به نام خدا

گزارش Lab4 سیستم عامل پیشرفته

استاد:

دکتر جوادی

دانشجویان:

سید علیرضا حسینی – ۴۰۱۱۳۱۰۰۵

امیررضا زارع – ۴۰۱۱۳۱۰۰۸

فهرست مطالب

مقدمه

در این آزمایش در بخش A، پشتیبانی چند پردازنده را به JOS اضافه می‌کنیم، برنامه‌ریزی دوره‌ای را پیاده‌سازی می‌کنیم و فراخوانی‌های سیستم مدیریت محیط اولیه را اضافه می‌کنیم (فراخوانی‌هایی که محیط‌ها را ایجاد می‌کنند و از بین می‌برند و حافظه را تخصیص می‌دهند). در بخش B، ما یک Fork() شبیه یونیکس را پیاده‌سازی خواهیم کرد که به یک محیط حالت کاربر اجازه می‌دهد تا کپی‌هایی از خودش ایجاد کند. در نهایت، در بخش C، پشتیبانی از ارتباطات بین فرآیندی (IPC)[1] را اضافه می‌کنیم که به محیط‌های حالت کاربری مختلف اجازه می‌دهد به طور صریح با یکدیگر ارتباط برقرار کرده و همگام شوند. همچنین پشتیبانی از وقفه‌های ساعت سخت‌افزاری را اضافه می‌کنیم. این آزمایش شامل تعدادی فایل جدید است که در شکل زیر مشاهده می‌کنید:

`kern/cpu.h`	Kernel-private definitions for multiprocessor support
`kern/mpconfig.c`	Code to read the multiprocessor configuration
`kern/lapic.c`	Kernel code driving the local APIC unit in each processor
`kern/mpentry.S`	Assembly-language entry code for non-boot CPUs
`kern/spinlock.h`	Kernel-private definitions for spin locks, including the big kernel lock
`kern/spinlock.c`	Kernel code implementing spin locks
`kern/sched.c`	Code skeleton of the scheduler that you are about to implement

بخش A: پشتیبانی از چند پردازنده و چند وظیفه‌ای مشترک

در بخش اول این آزمایش، ابتدا JOS را گسترش می‌دهیم تا روی یک سیستم چند پردازنده‌ای اجرا شود و سپس برخی از فراخوانی‌های سیستم هسته JOS جدید را پیاده‌سازی می‌کنیم تا به محیط‌های سطح کاربر اجازه ایجاد محیط‌های جدید اضافی را بدهند. ما همچنین زمانبند cooperative round-robin را پیاده‌سازی خواهیم کرد که به هسته اجازه می‌دهد تا زمانی که محیط فعلی به‌طور داوطلبانه از CPU استفاده نمی‌کند (یا کارش تمام شده است)، از یک محیط به محیط دیگر سوئیچ کند. بعداً در قسمت C، برنامه‌ریزی پیشگیرانه را پیاده‌سازی می‌کنیم، که به هسته اجازه می‌دهد تا کنترل CPU را پس از گذشت زمان معینی از یک محیط، دوباره به دست بگیرد.

بخش B: Copy-on-Write Fork

همانطور که قبلا ذکر شد، یونیکس فراخوانی سیستم fork() را به عنوان اولیه ایجاد فرآیند اولیه خود ارائه می‌کند. فراخوانی سیستم fork() فضای آدرس فرآیند فراخوانی (والد) را برای ایجاد یک فرآیند جدید (فرزند) کپی می‌کند. یونیکس fork() را با کپی کردن کل بخش داده والد در یک منطقه حافظه جدید اختصاص داده شده برای فرزند پیاده سازی می‌کند. این اساساً همان رویکردی است که dumbfork() اتخاذ می‌کند. کپی کردن آدرس والدین در فرزند گران‌ترین بخش عملیات fork() است. با این حال، فراخوانی fork() اغلب تقریباً بلافاصله با فراخوانی به exec() در فرآیند فرزند دنبال می‌شود که حافظه فرزند را با یک برنامه جدید جایگزین می‌کند. این همان کاری است که پوسته معمولا انجام می‌دهد. در این مورد، زمان صرف شده برای کپی کردن فضای آدرس والدین تا حد زیادی تلف می‌شود، زیرا پردازش فرزند قبل از فراخوانی exec() بسیار کمی از حافظه خود استفاده می‌کند. به همین دلیل، نسخه‌های بعدی یونیکس از سخت‌افزار حافظه مجازی استفاده کردند تا به والدین و فرزند اجازه دهند تا حافظه نگاشت شده را در

[1] Inter Process Comunication

فضای آدرس مربوطه خود به اشتراک بگذارند تا زمانی که یکی از فرآیندها واقعاً آن را تغییر دهد. این تکنیک به عنوان کپی در نوشتن شناخته می‌شود. برای انجام این کار، در fork() هسته نگاشت فضای آدرس را به جای محتویات صفحات نگاشت شده از والد به فرزند کپی می‌کند و در همان زمان صفحاتی که اکنون به اشتراک گذاشته شده‌اند فقط خواندنی هستند. هنگامی که یکی از دو فرآیند سعی می‌کند در یکی از این صفحات اشتراک گذاری شده بنویسد، فرآیند یک خطای صفحه را می‌گیرد. در این مرحله، هسته یونیکس متوجه می‌شود که صفحه واقعاً یک کپی «مجازی» یا «کپی در نوشتن» بوده است و بنابراین یک کپی جدید و خصوصی از صفحه برای فرآیند خطا ایجاد می‌کند. به این ترتیب، محتویات صفحات منفرد تا زمانی که واقعاً روی آنها نوشته نشده باشد، در واقع کپی نمی‌شوند. این بهینه‌سازی یک fork() به دنبال آن یک exec() را در فرزند بسیار ارزان‌تر می‌کند.

