به نام خدا

دانشگاه شهید بهشتی

رشته‌ی مهندسی کامپیوتر

درس آزمایشگاه سیستم عامل
استاد : دکتر آهوز

دستور کار شماره‌ی ۳ :

برنامه‌های کاربر

اعضای گروه :

متین ارجمند **امیرحسین جعفرزاده**
محدثه صفاری **علی نصرالله پور**

فهرست :

فاز اول – شناخت ساختار و آماده‌سازی :

۱. هدف فاز :

هدف این فاز آشنایی اولیه با ساختار Pintos و کامپوننت‌های اصلی زیرسیستم userprogاست. در این مرحله همچنان هیچ سیستم‌کالی پیاده‌سازی نمی‌شود و تنها عملکرد اولیه هسته بررسی می‌گردد.

فاز اول موارد زیر را پوشش می‌دهد:

- اجرای ابتدایی Pintos بدون هیچ تغییر
- مرور فایل‌های مهم در userprog شامل:
  - process.c
  - syscall.c
  - exception.c
- اجرای نمونه برنامه‌های کاربر مانند echo بدون آرگومان

۲. بررسی اجرای اولیه pintos : در این بخش Pintos بدون هیچ تغییری build و اجرا می‌شود:

```
cd pintos/src/userprog/build
make
pintos -- run 'echo'
```

که خروجی شامل :

- بوت شدن هسته
- پیام«system call!»
- کرش برنامه echo به دلیل عدم پیاده‌سازیsyscall ها

و نشان میدهد که که سیستم‌کال‌ها هنوز پیاده‌سازی نشده‌اند.

۲. مرور فایل‌های User Program Subsystem :

process.c :

- وظیفه بارگذاری برنامه کاربر(load)
- ایجاد thread جدید برای user process
- تابع مهم start_process
- تابع process_execute برای ایجاد فرآیند فرزند

syscall.c :

- ثبت وقفه 0x30 برایcall های کاربر
- تابع syscall_handler که هنوز فقط پیام "system call!" چاپ می‌کند
- جایگاه پیاده‌سازی تمام سیستم‌کال‌های پروژه

exception.c :

- رسیدگی به exception ها
- تابع kill()که هنگام page fault یا exception دیگر فرآیندها را به‌درستی خاتمه می‌دهد.

۳. اجرای اولیه برنامه‌های نمونه بدون آرگومان:

با دستور 'pintos -- run 'echo که خروجی : system call! همان دستوری است که نشان میدهد syscall هنوز پیاده سازی نشده.

جمع بندی فاز :

در این فاز، هیچ تغییر کدی اعمال نشده و هدف صرفاً آشنایی با ساختار Pintos و روند اجرای برنامه‌های کاربر بوده است. نتیجهٔ مهم این فاز:

- تأیید صحت build و اجرایPintos
- مشاهدهٔ فراخوانی0x30 interrupt

- آشنایی دقیق با مسیر اجرای برنامه:
process_execute → start_process → load → entry → syscall_handler
- آماده‌سازی برای فاز بعدی که در آن سیستم‌کال halt و exit پیاده‌سازی می‌شود.

فاز دوم – طراحی :

۱. انتخاب روش مناسب مدیریت حافظه‌ی کاربر:

هدف طراحی :

سیستم‌کال‌ها باید بتوانند به داده‌های کاربر (user memory) دسترسی پیدا کنند بدون اینکه باعث page fault در کرنل شوند. بنابراین لازم است روشی طراحی شود که:

- آدرس‌های دریافتی از کاربر معتبر باشند،
- دسترسی به صفحه‌های user فقط در صورت map بودن انجام شود،
- کرنل در هیچ حالتی روی آدرس نامعتبر کرش نکند.

طراحی پیشنهادی:

در این پروژه تصمیم گرفتیم سه تابع برای اعتبارسنجی حافظهٔ کاربر طراحی کنیم:

۱) validate_user_ptr (const void *uaddr)

- بررسی می‌کند آدرس کاربر:
  - در فضای user باشد(is_user_vaddr)
  - صفحهٔ مربوطه مقداردهی شده باشد(pagedir_get_page != NULL)
- در غیر این صورت، فوراً (sys_exit(-1صدا زده می‌شود.

```
// if uaddr is not a valid user address, end the process with exit(-1)
static void
validate_user_ptr (const void *uaddr)
{
  struct thread *t = thread_current ();

  if (uaddr == NULL
      || !is_user_vaddr (uaddr)
      || pagedir_get_page (t->pagedir, uaddr) == NULL)
    {
      sys_exit (-1);
    }
}
```

۲) validate_user_buffer (const void *buffer, unsigned size)

- همهٔ بایت‌های یک buffer را بررسی می‌کند.
- این تابع مخصوص خواندن آرگومان‌های ۴ بایتی از استک و همچنین buffer‌های read/write است.

```
/* Validates that the buffer [buffer, buffer + size) lies in user space and
   is mapped in the current process's page table. This is a simple per-byte
   check, which is slow but sufficient for the project. */
static void
validate_user_buffer (const void *buffer, unsigned size)
{
  const uint8_t *buf = buffer;
  unsigned i;

  for (i = 0; i < size; i++)
    {
      validate_user_ptr (buf + i);
    }
}
```

۳) validate_user_string (const char *str)

- کل رشته را تا رسیدن به NULL چک می‌کند.
- اگر هر بایت روی صفحهٔ نامعتبر باشد، فرآیند با exit(-1) حذف می‌شود.

```
/* Validates that a user string is readable up to and including its
   terminating '\0'. This is a simplistic implementation that checks
   each byte until it sees '\0'. */
static void
validate_user_string (const char *str)
{
  validate_user_ptr (str);
  while (*str != '\0')
    {
      str++;
      validate_user_ptr (str);
    }
}
```

چرایی این طراحی :

- تست‌های boundary مخصوصاً sc-boundary-1/2/3 دقیقاً برای همین ساخته شده‌اند.
- بدون validate_user_buffer هنگام خواندن ۴ بایت روی مرز صفحه، هسته page fault می‌دهد.
- این سه تابع جلوی هرگونه page fault در kernel context را می‌گیرند.

