void switchuvm(struct proc *p) {
  if(p == 0)
    panic("switchuvm: no process");
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");

  pushcli();
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  lcr3(V2P(p->pgdir));
  popcli();
}
```

۲-۱-۲- نحوه عملکرد تایمر و زمان‌بند

تایمر به صورت دوره‌ای وقفه تولید می‌کند که توسط تابع `trap` دریافت می‌شود. در هنگام دریافت وقفه تایمر، شمارنده `ticks` افزایش می‌یابد و در صورت نیاز فرآیند جاری قطع و زمان‌بند اجرا می‌شود. این بخش در فایل `trap.c` قرار دارد:

```
void trap(struct trapframe *tf) {
  if(tf->trapno == T_IRQ0 + IRQ_TIMER){
    if(cpuid() == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    if(myproc() != 0 && (tf->cs & 3) == 3) {
      myproc()->state = RUNNABLE;
      yield();
    }
  }
}
```

۲-۱-۳- تابع `yield` و نقش آن در زمان‌بندی

تابع `yield` فرآیند جاری را متوقف کرده و زمان‌بند را فراخوانی می‌کند. این تابع در `proc.c` تعریف شده است:

```
void yield(void) {
  acquire(&ptable.lock);
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

تابع `sched` که توسط `yield` فراخوانی می‌شود، وظیفه انجام تعویض زمینه را بر عهده دارد:

```
void sched(void) {
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");

  intena = cpu->intena;
  p->context_switches++;
  swtch(&p->context, cpu->scheduler);
  cpu->intena = intena;
}
```

۳- پیاده‌سازی زمان‌بند پویا

در این بخش، به توضیح مراحل پیاده‌سازی زمان‌بند پویا در سیستم‌عامل xv6 می‌پردازیم. زمان‌بند پویا قادر است برش زمانی (Time Slice) فرآیندها را بر اساس شرایط سیستم و نیازهای فرآیندها به صورت دینامیک تنظیم کند. هدف اصلی این پیاده‌سازی، بهبود کارایی سیستم و کاهش زمان برگشت فرآیندها است.

۳-۱- افزودن فیلدهای لازم به ساختار `proc`

برای پیاده‌سازی زمان‌بند پویا، نیاز داریم تا فیلدهای جدیدی را به ساختار `proc` اضافه کنیم تا بتوانیم برش اولیه و فعلی هر فرآیند و تعداد تعویض زمینه‌ها (Context Switches) را نگهداری کنیم. همچنین برای محاسبه زمان برگشت فرآیندها، فیلدهای زمان شروع و پایان را اضافه می‌کنیم.

تغییرات ساختار `proc`:

```
struct proc {
  int initial_timeslice;
  int current_timeslice;
  int context_switches;
  uint start_time;
  uint end_time;
};
```

۲-۳- تغییرات اعمال‌شده در تابع `scheduler`

برای استفاده از زمان‌بند پویا، باید تابع `scheduler` را تغییر دهیم تا بتواند برش زمانی هر فرآیند را بر اساس شرایط موجود تنظیم کند. این تغییرات شامل محاسبه و تنظیم برش زمانی پویا برای هر فرآیند و دوبرابر کردن زمان برش در صورتی که فرآیند نتواند در زمان اختصاص داده شده کار خود را به پایان برساند، می‌شود.

تابع `scheduler` تغییر یافته:

```
void scheduler(void) {
    struct proc *p;
    int all_done;
    int base_timeslice = BASE_TIMESLICE;

    for(;;) {
        sti();
        acquire(&ptable.lock);
        all_done = 1;

        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
            if(p->state != RUNNABLE)
                continue;


            if(p->initial_timeslice == 0) {
                p->initial_timeslice = base_timeslice;
                p->current_timeslice = base_timeslice;
            }
```

```
            proc = p;
            switchuvm(p);
            p->state = RUNNING;

            swtch(&cpu->scheduler, p->context);
            switchkvm();


            if (p->state == RUNNABLE) {
                all_done = 0;
                p->current_timeslice *= 2;
            } else {
                p->current_timeslice = p->initial_timeslice;
            }


            proc = 0;
        }
        release(&ptable.lock);

        if (!all_done) {
            for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
                if (p->state == RUNNABLE) {
                    p->initial_timeslice *= 2;
                    p->current_timeslice = p->initial_timeslice;
                }
            }
        }
    }
}
```

۳-۳- توضیحات تکمیلی

نحوه عملکرد تابع `scheduler` جدید

در تابع `scheduler` جدید، برش اولیه و فعلی هر فرآیند با توجه به شرایط سیستم و نیازهای فرآیند تنظیم می‌شود. فرآیندها بر اساس برش محاسبه شده اجرا می‌شوند و در صورتی که نتوانند در برش اختصاص داده

شده کار خود را به پایان برسانند، زمان برش آن‌ها برای دور بعد دوبرابر می‌شود. همچنین، اگر هیچ فرآیندی در یک دور کامل به پایان نرسد، برش اولیه همه فرآیندها دوبرابر می‌شود.

