به نام خدا

علی شیخ عطار و صبا کیانوش

پیاده سازی

قسمت اول

OS document

در صورت سوال خواسته شده که محتوای جدول ptable نمایش داده شود که طبق تحقیقات انجام شده فهمیدیم که این(ویرگول بزار اینجا) جدول فرایند ها در سیستم عامل xv6 را نشان میدهد.
ساختار این جدول به صورت زیر میباشد:

```
Struct {
        Struct spinlock lock;
        Struct proc proc[NROC];
}  ptable
```

ptable یک متغیر از نوع struct است که شامل یک آرایه از ساختارهای proc و یک قفل (lock) برای مدیریت همزمانی است. این قفل اطمینان می‌دهد که دسترسی به ptable به صورت امن و بدون تناقض انجام می‌شود.
ptable شامل تمام فرآیندهای موجود در سیستم است و هر ورودی در آرایه proc نشان‌دهنده یک فرآیند خاص است. با استفاده از ptable، سیستم عامل می‌تواند فرآیندهای جدید ایجاد کند، فرآیندهای موجود را مدیریت کند و اطلاعات مربوط به هر فرایند را به روز رسانی کند.
دسترسی به ptable باید به صورت همزمانی کنترل شود تا از تناقضات جلوگیری شود. برای این منظور، از یک قفل چرخشی (spinlock) استفاده می‌شود. این قفل باعث می‌شود که هر زمان که یک فرآیند در حال دسترسی به ptable است، سایر فرآیندها منتظر بمانند تا دسترسی به ptable آزاد شود.

- نصب و راه اندازی:

با استفاده از دستورات نوشته شده روی سیستم عامل لینوکس xv6 را clone کردیم و برای کامپایل کردن کد از QEMU استفاده می شود.

- چرایی استفاده از QEMU:

از QEMU برای شبیه سازی یک محیط سخت افزاری استفاده می‌شود که در آن ما می‌توانیم سیستم عامل خود را اجرا و تست کنید.
اجرای سیستم عامل: ما می‌توانیم سیستم عامل XV6 خود را بدون نیاز به سخت‌افزار واقعی روی QEMU اجرا کنید.
دیباگ کردن سیستم عامل: با استفاده از ابزارهای دیباگ مانند GDB و قابلیت‌های داخلی QEMU، ما می‌توانیم مشکلات سیستم عامل را پیدا و برطرف کنیم.

پشتیبانی از چندین سیستم عامل: QEMU می‌تواند چندین سیستم عامل را به طور همزمان اجرا کند، بنابراین ما می‌توانیم سیستم عامل XV6 خود را در کنار دیگر سیستم عامل‌ها تست کنیم.
به طور خلاصه، QEMU یک ابزار بسیار قدرتمند و انعطاف‌پذیر است که فرآیند توسعه، تست و دیباگ کردن سیستم عامل را به طور قابل توجهی ساده‌تر و امن‌تر می‌کند. این ویژگی‌ها باعث می‌شود که QEMU برای پروژه‌های آموزشی و توسعه سیستم عامل مانند XV6 بسیار مناسب باشد.

- پیاده سازی:

برای شروع پیاده سازی ابتدا باید فایل های proc.c و proc.h را مطالعه می کردیم و ارتباط بین proc struct و procstate enum با ptable را بفهمیم.
طبق مطالعه ی فایل proc.c:
در سیستم عامل XV6، فایل proc.c نقش بسیار مهمی در مدیریت فرآیندها ایفا می‌کند. این فایل شامل کدهای مربوط به ایجاد، زمان‌بندی، مدیریت و خاتمه فرآیندها است. به عبارت دیگر، proc.c بخشی از هسته سیستم عامل است که عملیات های مربوط به فرآیندها را پیاده سازی می‌کند.
توابع مربوط به سیستم کارهای مختلف مانند sbrk، kill، wait، exit، fork و sleep در این فایل قرار دارند. این توابع به کاربر اجازه می‌دهند تا با فرآیندها تعامل داشته باشد.

حال در فایل proc.h ساختار ptable را داریم که در بالاتر در توضیحات ptable ارده شده است.
در فایل proc.c در ptable یک آرایه proc دارد که آرایه ای از proc هاست که این proc در فایل proc.h تعریف شده است. که این ساختار proc شامل اطلاعات مختلفی مانند شناسه فرآیند (PID)، حالت فرآیند، اشاره‌گر به والد فرآیند، اندازه حافظه، و غیره است.