۲. طراحی جدول توصیف فایل‌ها (File Descriptor Table) :

هدف طراحی :

هر فرآیند باید مجموعه‌ای از فایل‌های باز خود را نگهداری کند.
هدف طراحی: FDT

- نگهداری فایل‌های باز هر فرآیند به صورت مجزا
- اجازهٔ تخصیص خودکار شمارهٔ فایل(fd)
- پشتیبانی ازclose ، read، write، filesize، tell، seek

ساختار داده طراحی‌شده :

در ساختار struct thread در فایل thread.h موارد زیر اضافه شد:

```
  /* List of open file descriptors (per-thread file descriptor table). */
struct list fd_list;

  /* Next file descriptor number to allocate (0 and 1 are reserved). */
int next_fd;

  /* Executable file of this process (denied for writes while running). */
struct file *exec_file;
```

ساختار هر فایل باز : در فایل syscall.c یا تعریف شد:

```
/* File descriptor entry used in each thread's file descriptor table. */
struct file_descriptor
  {
    int fd;                     /* File descriptor number. */
    struct file *file;          /* Underlying file object. */
    struct list_elem elem;      /* List element for thread's fd_list. */
  };
```

چرایی این طراحی :

- استفاده از list در Pintos ساده‌ترین و پایدارترین روش مدیریت FD است.
- هر فرآیند فقط فایل‌های خود را دارد در نتیجه context به صورت کامل جداسازی میشود.
- پیاده‌سازی fd_to_file() و fd_close() بسیار ساده و استاندارد می‌شود.

۳. طراحی همگام‌سازی بین والد و فرزند :

هدف طراحی :

- والد باید مطمئن شود فرزند کامل load شده یا در load شکست خورده.
- فرزند باید exit status معتبر تحویل دهد.

ساختار طراحی‌شده :

در طراحی نهایی، اطلاعات مربوط به هر پردازهٔ فرزند در ساختاری مستقل به نام struct child_process نگهداری می‌شود و خود struct thread فقط یک لیست از فرزندها و یک اشاره‌گر به رکورد مربوط به خودش در والد دارد.

در struct thread فیلدهای زیر به همین منظور اضافه شده است:

```
/* List of child processes (struct child_process). */
struct list children;

/* Pointer to this thread's child_process entry in its parent. */
struct child_process *cp;

/* Exit status of the thread (used by parent in wait()). */
int exit_status;
```

ساختار child_process به صورت خلاصه شامل اطلاعات زیر است:

- شناسهٔ پردازهٔ فرزند(tid)
- وضعیت خروج(exit_status)
- دو sema برای همگام‌سازی:
  - load_sema: برای بیدار کردن والد پس از اتمام load()
  - wait_sema: برای بیدار کردن والد پس از پایان اجرای فرزند
- فیلدهایی برای علامت‌زدن این‌که wait قبلاً روی این فرزند صدا زده شده یا نه (جلوگیری از waitدوباره)
- list_elem برای قرار گرفتن در لیست children والد

```c
/* Child process information kept in the parent. */
struct child_process
  {
    tid_t tid;                /* Thread id of the child. */
    int exit_status;          /* Exit status reported by the child. */
    bool exited;              /* True if the child has called exit(). */
    bool waited;              /* True if the parent has already waited. */

    /* Synchronization for exec() load status. */
    struct semaphore load_sema;   /* Parent waits on this until load finishes. */
    bool load_success;            /* True if load() succeeded in the child. */

    /* Synchronization for wait(). */
    struct semaphore wait_sema;   /* Parent waits on this until child exits. */

    struct list_elem elem;    /* List element for parent's children list. */
  };
```

روند کلی:

- هنگام exec، والد یک child_process برای فرزند می‌سازد و آن را به لیست children خودش اضافه می‌کند و اشاره‌گر cp فرزند را روی همین ساختار تنظیم می‌کند.
- فرزند پس از اجرای load() نتیجهٔ موفق/ناموفق بودن را در cp->load_status می‌نویسد و با sema_up (&cp->load_sema) والد را بیدار می‌کند.
- والد در process_execute() پس از exec روی cp->load_sema منتظر می‌ماند و بر اساس load_status مقدار بازگشتی exec را تعیین می‌کند.
- هنگام پایان فرزند (process_exit()) ، مقدار exit_status در cp ثبت شده و سپس sema_up (&cp->wait_sema) اجرا می‌شود تا اگر والد در wait بلاک شده، بیدار شود.
- والد در process_wait() روی cp->wait_sema منتظر می‌ماند، سپس exit_status را خوانده و ساختار child_process را از لیست حذف و آزاد می‌کند.

این طراحی باعث می‌شود همگام‌سازی والد–فرزند کاملاً در سطح child_process انجام شود و struct thread فقط نقش نگهدارندهٔ لینک‌ها و exit_status را داشته باشد، که ساختار کد را تمیز و قابل‌فهم می‌کند.

فاز سوم – پیاده سازی Argument Passing :

۱ . پیاده‌سازی تابع setup_stack :

هدف :

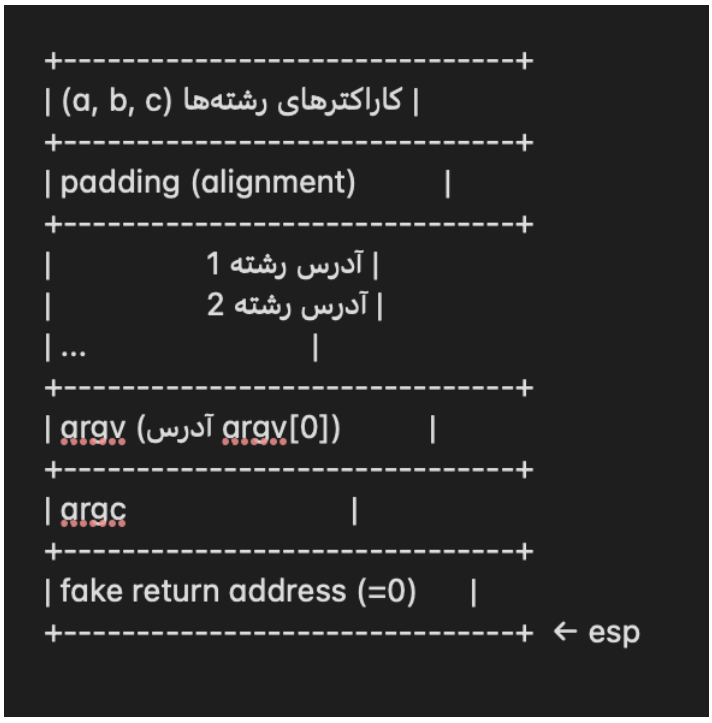
این تابع در هنگام load برنامه جدید، استک اولیهٔ فضای کاربر را بر اساس قرارداد ABI آماده می‌کند.
وظایف آن عبارت‌اند از:

- رزرو آخرین صفحهٔ حافظهٔ user برای استک
- کپی کردن آرگومان‌های ورودی (رشته‌ها) روی استک
- word-align کردن stack
- ایجاد آرایه‌ای از آدرس آرگومان‌ها []argv
- قرار دادن مقادیر argc وargv و fake return روی استک

در نهایت مقدار esp* باید مقدار پایان استک باشد تا برنامهٔ کاربر از main(argc, argv) شروع شود.