بخش C: چندوظیفه پیشگیرانه و ارتباطات بین فرآیندی (IPC)

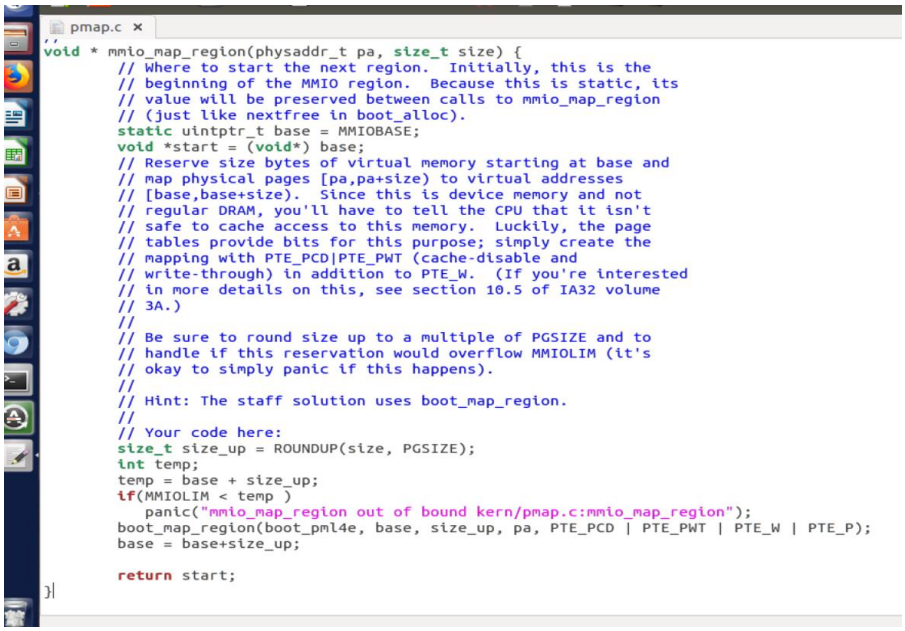
در بخش پایانی آزمایشگاه ۴، هسته را طوری تغییر می‌دهیم که از محیط‌های غیرهمکار جلوگیری کند و به محیط‌ها اجازه دهید پیام‌ها را به طور صریح به یکدیگر منتقل کنند.

تمرین ۱:

Exercise 1. Implement `mmio_map_region` in `kern/pmap.c`. To see how this is used, look at the beginning of `lapic_init` in `kern/lapic.c`. You'll have to do the next exercise, too, before the tests for `mmio_map_region` will run.

پیاده‌سازی تابعی مانند mmio_map_region همانطور که توضیح داده شد معمولاً شامل نوشتن کدی است که برای نگاشت ناحیه‌ای از حافظه که برای I/O با حافظه (MMIO) استفاده می‌شود، استفاده می‌شود. در سیستم عامل‌ها، MMIO روشی برای انجام ورودی/خروجی (I/O) بین CPU و دستگاه‌های جانبی در همان فضای آدرس حافظه برنامه است.

در MMIO، بخشی از حافظه فیزیکی به رجیسترهای برخی از دستگاه‌های ورودی/خروجی متصل می‌شود، بنابراین همان دستورالعمل‌های بارگیری/ذخیره‌ای که معمولا برای دسترسی به حافظه استفاده می‌شود، می‌تواند برای دسترسی به ثبات‌های دستگاه استفاده شود . به این منظور یک تابع برای تخصیص فضا و اختصاص دادن این فضا به MMIO پیاده‌سازی کردیم.



```
pmap.c ×
,,
void * mmio_map_region(physaddr_t pa, size_t size) {
        // Where to start the next region.  Initially, this is the
        // beginning of the MMIO region.  Because this is static, its
        // value will be preserved between calls to mmio_map_region
        // (just like nextfree in boot_alloc).
        static uintptr_t base = MMIOBASE;
        void *start = (void*) base;
        // Reserve size bytes of virtual memory starting at base and
        // map physical pages [pa,pa+size) to virtual addresses
        // [base,base+size).  Since this is device memory and not
        // regular DRAM, you'll have to tell the CPU that it isn't
        // safe to cache access to this memory.  Luckily, the page
        // tables provide bits for this purpose; simply create the
        // mapping with PTE_PCD|PTE_PWT (cache-disable and
        // write-through) in addition to PTE_W.  (If you're interested
        // in more details on this, see section 10.5 of IA32 volume
        // 3A.)
        //
        // Be sure to round size up to a multiple of PGSIZE and to
        // handle if this reservation would overflow MMIOLIM (it's
        // okay to simply panic if this happens).
        //
        // Hint: The staff solution uses boot_map_region.
        //
        // Your code here:
        size_t size_up = ROUNDUP(size, PGSIZE);
        int temp;
        temp = base + size_up;
        if(MMIOLIM < temp )
           panic("mmio_map_region out of bound kern/pmap.c:mmio_map_region");
        boot_map_region(boot_pml4e, base, size_up, pa, PTE_PCD | PTE_PWT | PTE_W | PTE_P);
        base = base+size_up;

        return start;
}
```

تمرین ۲:

Exercise 2. Read `boot_aps()` and `mp_main()` in `kern/init.c`, and the assembly code in `kern/mpentry.S`. Make sure you understand the control flow transfer during the bootstrap of APs. Then modify your implementation of `page_init()` in `kern/pmap.c` to avoid adding the page at `MPENTRY_PADDR` to the free list, so that we can safely copy and run AP bootstrap code at that physical address. Your code should pass the updated `page_check_free_list()` test (but might fail the `check_boot_pml4e()` test, which you will fix soon).

سوال ۲.۱: kern/mpentry.S را کنار هم با boot/boot.S مقایسه کنید. با در نظر گرفتن اینکه kern/mpentry.S برای اجرا در بالای KERNBASE درست مانند هر چیز دیگری در هسته کامپایل و پیوند داده شده است، هدف ماکرو MPBOOTPHYS چیست؟ چرا در kern/mpentry.S لازم است اما در boot/boot.S لازم نیست؟ به عبارت دیگر، اگر در kern/mpentry.S حذف شود، چه مشکلی وجود دارد؟

پاسخ:

از آنجایی که entry.S به داشتن آدرس‌های بالای KERNBASE پیوند داده شده است، حاوی ماکرو

#define RELOC(x) ((x) - KERNBASE) است تا آدرس‌های تولید شده توسط پیوند دهنده (که آدرس‌های مجازی هستند) را به فیزیکی ترجمه کند. آدرس‌ها (آدرس‌های بارگذاری)، جایی که داده‌ها در واقع ذخیره می‌شوند. برای مثال