- مراحل انجام بخش اول:

۱.ابتدا باید در فایل syscall.h سیستم کال مورد نظرمان را تعریف کنیم.
به این صورت:

```
#define  SYS_ps 22
```

2.سپس در مرحله بعد در فایل syscall.c سیستم کال جدید را اعلام کنید.

```
Extern int sys_ps(void);
```

در این خط در syscall.c تنها اعلام می‌کند که تابع sys_ps وجود دارد و در جای دیگری پیاده‌سازی شده است. این اعلامیه به کامپایلر اجازه می‌دهد تا از این تابع استفاده کند، حتی اگر تعریف کامل آن در فایل دیگری باشد. این قسمت از فرآیند اضافه کردن یک سیستم کال جدید به سیستم عامل XV6 است و برای مدیریت و دسترسی به سیستم کال ها ضروری است.

۳.سپس این سیستم کال را به جدول سیستم کال ها اضافه می کنیم:

```
[SYS_ps] sys_ps,
```

۴.حال در فایل proc.c تابع اصلی را پیاده سازی میکنیم:

```
int sys_ps(void) {
  struct proc *p;
  acquire(&ptable.lock);
  cprintf("PID\tState\t\tName\n");
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
      if(p->state == UNUSED)
          continue;
      cprintf("%d\t%s\t%s\n", p->pid, states[p->state], p->name);
  }
  release(&ptable.lock);
  return 0;
}
```

تابع sys_ps برای نمایش لیستی از فرایندهای فعال در سیستم به همراه اطلاعات مرتبط با آن‌ها مانند شناسه فرآیند (PID)، وضعیت فرآیند و نام فرآیند طراحی شده است. این تابع به عنوان یک سیستم‌کال تعریف شده است و با استفاده از قفل‌بندی (locking) از تداخل و مشکلات همزمانی جلوگیری می‌کند.

۵. در سیستم عامل XV6، فایل user.h شامل اعلان هایی برای سیستم کال ها و توابع کتابخانه ‌ایی است که در سطح کاربر استفاده می‌شوند. تابع زیر در فایل user.h نمایانگر اعلان سیستم کالی است که در فایل user.h تعریف شده و در فضای کاربر (user space) استفاده می‌شود.این تابع وقتی در سطح user صدا زده می شود به سیستم کال sys_ps در کرنل اشاره می شود و اجرا میشود.

```
Int ps(void);
```

۶. در این مرحله در فایل usys.S خط زیر را قرار میدهیم.این فایل رابط بین user و kernel است.این خط یک ماکرو (Macro) است که برای تعریف یک سیستم‌کال در فایل usys.s استفاده می‌شود. هدف اصلی این ماکرو ایجاد یک تابع با نام ps در فضای کاربر است که به طور مستقیم سیستم کال معادل آن را در فضای کرنل فراخوانی می‌کند.

```
SYSCALL(ps)
```

قسمت دوم

توضیح مرحله دوم پیاده سازی

هدف

بهبود سیستم کال ساده برای پذیرش آرگومان‌ها و فیلتر کردن فرآیندها بر اساس حالت و PID.

مراحل و پیاده سازی

۱. تغییر تعریف سیستم کال برای پذیرش آرگومان‌ها:

ابتدا باید اطمینان حاصل کنیم که سیستم کال ps به درستی در syscall.h تعریف شده است تا آرگومان‌های لازم را شامل شود. این خط کد سیستم کال ps را با شناسه 22 در سیستم شناسایی می‌کند.

```
#define SYS_ps 22
```

این تعریف اطمینان حاصل می‌کند که سیستم کال ما دارای شناسه منحصربه فرد 22 است.

۲. به روزرسانی جدول سیستم کال:

سپس، باید تابع sys_ps را در جدول سیستم کال در syscall.c اضافه کنیم. این بخش سیستم کال شماره 22 را به تابعی که در هسته اجرا می‌شود نگاشت می‌کند.

```
extern int sys_uptime(void);
extern int sys_ps(void);


static int (*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_sleep]   sys_sleep,
[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_ps]      sys_ps,
```

۳. به روزرسانی sys_ps برای مدیریت آرگومان‌ها:

