بسمه تعالی

گزارش تمرین عملی سوم – سوال اول – درس سیستم‌های عامل دکتر اسدی – نیمسال اول۰۴-۴۰۵

نویسندگان:

۱. سیداحمد موسوی‌اول – ۴۰۲۱۰۶۶۴۸

۲. عرفان تیموری – ۴۰۲۱۰۵۸۱۳

در این تمرین که ادامه تمرین قبلی است، باید دستور top را بهبود بدهیم و برای این کار باید دستورات top -t و top -tree را پیاده‌سازی می‌کردیم که درخت فرایندها را رسم کند و برای این کار دقیقا همانند دستورکار باید یک استراکت process_data داشته باشیم که شامل مواردی مانند pid, patent_id, head_size و ... شود و همچنین یک تابع به اسم next_process داشته باشیم که فرایند بعدی را به ما بدهد و این برای سادگی استفاده شده است.

در ادامه این تابع را ران می‌کنیم و اولین جایی که صفر شد یعنی این فرایند برگ است و فرزندی ندارد و در همین حین هرکسی طبق همان استراکتی که داشتیم آی‌دی پدرش را دارد و بنابراین می‌توانیم شروع به چاپ کردن کنیم.

ولی یک مشکلی که در این حین وجود دارد این است که ما به ترتیب از پدر به فرزند می‌خواهیم چاپ کنیم ولی الان برعکسش رو داریم که این هم با یک‌بار برگشت از فرزند برگ به پدر ریشه مشکل حل می‌شود و درست چاپ می‌شود.

در ادامه بخشی از کد مربوط به این قسمت را می‌توانید ببینید که در زیر کد proc.h را عوض کرده‌ایم و استراکت را به آن افزوده‌ایم:

```
kernel > C proc.h
 85  struct proc {
 89    enum procstate state;        // Process state
 90    void *chan;                  // If non-zero, sleeping on chan
 91    int killed;                  // If non-zero, have been killed
 92    int xstate;                  // Exit status to be returned to parent's wait
 93    int pid;                     // Process ID
 94
 95    // wait_lock must be held when using this:
 96    struct proc *parent;         // Parent process
 97
 98    // these are private to the process, so p->lock need not be held.
 99    uint64 kstack;               // Virtual address of kernel stack
100    uint64 sz;                   // Size of process memory (bytes)
101    pagetable_t pagetable;       // User page table
102    struct trapframe *trapframe; // data page for trampoline.S
103    struct context context;      // swtch() here to run process
104    struct file *ofile[NOFILE];  // Open files
105    struct inode *cwd;           // Current directory
106    char name[16];               // Process name (debugging)
107  };
108
109  struct process_data {
110    int pid;
111    int parent_id;
112    int head_size;
113    enum procstate state;
114    char name[16];
115  };
```

این هم تغییراتی‌ست که در کد syscall.c اضافه کردیم:

```
extern uint64 sys_mknod(void);
extern uint64 sys_unlink(void);
extern uint64 sys_link(void);
extern uint64 sys_mkdir(void);
extern uint64 sys_close(void);
extern uint64 sys_top(void);
extern uint64 sys_next_process(void);

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_pause]   sys_pause,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_top]     sys_top,
[SYS_next_process] sys_next_process,
};
```

این هم تغییراتی‌ست که در کد user.h دادیم:

```
int atoi(const char*);
int memcmp(const void *, const void *, uint);
void *memcpy(void *, const void *, uint);
char* sbrk(int);
char* sbrklazy(int);

// printf.c
void fprintf(int, const char*, ...) __attribute__ ((format (printf, 2, 3)));
void printf(const char*, ...) __attribute__ ((format (printf, 1, 2)));

// umalloc.c
void* malloc(uint);
void free(void*);

void hello(void);
void top(void);

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

struct process_data {
  int pid;
  int parent_id;
  int head_size;
  enum procstate state;
  char name[16];
};
int next_process(int before_pid, struct process_data* out);
```