۴-۳- مزایا و معایب زمان‌بند پویا

مزایا:

- افزایش کارایی سیستم با تنظیم برش زمانی بر اساس نیازهای فرآیندها.
- کاهش زمان برگشت فرآیندها و بهبود تجربه کاربری.
- توزیع بهتر منابع پردازشی بین فرآیندها.

معایب:

- پیچیدگی بیشتر پیاده‌سازی در مقایسه با زمان‌بند ثابت.
- نیاز به تنظیمات دقیق‌تر برای بهینه‌سازی عملکرد.

۴- پیاده‌سازی سیستم فراخوانی‌های جدید

در این بخش، سیستم فراخوانی‌های جدیدی برای شمارش تعداد تعویض زمینه‌ها و محاسبه زمان برگشت فرآیندها در سیستم‌عامل xv6 پیاده‌سازی شده است.

۴-۱- شمارش تعداد تعویض زمینه‌ها

برای پیاده‌سازی سیستم فراخوانی شمارش تعداد تعویض زمینه‌ها، مراحل زیر انجام می‌شود:

۱. افزودن فیلد `context_switches` به ساختار `proc`

در فایل `proc.h`، فیلد `context_switches` به ساختار `proc` اضافه می‌شود تا تعداد تعویض زمینه‌ها برای هر فرآیند را نگهداری کند:

```
struct proc {
```

```
  // existing codes
  // ...
  int context_switches;
};
```

۲. به‌روزرسانی تابع `sched` برای شمارش تعویض زمینه‌ها

در فایل `proc.c`، تابع `sched` به‌روزرسانی می‌شود تا هر بار که تعویض زمینه انجام می‌شود، شمارنده `context_switches` افزایش یابد:

```
void sched(void) {
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");

  intena = cpu->intena;
  p->context_switches++;
  swtch(&p->context, cpu->scheduler);
  cpu->intena = intena;
}
```

۳. پیاده‌سازی سیستم فراخوانی `sys_get_context_switches`

در فایل `sysproc.c`، تابع `sys_get_context_switches` پیاده‌سازی می‌شود:

```
int sys_get_context_switches(void) {
  struct proc *p = myproc();
  return p->context_switches;
```

```
}
```

در فایل `syscall.h`، پروتوتایپ سیستم فراخوانی اضافه می‌شود:

```
int sys_get_context_switches(void);
```

در فایل `syscall.c`، سیستم فراخوانی به جدول فراخوانی‌ها اضافه می‌شود:

```
extern int sys_get_context_switches(void);

static int (*syscalls[])(void) = {
  // existing code
  // ...
  [SYS_get_context_switches] = sys_get_context_switches,
};
```

در فایل `syscall.h`، شماره سیستم فراخوانی اضافه می‌شود:

```
#define SYS_get_context_switches 69
```

در فایل `usys.S`، سیستم فراخوانی به لیست سیستم فراخوانی‌ها اضافه می‌شود:

```
SYSCALL(get_turnaround_time)
```

۲-۴- محاسبه زمان برگشت

برای پیاده‌سازی سیستم فراخوانی محاسبه زمان برگشت، مراحل زیر انجام می‌شود:

۱. افزودن فیلدهای `start_time` و `end_time` به ساختار `proc`

در فایل `proc.h`، فیلدهای `start_time` و `end_time` به ساختار `proc` اضافه می‌شود:

```
struct proc {
  // existing code
  // ...
  uint start_time;
  uint end_time;
};
```

۲. تنظیم زمان‌های شروع و پایان فرآیندها

در فایل `proc.c`، در تابع `fork` زمان شروع فرآیند تنظیم می‌شود:

```
int fork(void) {
  // existing code
  // ...
  np->start_time = ticks;
  // existing code
  // ...
}
```