حالا تابع sys_ps را در proc.c به روزرسانی می‌کنیم تا آرگومان‌ها را مدیریت کرده و اطلاعات فرآیندها را بر اساس فیلترهای حالت و PID برگرداند.

```c
int sys_ps(void) {
    else if (p->state == ZOMBIE)
      state_name = "ZOMBIE";
    else
      state_name = "???";

    if (p->parent->state == UNUSED)
      parent_state_name = "UNUSED";
    else if (p->parent->state == EMBRYO)
      parent_state_name = "EMBRYO";
    else if (p->parent->state == SLEEPING)
      parent_state_name = "SLEEPING";
    else if (p->parent->state == RUNNABLE)
      parent_state_name = "RUNNABLE";
    else if (p->parent->state == RUNNING)
      parent_state_name = "RUNNING";
    else if (p->parent->state == ZOMBIE)
      parent_state_name = "ZOMBIE";
    else
      parent_state_name = "???";

    // Filter based on state and PID
    if ((state_t == -1 || p->state == state_t) && (pid_t == -1 || p->pid == pid_t)) {
      cprintf("pid: %d state: %s name: %s ppid: %d pstate: %s\n", p->pid, state_name, p->name,
                                                                  p->parent->pid,parent_state_name);
      //process_info_t->pid = p->pid;
      //process_info_t->state = p->state;
      //*process_info_t->parent = *p->parent;
      found = 1;
      //char* proc_name = (char*)malloc(sizeof(*p->name));
      //snprintf(process_info_t->name, sizeof(process_info_t->name), "%s", p->name);


    }
    if (found == 0)
    {
      if (elected_state_id  == UNUSED)
        state_name = "UNUSED";
      else if (elected_state_id == EMBRYO)
        state_name = "EMBRYO";
      else if (elected_state_id == SLEEPING)
        state_name = "SLEEPING";
      else if (elected_state_id == RUNNABLE)
        state_name = "RUNNABLE";
      else if (elected_state_id == RUNNING)
        state_name = "RUNNING";
      else if (elected_state_id == ZOMBIE)
        state_name = "ZOMBIE";
      else
        state_name = "???";

      cprintf("pid: %d state: %s name: %s\n", elected_id, state_name, elected_name);
    }


  }
  return 0;
}
```

```c
nt sys_ps(void) {
 int state_t;
 int pid_t;
 //struct proc_info* process_info_t = (struct proc_info*) malloc(sizeof(struct proc_info));
 struct proc *p;
 char *state_name;
 char *parent_state_name;
 int min_dis = __INT16_MAX__;
 // struct proc *elected_p = (struct proc*)0x32123456;
 int elected_id = 0;      You, 2 days ago • Uncommitted changes
 char* elected_name = "ss";
 enum procstate elected_state_id = UNUSED;
 int found = 0;
 // Get the arguments from user space
 if (argint(0, &state_t) < 0)
   return -1;
 if (argint(1, &pid_t) < 0)
   return -1;
 //if (argptr(1, (char**)process_info_t, sizeof(struct proc_info)) < 0)
  // return -1;

 // Iterate through the process table
 for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
   if (p->state == state_t)
   {
     if (pid_t != -1){
       if (ABS(p->pid ,pid_t) <= min_dis && ABS(p->pid ,pid_t) != 0){
         min_dis = ABS(p->pid ,pid_t);
         elected_id = p->pid;
         elected_name = p->name;
         elected_state_id = p->state;
       }
     }
   }




   if (p->state == UNUSED)
     state_name = "UNUSED";
   else if (p->state == EMBRYO)
     state_name = "EMBRYO";
   else if (p->state == SLEEPING)
     state_name = "SLEEPING";
   else if (p->state == RUNNABLE)
     state_name = "RUNNABLE";
   else if (p->state == RUNNING)
     state_name = "RUNNING";
   else if (p->state == ZOMBIE)
     state_name = "ZOMBIE";
   else
     state_name = "???";

   if (p->parent->state == UNUSED)
     parent_state_name = "UNUSED";
   else if (p->parent->state == EMBRYO)
     parent_state_name = "EMBRYO";
```

این تابع، جدول فرآیندها (ptable) پیمایش می‌شود و اگر فرآیندی با شرایط فیلتر شده تطابق داشته باشد، اطلاعات آن در آرایه info ذخیره می‌شود.