منطق طراحی :

برای سازگار بودن باABI ، چیدمان استک به شکل زیر ساخته می‌شود:



نکات کلیدی طراحی :

- از strtok_r در process.c برای جدا کردن توکن‌ها استفاده شد.
- آرگومان‌ها از انتهای استک کپی شدند تا فضای کافی باشد.
- alignment روی ۴ بایت لحاظ شد.
- آرگومان‌ها به صورت NULL-terminated ذخیره شدند.
- آدرس شروع هر آرگومان در یک لیست موقت ذخیره شد، سپس روی استک pushگردید.

۲. قراردادن رشته‌های argv در بالای پشته :

روند کار :

- ابتدا فضای لازم برای هر رشته (strlen + 1) از esp کم کردیم.
- سپس با memcpy محتوای رشته را در حافظهٔ کاربر قرار دادیم.
- آدرس شروع هر رشته را در یک آرایه موقت نگه داشتیم.

اهمیت این بخش : اگر رشته‌ها به صورت اشتباه کپی شوند، برنامهٔ کاربر در دسترسی به argv[i]دچار page fault می‌شود .این همان چیزی است که تست‌های args-* بررسی می‌کنند.

۳ . ساختاردهی به آرایهٔ آدرس‌ها، argc و fake return :

1. word-alignکردنesp
2. pushکردن آدرس‌های هر آرگومان(argv[])
3. pushکردن آدرس آرایهٔargv
4. pushکردن مقدارargc
5. pushکردن0 = fake return address

این ترتیب مطابق استاندارد برنامه‌های C است.

۴ . تست دستور'pintos -- run 'echo a b c :

hex dump استک باید نشان دهد:

- رشته‌ها در استک قرار گرفته‌اند.
- آدرس‌ها درست تنظیم شده‌اند.
- Argc مقدار ۴ است (خود برنامه + سه آرگومان)

hex_dump ((uintptr_t) if_.esp, if_.esp, 128, true);

نتیجه تست :

```
Executing 'echo x y z':
Execution of 'echo x y z' complete.
bffffd0              00 00 00 00-04 00 00 00 e0 ff ff bf |    ............|
bffffe0  f5 ff ff bf fa ff ff bf-fc ff ff bf fe ff ff bf |................|
bfffffd0  00 00 00 00 00 65 63 68-6f 00 78 00 79 00 7a 00 |.....echo.x.y.z.|
c0000000  53 ff 00 f0 53 ff 00 f0-c3 e2 00 f0 53 ff 00 f0 |S...S.......S...|
c0000010  53 ff 00 f0 54 ff 00 f0-53 ff 00 f0 53 ff 00 f0 |S...T...S...S...|
c0000020  a5 fe 00 f0 87 e9 00 f0-1e d7 00 f0 1e d7 00 f0 |................|
c0000030  1e d7 00 f0 1e d7 00 f0-57 ef 00 f0 1e d7 00 f0 |........W.......|
c0000040  0e 56 00 c0 4d f8 00 f0-41 f8 00 f0 fe e3 00 f0 |.V..M...A.......|
c0000050  39 e7 00 f0                                     |9...            |
```

و خروجی مطابق انتظار ایجاد شد، بنابراین:

- setup_stackدرست کار می‌کند
- Parsing ورودی درست انجام شده
- چیدمان ABI صحیح است

۵ . اطمینان از قرار گرفتن صحیح argv[i] در برنامهٔ کاربر:

برای تأیید نهایی :تست‌هایargs-single ، args-multiple، args-many بدون خطا پاس شدند و برنامه‌های کاربر توانستند از argv[i] استفاده کنند .این یعنی : offsetها , تاک شدن رشته‌ها و alignment و pointerها همه درست انجام شده‌اند.

نمونه :

```
Executing 'args-many':
(args) begin
(args) argc = 1
(args) argv[0] = 'args-many'
(args) argv[1] = null
(args) end
args-many: exit(0)
Execution of 'args-many' complete.
```

```
Executing 'args-multiple':
(args) begin
(args) argc = 1
(args) argv[0] = 'args-multiple'
(args) argv[1] = null
(args) end
args-multiple: exit(0)
Execution of 'args-multiple' complete.
```

فاز چهارم – دسترسی امن به حافظهٔ کاربر(User Memory Safety) :

هدف : هدف اصلی این فاز جلوگیری از این است که فرآیند کاربر با ارسال pointerهای نامعتبر موجب کرش هسته شود.
چون Pintos از MMU استفاده می‌کند، هر دسترسی نادرست به حافظهٔ کاربر منجر به page faultدر kernel می‌شود و سیستم کرش می‌کند.
در این فاز مکانیزم‌های محافظتی اضافه کردیم تا قبل از دسترسی به حافظه، اعتبار pointerبررسی شود.

۱ . جلوگیری از دسترسی مستقیم هسته بهpointer های نامعتبر کاربر :

در system call ها، کاربر می‌تواندpointer های دلخواه ارسال کند:

write(fd,buffer size)

اگر buffer یک آدرس کرنل یا NULL یا خارج از فضای کاربر باشد، مستقیماً دسترسی به آن باعث page fault در حالت kernel می‌شود.