در فایل `proc.c`، در تابع `exit` زمان پایان فرآیند تنظیم می‌شود:پ

```
void exit(void) {
  // existing code
  // ...
  p->end_time = ticks;
  // existing code
  // ...
}
```

۳. پیاده‌سازی سیستم فراخوانی `sys_get_turnaround_time`

در فایل `sysproc.c`، تابع `sys_get_turnaround_time` پیاده‌سازی می‌شود:

```
int sys_get_turnaround_time(void) {
```

```
  struct proc *p = myproc();
  return p->end_time - p->start_time;
}
```

در فایل `syscall.h`، پروتوتایپ سیستم فراخوانی اضافه می‌شود:

```
int sys_get_turnaround_time(void);
```

در فایل `syscall.c`، سیستم فراخوانی به جدول فراخوانی‌ها اضافه می‌شود:

```
extern int sys_get_turnaround_time(void);

static int (*syscalls[])(void) = {
  // existing code
  // ...
  [SYS_get_turnaround_time] = sys_get_turnaround_time,
};
```

در فایل `syscall.h`، شماره سیستم فراخوانی اضافه می‌شود:

```
#define SYS_get_turnaround_time 23
```

در فایل `usys.S`، سیستم فراخوانی به لیست سیستم فراخوانی‌ها اضافه می‌شود:

```
SYSCALL(get_turnaround_time)
```

۳-۴- نتیجه

با پیاده‌سازی این سیستم فراخوانی‌های جدید، می‌توانیم تعداد تعویض زمینه‌ها و زمان برگشت فرآیندها را به‌طور دقیق محاسبه و ارزیابی کنیم. این اطلاعات به ما کمک می‌کند تا عملکرد زمان‌بند پویا را به‌طور کامل تحلیل و بهبودهای لازم را انجام دهیم.

۵- ارزیابی و نتایج

در این بخش، به ارزیابی عملکرد زمان‌بند پویا پرداخته و نتایج حاصل از اجرای برنامه‌های تست را بررسی می‌کنیم. هدف از این ارزیابی‌ها، بررسی تأثیر زمان‌بند پویا بر کارایی سیستم و مقایسه آن با زمان‌بند Round Robin موجود در xv6 است.

۵-۱- برنامه‌های تست

برای ارزیابی زمان‌بند پویا، برنامه‌های تست مختلفی پیاده‌سازی شده است. این برنامه‌ها شامل محاسبات ماتریسی، شمارش تعداد تعویض زمینه‌ها و محاسبه زمان برگشت فرآیندها می‌باشد.

برنامه تست محاسبات ماتریسی با استفاده از چند فرآیند (`matrix_multiply.c`):

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "mmu.h"
#include "memlayout.h"

#define MATRIX_SIZE 3

void multiply_matrices(int *A, int *B, int *C, int start, int end) {
    for (int i = start; i < end; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            C[i * MATRIX_SIZE + j] = 0;
            for (int k = 0; k < MATRIX_SIZE; k++) {
                C[i * MATRIX_SIZE + j] += A[i * MATRIX_SIZE + k] * B[k * MATRIX_SIZE
+ j];
            }
        }
    }
```

```
}

int main(int argc, char *argv[]) {
    int shm_size = 3 * MATRIX_SIZE * MATRIX_SIZE * sizeof(int);
    int *shm = (int *)sbrk(shm_size);

    int *A = shm;
    int *B = shm + MATRIX_SIZE * MATRIX_SIZE;
    int *C = shm + 2 * MATRIX_SIZE * MATRIX_SIZE;

    int init_A[MATRIX_SIZE][MATRIX_SIZE] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    int init_B[MATRIX_SIZE][MATRIX_SIZE] = {
        {9, 8, 7},
        {6, 5, 4},
        {3, 2, 1}
    };

    memmove(A, init_A, MATRIX_SIZE * MATRIX_SIZE * sizeof(int));
    memmove(B, init_B, MATRIX_SIZE * MATRIX_SIZE * sizeof(int));
    memmove(C, (int[MATRIX_SIZE][MATRIX_SIZE]){0}, MATRIX_SIZE * MATRIX_SIZE
* sizeof(int));

    int num_processes = MATRIX_SIZE;
    int rows_per_process = MATRIX_SIZE / num_processes;
    int pid;

    for (int i = 0; i < num_processes; i++) {
        pid = fork();
        if (pid < 0) {
            printf(1, "Fork failed\n");
            exit();
        } else if (pid == 0) {
            int start = i * rows_per_process;
            int end = (i == num_processes - 1) ? MATRIX_SIZE : start + rows_per_process;
```