۴. تعریف ساختار processInfo در proc.h:

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
You, 2 days ago | 1 author (You)
struct proc_info {
    enum procstate state;  // Process state
    int pid;               // Process ID
    struct proc *parent;   // Parent process
    char name[16];         // Process name (debugging)
};
```

ساختار processInfo را در proc.h تعریف می کنیم تا اطلاعات فرآیند را ذخیره کند.
این ساختار شامل شناسه فرآیند، حالت فرآیند و نام فرآیند و فرآیند پدر است.

۵. به روز رسانی برنامه کاربری (ps.c):
حالا برنامه کاربری ps.c را به روزرسانی کنید تا سیستم‌کال جدید را با آرگومان‌ها فراخوانی کند.

```
int main(int argc, char *argv[]) {
  int state_t = -1; // -1 means do not filter by state
  int pid_t = -1;   // -1 means do not filter by PID      You, 2 days ago • sec
  if (argc > 1) {
    state_t = atoi(argv[1]);
  }
  if (argc > 2) {
    pid_t = atoi(argv[2]);
  }

// Assuming enum procstate is defined elsewhere in the code

struct proc_info *process_info_t;
  process_info_t = (struct proc_info*) malloc(sizeof(struct proc_info));
  ps(state_t, pid_t, process_info_t);
  //printf("pid: %d state: %d name: %s\n", (int)process_info_t->pid, (int)proce
  exit();
  free(process_info_t);
}
```

این برنامه کاربری آرگومان‌های ورودی را می‌خواند، سیستم‌کال ps را فراخوانی می‌کند و اطلاعات فرآیندها را چاپ می‌کند.
در این مرحله، سیستم‌کال ps را بهبود دادیم تا بتواند آرگومان‌هایی برای فیلتر کردن فرآیندها بر اساس حالت و PID پذیرش کند. این پیاده‌سازی شامل به‌روزرسانی سیستم‌کال، تعریف ساختارهای لازم، و نوشتن برنامه کاربری برای تست سیستم‌کال بود. این رویکرد ساختار یافته به ما کمک می‌کند تا درک عمیقی از عملکرد داخلی سیستم عامل داشته باشیم.

تحقیق درباره ی *QEMU*

QEMU یک ماشین مجازی عمومی و متن باز است که می‌تواند به چندین صورت مختلف مورد استفاده قرار گیرد. روش معمول برای استفاده از QEMU شبیه‌سازی سیستم است که در آن یک مدل مجازی از یک ماشین کامل (پردازنده ، حافظه و دستگاه‌های شبیه سازی شده) برای اجرای سیستم عامل guest فراهم می‌کند. در این حالت ، پردازنده ممکن است کاملاً شبیه سازی شود ، یا ممکن است با یک هایپروایزر مانند KVM ، Xen یا Hypervisor.Framework کار کند تا سیستم عامل guest بتواند مستقیماً روی پردازنده host اجرا شود.

روش دوم پشتیبانی شده برای استفاده از QEMU شبیه سازی حالت کاربر است ، که در آن QEMU می‌تواند پردازش‌هایی که برای یک پردازنده کامپایل شده اند را روی پردازنده دیگری اجرا کند. در این حالت ، پردازنده همیشه شبیه سازی می‌شود.[1]

QEMU همچنین تعدادی ابزار مستقل خط فرمان را فراهم می‌کند ، مانند ابزار qemu-img disk image که به شما اجازه می‌دهد تا disk image را ایجاد ، تبدیل و تغییر دهید.

Source: Conversation with Bing, 5/19/2024
https://www.qemu.org/docs/master/about/index.html.
https://it-planet.ir/40770.
https://www.qemu.org/.

مطالعه ی *proc.h , struct proc & enum procstate*

1. بررسی فایل proc.h

فایل proc.h شامل تعریف ساختار فرایند (struct proc) و نوع حالت فرایند (enum procstate) است.

ساختار enum procstate

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

این enum نشان‌دهنده حالات مختلف یک فرایند است که شامل موارد زیر است:

UNUSED: فرایند استفاده نمی‌شود.

EMBRYO: فرایند در حال ایجاد شدن است.

SLEEPING: فرایند در حال خواب است و منتظر یک رویداد است.

RUNNABLE: فرایند آماده اجرا است و می‌تواند به صف اجرای CPU وارد شود.

RUNNING: فرایند در حال اجرا روی CPU است.

ZOMBIE: فرایند خاتمه یافته است ولی هنوز منابع آن آزاد نشده است.

ساختار struct proc