در ادامه نیاز بود تا تغییراتی هم است تا اضافه کردیم تا بتوانیم دستورات را اجرا کنیم و به دلیل کم بودن آنها و صرفا اضافه کردن یک‌سری syscall دیگر آنها را در فایل عوض نکردیم و انجام دادیم که آنها را می‌توانید در زیر ببینید:

```
#define SYS_top    22
#define SYS_next_process 23
```

```
void            top(void);
int             next_process(int before_pid, struct process_data* out);
```

در این کد به این دقت کردیم که ابتدا سه تا fork زدیم تا تعدادی روابط پدر فرزندی ایجاد شود و خوب دیده شود؛

در ادامه یک تابع fibo(40) اجرا کردیم تا زمانی طول بکشد و با توجه به زمان آن بتواند بفهمد چه کسی فرزند چه کسی است و در واقع سریع تمام نشود و این موضوع را نفهمد

به همین دلیل تفاوت اجرا بدون و با وجود این تابع را می‌توانید در زیر ببینید که در بالایی تا دو لایه فرزندان را شناسایی کرده‌است ولی در پایینی فقط تا یک لایه شناسایی کرده و بعضی فرزندان اشتباه نشان داده شده‌اند:

```
$ top -t
-------------------------------------------------------------
PID     Parent  State           Size    Name
-------------------------------------------------------------
1 | 0 | SLEEPING | 16384 | init
    2 | 1 | SLEEPING | 20480 | sh
        27 | 2 | RUNNING | 86016 | top
            28 | 27 | RUNNABLE | 86016 | top
            29 | 27 | RUNNABLE | 86016 | top
            30 | 27 | RUNNABLE | 86016 | top
    20 | 1 | RUNNING | 86016 | top
        23 | 20 | RUNNABLE | 86016 | top
            26 | 23 | RUNNING | 86016 | top
        24 | 20 | RUNNABLE | 86016 | top
    25 | 1 | RUNNABLE | 86016 | top
-------------------------------------------------------------
$ top -t
-------------------------------------------------------------
PID     Parent  State           Size    Name
-------------------------------------------------------------
1 | 0 | SLEEPING | 16384 | init
    2 | 1 | SLEEPING | 20480 | sh
        35 | 2 | RUNNING | 86016 | top
            36 | 35 | RUNNABLE | 86016 | top
            37 | 35 | RUNNABLE | 86016 | top
            38 | 35 | RUNNABLE | 86016 | top
    28 | 1 | RUNNING | 86016 | top
        31 | 28 | ZOMBIE | 86016 | top
        32 | 28 | RUNNING | 86016 | top
    29 | 1 | RUNNABLE | 86016 | top
        33 | 29 | RUNNABLE | 86016 | top
    34 | 1 | RUNNABLE | 86016 | top
-------------------------------------------------------------
```

کدهای زده شده نیز می‌توانید بعضی از آنها را در زیر ببینید که در کنار همین فایل هم قرار داده‌شده‌اند.

این کد sysfile.c:

```
uint64
sys_top(void)
{
  top();
  return 0;
}

uint64
sys_next_process(void)
{
  int before_pid;
  uint64 proc_addr;
  int return_pid;

  argaddr(1, &proc_addr);
  argint(0, &before_pid);

  struct process_data out;

  return_pid = next_process(before_pid, &out);

  if (return_pid > 0) {
    if (copyout(myproc()->pagetable, proc_addr, (char *)&out, sizeof(out))
  < 0) {
      return 1;
    }
  }

  return return_pid;
}
```

این کد top.c:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include <time.h>
#include "user/user.h"

#define INITIAL_CAP 32
#define MAX_CAP 100

char*
get_state_string(enum procstate state)
{
    switch (state) {
        case SLEEPING:  return "SLEEPING";
        case RUNNABLE:  return "RUNNABLE";
```

```
15.         case RUNNING:   return "RUNNING";
16.         case ZOMBIE:    return "ZOMBIE";
17.         case USED:      return "USED";
18.         case UNUSED:    return "UNUSED";
19.         default:        return "UNKNOWN";
20.     }
21. }
22.
23. int fibo(int n)
24. {
25.   if (n == 1 || n == 0) {
26.     return 1;
27.   }
28.   return fibo(n-1) + fibo(n-2);
29. }
30.

