آزمایشگاه سیستم عامل

گزارش پروژه سوم

- **آرین باستانی** - 810100088
- **مهدیار هرندی** - 810199596
- **محمدرضا نعمتی** - 810100226

- https://github.com/AryanBastani/OS-Lab_CAs
- Last commit hash: 440363f15998645f2321729399bd159033b132bb

1_ این تابع زمانی فراخوانی می شود که یک پردازه آماده به اجرا شود. اولین کار این است که شرایط پردازنده را چک کند. برای این کار، از تابع holding استفاده می کند تا مطمئن شود که ptable قفل نیست و پردازه در حال اجرا نیست (بلکه در حالت RUNNABLE است). سپس، flag های مربوطه را بررسی می کند و اگر مشکلی وجود داشته باشد، با تابع panic خطا را گزارش می دهد. بعد از آن، تابع switch context را اجرا می کند تا context کنونی را ذخیره کند و سپس به تابع scheduler می رود تا پردازه را انتخاب کند و حالت آن را از RUNNABLE به RUNNING تغییر دهد.

2_ لینوکس از یک درخت قرمز-سیاه برای نگهداری پردازه های در حال اجرا استفاده می کند. در این درخت، پردازه ای که کمترین مقدار vruntime را دارد، در گره کوچکترین درخت قرار می گیرد. (vruntime نشان دهنده زمان اجرای پردازه است که در ساختار task_struct ذخیره می شود)

3_ لینوکس و xv6 دو روش متفاوت برای زمانبندی پردازه ها روی پردازنده ها دارند. در xv6، تمام پردازنده ها از یک صف مشترک برای انتخاب پردازه ها استفاده می کنند:

```
10    struct {
11      struct spinlock lock;
12      struct proc proc[NPROC];
13    } ptable;
14
```

این صف حاوی حداکثر 64 پردازه است که با ساختار proc struct نشان داده می شوند. برای جلوگیری از تداخل بین پردازنده ها در دسترسی به این صف، یک spinlock به نام lock.ptable وجود دارد که باید قبل و بعد از هر تغییر در صف، قفل و باز شود. این روش زمانبندی ساده است ولی ممکن است باعث کاهش عملکرد سیستم شود. زیرا هر بار که یک پردازنده یک پردازه را اجرا می کند و به پردازه دیگری می رود، cache پردازنده ممکن است نیاز به بروزرسانی داشته باشد.

در لینوکس، هر پردازنده یک صف مخصوص خودش را برای زمانبندی پردازه ها دارد. پردازه ها به صورت جداگانه در این صف ها قرار می گیرند. این روش زمانبندی پیچیده تر است ولی ممکن است باعث افزایش عملکرد سیستم شود. زیرا هر پردازنده می تواند cache خودش را بهینه کند و پردازه ها را با توجه به اولویت آن ها اجرا کند. این روش زمانبندی نیازمند یک مکانیزم برای balancing load است. یعنی اگر یک پردازنده پردازه ای نداشته باشد و یک پردازنده دیگر پردازه های زیادی داشته باشد، باید بتوانند پردازه ها را به یکدیگر منتقل کنند. این کار در صف مشترک لازم نیست.

4_ گاهی اوقات، هیچ پردازه ای آماده به اجرا نیست و تمام پردازه ها منتظر ورودی یا خروجی هستند. اگر در این حالت، هیچ وقفه ای رخ ندهد و فعال نشود، پردازه ها همیشه در انتظار می مانند و ورودی یا خروجی تکمیل نمی شود. برای جلوگیری از این مشکل، در هر دوره، وقفه ای برای مدت کوتاهی فعال می شود تا پردازه ها را بررسی کند و اگر لازم باشد، حالت آن ها را تغییر دهد. این مسئله در سیستم های یک هسته ای هم ممکن است پیش بیاید.

_5

1. FLIH (first level interrupt handler)
2. SLIH (second level interrupt handler)

همچنین به بخش بالایی و پایینی این دو سطح upper half و lower half نیز گفته می شود. FLIH کارش این است که وقفه هایی را که زودتر باید رسیدگی شوند، مدیریت کند. این کار را با دو روش می تواند انجام دهد:

یا خودش وقفه را حل کند یا اطلاعات مهمی که فقط در زمان وقوع وقفه قابل دسترسی است را ذخیره کند و یک SLIH را برای ادامه کار زمانبندی کند. برای این کار، یک switch context اتفاق می افتد و کد مربوط به هندلر وقفه بارگذاری و اجرا می شود. SLIH کارش این است که وقفه هایی را که زمان بیشتری می خواهند، پردازش کند. این کار را مانند یک پردازه انجام می دهد. یعنی یا یک thread خاص در سطح کرنل برای هر هندلر دارد یا توسط یک pool thread کنترل می شود. سپس، این thread ها در یک صف قرار می گیرند و منتظر می مانند تا اجرا شوند. این thread ها هم مانند پردازه ها زمانبندی می شوند. برای جلوگیری از مشکل starvation در سیستم های real-time، از روش aging استفاده می کنند. این روش این است که هر چه یک پردازه با اولویت کمتر بیشتر منتظر بماند، اولویت آن را بالا می برند تا در نهایت اجرا شود.