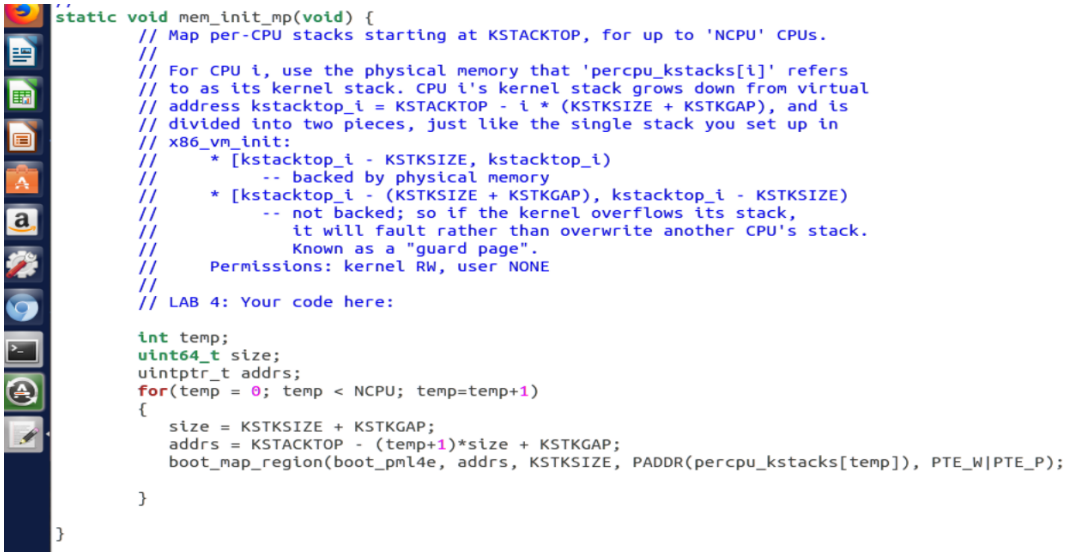
movl $(RELOC(entry_pgdir)), %eax برای استفاده از ماکرو RELOC مورد نیاز است زیرا نمی‌خواهیم آدرس مجازی enter_pgdir را در eax بارگیری کنیم، بلکه آدرس فیزیکی (بارگذاری شده) را بارگذاری کنیم. بنابراین این ماکرو RELOC برای محاسبه آدرس‌های فیزیکی داده‌های تعریف شده در داخل خود فایل مفید است. مفهوم مشابهی در mpentry.S وجود دارد

که در عوض ما باید از MPBOOTPHYS برای ارجاع به آدرس‌های داخل خود mpentry.S استفاده کنیم. توجه داشته باشید که ما همچنان از RELOC در داخل mpentry.S برای چیزهایی که خارج از mpentry.S تعریف شده اند، مانند enter_pgdir استفاده می کنیم. اگر بخواهیم آن را حذف کنیم، آدرس های مجازی دریافت می‌کنیم در حالی که CPU AP هنوز صفحه‌بندی را فعال نکرده است. این باعث می‌شود کارها شکست بخورند.

تمرین ۳:

Exercise 3. Modify `mem_init_mp()` (in `kern/pmap.c`) to map per-CPU stacks starting at `KSTACKTOP`, as shown in `inc/memlayout.h`. The size of each stack is `KSTKSIZE` bytes plus `KSTKGAP` bytes of unmapped guard pages. Your code should pass the new check in `check_boot_pml4e()()`.

تغییرات خواسته شده انجام شد.



```
static void mem_init_mp(void) {
        // Map per-CPU stacks starting at KSTACKTOP, for up to 'NCPU' CPUs.
        //
        // For CPU i, use the physical memory that 'percpu_kstacks[i]' refers
        // to as its kernel stack. CPU i's kernel stack grows down from virtual
        // address kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP), and is
        // divided into two pieces, just like the single stack you set up in
        // x86_vm_init:
        //     * [kstacktop_i - KSTKSIZE, kstacktop_i)
        //          -- backed by physical memory
        //     * [kstacktop_i - (KSTKSIZE + KSTKGAP), kstacktop_i - KSTKSIZE)
        //          -- not backed; so if the kernel overflows its stack,
        //             it will fault rather than overwrite another CPU's stack.
        //             Known as a "guard page".
        //     Permissions: kernel RW, user NONE
        //
        // LAB 4: Your code here:

        int temp;
        uint64_t size;
        uintptr_t addrs;
        for(temp = 0; temp < NCPU; temp=temp+1)
        {
           size = KSTKSIZE + KSTKGAP;
           addrs = KSTACKTOP - (temp+1)*size + KSTKGAP;
           boot_map_region(boot_pml4e, addrs, KSTKSIZE, PADDR(percpu_kstacks[temp]), PTE_W|PTE_P);

        }

}
```

تمرین ۴:

Exercise 4. The code in `trap_init_percpu()` (`kern/trap.c`) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs. Change the code so that it can work on all CPUs. (Note: your new code should not use the global `ts` variable any more.)

حال با توجه به این که در Labهای قبلی فقط به پیاده‌سازی تک پردازنده پرداخته بودیم، تابع percpu_init_trap() به درستی کار نمی‌کند و برای این کار این تابع نیز باید دچار تغییر شود. برای جلوگیری از condition race ابتدا یک قفل گلوبال باید درنظر بگیریم که به این منظور برای پیاده‌سازی از تابع kernel_lock (void) و برای آزادسازی قفل از kernel_unlock (void) استفاده می‌کنیم. تغییرات انجام شد و تمام.

تمرین ۵:

Exercise 5. Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

سوال ۵.۱: به نظر می‌رسد که استفاده از قفل هسته بزرگ تضمین می‌کند که تنها یک CPU می‌تواند کد هسته را در یک زمان اجرا کند. چرا هنوز برای هر CPU به پشته‌های کرنل جداگانه نیاز داریم؟ سناریویی را توصیف کنید که در آن استفاده از پشته هسته مشترک، حتی با محافظت از قفل هسته بزرگ، اشتباه می‌کند.