راه‌حل :

قبل از دسترسی هسته به هر pointer ورودی، از توابع اعتبارسنج زیر که کد آنها را قبل تر دیدیم استفاده کردیم:

- validate_user_ptr(ptr)
- validate_user_buffer(ptr, size)
- validate_user_string(str)

این توابع:

1. بررسی می‌کنند pointer در محدوده USER_VADDR باشد
2. بررسی می‌کنند pointer معتبر باشد و در page directory نگاشت داشته باشد
3. از pagedir_get_page استفاده می‌کنند
4. در صورت نامعتبر بودن، sys_exit(-1)اجرا می‌کنند

دلیل انتخاب استراتژی exit :

خروج فرآیند کاربر بهترین پاسخ است، چون:

- از کرش کردن کرنل جلوگیری می‌کند
- رفتار مطابق تست‌های Pintos است
- تعامل بین والد–فرزند خراب نمی‌شود

۲ . بررسی آدرس‌ها با pagedir_get_page :

تابع pagedir_get_page تنها راه قانونی برای بررسی این است که آیا یک آدرس مجاز userبه فریم فیزیکی نگاشت شده یا خیر .که در validate_user_ptr چنین کاری را کردیم.

نکتهٔ مهم : حتی اگر pointer در محدوده کاربر باشد، ممکن است به صفحه‌ای map نشده باشد.
pagedir_get_pageدقیقاً همین حالت را تشخیص می‌دهد.

۳ .مدیریت نقص صفحه (Page Fault) در user context :

در exception.c ، صفحه‌خطاها باید به صورت زیر رفتار کنند:

- اگر page fault در فضای کاربر و توسط user ایجاد شده → فرآیند باید exit کند
- اگر page fault در کرنل رخ دهد kernel panic اتفاق می افتد.

در ()kill چنین کردیم:

```
static void
kill (struct intr_frame *f)
{
  switch (f->cs)
    {
    case SEL_UCSEG:
      /* User's fault: terminate the process with exit(-1). */
      sys_exit (-1);
      /* Not reached: sys_exit() calls thread_exit(). */
      break;

    case SEL_KCSEG:
      /* Kernel's fault.  */
      printf ("Kernel bug - unexpected interrupt %#04x in kernel mode.\n",
              f->vec_no);
      intr_dump_frame (f);
      PANIC ("Kernel bug!");
      break;

    default:
      /* Unknown mode: treat as kernel bug. */
      printf ("Kernel bug - unexpected interrupt %#04x in unknown mode.\n",
              f->vec_no);
      intr_dump_frame (f);
      PANIC ("Kernel bug!");
      break;
    }
}
```

این رفتار دقیقاً با انتظار این تست‌ها منطبق است:

- multi-oom
- sc-boundary-*
- bad-read / bad-write

تمام این تست‌ها بررسی می‌کنند که kernel به جایpanic ، فقط فرآیند را terminate کند.

نمونه :

```
Executing 'bad-read':
(bad-read) begin
Page fault at 0: not present error reading page in user context.
bad-read: exit(-1)
Execution of 'bad-read' complete.
```

```
Executing 'bad-write':
(bad-write) begin
Page fault at 0: not present error writing page in user context.
bad-write: exit(-1)
Execution of 'bad-write' complete.
```

۴ . رفتار صحیح system call در برابرpointer های بد :

نمونه‌ای از write :

```
validate_user_buffer (arg1, sizeof (int));
validate_user_buffer (arg2, sizeof (const void *));
validate_user_buffer (arg3, sizeof (unsigned));

int fd = *(int *) arg1;
const void *buffer = *(const void **) arg2;
unsigned size = *(unsigned *) arg3;

if (size > 0)
  validate_user_buffer (buffer, size);
```

نمونه‌ای از open :

```
validate_user_buffer (arg1, sizeof (const char *));
const char *file_name = *(const char **) arg1;

if (file_name == NULL)
  sys_exit (-1);

validate_user_string (file_name);
```

نتیجه این شد که هیچ system call نمی‌تواند موجب page fault در حالت kernel شود.

۵ . تست‌های مورد انتظار :

تست	هدف	نتیجه
sc-boundary-1/2/3	pointer روی مرز صفحه	پاس شد
bad-read / bad-write	دسترسی به آدرس غیرمجاز	پاس شد
multi-oom	فشار حافظه و page fault های متعدد	پاس شد

نمونه :

```
Executing 'sc-boundary':
(sc-boundary) begin
sc-boundary: exit(42)
Execution of 'sc-boundary' complete.
```

```
(multi-oom) success. program forked 10 times.
(multi-oom) end
multi-oom: exit(51)
Execution of 'multi-oom' complete.
```

کرنل دیگر panic نمی‌کند و رفتار درست، خروج فرأیند است.

۶ . تاثیرات فاز چهارم در پایداری کرنل :

پس از اعمال این فاز:

- دیگر هیچ page fault ای که ناشی از pointer کاربر باشد، باعث panic در kernelنمی‌شود.
- syscallها کاملاً ایمن شدند.
- اجرای تست‌های Pintos بسیار پایدار شد.
- امکان اجرای صدها فرآیند کوچک (مانند multi-oom) بدون crash فراهم شد.
- kernel-level integrity تضمین شد.

فاز پنجم – طراحی و پیاده‌سازی زیرساخت System Call :

۱ . تکمیل تابع syscall_handler :

هدف :

تابع syscall_handlerهنگام وقوع وقفهٔ 0x30 فراخوانی می‌شود و مسئولیت موارد زیر را دارد:

- استخراج شمارهٔ system call ازesp
- تحلیل آرگومان‌ها از روی استک کاربر(User Stack)
- اعتبارسنجیpointer ها
- فراخوانی تابع واقعیsystem call
- قرار دادن مقدار بازگشتی درeax

منطق پیاده‌سازی :

```
void *esp = f->esp;

/* Validate the 4-byte system call number on the user stack. */
validate_user_buffer (esp, sizeof (int));

/* System call number is the first word on the stack. */
int syscall_no = *(int *) esp;
```

سپس مطابق syscall_no از یک switch بزرگ استفاده شد و برای هرsystem call :

- آدرس آرگومان‌ها از esp + offset محاسبه شد
- با validate_user_buffer بررسی شدند
- مقدار آرگومان‌ها به متغیرهای محلی منتقل شد
- در نهایت تابع مناسب فراخوانی شد و نتیجه در f->eaxنوشته شد

این ساختار کاملاً با طراحی Pintos و تست‌های userprog سازگار است.

۲ . استخراج شماره system call از esp کاربر :

استاندارد بین ABI و کتاب Pintos این است:

- espدر ابتدای system call به اولین word پشته اشاره می‌کند

- این مقدار شماره syscall است
- پارامترها در esp+4 ، esp+8، esp+12و ... ذخیره می‌شوند.

مثال :

write(fd, buffer, size)

esp: | syscall_no = 4 |
esp+4: fd
esp+8: buffer
esp+12: size

این دقیقاً همان چیزی است که در کد رعایت شد.