31. void
32. print_table_simple(struct process_data *procs, int count)
33. {
34.     for (int i = 0; i < count; i++) {
35.         printf("%d\t%d\t%s\t%d\t%s\n",
36.                procs[i].pid,
37.                procs[i].parent_id,
38.                get_state_string(procs[i].state),
39.                procs[i].head_size,
40.                procs[i].name);
41.     }
42. }
43.
44. void
45. print_tree_recursive(struct process_data *procs, int count, int
    parent_id, int depth)
46. {
47.     for (int i = 0; i < count; i++) {
48.         if (procs[i].parent_id == parent_id) {
49.             for (int j = 0; j < depth; j++) printf("    ");
50.             printf("%d | %d | %s | %d | %s\n",
51.                    procs[i].pid,
52.                    procs[i].parent_id,
53.                    get_state_string(procs[i].state),
54.                    procs[i].head_size,
55.                    procs[i].name);
56.
57.             print_tree_recursive(procs, count, procs[i].pid, depth + 1);
58.             // printf("depth: %d", depth);
59.         }
60.     }
61. }
62.
63. int
64. main(int argc, char *argv[])
```

```
65. {
66.     int tree_mode = (argc > 1 && (strcmp(argv[1], "-t") == 0 ||
    strcmp(argv[1], "--tree") == 0));
67. 
68.     int cap = MAX_CAP;
69.     struct process_data *all_procs = malloc((uint)(sizeof(struct
    process_data) * cap));
70.     int count = 0;
71. 
72.     int last_pid = 0;
73.     struct process_data current_proc_data;
74.     int ret;
75. 
76.     int pid_root = getpid();
77. 
78.     fork();
79.     fork();
80.     fork();
81. 
82.     if (getpid() != pid_root) {
83.       fibo(35);
84.       exit(0);
85.     }
86. 
87.     if (getpid() == pid_root) {
88.       while (1) {
89.         ret = next_process(last_pid, &current_proc_data);
90.         if (ret == 0) {
91.             break;
92.         } else if (ret < 0) {
93.             break;
94.         }
95. 
96.         if (count >= cap) {
97.             break;
98.         }
99.         all_procs[count++] = current_proc_data;
100. 
101.         last_pid = ret;
102.       }
103. 
104.       printf("------------------------------------------------------------
    \n");
105.       printf("PID\tParent\tState\t\tSize\tName\n");
106.       printf("------------------------------------------------------------
    \n");
107.       if (tree_mode) {
108.         print_tree_recursive(all_procs, count, 0, 0);
109.       } else {
110.         print_table_simple(all_procs, count);
111.       }
```

```
      printf("------------------------------------------------------------
    \n");
      free(all_procs);
      exit(0);
    }
}
```

این هم کد proc.c:

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

  printf("\n--- Process List ---\n");
  printf("PID\tSIZE\tSTATE\tNAME\n");

  for(p = proc; p < &proc[NPROC]; p++){
    if(p->state == UNUSED)
      continue;
    if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
      state = states[p->state];
    else
      state = "???";
    printf("%d\t%s\t%s\t%lu", p->pid, state, p->name, p->sz);
    printf("\n");
  }
}

int
next_process(int before_pid, struct process_data *out)
{
  struct proc *p;
  struct proc *best = 0;
  int best_pid = 1000000000;

  acquire(&pid_lock);

  for(p = proc; p < &proc[NPROC]; p++){
    acquire(&p->lock);

    if(p->state != UNUSED && p->pid > before_pid) {
```

```
      if(p->pid < best_pid) {
        best = p;
        best_pid = p->pid;
      }
    }

    release(&p->lock);
  }

  if(best) {
    acquire(&best->lock);

    out->pid = best->pid;
    out->parent_id = best->parent ? best->parent->pid : 0;
    out->head_size = best->sz;
    out->state = best->state;
    safestrcpy(out->name, best->name, sizeof(out->name));

    release(&best->lock);
  }

  release(&pid_lock);

  return best_pid == 1000000000 ? 0 : best_pid;
}
```

این کدها تماما در کنار همین فایل قرار گرفته‌اند.