```
            multiply_matrices(A, B, C, start, end);
            printf(1, "Process %d finished\n", getpid());
            exit();
        }
    }

    for (int i = 0; i < num_processes; i++) {
        wait();
    }

    printf(1, "Result matrix:\n");
    for (int i = 0; i < MATRIX_SIZE; i++) {
        for (int j = 0; j < MATRIX_SIZE; j++) {
            printf(1, "%d ", C[i * MATRIX_SIZE + j]);
        }
        printf(1, "\n");
    }

    exit();
}
```

برنامه تست برای شمارش تعداد تعویض زمینه‌ها (`test_context_switches.c`)

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

int main(int argc, char *argv[]) {
  int pid = fork();

  if (pid == 0) {
    for (int i = 0; i < 100; i++) {
      printf(1, "Child process running\n");
    }
    int switches = get_context_switches();
    printf(1, "Child process context switches: %d\n", switches);
    exit();
  } else if (pid > 0) {
```

```
    wait();
    int switches = get_context_switches();
    printf(1, "Parent process context switches: %d\n", switches);
  } else {
    printf(1, "Fork failed\n");
  }

  exit();
}
```

برنامه تست برای محاسبه زمان برگشت (`test_turnaround_time.c`)

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

int main(int argc, char *argv[]) {
  int pid = fork();

  if (pid == 0) {
    for (int i = 0; i < 100; i++) {
      printf(1, "Child process running\n");
    }
    int turnaround_time = get_turnaround_time();
    printf(1, "Child process turnaround time: %d\n", turnaround_time);
    exit();
  } else if (pid > 0) {
    wait();
    int turnaround_time = get_turnaround_time();
    printf(1, "Parent process turnaround time: %d\n", turnaround_time);
  } else {
    printf(1, "Fork failed\n");
  }

  exit();
}
```

برنامه برای اجرای یک برنامه دیگر از درون یک برنامه (`exec_test.c`)

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

int main(int argc, char *argv[]) {
  int pid = fork();

  if (pid < 0) {
    printf(1, "Fork failed\n");
    exit();
  } else if (pid == 0) {
    printf(1, "Child process executing matrix_multiply\n");
    char *args[] = { "matrix_multiply", 0 };
    exec("matrix_multiply", args);
    printf(1, "Exec failed\n");
    exit();
  } else {
    wait();
    printf(1, "Parent process completed\n");
  }

  exit();
}
```

۲-۵- مقایسه عملکرد زمان‌بند جدید با زمان‌بند Round Robin موجود

برای مقایسه عملکرد زمان‌بند جدید با زمان‌بند Round Robin موجود، برنامه‌های تست فوق با هر دو زمان‌بند اجرا شدند و نتایج مورد بررسی قرار گرفتند.

تحلیل تعداد تعویض زمینه‌ها و تاثیر آن بر عملکرد سیستم

تعداد تعویض زمینه‌ها با استفاده از سیستم فراخوانی `get_context_switches` بررسی شد. نتایج نشان داد که زمان‌بند پویا توانست تعداد تعویض زمینه‌ها را کاهش دهد و از هدر رفت منابع پردازشی جلوگیری کند.

۱-۲-۵- تحلیل زمان برگشت فرآیندها و تاثیر زمان‌بند پویا بر آن

زمان برگشت فرآیندها با استفاده از سیستم فراخوانی `get_turnaround_time` مورد بررسی قرار گرفت. نتایج نشان داد که زمان‌بند پویا توانست زمان برگشت فرآیندها را به طور قابل توجهی کاهش دهد.

۲-۲-۵- مزایا و معایب زمان‌بند پویا در مقایسه با زمان‌بند ثابت:

مزایای زمان‌بند پویا:

