OS final project document

400521171 – 400521054

رفع مشکلات هنگام اجرای make qemu-nox:

https://stackoverflow.com/questions/56507764/error-couldnt-find-a-working-qemu-executable

https://stackoverflow.com/questions/43335499/unable-to-locate-package-libvirt-clients-error-on-ubuntu

نکات کلی:

- **تمام توابع و** struct **های تعریف شده، در فایل** declare defs.h **شوند.**
- **برای استفاده از** type **های تعریف شده نیاز به** include **کردن** types.h **است.**
- **برای خروج از ترمینال بعد از اجرای** qemu-nox: ‘CTRL + A’ followed by ‘X’
- **برای اضافه کردن** system call **ها، در فایل** syscall.h **یک** id **برای آنها در نظر گرفته شود و در فایل** declaration syscall.c **های آنها اضافه شود،** definition **های آنها باید در فایل** sysproc.c **اضافه شوند.**
- **برای اضافه کردن توابع سیتمی و فضای کاربری، به فایل** definition user.h **ها اضافه می شوند.**
- **برای اضافه کردن فایل های تست، از** forktest **الهام گرفته شده است.**

- در توابع sys_x ابتدا argument ها retrieve می شوند و validity آنها چک می شود، سپس تابع x با آرگومان های مورد نظر صدا زده می شود.

مراحل پیاده سازی:

Clone :

```
int
clone(void(*fcn)(void*,void*), void *arg1, void *arg2, void* stack)
{
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process data to the new thread
```

این تابع `clone` در سیستم عامل xv6 برای ایجاد (kernel threads) استفاده می‌شود.

1. ابتدا، یک ساختار جدید برای thread جدید ایجاد می‌شود با فراخوانی تابع `allocproc`() که یک پردازه جدید از جدول پردازه‌ها (`ptable`) به طور دینامیک جلب می‌کند.

2. سپس اطلاعات مربوط به process اصلی (parent) به نخ جدید کپی می‌شود. این اطلاعات شامل فضای آدرس (`pgdir`)، اندازه (`sz`)، (`parent`) و ساختار فرکانس (`tf`) است.

3. در این مرحله، آرگومان‌های تابع (`fcn`) به عنوان آرگومان‌های ورودی برای thread جدید درون استک قرار می‌گیرند.

4. اطلاعات مربوط به فایل‌ها و دایرکتوری‌ها نیز کپی می‌شوند تا thread جدید از همان فایل‌ها و دایرکتوری‌ها استفاده کند.

5. پردازه جدید به وضعیت `RUNNABLE` تنظیم می‌شود و سپس به جدول پردازه اضافه می‌شود. این مرحله تضمین می‌کند که thread جدید به زمان اجرای آینده تعلق گیرد.

6. در نهایت، pid جدید به عنوان نتیجه تابع برگشت داده می‌شود تا از طریق این شناسه بتوان به thread جدید متصل شد و با آن ارتباط برقرار کرد.

: Join

```
int
join(void** stack)
{
  struct proc *p;
  int havekids, pid;
  struct proc *cp = myproc();
  acquire(&ptable.lock);
  for(;;){
    // Scan through table looking for zombie children.
    havekids = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {

      // Check if this is a child thread (parent or shared address sp
      if(p->parent != cp || p->pgdir != p->parent->pgdir)
```

تابع `join` برای ایجاد انتظار (kernel threads) استفاده می‌شود. در این توابع:

- ابتدا، اطلاعات مربوط به process جاری (فرزند) با استفاده از تابع `myproc()` گرفته می‌شود.

- سپس با (`ptable.lock`)، یک حلقه برای جستجوی فرزندانی که در وضعیت `ZOMBIE` قرار دارند ایجاد می‌شود.

- در داخل حلقه، برای هر process در جدول ، بررسی می‌شود که آیا این process یک فرزند از process جاری است و آیا در وضعیت `ZOMBIE` قرار دارد.

- اگر یک فرزند باشد که در وضعیت `ZOMBIE` است، اقدامات لازم برای حذف اطلاعات مربوط به آن فرزند از جدول انجام می‌شود.

- در صورتی که هیچ فرزندی در وضعیت `ZOMBIE` نباشد، اما حداقل یک فرزند وجود داشته باشد یا پردازه جاری (`killed`) باشد، از حلقه خارج شده و -1 به عنوان نتیجه تابع برگشت داده می‌شود.