```
struct proc {
  uint sz;                     // Size of p
  pde_t* pgdir;                // Page tabl
  char *kstack;                kernel stack for
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         / Parent process
  struct trapframe *tf;        for current sysc
  struct context *context;     e to run process
  void *chan;                  sleeping on cha
  int killed;                  , have been kill
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           ent directory
  char name[16];               e (debugging)
};
```

این ساختار شامل اطلاعات مختلفی در مورد فرایند است، از جمله اندازه حافظه، جدول صفحات، پشته کرنل، حالت فرایند، شناسه فرایند، اشاره‌گر به فرایند والد، و غیره.

2. بررسی فایل proc.c

در فایل proc.c، ptable به عنوان جدول فرایندها تعریف شده است. این جدول شامل آرایه‌ای از ساختارهای proc است که تمام فرایندهای سیستم را نگهداری می‌کند.

تعریف ptable

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

3. ارتباط ساختارهای proc و procstate با ptable

ptable شامل آرایه‌ای از proc است که هر عنصر آن یک فرایند در سیستم را نشان می‌دهد. هر فرایند دارای یک حالت (state) است که از نوع procstate است. این حالت می‌تواند یکی از مقادیر UNUSED، EMBRYO، RUNNING، RUNNABLE، SLEEPING یا ZOMBIE باشد.

4. ماشین حالت (State Machine) فرایند

هر فرایند در سیستم عامل به صورت ماشین حالت عمل می‌کند، به این صورت که در یک زمان در یکی از حالات procstate قرار دارد و بر اساس رویدادهای مختلف می‌تواند بین این حالات جابجا شود.

تغییر حالات فرایندها

ایجاد فرایند جدید: حالت EMBRYO -> حالت RUNNABLE

زمان‌بندی فرایند: حالت RUNNABLE -> حالت RUNNING

مسدود شدن فرایند: حالت RUNNING -> حالت SLEEPING

بیدار شدن فرایند: حالت SLEEPING -> حالت RUNNABLE

خاتمه فرایند: حالت RUNNING -> حالت ZOMBIE

نمونه کد برای استفاده از procstate و ptable

برای مثال، کد زیر تمامی فرایندهای موجود در سیستم را چاپ می‌کند:

```
void list_processes(void) {
    struct proc *p;

    acquire(&ptable.lock);
    for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if (p->state != UNUSED) {
            cprintf("pid: %d state: %d name: %s\n", p->pid,
p->state, p->name);
        }
    }
    release(&ptable.lock);
}
```

این کد با استفاده از قفل ptable، از هر فرایند در جدول ptable عبور کرده و اطلاعات فرایندهایی که در حالت UNUSED نیستند را چاپ می‌کند.

با مطالعه ساختارهای proc و procstate در فایل proc.h و بررسی ارتباط آن‌ها با ptable در فایل proc.c، می‌توانیم ماشین حالت فرایندها را درک کنیم و با استفاده از این اطلاعات، کدهایی برای مدیریت و نمایش فرایندها در سیستم عامل XV6 بنویسیم.

مطالعه ی *Makefile* و *Linker.ld* و ارتباط آن با *Qemu* و *Boot*

فایل `Makefile` و فایل `linker.ld` نقش مهمی در فرایند کامپایل و لینک کردن پروژه های نرم افزاری ایفا می‌کنند. این فایل ها تنظیمات و دستورات مورد نیاز برای ساخت پروژه و تنظیمات لینک کردن را فراهم می‌کنند. در ادامه به بررسی مفاهیم کلی و نحوه ارتباط آن‌ها با `QEMU` و فرآیند Boot می‌پردازیم.

بررسی فایل `Makefile`

فایل `Makefile` در زبان C به منظور خودکارسازی فرایند کامپایل و ساخت پروژه استفاده می‌شود. این فایل شامل دستوراتی است که به `make` می‌گوید چگونه کد منبع را به فایل‌های اجرایی تبدیل کند.

بخش‌های اصلی یک `Makefile`

تعریف متغیرها:

```
CC = gcc
CFLAGS = -Wall -Werror
```

متغیرها برای ذخیره مسیرها، نام فایل ها و پارامترهای کامپایلر استفاده می‌شوند. قواعد (Rules): هر قاعده شامل یک هدف (target)، وابستگی ها (dependencies) و دستورات (commands) است.