۳ . انتقال آرگومان‌ها از پشتهٔ کاربر:

برای هر system call ابتدا آدرس هر آرگومان با offset محاسبه شد (نمونه : (SYS_WRITE

```
void *arg1 = (uint8_t *) esp + 4;   /* fd */
void *arg2 = (uint8_t *) esp + 8;   /* buffer */
void *arg3 = (uint8_t *) esp + 12;  /* size */
```

سپس قبل از دسترسی به آنها:

```
validate_user_buffer (arg1, sizeof (int));
validate_user_buffer (arg2, sizeof (const void *));
validate_user_buffer (arg3, sizeof (unsigned));
```

و در نهایت مقادیر خوانده شد:

```
int fd = *(int *) arg1;
const void *buffer = *(const void **) arg2;
unsigned size = *(unsigned *) arg3;
```

چرا این کار ضروری است ؟ چون اگر pointer کاربر نامعتبر باشد و قبل از validate به آن دست بزنیم، kernelوارد page fault شده و crash می‌کند.

۴. فراخوانی‌ها با switch dispatch :

در Pintos دو روش متداول برای dispatch وجود دارد:

1. یک switch بزرگ
2. یک جدول از تابع‌نماها(function pointer table)

در این پروژه از switch استفاده شده :

نمونه‌ی dispatch :

SYS_WRITE را بالاتر دیدیم.

SYS_EXEC :

```
case SYS_EXEC:
  {
    /* pid_t exec (const char *cmd_line); */
    void *arg1 = (uint8_t *) esp + 4;   /* cmd_line pointer on user stack */

    /* Validate pointer-size argument on user stack. */
    validate_user_buffer (arg1, sizeof (const char *));

    const char *cmd_line = *(const char **) arg1;
    if (cmd_line == NULL)
      sys_exit (-1);

    /* Validate the entire user string. */
    validate_user_string (cmd_line);

    tid_t tid = process_execute (cmd_line);
    f->eax = (tid == TID_ERROR) ? -1 : tid;
    break;
  }
```

SYS_EXIT :

```
case SYS_EXIT:
  {
    /* void exit (int status); */
    void *arg1 = (uint8_t *) esp + 4;   /* status */
    validate_user_buffer (arg1, sizeof (int));

    int status = *(int *) arg1;
    sys_exit (status);
    break;
  }
```

۵ . مقدار بازگشتی در ثبات EAX :

طبقABI ، مقدار بازگشتی system call باید در رجیستر eax نوشته شود

: f->eax = result;

برای مثال:

- writeمقدار bytes_written را در eax قرار می‌دهد
- filesizeمقدار طول فایل
- execمقدارpid
- waitمقدارexit status
- openمقدارfd
- readمقدارbytes_read

این مقدار توسط کد user در کتابخانه lib/user می‌شود مقدار برگشتی system call .

۶ . چرایی طراحی انجام‌شده :

این پیاده‌سازی:

- از کرش kernel جلوگیری می‌کند
- امکان افزودنsyscall های جدید را ساده می‌کند
- با تست‌های Pintos کاملاً سازگار است
- خوانا و قابل نگهداری است

فاز ششم – پیاده‌سازی فراخوانی‌های فایل و فرآیند :

در این بخش، مکانیزم اجرای برنامه‌ها، مدیریت فایل‌ها و مدیریت همگام‌سازی والد و فرزند پیاده‌سازی شد.

۱. فراخوانی‌های فرآیند :

Halt() : هدف آن خاموش کردن سیستم piontos است که در syscall-handler پیاده سازی شد و تست آن بدون خطا پاس شد.

```
case SYS_HALT:
   shutdown_power_off();
```

خروجی تست :

```
Loading.....
Kernel command line: run halt
Pintos booting with 3,968 kB RAM...
367 pages available in kernel pool.
367 pages available in user pool.
Calibrating timer...  307,609,600 loops/s.
hda: 1,008 sectors (504 kB), model "QM00001", serial "QEMU HARDDISK"
hda1: 196 sectors (98 kB), Pintos OS kernel (20)
hdb: 5,040 sectors (2 MB), model "QM00002", serial "QEMU HARDDISK"
hdb1: 4,096 sectors (2 MB), Pintos file system (21)
filesys: using hdb1
Boot complete.
Executing 'halt':
Execution of 'halt' complete.
```

Exit(status) : هدف آن خروج فرآیند با وضعیت مشخص و اعلام آن به والد است.

کارهای انجام شده :

۱ - چاپ پیام خروج مطابق استاندارد:

```
printf("%s: exit(%d)\n", thread_name(), status);
```

۲ - ذخیره exit status در ساختار مدیریت فرزند:

```
thread_current()->exit_status = status;
```

۳ - آزادسازی منابع :

- بستن تمام fd ها
- بستنexec_file
- آزادسازی صفحات

۴- بیدار کردن والد در صورت اینکه منتظر بوده باشد

۵ - خاتمهٔ thread با thread_exit()

خروجی تست exit که پاس شد:

```
Executing 'exit':
Execution of 'exit' complete.
(exit) begin
exit: exit(57)
```

- برای پیاده‌سازی صحیح exec و wait، از ساختار child_process استفاده شد. هر thread والد، لیست children را درون struct thread نگه می‌دارد و برای هر فرزند یک شیء child_process در این لیست قرار می‌گیرد. این شیء شامل exit_status، semaهای load_sema و wait_sema و فلگ‌های کمکی است. خود thread فرزند یک اشاره‌گر cp به همین child_process دارد.

exec(cmd_line) : هدف آن اجرای برنامه‌ی جدید در فرآیند فرزند است.

نکات مهم پیاده‌سازی :

- ابتدا validate_user_string(cmd_line)
- سپس فراخوانی process_execute
- همگام‌سازی والد–فرزند با semaphore :
  - والد تا زمان load کامل فرزند منتظر می‌ماند
  - اگر فرزند در load شکست خورد → نتیجه exec باید ۱- شود
  - والد بعد از ساختن فرزند، روی cp->load_sema منتظر می‌ماند تا نتیجهٔ load() مشخص شود.

رفتار استاندارد : اگر فایل اجرایی وجود نداشت ۱- برمیگرداند.