پاسخ:

هنگامی که یک استثنا یا وقفه ایجاد می‌شود، قبل از اینکه کنترل‌کننده تله قفل هسته بزرگ را بدست آورد، هسته برخی از اطلاعات را به استک پوش می‌کند (در اینجا TrapFrame است). بنابراین CPU های دیگر می‌توانند یک TrapFrame را هنگامی که یک CPU در حالت هسته است پوش کنند. اگر همه CPUها یک پشته هسته مشترک داشته باشند، ممکن است چیزی شبیه به این اتفاق بیفتد. CPU0 در حال مدیریت یک تله است، در این زمان، یک تله در CPU1 فعال می‌شود و TrapFrame خود را روی پشته پوش می‌کند. وقتی CPU0 کار را انجام داد، پشته‌ای را که CPU1 پوش می‌کند، به جای پشته‌ی خودش باز می‌کند. در این شرایط، پشته هسته خراب می‌شود و جریان کنترل آنطور که انتظار می‌رود منتقل نمی‌شود.

تمرین ۶:

Exercise 6. Implement round-robin scheduling in `sched_yield()` as described above. Don't forget to modify `syscall()` to dispatch `sys_yield()`.

Modify `kern/init.c` to create two (or more!) environments that all run the program `user/yield.c`. You should see the environments switch back and forth between each other five times before terminating, like this:

```
...
Hello, I am environment 00001008.
Hello, I am environment 00001009.
Hello, I am environment 0000100a.
Back in environment 00001008, iteration 0.
Back in environment 00001009, iteration 0.
Back in environment 0000100a, iteration 0.
```

```
Back in environment 00001008, iteration 1.
Back in environment 00001009, iteration 1.
Back in environment 0000100a, iteration 1.
...
```

After the `yield` programs exit, when only idle environments are runnable, the scheduler should invoke the JOS kernel monitor. If all this does not happen, then fix your code before proceeding.

سوال ۶.۱: در پیاده سازی env_run() باید lcr3() را فراخوانی می‌کردید. قبل و بعد از فراخوانی lcr3() ، کد شما به متغیر e ارجاع می‌دهد، پس از بارگیری رجیستر cr3٪، زمینه آدرس دهی مورد استفاده توسط MMU فوراً تغییر می‌کند. اما یک آدرس

مجازی (یعنی e) نسبت به یک زمینه آدرس معین معنی دارد. چرا اشاره گر e هم قبل و هم بعد از سوئیچ آدرس دهی قابل حذف است؟

پاسخ: در env_setup_vm()، کامنت می‌گوید که فضای آدرس مجازی همه محیط‌ها از UTOP تا UVPT و همچنین فضای آدرس هسته یکسان است. آدرس مجازی e همیشه در هر فضای آدرسی که باشد یکسان است، بنابراین می‌توان آن را هم قبل و هم بعد از سوئیچ آدرس دهی ارجاع داد.

سوال ۶.۲: هر زمان که هسته از یک محیط به محیط دیگر سوئیچ می‌کند، باید اطمینان حاصل کند که رجیسترهای محیط قدیمی ذخیره شده‌اند تا بتوانند بعداً به درستی بازیابی شوند. چرا؟ کجا این اتفاق می‌افتد؟

پاسخ: سوئیچ زمینه باید اطمینان حاصل کند که محیط می‌تواند اجرا را دقیقاً در جایی که متوقف می‌شود از سر بگیرد زیرا سوئیچ هرگز اتفاق نیفتاده است. بنابراین همه رجیسترها باید ذخیره شوند. هنگامی که سیستم‌کال sys_yield() را راه‌اندازی می‌کند، روی پشته پوش داده می‌شوند و سپس کنترل‌کننده دام (در اینجا kern/trap.c:trap()) آن‌ها را در env_tf ذخیره می‌کند. و زمانی که env_run() اجرا شد توسط env_pop_tf() بازیابی می‌شوند.

تمرین ۷:

Exercise 7. Implement the system calls described above in `kern/syscall.c`. You will need to use various functions in `kern/pmap.c` and `kern/env.c`, particularly `envid2env()`. For now, whenever you call `envid2env()`, pass 1 in the `checkperm` parameter. Be sure you check for any invalid system call arguments, returning `-E_INVAL` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

موارد خواسته شده را پیاده‌سازی کردیم.

تمرین ۸:

Exercise 8. Implement the `sys_env_set_pgfault_upcall` system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

برای رسیدگی به خطاهای صفحه یک محیط کاربری باید یک نقطه ورودی کنترل کننده خطای صفحه را با هسته JOS ثبت می‌کند. محیط کاربر نقطه ورود خطای صفحه خود را از طریق فراخوانی سیستم جدید upcall_pgfault_set_env_sys ثبت می‌کند. ما یک عضو جدید به ساختار Env اضافه کرده‌ایم، upcall_pgfault_env، تا این اطلاعات را ثبت کند. پیاده‌سازی انجام شد.

تمرین ۹:

Exercise 9. Implement the code in `kern/trap.c` required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

پیاده‌سازی خواسته شده انجام شد.

تمرین ۱۰:

Exercise 10. Implement the `_pgfault_upcall` routine in `lib/pfentry.s`. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the RIP.

آنچه در اینجا باید بنویسیم روش بازگشت به نقطه اصلی در کد کاربر است که باعث خطای صفحه شده است، بنابراین باید دید واضحی از نحوه انجام انتقال کنترل در سطح ماشین داشته باشیم. در اینجا از آنجایی که این یک فراخوانی معمولی نیست، ما باید قاب پشته فراخوانی کننده را طوری تغییر دهیم که eip را به فریم پشته فراخوانی کننده پوش کند. یعنی برای قرار دادن eip ذخیره شده باید پشته فراخوانی کننده را ۴ بایت بزرگ کنیم و به همین دلیل است که ما به کلمه ۳۲ بیتی خالی برای خطاهای صفحه تو در تو نیاز داریم. سپس باید به صورت دستی رجیسترهای ذخیره شده را در یک قاب پشته استثنای کاربر به عقب برگردانیم و در نهایت، کنترل را به جایی که با خطای صفحه مواجه شده است برگردانیم. موارد خواسته شده پیاده‌سازی شد و تمام.

تمرین ۱۱:

Exercise 11. Finish `set_pgfault_handler()` in `lib/pgfault.c`.

مورد خواسته شده انجام شد و تمام.