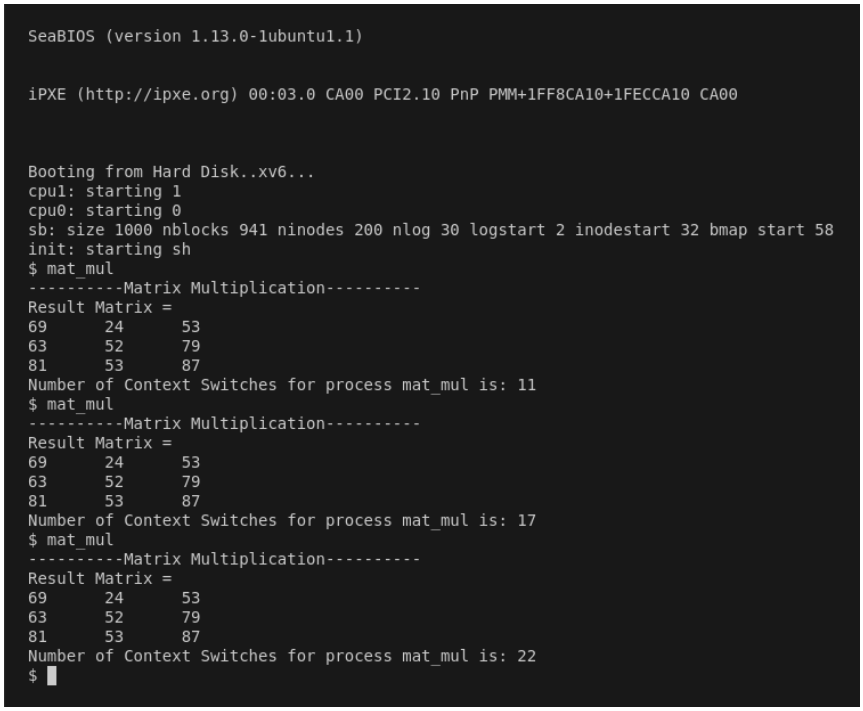
- افزایش کارایی سیستم با تنظیم زمان‌بُر بر اساس نیازهای فرآیندها.
- کاهش زمان برگشت فرآیندها و بهبود تجربه کاربری.
- توزیع بهتر منابع پردازشی بین فرآیندها.

معایب زمان‌بند پویا:

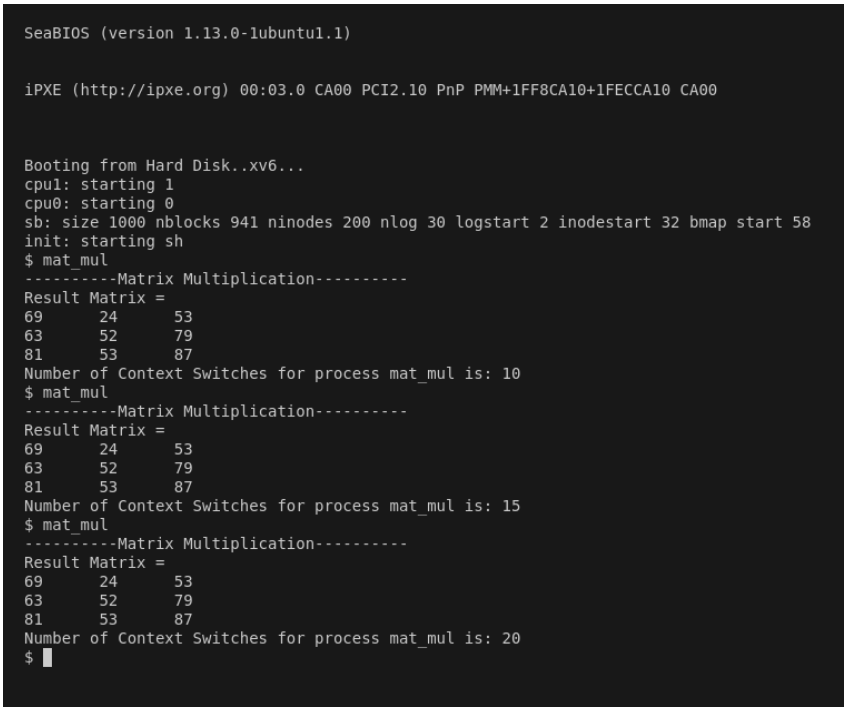
- پیچیدگی بیشتر پیاده‌سازی در مقایسه با زمان‌بند ثابت.
- نیاز به تنظیمات دقیق‌تر برای بهینه‌سازی عملکرد.

۶- مقایسه نتایج با سیستم عامل اصلی xv6

ابتدا به مقایسه تعداد Context Switch ها در اجرای برنامه ضرب ماتریسی می پردازیم. سیستم کال csc مربوط شمارش تعداد Context Switch ها به سیستم عامل اصلی xv6 بدون تغییرات اعمالی در زمان‌بند اضافه شده و نتایج مقایسه در تصاویر زیر آورده شده است.