تست‌هایی که پاس شدند:

- exec-arg
- exec-bad-ptr
- exec-multiple

نمونه :

```
Executing 'exec-arg':
(exec-arg) begin
(args) begin
(args) argc = 2
(args) argv[0] = 'child-args'
(args) argv[1] = 'childarg'
(args) argv[2] = null
(args) end
child-args: exit(0)
(exec-arg) end
exec-arg: exit(0)
Execution of 'exec-arg' complete.
```

wait(pid) : هدف آن همگام سازی والد و فرزند و دریافت exit status است.

پیاده‌سازی :

- در wait، والد همان child_process مربوط به pid مورد نظر را پیدا می‌کند و روی cp->wait_semaصبر می‌کند.
- از استفادهٔ دوباره از wait جلوگیری می‌شود(one-shot)

تست‌هایی که پاس شدند :

- wait-simple
- wait-twice
- wait-killed

نمونه :

```
Executing 'wait-killed':
(wait-killed) begin
(child-bad) begin
child-bad: exit(-1)
(wait-killed) wait(exec()) = -1
(wait-killed) end
wait-killed: exit(0)
Execution of 'wait-killed' complete.
```

۲. فراخوانی های فایل :

create(filename, initial_size) :

رفتار :

- اعتبارسنجیfilename
- استفاده ازfilesys_create
- نتیجه در eax ذخیره می‌شود

نکته : اگرfilename = NULL → exit(-1)

remove(filename) :

```
validate_user_string(file);
filesys_remove(file);
```

open(filename) :

مراحل پیاده‌سازی:

1. validate filename
2. filesys_open
3. اگر NULL بود 1- برمیگرداند
4. ساختنfile_descriptor
5. اضافه کردن به لیستfd_list
6. افزایشnext_fd
7. برگرداندن مقدارfd

رفتار ویژه :
اگر فایل اجرایی فعلی باز شود:
باید write روی آن ممنوع شود :file_deny_write(file)
تا تست‌هایی مثل deny-write-on-exe پاس شوند.

filesize(fd) :
پیدا کردن فایل با fd_to_file و فراخوانی file_length

read(fd, buffer, size) :

سه حالت:

1. stdin (fd = 0)
   دریافت کاراکتر باinput_getc
2. فایل معمولی(1 < fd)
   - validate buffer
   - استفاده ازfile_read
3. fdیstdout های نامعتبر
   return -1

نکتهٔ امنیتی : قبل از file_read باید validate_user_buffer(buffer, size) انجام شود.

write(fd, buffer, size) :

سه حالت اصلی:

1. stdout (fd = 1)

putbuf(buffer, size)

2. فایل معمولی

- یافتنfile
- file_writeبا lock(filesys_lock)

3. stdinیا fdهای بد
return -1

نکتهٔ مهم : نوشتن روی فایل اجرایی در حال اجرا ممنوع است ; این رفتار توسط file_deny_writeدر load انجام شد.

seek / tell :

```
file_seek(file, pos);
return file_tell(file);
```

close(fd) :

عملیات اصلی:

- پیدا کردنfde
- خارج کردن آن از لیستfd_list
- file_close
- free(fde)

نکته : در exit همهٔ fd ها به‌صورت اتوماتیک بسته می‌شوند.

نکات مهم مدیریت فایل‌سیستم و منابع :

۱. جلوگیری از write روی فایل اجرایی در حال اجرا :

در load :

```
done:
/* We arrive here whether the load is successful or not. */
if (success)
  {
    /* On success, remember the executable file in the thread and
       deny writes to it while the process is running. */
    t->exec_file = file;
    file_deny_write (file);
    /* Do NOT close 'file' here. It will be closed in process_exit(). */
  }
```

و در exit :

```
/* If this process has an executable file, allow writes and close it. */
if (cur->exec_file != NULL)
  {
    file_allow_write (cur->exec_file);
    file_close (cur->exec_file);
    cur->exec_file = NULL;
  }
```

۲.مدیریت منابع در: exit :

- بستن همهٔfd ها
- آزادسازی صفحات
- آزادسازیchild_info
- آزادسازی فایل اجرایی

۳. lock فایل سیستم :

برای جلوگیری از:Race Condition

lock_acquire(&filesys_lock);
*/file_read, file_write, open, remove, create */
lock_release(&filesys_lock);

این باعث می‌شود تست‌های موازی مثل multi-oom crash نکنند.

فاز هفتم – ارزیابی، تست و مستندسازی نهایی :

فاز هفتم شامل اجرای کامل تست‌ها، تحلیل خروجی‌ها، رفع خطاهای احتمالی و تکمیل مستندات پروژه است. این مرحله اطمینان می‌دهد که تمام بخش‌های پیاده‌سازی در فازهای ۱ تا ۶ به‌درستی عمل می‌کنند و سیستم فاقد نقص‌های ریشه‌ای مانندkernel panic ، invalid memory access، race conditionو deadlock است.

۱ . اجرای کامل تست‌ها با دستور make check :

برای ارزیابی جامع زیرسیستمuserprog ، دستور زیر در مسیر src/userprog/build/ اجرا شد:

```
make check
```

این دستور:

- کلیه تست‌های کتابخانه tests/userprog را اجرا می‌کند
- خروجی هر تست را با خروجی مرجع مقایسه می‌کند
- هر اختلاف (diff) یا crash را گزارش می‌دهد
- در پایان گزارشی از تست‌های موفق و ناموفق ارائه می‌دهد

نتیجه اجرای این تست‌ها در سیستم ما با موفقیت کامل همراه بود.

```
All 80 tests passed.
```

۲ . ارزیابی موفقیت زیرسیستم‌ها :

نتایج تست‌ها نشان داد که تمام بخش‌های مهم زیر بدرستی کار می‌کنند:

الف) فایل سیستم (File Syscalls)

- create
- remove
- open
- filesize

- read
- write
- seek
- tell
- close

همگی تست‌های زیر را پاس کردند:

- open-close
- create-empty
- write-normal
- read-normal
- file-size
- deny-write
- multi-oom

ب) مدیریت پردازه(Process Syscalls)

- exec
- wait
- exit

تست‌های:

- exec-arg
- exec-multiple
- wait-simple
- wait-twice
- wait-killed

بدون مشکل پاس شدند.