تمرین ۱۲:

Exercise 12. Implement `fork`, `duppage` and `pgfault` in `lib/fork.c`.

Test your code with the `forktree` program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.

```
1008: I am ''
1009: I am '0'
2008: I am '00'
2009: I am '000'
100a: I am '1'
3008: I am '11'
3009: I am '10'
200a: I am '110'
4008: I am '100'
100b: I am '01'
5008: I am '011'
4009: I am '010'
100c: I am '001'
100d: I am '111'
100e: I am '101'
```

موارد خواسته شده پیاده‌سازی شدند و تمام.

تمرین ۱۳:

Exercise 13. Modify `kern/trapentry.S` and `kern/trap.c` to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in `env_alloc()` in `kern/env.c` to ensure that user environments are always run with interrupts enabled.

The processor never pushes an error code or checks the Descriptor Privilege Level (DPL) of the IDT entry when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3, at this time.

After doing this exercise, if you run your kernel with any test program that runs for a non-trivial length of time (e.g., `spin`), you should see the kernel print trap frames for hardware interrupts. While interrupts are now enabled in the processor, JOS isn't yet handling them, so you should see it misattribute each interrupt to the currently running user environment and destroy it. Eventually it should run out of environments to destroy and drop into the monitor. If you see an infinite cascade of hardware interrupts, you're probably running with interrupts enabled in the kernel and you should fix this before proceeding.

این تمرین نیاز به اضافه کردن کنترل کننده وقفه بیشتری در kern/trapentry.S و kern/trap.c:trap() دارد. موارد خواسته شده انجام شد.

تمرین ۱۴:

Exercise 14. Modify the kernel's `trap_dispatch()` function so that it calls `sched_yield()` to find and run a different environment whenever a clock interrupt takes place.

You should now be able to get the `user/spin` test to work: the parent environment should fork off the child, `sys_yield()` to it a couple times but in each case regain control of the CPU after one time slice, and finally kill the child environment and terminate gracefully.

اکنون از ما خواسته شده که kern/trap.c:trap_dispatch() را برای فراخوانی sched_yield() تغییر دهیم تا وقفه ساعت را مدیریت کنیم. به سادگی یک مورد اضافه می‌کنیم که این کار را انجام می‌دهد. موارد خواسته شده را پیاده‌سازی کردیم و تمام.

تمرین ۱۵:

Exercise 15. Implement `sys_ipc_recv` and `sys_ipc_try_send` in `kern/syscall.c`. Read the comments on both before implementing them, since they have to work together. When you call `envid2env` in these routines, you should set the `checkperm` flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.

Then implement the `ipc_recv` and `ipc_send` functions in `lib/ipc.c`.

Use the `user/sendpage`, `user/pingpong`, and `user/primes` functions to test your IPC mechanism. You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

پیاده سازی sys_ipc_recv() در kern/sycall.c نسبتا آسان است. اگرچه کامنت می‌گوید که باید CPU را رها کنیم، برای انجام این کار نیازی به فراخوانی صریح sched_yield() نداریم زیرا علامت گذاری محیط غیر قابل اجرا به طور ضمنی این کار را انجام می‌دهد و اگر آن را فراخوانی کنیم، مقدار بازگشتی syscall نمی‌تواند به محیط کاربر ارسال شود. تمام مواردی که تمرین خواسته را پیاده‌سازی کردیم.

چالش ۱:

Challenge 1! (up to 10 points, depending on how elegant and correct your fine-grained locking is) The big kernel lock is simple and easy to use. Nevertheless, it eliminates all concurrency in kernel mode. Most modern operating systems use different locks to protect different parts of their shared state, an approach called *fine-grained locking*. Fine-grained locking can increase performance significantly, but is more difficult to implement and error-prone. If you are brave enough, drop the big kernel lock and embrace concurrency in JOS!

It is up to you to decide the locking granularity (the amount of data that a lock protects). As a hint, you may consider using spin locks to ensure exclusive access to these shared components in the JOS kernel:

- The page allocator.
- The console driver.
- The scheduler.
- The inter-process communication (IPC) state that you will implement in the part C.

برای مقابله با چالش اجرای قفل ریز در هسته سیستم عامل JOS، باید بخش‌های مهمی را که توسط بخش‌های مختلف هسته به طور همزمان به آن دسترسی دارند شناسایی کنید و با استفاده از قفل‌های چرخشی یا سایر موارد اولیه هماهنگ‌سازی مناسب برای محیط خود از آنها محافظت کنید. در اینجا چند مرحله و ملاحظات سطح بالا برای هر مؤلفه‌ای که ذکر کردید آورده شده است.
تخصیص دهنده صفحه: همه عملکردهایی را که به ساختارهای داده مشترک مانند لیست رایگان دسترسی دارند، شناسایی کنید. یک دانه بندی مناسب برای قفل کردن انتخاب کنید. یک قفل برای کل لیست رایگان ممکن است خیلی درشت باشد، اما یک قفل برای هر صفحه ممکن است خیلی خوب باشد. برای محافظت از بخش های حیاتی در این توابع تخصیص دهنده، قفل های اسپین را اجرا کنید. اطمینان حاصل کنید که قفل ها تا حد امکان کوتاه نگه داشته می شوند تا اختلاف را به حداقل برسانید.

درایور کنسول: از وضعیت مشترک کنسول مانند بافر یا موقعیت/شاخصی که در آن کاراکترها نوشته شده است محافظت کنید. قفل‌هایی را در اطراف کدهایی که از کنسول می‌خوانند یا روی آن می‌نویسند، پیاده‌سازی کنید. در صورت وجود اختلاف نظر، از ساختارهای داده بدون قفل یا بافرهای حلقه استفاده کنید. زمان‌بند: قفل‌های اسپین را در اطراف کدی که مسئول انتخاب رشته بعدی برای اجرا، دستکاری صف‌های آماده، و مدیریت رشته‌های خواب و بیدار است، اعمال کنید. اطمینان حاصل کنید که سوئیچ‌های بافت رشته به درستی اتفاق می‌افتند و هیچ رشته دیگری نمی‌تواند زمان‌بندی را در زمانی که وضعیت CPU تغییر می‌کند، پیشی بگیرد.