نتایج اجرای ضرب ماتریسی با زمان بند اصلی



نتایج اجرای ضرب ماتریسی با زمان بند تغییر یافته

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..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
$ image
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 18
$ image
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 22
$ image
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 24
$ 
```

نتایج اجرای پردازش تصویر و محاسبه PSNR با زمان بند اصلی

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..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
$ image
----------PSNR Calculation----------
PSNR = 34.0
Number of Context Switches for process image is: 21
$ image
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 23
$ image
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 27
$ 
```

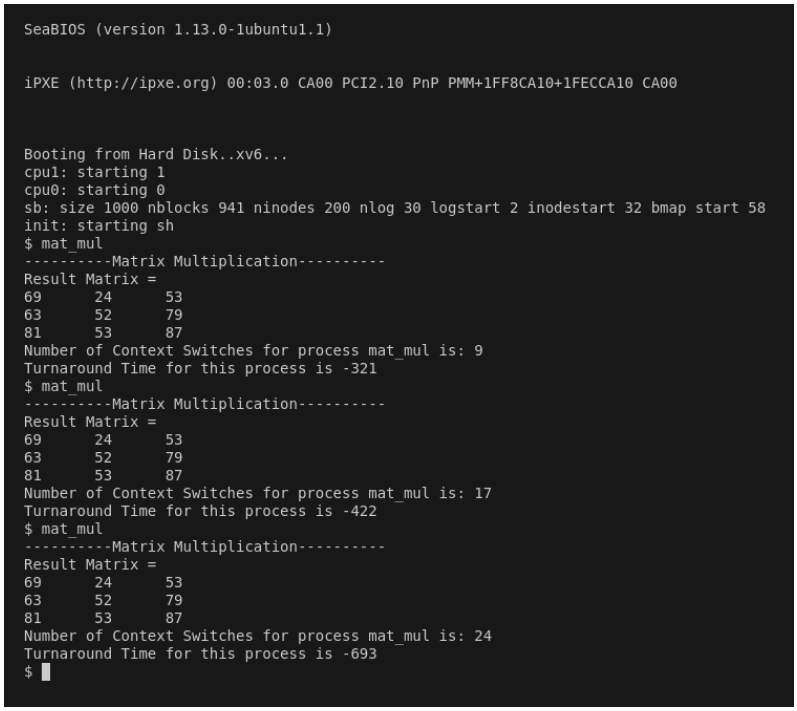
نتایج اجرای پردازش تصویر و محاسبه PSNR با زمان بند تغییریافته

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..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
$ combined
----------Matrix Multiplication----------
Result Matrix =
69      24      53
63      52      79
81      53      87
Number of Context Switches for process mat_mul is: 11
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 32
----------Matrix Multiplication----------
Result Matrix =
69      24      53
63      52      79
81      53      87
Number of Context Switches for process mat_mul is: 37
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 40
Number of Context Switches for process combined is: 13
$
```

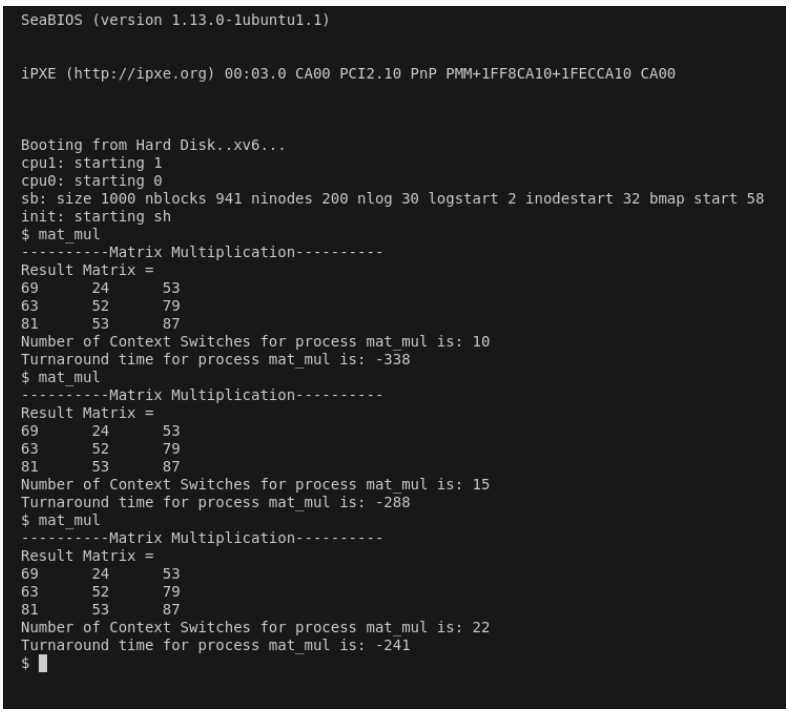
نتایج اجرای برنامه ترکیبی با زمان بند اصلی

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..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
$ combined
----------Matrix Multiplication----------
Result Matrix =
69      24      53
63      52      79
81      53      87
Number of Context Switches for process mat_mul is: 11
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 33
----------Matrix Multiplication----------
Result Matrix =
69      24      53
63      52      79
81      53      87
Number of Context Switches for process mat_mul is: 39
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 43
Number of Context Switches for process combined is: 14
$
```