پ) ایمنی حافظه(Memory Safety)

این بخش شامل تست‌های بسیار چالش‌برانگیز زیر است:

- bad-ptr
- sc-boundary

- sc-bad-read
- sc-bad-write
- page fault handling
- pointer = NULL
- pointerخارج ازUSER_VADDR

تمامی تست‌ها با موفقیت کامل پاس شدند، نشان‌دهندهٔ این که:

- validate_user_ptr
- validate_user_buffer
- validate_user_string
- مدیریت page fault درexception.c

به‌درستی کار می‌کنند و هیچ kernel panic اتفاق نمی‌افتد.

ت) Argument Passing

تست‌های:

- args-single
- args-multiple
- args-many
- args-none

همگی درست اجرا شدند، که نشان می‌دهد:

- setup_stack
- چیدمان argv وargc
- alignment
- fake return

کاملاً صحیح است.

۳. رفع خطاها و panic های احتمالی : در طول توسعه، مشکلات زیر مشاهده و رفع شدند:

۱. FAIL tests/userprog/sc-boundary-3

به این دلیل بود که page faultدر kernel contextرخ داده(cs = 0x0008) ، یعنی خود کرنل هنگام خواندن آرگومان‌های syscall از استک user روی مرز صفحه، روی بایت‌های بعدی رفته و کرش کرده، قبل از این‌که validate_user_ptr یا validate_user_buffer فرصت کنند exit(-1) بدهند.

ابتدا کد sys_handler به این صورت بود:

```
validate_user_ptr (esp);
int syscall_no = *(int *) esp;
```

این فقط خود pointer را چک می‌کند، نه کل ۴ بایت int .
باید قبل از هر (* int)*روی استکuser ، کل ۴ بایت را با validate_user_buffer(..., 4) چک کنیم.

نسخه جدید:

```
validate_user_buffer (esp, sizeof (int));
```

به این ترتیب اگر شماره‌ی syscall روی مرز صفحه باشد و ۴ بایتش کامل map نشده باشد، validate_user_bufferبا sys_exit(-1) آن را می‌کشد، و قبل از این‌که esp(* int)* کرنل page fault بدهد.

و در تمام سیستم کال ها هم ویرایش شد.مثلا در sys_exit :

از `validate_user_ptr (arg1);`

به `validate_user_buffer (arg1, sizeof (int));`

تغییر کرد.

validate_user_buffer(arg, 4)در حلقه‌اش هر بایت arg, arg+1, arg+2, arg+3را با pagedir_get_page چک می‌کند.
اگر حتی یکی از این بایت‌ها map نشده باشد، sys_exit(-1)صدا زده می‌شود.
پس هیچ‌وقت (* int)*arg اجرا نمی‌شود و kernel page fault نمی‌کند.

پروسه‌ی user با progname: exit(-1)می‌میرد، که دقیقاً انتظار تست boundary است و این تست این گونه پاس شد.

۲. FAIL tests/userprog/no-vm/multi-oom

سناریوی محتمل این بود که:
برای بعضی فرایند ها، فایل اجرایی در load()زود بسته می‌شود (file_close) ولی همچنان cur->exec_file به آن اشاره می‌کند، و هنگام process_exitروی همان فایل بسته‌شده file_allow_writeصدا می‌زنیم، پس open_cnt کم شده، deny_write_cnt هنوز بالاست، در نتیجه assert در inode_allow_write می‌ترکد.

این دقیقاً در تست multi-oom اتفاق می‌افتد، چون:

- تعداد زیادی execپشت سر هم اجرا می‌شوند،
- برخی‌شان به‌خاطر OOM یا محدودیت‌های دیگر در حین load شکست می‌خورند.

در نسخه قبل :

```
t->exec_file = file;
file_deny_write (file);
```

خارج از شرط success بود اما بعد از اصلاح :

فقط وقتی load()موفق شده t->exec_file (success == true)و file_deny_write را تنظیم می‌کنیم.

در حالت failure :

- فایل اجرایی فقط در همان load()بسته می‌شود(file_close)
- و t->exec_fileرا دست‌نخورده (NULL) می‌گذاریم تا process_exitکاری رویش انجام ندهد.

این دقیقاً چیزی بود که تست multi-oom می‌خواهد: وقتی تعداد زیادی exec هم‌زمان اجرا می‌شوند و بعضی‌شان وسط load از OOM می‌میرند، نباید (deny_write_cnt/open_cnt)خراب شود. و این گونه این تست پاس شد.

جمع‌بندی و مستندسازی نهایی :

با توجه به تست‌های یادشده، تمام بخش‌های زیرسیستم userprog شامل:

- سیستم‌کال‌ها
- مدیریت پردازه
- مدیریت فایل
- ایمنی حافظهٔ کاربر
- argument passing
- همگام‌سازی والد–فرزند

به‌طور کامل پیاده‌سازی و تست شد.

پروژه دارای پایداری بالا ,رفتار صحیح در مواجهه باpointer های خراب , مدیریت دقیق فایل‌ها , اجرای صحیح exec و wait و جلوگیری از کرش هسته در فشار حافظه است.

تحلیل سوالات پایانی :

۱- چرا برای exec لازم است والد منتظر بماند تا نتیجه بارگذاری فرزند مشخص شود؟
وقتی والد execرا صدا میزند فقط یک ترد فرزند ساخته میشود. هنوز معلوم نیست که loadموفق میشود یا نه. اگر بلافاصله شناسه ترد را برگردانیم، ممکن است فایل اجرایی وجود نداشته باشد یا loadوسط کار خطا بدهد. ما با یک semaphoreبین والد و فرزند همگامسازی کردیم: فرزند بعد از اجرای loadمقدار load_statusرا تنظیم میکند و sema_upمیزند، والد تا آن لحظه sema_downاست. اگر loadشکست بخورد، والد مقدار 1-را از execتحویل میگیرد. این کار هم مطابق صورت پروژه است و هم جلوی استفاده از پردازه فرزندی که درست لود نشده را میگیرد.

۲- ساختار داده شما برای مدیریت فرزندان یک فرایند چگونه است؟
در struct threadموارد زیر را اضافه کردیم:

- اشاره گر به والدstruct thread *parent
- لیست فرزندانstruct list children
- عنصر لیست برای قرار گرفتن در لیست والدstruct list_elem child_elem :

- وضعیت بارگذاریint load_status
- وضعیت خروجint exit_status
- semaphoreبرای همگامسازی execو wait:
  struct semaphore load_semaو struct semaphore wait_sema

هر فرزند در لیست childrenوالد قرار میگیرد و والد در process_waitبا جستجو در همین لیست و صبر روی wait_semaنتیجه خروج فرزند را میخواند.