پیاده سازی سازوکار افزایش سن:

تابع handle_procs_age را در فایل proc.c اضافه میکنیم. این تابع در تابع trap صدا زده می شود و پراسس ها را بررسی میکند. اگر سن یک پراسس از لیمیت تعریف شده بیشتر بود, صف آن را به صف اول که در اینجا round robin است تغییر میدهد. مقدار last_run که نشان دهنده آخرین زمان اجرای این پراسس است در تابع scheduler مقداردهی می شود.

برای تغییر صف پراسس از تابع change_sched_queue استفاده کردیم که در بخش سیستم کال ها به توضیح آن می پردازیم.

OS-Lab-CAs - proc.c

```
703  void
704  handle_procs_age(int ticks)
705  {
706    struct proc *p;
707    acquire(&ptable.lock);
708
709    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
710      if (p->state == RUNNABLE && p->sched_info.queue != ROUND_ROBIN)
711        if (ticks - p->sched_info.last_run > MAX_AGING_LIMIT)
712        {
713          release(&ptable.lock);
714          change_sched_queue(p->pid, ROUND_ROBIN);
715          acquire(&ptable.lock);
716        }
717
718    release(&ptable.lock);
719  }
```

تغییرات مورد نیاز در استراکت proc:

برای اینکه پراسس ها را به صف های زمانبندی اساین کنیم نیاز داریم فیلد های جدیدی به استراکت پراسس در فایل proc.h اضافه کنیم.

توضیحات فیلد ها در کامنت ها آمده و نیاز به تکرار نیست. :)

OS-Lab-CAs - proc.h

```h
enum schedulequeue { UNSET, ROUND_ROBIN, LCFS, BJF };
#define MAX_AGING_LIMIT 8000
#define BJF_PRIORITY 3

struct bjfinfo {
  int priority;
  float priority_ratio;
  int arrival_time;
  float arrival_time_ratio;
  float executed_cycle;
  float executed_cycle_ratio;
  int process_size;
  float process_size_ratio;
};

struct scheduleinfo {
  enum schedulequeue queue; // Process's queue
  struct bjfinfo bjf;       // BJF scheduling information
  int last_run;             // Last time process was running
};

enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
// 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 creation_time;           // Creation time of the process
  struct scheduleinfo sched_info; // Scheduling information
};
```

سطح اول: Round Robin

برای پیاده سازی این زمانبند, تابع roundrobin را به proc.c اضافه میکنیم.

این تابع در واقع آخرین پراسسی که زمانبندی شده* را دریافت میکند و به پراسس های بعدی می رود تا زمانی که به پراسسی برسد که runnable و مربوط به صف round robin است. سپس آن را به عنوان پراسس مورد نظر برای ران شدن در تابع scheduler ریترن میکند.

*در واقع چون در MFQS ابتدا تمامی پراسس های مربوط به صف اول اجرا می شوند, سپس پراسس های صف های بعدی به ترتیب, پس میتوان گفت پراسسی که این تابع دریافت میکند هم توسط round robin زمانبندی شده است.

OS-Lab-CAs - proc.c

```
331  struct proc*
332  roundrobin(struct proc *last_sched_proc)
333  {
334    struct proc *p = last_sched_proc;
335    while (1)
336    {
337      p++;
338      if (p >= &ptable.proc[NPROC])
339        p = ptable.proc;
340
341      if (p->state == RUNNABLE && p->sched_info.queue == ROUND_ROBIN)
342        return p;
343
344      if (p == last_sched_proc)
345        return 0;
346    }
347  }
```

سطح دوم: Last Come First Serve

برای این زمانبندی تابع lcfs را به فایل proc.c اضافه میکنیم.

پیاده سازی این زمانبندی به این صورت است که در بین تمام پراسس ها به دنبال پراسس هایی میگردد که runnable و متعلق به صف LCFS هستند. سپس از بین این پراسس ها, پراسسی که کمترین عمر را دارد (یعنی فیلد creation time آن بیشترین است) را انتخاب میکند و به عنوان پراسسی که باید اجرا شود به تابع scheduler ریترن میکند.