وضعیت ارتباطات بین فرآیندی (IPC): هنگامی که مکانیسم IPC خود را پیاده سازی کردید، از هرگونه داده مشترک مانند بافرهای پیام یا وضعیت فرستنده/گیرنده محافظت کنید. قفل کردن عملکردهای ارسال و دریافت برای جلوگیری از شرایط مسابقه بسیار مهم است.

نکات کلی زیر برای قفل کردن ریزدانه مهم هستند: تمام مسیرهای کد ممکن را به دقت تجزیه و تحلیل کنید تا از بن بست جلوگیری کنید. محدوده قفل (بخش بحرانی) را تا حد امکان کوچک نگه دارید. در صورت امکان از نگه داشتن چندین قفل به طور همزمان خودداری کنید، اما در صورت نیاز، همیشه آنها را به ترتیب ثابت تهیه کنید. مدیریت وقفه را در ذهن داشته باشید. اگر یک قفل را نگه دارید و وقفه ای وارد شود که سعی کند همان قفل را بدست آورد، سیستم شما به بن بست می رسد. در نهایت، هنگام استفاده از قفل های ریز دانه، ضروری است که هسته خود را به طور کامل آزمایش کنید تا مطمئن شوید که فاقد بن بست و شرایط مسابقه است. تحت بارهای مختلف تست کنید و از ابزارها استفاده کنید یا تکنیک‌هایی مانند نمودارهای وابستگی قفل را توسعه دهید تا صحت قفل را بررسی کنید.

چالش ۲:

Challenge 2! (5 points) Add a less trivial scheduling policy to the kernel, such as a fixed-priority scheduler that allows each environment to be assigned a priority and ensures that higher-priority environments are always chosen in preference to lower-priority environments. If you're feeling really adventurous, try implementing a Unix-style adjustable-priority scheduler or even a lottery or stride scheduler. (Look up "lottery scheduling" and "stride scheduling" in Google.)

Write a test program or two that verifies that your scheduling algorithm is working correctly (i.e., the right environments get run in the right order). It may be easier to write these test programs once you have implemented `fork()` and IPC in parts B and C of this lab.

برای پیاده‌سازی یک زمان‌بندی با اولویت ثابت در هستهJOS ، این مراحل را دنبال کنید:

اولویت‌ها را تعیین کنید: ساختار Env را گسترش دهید تا یک فیلد اولویت را شامل شود. کد اولیه سازی هسته را تغییر دهید تا یک اولویت پیش فرض به هر محیط اختصاص دهید یا یک فراخوانی سیستمی ایجاد کنید تا امکان تنظیم اولویت از فضای کاربر را فراهم کند.

Modify the Scheduler: تابع زمانبندی را در هسته تغییر دهید تا محیط ها را بر اساس اولویت ها مرتب کرده یا در صف قرار دهید. اطمینان حاصل کنید که زمان‌بند همیشه محیط قابل اجرا را با بالاترین اولویت انتخاب می‌کند

Maintain Preemption: با اولویت‌های ثابت، باید مطمئن شوید که زمان‌بندی به‌اندازه کافی مکرر اجرا می‌شود تا ارزیابی کند که کدام محیط باید اجرا شود. بسته به طراحی خود، از یک وقفه تایمر برای فراخوانی دوره‌ای زمان‌بندی استفاده کنید. اگر یک زمان‌بندی با اولویت قابل تنظیم یا زمان‌بندی پیچیده‌تر انتخاب کرده‌اید.

برنامه‌ریزی با اولویت قابل تنظیم: به‌طور دوره‌ای اولویت محیط‌های در حال اجرا را بر اساس عواملی مانند میزان زمان استفاده از CPUیا رفتار I/O آنها تنظیم کنید.

زمان‌بندی قرعه‌کشی: به هر محیط تعدادی بلیط قرعه‌کشی را بر اساس اولویت یا منابع مورد نیاز آن‌ها اختصاص دهید، و سپس از زمان‌بندی‌کننده بخواهید به‌طور تصادفی یک بلیط را انتخاب کند تا تصمیم بگیرد که کدام فرآیند بعدی اجرا شود.

برنامه ریزی گام: به هر محیطی یک مقدار گام بر اساس اولویت آن اختصاص دهید و از زمانبندی بخواهید که محیطی را با کمترین مقدار پاس برای اجرای بعدی انتخاب کند و پس از هر تصمیم زمانبندی میزان عبور آن را با گام افزایش دهد.

ایجاد محیط‌های آزمایشی: محیط‌های آزمایشی (فرآیندها) را با اولویت‌های مختلف بنویسید که وظایف قابل اندازه‌گیری را به راحتی انجام می‌دهند، مانند افزایش شمارنده.

مشاهده رفتار: محیط های آزمون را به طور همزمان اجرا کنید و ترتیب انجام وظایف آنها را که باید با اولویت های آنها مطابقت داشته باشد، رعایت کنید

Quantify Scheduling: برای تعیین کمیت صحت زمان‌بندی خود، از محیط‌های آزمایشی بخواهید گزارش‌های خروجی را در کنسول ارسال کنند یا در یک منطقه حافظه خاص بنویسید که می‌تواند توسط یک محیط با اولویت بالاتر یا پس از اجرای همه محیط‌های آزمایشی بررسی شود. Starvation را در نظر بگیرید. مطمئن شوید که محیط‌های با اولویت پایین‌تر از زمان CPU استفاده نمی‌کنند، مگر اینکه این یک تصمیم آگاهانه بر اساس خط‌مشی زمان‌بندی شما باشد (که معمولاً نباید چنین باشد).

Edge Cases: تست‌های طراحی برای موارد لبه، مانند بسیاری از محیط‌های با اولویت بالا برای اجرا، یا موقعیت‌هایی که اولویت‌ها اغلب تغییر می‌کنند. پس از ایجاد زمان‌بندی، باید آن را به طور گسترده آزمایش کنید تا مطمئن شوید که مطابق انتظار عمل می‌کند، حتی زمانی که عملکردهای پیچیده‌تری را در بخش‌های B و C آزمایشگاه خود، مانند ()fork و IPC ترکیب می‌کنید. به یاد داشته باشید که با بارهای زیاد و کم و همچنین با سطوح اولویت مختلف تست کنید تا مطمئن شوید که زمان‌بندی شما تحت هر شرایطی کار می‌کند.