```
target: dependencies command
```

بررسی فایل `linker.ld`

فایل `linker.ld` فایل تنظیمات لینک کردن است که به لینک کننده (Linker) می‌گوید چگونه بخش های مختلف برنامه را به هم پیوند دهد و فایل اجرایی نهایی را ایجاد کند.

ارتباط با `QEMU` و فرآیند Boot

`QEMU` یک شبیه ساز سخت افزار است که می‌تواند سیستم عامل ها و برنامه ها را در یک محیط مجازی اجرا کند. فایل `Makefile` و `linker.ld` در فرآیند Boot و اجرای سیستم عامل در `QEMU` نقش مهمی دارند.

فرآیند Boot با `QEMU`

1. کامپایل و لینک کردن:
   - `Makefile` فایل های منبع را کامپایل کرده و با استفاده از تنظیمات `linker.ld` فایل اجرایی نهایی را ایجاد می‌کند.
2. اجرای سیستم عامل در `QEMU`:
   - فایل اجرایی نهایی که توسط `Makefile` و `linker.ld` تولید شده، توسط `QEMU` بارگذاری و اجرا می‌شود.
   - `QEMU` مراحل اولیه Boot را شبیه سازی می‌کند و کنترل را به سیستم عامل منتقل می‌کند.

مثال اجرای سیستم عامل با `QEMU`

فرض کنید که فایل اجرایی نهایی به نام `kernel.img` ایجاد شده است

این دستور `QEMU` را راه‌اندازی کرده و فایل `kernel.img` را به عنوان هسته سیستم عامل بارگذاری و اجرا می‌کند.

نتیجه گیری

فایل‌های `Makefile` و `linker.ld` برای مدیریت فرآیند ساخت و لینک کردن پروژه های نرم افزاری بسیار حیاتی هستند. این فایل‌ها به `make` و لینک کننده دستور می‌دهند که چگونه کد منبع را به فایل اجرایی تبدیل کنند. در ارتباط با `QEMU`، این فایل‌ها نقش مهمی در شبیه سازی و اجرای سیستم عامل ایفا می‌کنند و مراحل اولیه Boot سیستم را مدیریت

می‌کنند. مطالعه و درک این فایل‌ها به شما کمک می‌کند تا بهتر بتوانید پروژه های بزرگ نرم افزاری را مدیریت و کامپایل کنید.

تحقیق درباره ی پوینتر پاس‌داده شده از فضای کاربر به عنوان آرگومان

برای پاسخ به این سوال که آیا پوینتر پاس داده شده از فضای کاربر به عنوان آرگومان سیستم کال هنوز به عنوان یک حافظه معتبر در فضای کرنل معتبر است یا خیر، باید به جزئیات مکانیزم سیستم کال و مدیریت حافظه در سیستم عامل بپردازیم.

توضیح مکانیزم سیستم کال و مدیریت حافظه

هنگامی که یک سیستم کال از فضای کاربر فراخوانی می‌شود، کنترل از فضای کاربر به فضای کرنل منتقل می‌شود. در این انتقال، CPU حالت خود را از user mode به kernel mode تغییر می‌دهد و کرنل شروع به اجرای کد مربوط به سیستم کال می‌کند. پرینترهایی که به عنوان آرگومان به

سیستم کال ها پاس داده می‌شوند، آدرس‌های مجازی هستند که در فضای آدرس فرایند کاربر معتبر هستند. اما این آدرس ها به صورت مستقیم در فضای کرنل معتبر نیستند و نیاز به ترجمه دارند.

اعتبار پوینترها در فضای کرنل

در بسیاری از سیستم عامل‌ها، برای اطمینان از امنیت و ثبات سیستم، کرنل نمی‌تواند به صورت مستقیم از پوینترهای فضای کاربر استفاده کند. ابتدا باید این پوینترها را اعتبارسنجی و سپس ترجمه کند.

مراحل اعتبارسنجی و ترجمه پوینترها

1. اعتبارسنجی پوینتر:

بررسی می‌شود که آدرس مجازی پوینتر کاربر در محدوده فضای آدرس مجازی فرآیند کاربر قرار دارد و به ناحیه ای از حافظه که به فرآیند کاربر تخصیص داده شده است، اشاره می‌کند.

2. ترجمه آدرس مجازی به آدرس فیزیکی:

کرنل از جدول صفحات (page tables) فرآیند کاربر استفاده می‌کند تا آدرس مجازی پوینتر را به آدرس فیزیکی ترجمه کند.

3. دسترسی به داده ها:

کرنل می‌تواند با استفاده از آدرس فیزیکی، به داده‌های مورد نظر دسترسی پیدا کند.