OS-Lab-CAs - proc.c

```
350  struct proc*
351  lcfs(void)
352  {
353    struct proc *maxP = 0;
354    struct proc* p;
355    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
356      if(p->state != RUNNABLE || p->sched_info.queue != LCFS)
357        continue;
358
359      if (maxP!=0){
360        if(p->creation_time > maxP->creation_time)
361          maxP = p;
362      }
363      else
364        maxP = p;
365    }
366    return maxP;
367  }
368
```

سطح سوم: Best Job First

برای این زمانبندی تابع bestjobfirst را به فایل proc.c اضافه میکنیم.

پیاده سازی این زمانبندی به این صورت است که در بین تمام پراسس ها به دنبال پراسس هایی میگردد که runnable و متعلق به صف BJF هستند. سپس از بین این پراسس ها, پراسسی که کمترین رنک را دارد را انتخاب میکند و به عنوان پراسسی که باید اجرا شود به تابع scheduler ریترن میکند.

برای محاسبه رنک پراسس از تابع calc_bjf_rank استفاده میکنیم که با توجه به فرمول داده شده و فیلد های در دسترس, رنک را بدست میاورد.

OS-Lab-CAs - proc.c

```
368  static inline
369  float
370  calc_bjf_rank(struct proc* p)
371  {
372    return p->sched_info.bjf.priority * p->sched_info.bjf.priority_ratio +
373           p->sched_info.bjf.arrival_time * p->sched_info.bjf.arrival_time_ratio +
374           p->sched_info.bjf.executed_cycle * p->sched_info.bjf.executed_cycle_ratio +
375           p->sched_info.bjf.process_size * p->sched_info.bjf.process_size_ratio;
376  }
377
378  struct proc*
379  bestjobfirst(void)
380  {
381    struct proc* res_p = 0;
382    struct proc* p;
383    float minrank, rank;
384
385    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
386      if(p->state != RUNNABLE || p->sched_info.queue != BJF)
387        continue;
388      rank = calc_bjf_rank(p);
389      if(res_p == 0 || rank < minrank){
390        res_p = p;
391        minrank = rank;
392      }
393    }
394
395    return res_p;
396  }
```

پیاده سازی MFQ:

تابع scheduler را به این صورت تغییر میدهیم که ابتدا از صف round robin پراسس میگیرد و سعی میکند اجرا کند. اگر پراسسی در این صف نبود به سراغ LCFS میرود دوباره اگر پراسسی نبود به سراغ صف آخر یعنی BJF میرود. همچنین در هر مرحله فیلد های مربوط به bjf پراسس را تغییر میدهیم.

OS-Lab-CAs - proc.c

```
406  void
407  scheduler(void)
408  {
409    struct proc *p;
410    struct proc *last_round_robin = &ptable.proc[NPROC - 1];
411    struct cpu *c = mycpu();
412    c->proc = 0;
413
414    for(;;){
415      // Enable interrupts on this processor.
416      sti();
417
418      // Loop over process table looking for process to run.
419      acquire(&ptable.lock);
420
421      p = roundrobin(last_round_robin);
422      if(p)
423        last_round_robin = p;
424
425      else{
426        p = lcfs(); // TODO LCFS
427        if(!p){
428          p = bestjobfirst();
429          if(!p){
430            release(&ptable.lock);
431            continue;
432          }
433        }
434      }
435
436      c->proc = p;
437      switchuvm(p);
438      p->state = RUNNING;
439
440      p->sched_info.last_run = ticks;
441      p->sched_info.bjf.executed_cycle += 0.1f;
442
443
444      swtch(&(c->scheduler), p->context);
445      switchkvm();
446
447      c->proc = 0;
448      release(&ptable.lock);
449
450    }
451  }
```

سیستم کال های اضافه شده:

تغییر صف پراسس ها:

تابع نهایی این سیستم کال change_sched_queue است.

این سیستم کال ابتدا چک میکند اگر صف ورودی UNSET بود, یعنی پراسس تازه ایجاد شده است. در صورتی که اولین پراسس است آن را به round robin در غیر این صورت به صف LCFS اساین میکند.

اگر صف ورودی غیر UNSET بود, بین پراسس ها سرچ میکند و طبق pid ورودی پراسس را پیدا میکند و آن را به صف ورودی اضافه میکند.