چالش ۳:

Challenge 3! (5 points) The JOS kernel currently does not allow applications to use the x86 processor's x87 floating-point unit (FPU), MMX instructions, or Streaming SIMD Extensions (SSE). Extend the `Env` structure to provide a save area for the processor's floating point state, and extend the context switching code to save and restore this state properly when switching from one environment to another. The `FXSAVE` and `FXRSTOR` instructions may be useful, but note that these are not in the old i386 user's manual because they were introduced in more recent processors. Write a user-level test program that does something cool with floating-point.

برای فعال کردن پشتیبانی ازFPU ، MMX و SSE در هستهJOS ، باید چند مرحله را با محوریت مدیریت وضعیت ممیز شناور در طول سوئیچ‌های زمینه انجام دهید. در اینجا وظایف اولیه ای هستند که باید انجام دهید.

گسترش ساختارEnv

راه اندازی واحد نقطه شناور

ذخیره و بازیابی وضعیت ممیز شناور

تنظیم کننده خطای صفحه

فعال کردن دستورالعمل های SSE

چالش ۴:

Challenge 4! (5 points) Add the additional system calls necessary to *read* all of the vital state of an existing environment as well as set it up. Then implement a user mode program that forks off a child environment, runs it for a while (e.g., a few iterations of `sys_yield()`), then takes a complete snapshot or *checkpoint* of the child environment, runs the child for a while longer, and finally restores the child environment to the state it was in at the checkpoint and continues it from there. Thus, you are effectively "replaying" the execution of the child environment from an intermediate state. Make the child environment perform some interaction with the user using `sys_cgetc()` or `readline()` so that the user can view and mutate its internal state, and verify that with your checkpoint/restart you can give the child environment a case of selective amnesia, making it "forget" everything that happened beyond a certain point.

برای مقابله با این چالش، شما باید چند کار مجزا را انجام دهید: بهبود رابط تماس سیستم، ایجاد قابلیت چک پوینت، و نوشتن برنامه‌های آزمایشی. بیایید آن را گام به گام تجزیه کنیم.

گسترش تماس‌های سیستمی: فراخوان‌های سیستمی جدیدی را پیاده‌سازی کنید که به محیط والدین اجازه می‌دهد وضعیت یکی از فرزندان خود را بخواند و بنویسد. حالتی که باید خوانده یا نوشته شود شامل رجیسترهای همه منظوره، اشاره گر دستورالعمل، اشاره گر پشته و هر حالت پردازشگر دیگری است که ممکن است برای شروع مجدد اجرای یک فرآیند به طور دقیق مورد نیاز باشد (این می‌تواند شامل ثبت پرچم‌ها و حتی وضعیت ممیز شناور باشد، اگر شما چالش ۳ را تکمیل کردید). اطمینان حاصل کنید که امنیت و مجوزها در نظر گرفته شده است. یک محیط نباید بتواند وضعیت محیطی را که متعلق به خود نیست تغییر دهد. Snapshoting/System Checkpointing یک ساختار داده ایجاد کنید که بتواند کل وضعیت یک محیط را در یک نقطه

از زمان نگه دارد. اجرای عملکرد برای پر کردن این ساختار با وضعیت فعلی یک محیط با استفاده از فراخوانی سیستم ایجاد شده در مرحله ۱. اجرای عملکرد برای بازگرداندن وضعیت.

چالش ۸:

Challenge 8! (5 points) Why does `ipc_send` have to loop? Change the system call interface so it doesn't have to. Make sure you can handle multiple environments trying to send to one environment at the same time.

برای حل این چالش، باید رابط تماس سیستم را طوری تغییر دهیم که ipc_send نیازی به حلقه زدن نداشته باشد، اما بتواند چندین محیط را که در تلاش برای ارسال همزمان به یک محیط هستند، مدیریت کند. این احتمالاً شامل تقویت مکانیسم IPC برای پشتیبانی از پیام‌های ناهمزمان و روشی کارآمدتر برای مدیریت چندین فرستنده به یک گیرنده است.

در اینجا یک رویکرد سطح بالا آورده شده است:

از پیام‌رسانی ناهمزمان استفاده کنید: به‌جای مسدود کردن ipc_send، به فرستنده اجازه دهید تا عملکرد برگشت به فراخوانی یا روشی را برای اطلاع در هنگام ارسال موفقیت‌آمیز پیام مشخص کند. به این ترتیب، فرستنده نیازی به حلقه زدن ندارد و می‌تواند در حالی که منتظر ارسال پیام است، به کارهای دیگر ادامه دهد.

Implement Queues: برای هر محیط یک صف پیام معرفی کنید که بتواند چندین پیام را در خود جای دهد. به این ترتیب، حتی اگر چندین محیط سعی کنند به طور همزمان پیام ها را به یک محیط ارسال کنند، می توان پیام ها را در صف قرار داد و محیط دریافت کننده می تواند آنها را یکی یکی پردازش کند.

Handle Multiple Senders: مطمئن شوید که مکانیسم IPC شما می‌تواند سناریوهایی را که در آن چندین محیط سعی می‌کنند پیام‌ها را به یک محیط ارسال کنند، کنترل کند. این ممکن است شامل مکانیسم های قفل مناسب یا سایر تکنیک های همگام سازی برای جلوگیری از شرایط مسابقه باشد.

چالش ۹:

Challenge 9! (5 points) The prime sieve is only one neat use of message passing between a large number of concurrent programs. Read C. A. R. Hoare, ``Communicating Sequential Processes,'' *Communications of the ACM* 21(8) (August 1978), 666-667, and implement the matrix multiplication example.

این چالش شامل اجرای یک مثال ضرب ماتریس با استفاده از ارسال پیام بین تعداد زیادی از برنامه‌های همزمان است، همانطور که از C. A. R. Hoare "Communicating Sequential Processes" (CSP) الهام گرفته شده است. در اینجا مراحل کلی وجود دارد که می‌توانیم برای رویارویی با این چالش برداریم:

مقاله C. A. R. Hoare ، مقاله‌ای با عنوان «ارتباط فرآیندهای متوالی» است توسط C. A. R. Hoare نوشته شده و مفهوم ارتباط فرآیندهای متوالی (CSP) را معرفی می‌کند و بینش‌هایی را در مورد برنامه‌نویسی همزمان با استفاده از ارسال پیام ارائه می‌دهد.