- در صورتی که هیچ فرزندی در وضعیت `ZOMBIE` نباشد و هیچ فرزندی نیز وجود نداشته باشد، thread جاری به وضعیت `SLEEPING` می‌رود و قفل آزاد می‌شود تا بقیه پردازه‌ها فرصت اجرا داشته باشند.

این تابع به thread جاری این امکان را می‌دهد که منتظر اتمام یکی از فرزندانش باشد. وقتی یک فرزند در وضعیت `ZOMBIE` قرار بگیرد، از حالت `SLEEPING` خارج شده و ادامه اجرای برنامه امکان‌پذیر خواهد بود.

```
int
sys_clone(void)
{
  int fcn, arg1, arg2, stack;
  if(argint(0, &fcn)<0 || argint(1, &arg1)<0 || argint(2, &arg2)<0 || argint(3, &stack)<0)
    return -1;
  return clone((void *)fcn, (void *)arg1, (void *)arg2, (void *)stack);
}

int
sys_join(void)
{
  void **stack;
  int stackArg;
  stackArg = argint(0, &stackArg);
  stack = (void**) stackArg;
  return join(stack);
}
```

این دو تابع `sys_clone` و `sys_join` به ترتیب تابع `clone` و `join` را از فضای کاربری به فضای کرنل انتقال می‌دهند.

```c
int thread_create(void (*worker)(void *, void *), void *arg1, void *arg2)
{
  void *stack;
  stack = malloc(PGSIZE);

  return clone(worker, arg1, arg2, stack);
}

int thread_join()
{
  void *stackPtr;
  int x = join(&stackPtr);
  return x;
}
```

در اینجا دو تابع `thread_create` و `thread_join` که به ترتیب تابع `clone` و `join` را از فضای کاربری فراخوانی می‌کنند، ارائه شده‌اند:

1. **`thread_create`**:

- این تابع یک thread کرنل جدید ایجاد می‌کند.

- ابتدا با استفاده از تابع `malloc` یک فضای حافظه به اندازه یک صفحه (`PGSIZE`) برای استفاده به عنوان استک thread ایجاد می‌کند.

- سپس تابع `clone` را با استفاده از تابع `start_routine` و آرگومان‌های `arg1` و `arg2` فراخوانی می‌کند و ایدی thread جدید ایجاد شده را به عنوان نتیجه تابع برگشت می‌دهد.

2. **`thread_join`**:

- این تابع برای انتظار اتمام یک thread کرنلی فراخوانی می‌شود.

- از تابع `join` برای ایجاد اتصال به thread در حالت `ZOMBIE` و به دست آوردن آدرس استک آن استفاده می‌کند.

- نتیجه تابع `join` (ایدی thread فراخوانی شده) به عنوان خروجی تابع برگشت داده می‌شود.

```c
int lock_init(lock_t *lk)
{
  lk->flag = 0;
  return 0;
}

void lock_acquire(lock_t *lk)
{
  // Spin until the lock is acquired
  while (xchg(&lk->flag, 1) != 0)
    ;
}

void lock_release(lock_t *lk)
{
  xchg(&lk->flag, 0);
}
```

در اینجا سه تابع مربوط به مدیریت قفل‌ها (locks) ارائه شده‌اند:

1. **`lock_init`:**

- این تابع برای مقداردهی اولیه یک قفل استفاده می‌شود.

- مقدار `flag` قفل را به صفر تنظیم می‌کند که نشان‌دهنده این است که قفل در حال حاضر در دسترس و آزاد است.

- هیچ خطایی گزارش نمی‌شود و تابع همواره مقدار 0 را برمی‌گرداند.

2. **`lock_acquire`:**

- این تابع برای گرفتن (بستن) یک قفل استفاده می‌شود.

- این تابع از یک حلقه بی‌نهایت استفاده می‌کند تا در صورتی که قفل قبلاً در دسترس نبوده باشد، منتظر تا زمانی که قفل در دسترس شود (مقدار `flag` برابر با 0 باشد)، بماند.

- برای انجام این کار از تابع `xchg` (exchange) استفاده شده است که مقدار `flag` را با مقدار 1 جایگزین می‌کند و مقدار قبلی را برمی‌گرداند. حلقه تا زمانی که مقدار قبلی صفر نباشد (یعنی قفل در دسترس نباشد) ادامه پیدا می‌کند.

3. **`lock_release`:**

- این تابع برای آزاد کردن (باز کردن) یک قفل استفاده می‌شود.

- با استفاده از تابع `xchg`، مقدار `flag` را با صفر جایگزین می‌کند که نشان‌دهنده آزاد بودن قفل است.