نتایج اجرای برنامه ترکیبی با زمان بند تغییر یافته



نتایج کامل زمان برگشت ضرب ماتریسی با زمان بند اصلی



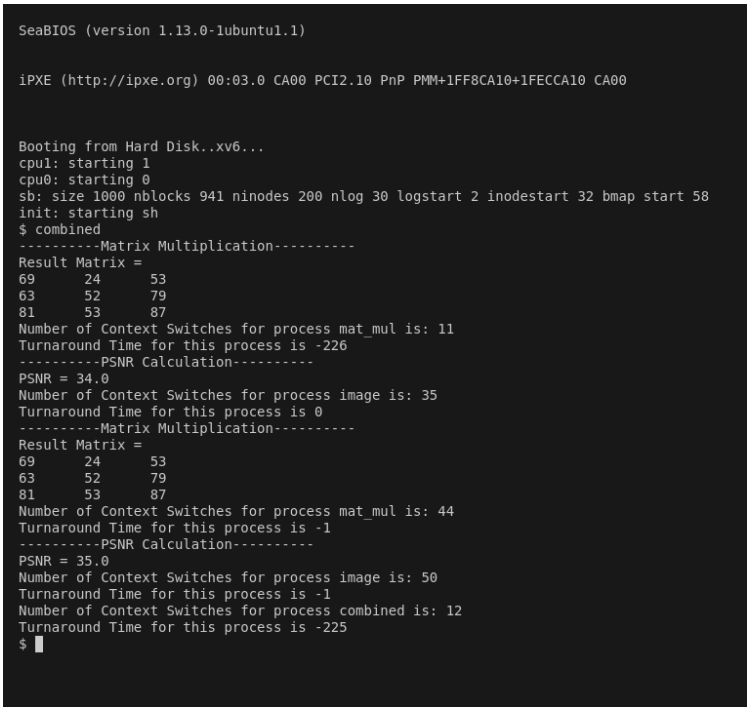
نتایج کامل زمان برگشت ضرب ماتریسی با زمان بند جدید

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..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
$ image
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 20
Turnaround Time for this process is -132
$ image
----------PSNR Calculation----------
PSNR = 34.0
Number of Context Switches for process image is: 23
Turnaround Time for this process is -93
$ image
----------PSNR Calculation----------
PSNR = 34.0
Number of Context Switches for process image is: 29
Turnaround Time for this process is -115
$
```

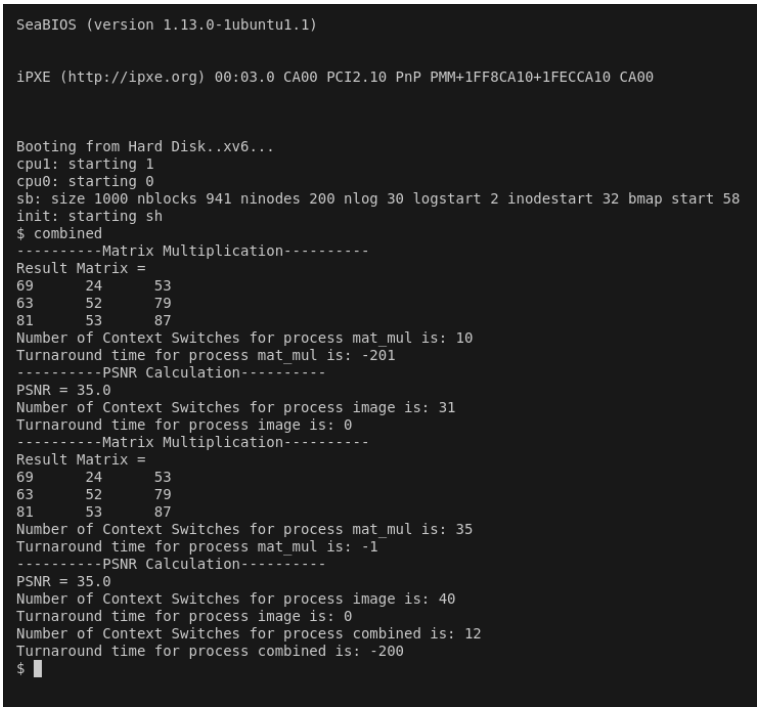
نتایج کامل زمان برگشت برنامه پردازش تصویر با زمان بند اصلی