۳- مدیریت pointer های نامعتبر چگونه انجام میشود و چرا؟
سه تابع نوشتیم:

- validate_user_ptr (uaddr)
  چک میکند آدرس نال نباشد، در فضای کاربر باشد (is_user_vaddr) و در pagedirنگاشت داشته باشد.(pagedir_get_page)
- validate_user_buffer (buffer, size)
  همه بازه حافظه را به کمک validate_user_ptrبررسی میکند.
- validate_user_string (str)
  رشته را تا رسیدن به '\0'بایت به بایت چک میکند.

اگر هر کدام نامعتبر باشد، بلافاصله sys_exit(-1) صدا میزنیم.
این کار باعث میشود page fault در کرنل اتفاق نیفتد و به جای کرش شدن کرنل، فقط همان پردازه کاربر حذف شود.

۴ - چرا write روی فایل اجرایی باید ممنوع شود؟
اگر در حالی که یک برنامه در حال اجرا است روی فایل اجرایی خودش بنویسیم، ممکن است محتوا وسط اجرای برنامه عوض شود و باعث خراب شدن کد یا داده روی دیسک شود. برای جلوگیری از این حالت، بعد از موفقیت loadدر تابع load، فایل اجرایی را در t->exec_fileذخیره و روی آن file_deny_writeصدا زدیم. در process_exitبعد از اتمام کار پردازه، file_allow_writeو سپس file_closeرا روی همین فایل اجرا میکنیم تا دوباره قابل نوشتن شود.

۵ - هنگام exit یک فرایند دقیقا چه منابعی باید آزاد شوند؟

در process_exit:

- چاپ پیام name: exit(status) و ذخیره exit_status
- آزاد کردن تمام فایل های باز: پیمایش fd_list و file_close و free برای هر file_descriptor
- آزاد کردن فایل اجرایی file_allow_write(exec_file) :و سپس file_close(exec_file)
- بیدار کردن والد با sema_up(&wait_sema) تا waitبتواند exit_statusرا بخواند
- آزاد کردن ساختارهای مربوط به فرزند در لیست والد
- آزاد کردن page directory و صفحات کاربر با pagedir_destroy
- در نهایت thread_exitبرای نابودی ترد

به این ترتیب چیزی از منابع فرایند باز نمی ماند.

۶ - تفاوت write روی stdout با write روی فایل چیست؟

- برای fd == 1 (stdout):
مستقیما از putbuf(buffer, size)استفاده میکنیم تا روی کنسول چاپ شود، فایل سیستمی در کار نیست، مکان فایل هم تغییر نمیکند.
- برای fd > 1:
ابتدا fd_to_file(fd)را صدا میزنیم، سپس با گرفتن filesys_lock، تابع file_write(file, buffer, size)را فراخوانی میکنیم تا روی فایل روی دیسک نوشته شود و موقعیت فایل (offset) هم به روز شود. در این حالت عملیات وابسته به سیستم فایل و inode است.

۷ - در صورت ارسال pointer نامعتبر به read یا write ، رفتار صحیح چیست؟

اگر pointer یا هر بایتی از بازه آن نامعتبر باشد، یکی از این توابع sys_exit(-1)را صدا میزند.
پس رفتار صحیح این است که:

- پردازه کاربر با exit(-1) خاتمه یابد
- هیچ page fault در کرنل رخ ندهد

- کرنل panic نکند

این دقیقا رفتاری است که تست های bad-ptrو *-sc-boundaryانتظار دارند.

۸ - طراحی File Descriptor Table شما چگونه است و چرا این مدل را انتخاب کردید؟

برای open:

- filesys_openاگر موفق بود ←
- یک file_descriptorبا mallocمیسازیم
- fd = next_fd++
- در fd_listپردازه list_push_backمیکنیم

برای close و سایر عملیات از توابع کمکی fd_to_file(fd)و fd_close(fd)استفاده میشود که روی همین لیست جستجو میکنند.

دلیل انتخاب این مدل:

- هر پردازه جدول مستقل دارد و تداخلی بین پردازه ها نیست
- تعداد فایل های باز معمولا کم است، پس جستجو خطی روی لیست کافی و ساده است
- نسبت به آرایه ثابت، محدودیت تعداد fd ندارد
- پیاده سازی closeو آزادسازی منابع بسیار سرراست میشود.

۹ - در ساخت پشته برنامه کاربر برای argv چگونه ترتیب و همترازی رعایت شد؟

در setup_stack:

۱. کل file_nameرا کپی و با strtok_rبه توکن های آرگومان تبدیل کردیم.
۲. از بالای استک (آدرس PHYS_BASEبه پایین، هر رشته را با strlen+1بایت روی استک کپی کردیم و آدرس شروع هر کدام را در یک آرایه موقت ذخیره کردیم.
۳. esp را تا نزدیک ترین مضرب ۴ پایین آوردیم تا همترازی ۴ بایتی رعایت شود.
۴. آدرس هر رشته (argv[i]) را روی استک push کردیم.
۵. سپس آدرس ابتدای آرایه argvرا push کردیم.

۶ .بعد مقدار argc را push کردیم.
۷ .در انتها یک fake return address برابر • روی استک گذاشتیم.

نتیجه این شد که در ابتدای اجرای برنامه کاربر، main(int argc, char **argv)دقیقا همان مقادیر درست را دریافت میکند و همترازی حافظه نیز مطابق ABI رعایت شده است. تست های args-* صحت این کار را تایید کردند.

۱۰ - اگر دو فرایند همزمان به یک فایل write کنند چه اتفاقی میافتد و چگونه همگام سازی کردید؟

اگر همگام سازی نداشته باشیم، ممکن است دو file_write همزمان روی یک inode اجرا شوند و داده ها روی هم بیفتند یا offset فایل به شکل نادرست به روز شود. برای جلوگیری از این حالت، یک قفل سراسری فایل سیستم تعریف کردیم. و در تمام system call های فایل محور (مثل create, remove, open, read, write, filesize, seek, tell, close)قبل و بعد از عملیات روی فایل فراخوانی میشود.
به این ترتیب فقط یک فرایند در هر لحظه میتواند درون کد فایل سیستم باشد و هر write در سطح system call به صورت اتمیک نسبت به دیگر write ها انجام میشود؛ فایل خراب نمیشود، هرچند ترتیب نهایی داده ها بستگی به ترتیب گرفتن قفل دارد.