OS-Lab-CAs - proc.c

```
673  int
674  change_sched_queue(int pid, int new_queue) {
675    struct proc *p;
676    if (new_queue == UNSET)
677    {
678      if (pid == 1)
679        new_queue = ROUND_ROBIN;
680      else if (pid > 1)
681        new_queue = LCFS;
682      else
683        return -1;
684    }
685    int old_queue = -1;
686    acquire(&ptable.lock);
687    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
688      if(p->pid == pid){
689        old_queue = p->sched_info.queue;
690        p->sched_info.queue = new_queue;
691        break;
692      }
693    }
694    release(&ptable.lock);
695    return old_queue;
696  }
```

تغییر دادن پارامتر های BJF در سطح پراسس و سیستم:

توابع نهایی برای تغییر دادن پارامتر های bjf در هر دو سطح به صورت زیر هستند. تمپلیت کلی هر دو یکسان است.

تابع set_proc_bjf_params بر اساس pid پراسس مورد نظر را پیدا میکند و پارامترها را ست میکند. اگر پراسس پیدا نشد -1 ریترن می شود که نشان دهنده ارور است.

OS-Lab-CAs - proc.c

```c
int
set_proc_bjf_params(int pid, float priority_ratio, float arrival_time_ratio,
                    float executed_cycles_ratio, float process_size_ratio)
{
  acquire(&ptable.lock);
  struct proc* p;
  int found = -1;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->sched_info.bjf.priority_ratio = priority_ratio;
      p->sched_info.bjf.arrival_time_ratio = arrival_time_ratio;
      p->sched_info.bjf.executed_cycle_ratio = executed_cycles_ratio;
      p->sched_info.bjf.process_size_ratio = process_size_ratio;
      found = 0;
      break;
    }
  }
  release(&ptable.lock);
  return found;
}

void
set_global_bjf_params(float priority_ratio, float arrival_time_ratio,
                      float executed_cycles_ratio, float process_size_ratio)
{
  acquire(&ptable.lock);
  struct proc* p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    p->sched_info.bjf.priority_ratio = priority_ratio;
    p->sched_info.bjf.arrival_time_ratio = arrival_time_ratio;
    p->sched_info.bjf.executed_cycle_ratio = executed_cycles_ratio;
    p->sched_info.bjf.process_size_ratio = process_size_ratio;
  }
  release(&ptable.lock);
}
```

چاپ اطلاعات:

در آخر با تابع show_proc_info اطلاعات مربوط به پراسس ها را پرینت می کنیم. خود این تابع صرفا تعدادی پرینت و فرمت کردن دیتا ها است و محتوای خاصی ندارد. در برنامه test_sched از آن استفاده میکنیم.

برنامه های سطح کاربر:

دو برنامه سطح کاربر foo و test_sched را برای بررسی فیچر های اضافه شده به سیستم اضافه میکنیم.

برنامه foo به صورت زیر تعدادی پراسس می سازد که برای اینکه به سرعت تمام نشوند, مقداری عملیات های طولانی انجام میدهند.

OS-Lab-CAs - foo.c

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

#define INF 1000000000000

void make_very_long_calculations()
{
  for (int i = 0; i < 6; i++)
  {
    int pid = fork();
    if (pid > 0)
      continue;
    if (pid == 0)
    {
      sleep(5555);
      for (int j = 0; j < i * 123; j++)
      {
        long long num = 1;
        for (long long k = 0; k < INF; k++)
          num = num << 1;
      }
      exit();
    }
  }
}

int main()
{
  make_very_long_calculations();
  while (wait() != -1)
    ;
  exit();
}
```

برنامه test_sched یک اینترفیس برای کاربر مهیا میکند که با کمک آن بتوانیم عملیات های مختلفی که در این آزمایش در سیستم پیاده کردیم را تست کنیم.

بخشی از نتایج اجرای این دو برنامه در زیر آمده است.