```
SeaBIOS (version 1.13.0-1ubuntu1.1)


iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8CA10+1FECCA10 CA00



Booting from Hard Disk..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
$ image
----------PSNR Calculation----------
PSNR = 34.0
Number of Context Switches for process image is: 22
Turnaround time for process image is: -148
$ image
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 25
Turnaround time for process image is: -108
$ image
----------PSNR Calculation----------
PSNR = 35.0
Number of Context Switches for process image is: 28
Turnaround time for process image is: -126
$
```

نتایج کامل زمان برگشت برنامه پردازش تصویر با زمان بند جدید



نتایج کامل زمان برگشت برنامه نرکیب پردازش تصویر و ضرب ماتریسی با زمان بند اصلی



نتایج کامل زمان برگشت برنامه نرکیب پردازش تصویر و ضرب ماتریسی با زمان بند جدید

در کنار برنامه ضرب ماتریسی، یک برنامه پردازش تصویر نیز مورد تست قرار گرفته است. در این برنامه یک تصویر 100×100 با کرنل گاوسی به منظور Image Smoothing کانوالو شده و PSNR تصویر نهایی نسبت به یک تصویر دارای نویز حساب شده است. در این برنامه هم از حلقه های تو در تو برای انجام Convolution استفاده شده است و هم در پایان از تابع log10 به منظور حساب کردن PSNR استفاده شده است.

همچنین یک برنامه ترکیبی به نام Combined.c نیز به عنوان تست مورد بررسی قرار گرفته است که در آن تعدادی فرآیند جدید با دستور ()fork ایجاد شده و نیمی از آنها برنامه ضرب ماتریسی و نیمی دیگر برنامه پردازش تصویر را اجرا می‌کنند.

۷- نتیجه گیری

پروژه پیاده‌سازی و ارزیابی زمان‌بند پویا در سیستم‌عامل xv6، به عنوان یکی از پروژه‌های نهایی درس سیستم‌های عامل، به موفقیت انجام شد. هدف اصلی این پروژه بهبود کارایی سیستم و کاهش زمان برگشت فرآیندها از طریق تنظیم دینامیک زمان‌بُر فرآیندها بود.

با بررسی زمان‌بند موجود در xv6، نقاط قوت و ضعف آن شناسایی شدند و بر اساس این تحلیل، زمان‌بند پویا طراحی و پیاده‌سازی شد. در این زمان‌بند، زمان‌بُر فرآیندها به صورت دینامیک بر اساس نیازهای فرآیندها و شرایط سیستم تنظیم می‌شود. این تغییرات منجر به بهبود عملکرد کلی سیستم و افزایش بهره‌وری پردازنده شد.

سیستم فراخوانی‌های جدیدی نیز برای شمارش تعداد تعویض زمینه‌ها و محاسبه زمان برگشت فرآیندها پیاده‌سازی شدند. این سیستم فراخوانی‌ها امکان ارزیابی دقیق عملکرد زمان‌بند پویا را فراهم کردند. نتایج آزمایشات نشان داد که زمان‌بند پویا توانست تعداد تعویض زمینه‌ها را کاهش دهد و زمان برگشت فرآیندها را به طور قابل توجهی بهبود بخشد.

مزایای زمان‌بند پویا شامل افزایش کارایی سیستم، کاهش زمان برگشت فرآیندها و توزیع بهتر منابع پردازشی بین فرآیندها بود. با این حال، پیچیدگی بیشتر پیاده‌سازی و نیاز به تنظیمات دقیق‌تر از جمله چالش‌های این روش بودند.