```
mmd@mmd ~/Uni/OS-Lab-CAs/source_code main make qemu
qemu-system-i386 -serial mon:stdio -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
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
Group #3:
1. Mohammad Reza Nemati
2. Aryan Bastani
3. Mahdiar Harandi
$ foo&
$ test_sched info
Process_Name  |  PID   |  State  |  Queue  |  Cycle  |  Arrival  |  Priority  |  R_Prty  |  R_Arvl  |  R_Exec  |  R_Size  |  Rank
-------------------------------------------------------------------------------------------------------------------------------
init            1        sleeping    1         2         0           3          1          1          1          0          5
sh              2        sleeping    1         2         4           3          1          1          1          0          9
foo             5        sleeping    2         56        619         3          1          1          1          0          678
foo             4        sleeping    2         1         618         3          1          1          1          0          622
foo             6        runnable    2         56        620         3          1          1          1          0          679
foo             7        sleeping    2         56        620         3          1          1          1          0          679
foo             8        sleeping    2         56        621         3          1          1          1          0          680
foo             9        sleeping    2         56        621         3          1          1          1          0          680
foo             10       sleeping    2         56        621         3          1          1          1          0          680
test_sched      11       running     2         2         1177        3          1          1          1          0          1182
$ test_sched
usage: test_sched <command> <arg1> <arg2>...
Commands and Arguments: [
  info
  set_queue <pid> <new_queue>
  set_proc_bjf <pid> <priority_ratio> <arrival_time_ratio> <executed_cycle_ratio> <process_size_ratio>
  set_global_bjf <priority_ratio> <arrival_time_ratio> <executed_cycle_ratio> <process_size_ratio>
]
$ test_sched set_queue 6 3
Queue changed successfully
$ test_sched info
exec: fail
exec test_sched failed
$ test_sched info
Process_Name  |  PID   |  State  |  Queue  |  Cycle  |  Arrival  |  Priority  |  R_Prty  |  R_Arvl  |  R_Exec  |  R_Size  |  Rank
-------------------------------------------------------------------------------------------------------------------------------
init            1        sleeping    1         2         0           3          1          1          1          0          5
sh              2        sleeping    1         3         4           3          1          1          1          0          10
foo             5        runnable    2         533       619         3          1          1          1          0          1155
foo             4        sleeping    2         1         618         3          1          1          1          0          622
foo             6        sleeping    3         533       620         3          1          1          1          0          1156
foo             7        sleeping    2         533       620         3          1          1          1          0          1156
foo             8        sleeping    2         533       621         3          1          1          1          0          1157
foo             9        sleeping    2         533       621         3          1          1          1          0          1157
foo             10       sleeping    2         533       621         3          1          1          1          0          1157
test_sched      15       running     2         0         5947        3          1          1          1          0          5950
$ zombie!
```

```
]
$ test_sched info
Process_Name  |  PID   |  State  |  Queue  |  Cycle  |  Arrival  |  Priority  |  R_Prty  |  R_Arvl  |  R_Exec  |  R_Size  |  Rank
-------------------------------------------------------------------------------------------------------------------------------
init            1        sleeping    1         2         0           3          1          1          1          0          5
sh              2        sleeping    1         5         4           3          1          1          1          0          12
test_sched      31       running     2         0         19591       3          1          1          1          0          19594
$ foo&
$ test_sched info
Process_Name  |  PID   |  State  |  Queue  |  Cycle  |  Arrival  |  Priority  |  R_Prty  |  R_Arvl  |  R_Exec  |  R_Size  |  Rank
-------------------------------------------------------------------------------------------------------------------------------
init            1        sleeping    1         2         0           3          1          1          1          0          5
sh              2        sleeping    1         5         4           3          1          1          1          0          12
foo             34       sleeping    2         34        20286       3          1          1          1          0          20323
foo             33       sleeping    2         0         20285       3          1          1          1          0          20288
foo             35       sleeping    2         35        20286       3          1          1          1          0          20324
foo             36       sleeping    2         35        20286       3          1          1          1          0          20324
foo             37       sleeping    2         35        20287       3          1          1          1          0          20325
foo             38       sleeping    2         35        20287       3          1          1          1          0          20325
foo             39       sleeping    2         35        20288       3          1          1          1          0          20326
test_sched      40       running     2         0         20632       3          1          1          1          0          20635
$ test_sched set_global_bjf 3 4 5 1
global BJF params set successfully
$ test_sched info
Process_Name  |  PID   |  State  |  Queue  |  Cycle  |  Arrival  |  Priority  |  R_Prty  |  R_Arvl  |  R_Exec  |  R_Size  |  Rank
-------------------------------------------------------------------------------------------------------------------------------
init            1        sleeping    1         2         0           3          4          5          1          0          14
sh              2        sleeping    1         6         4           3          4          5          1          0          38
foo             34       runnable    2         249       20286       3          4          5          1          0          101691
foo             33       sleeping    2         0         20285       3          4          5          1          0          101437
foo             35       sleeping    2         249       20286       3          4          5          1          0          101691
foo             36       sleeping    2         249       20286       3          4          5          1          0          101691
foo             37       sleeping    2         249       20287       3          4          5          1          0          101696
foo             38       sleeping    2         249       20287       3          4          5          1          0          101696
foo             39       sleeping    2         249       20288       3          4          5          1          0          101701
test_sched      42       running     2         0         22780       3          1          1          1          0          22783
```