از اصول CSP برای طراحی یک برنامه همزمان برای ضرب ماتریس استفاده کنید. به این فکر کنید که چگونه فرآیندهای مختلف (برنامه های همزمان) از طریق ارسال پیام برای انجام ضرب ارتباط برقرار می کنند.

تعریف فرآیندها و کانال های ارتباطی: فرآیندهای درگیر در ضرب ماتریس را شناسایی کنید (به عنوان مثال، فرآیندهای محاسبه عناصر منفرد ماتریس نتیجه). کانال‌های ارتباطی بین این فرآیندها را برای تبادل پیام تعریف کنید.

پیاده سازی ارسال پیام: سیستم عامل JOS را تغییر دهید یا گسترش دهید تا از ارسال پیام بین فرآیندها پشتیبانی کند. اجرای عملکردهای لازم برای ارسال و دریافت پیام.

اجرای ضرب ماتریس: کد ضرب ماتریس را در متن برنامه همزمان خود بنویسید. هر فرآیند باید یک کار خاص مربوط به ضرب را انجام دهد و آنها باید برای به اشتراک گذاشتن نتایج میانی ارتباط برقرار کنند.

تست و رفع اشکال: اجرای خود را با ماتریس های مختلف آزمایش کنید تا از صحت اطمینان حاصل کنید. اشکال زدایی هر مشکلی که در طول آزمایش ایجاد می‌شود.

چالش ۱۰:

Challenge 10! (5 points) Probably the most impressive example of the power of message passing is Doug McIlroy's power series calculator, described in M. Douglas McIlroy, ``Squinting at Power Series,'' *Software--Practice and Experience*, 20(7) (July 1990), 661-683. Implement his power series calculator and compute the power series for *sin*(*x*+*x*^3).

چالش ۱۰ شامل پیاده‌سازی ماشین‌حساب سری قدرت داگ مک‌ایلروی است، همانطور که در مقاله او «چشم‌اندازی در سری‌های توان» (نرم‌افزار – تمرین و تجربه، ژوئیه ۱۹۹۰) توضیح داده شده است. علاوه بر این، باید سری توان را برای sin(x + x^3) محاسبه کنیم. در ادامه یک رویکرد سطح بالا را برای حل این چالش بیان می‌کنیم.

درک محاسبات سری توان: درک کاملی از نحوه محاسبه سری های توان و نحوه استفاده رویکرد مک ایلروی از ارسال پیام برای این منظور به دست آورید.

ماشین حساب سری Power را طراحی کنید: مفاهیم مقاله مک ایلروی را برای طراحی ماشین حساب سری قدرت خود به کار ببرید. فرآیندهای درگیر را شناسایی کنید و کانال های ارتباطی را برای ارسال پیام تعریف کنید.

پیاده سازی ارسال پیام: سیستم عامل JOS را تغییر دهید یا گسترش دهید تا از ارسال پیام بین فرآیندها پشتیبانی کند. پیاده‌سازی توابع برای ارسال و دریافت پیام.

اجرای محاسبه سری قدرت: کد سری های توان محاسباتی را بنویسید. بر پیاده سازی الگوریتم‌های توصیف شده توسط مک ایلروی تمرکز کنید و اطمینان حاصل کنید که ارتباط بین فرآیندها به درستی انجام می‌شود.

محاسبه sin (x + x^3): به طور خاص، محاسبه sin(x + x^3) را با استفاده از ماشین‌حساب سری power که ساخته‌اید، اجرا کنید. این شامل ارزیابی سری برای عبارت داده شده است.

تست و رفع اشکال: ماشین حساب سری پاور خود را با ورودی های مختلف از جمله sin(x + x^3) تست کنید تا از صحت آن اطمینان حاصل کنید. اشکال زدایی هر مشکلی که در طول آزمایش ایجاد می‌شود.

چالش ۱۱:

Challenge 11! (5 points) Make JOS's IPC mechanism more efficient by applying some of the techniques from Liedtke's paper, "Improving IPC by Kernel Design", or any other tricks you may think of. Feel free to modify the kernel's system call API for this purpose, as long as your code is backwards compatible with what our grading scripts expect.

چالش ۱۱ شامل کارآمدتر کردن مکانیسم IPC (ارتباط بین فرآیندی) JOS با استفاده از تکنیک‌های مقاله Liedtke، «بهبود IPC توسط طراحی هسته» یا هر ترفند دیگری است که ممکن است کارایی را افزایش دهد. ما همچنین مجاز به تغییر API فراخوانی سیستم هسته برای این منظور هستیم و باید از سازگاری با اسکریپت‌های درجه بندی مورد انتظار اطمینان حاصل کنیم. در ادامه یک رویکرد سطح بالا برای حل این چالش را مطرح می‌کنیم.

شناسایی تکنیک‌های کلیدی: شناسایی تکنیک‌ها و استراتژی‌های کلیدی پیشنهاد شده توسط Liedtke که می‌تواند برای مکانیسم IPC JOS قابل اجرا باشد. روی مواردی تمرکز کنید که با اهداف کارایی و بهبود عملکرد همسو هستند.

مشخصات مکانیزم IPC موجود: مشکلات فعلی و مشکلات عملکرد در مکانیسم IPC JOS را درک کنید. نمایه سیستم را برای شناسایی مناطقی که می‌توان بهینه کرد.

استفاده از تکنیک‌های IPC کارآمد: پیاده‌سازی تکنیک‌های شناسایی شده برای بهبود کارایی مکانیزم IPC JOS. این ممکن است شامل تغییراتی در نحوه ارسال و دریافت پیام‌ها، بهینه‌سازی در ارسال پیام، یا سایر پیشرفت‌های سطح هسته باشد.

تغییر API تماس سیستم: در صورت لزوم، API فراخوانی سیستم هسته را تغییر دهید تا تغییرات IPC کارآمد را در خود جای دهد. از سازگاری با اسکریپت‌ها و برنامه‌های موجود که به API قبلی متکی هستند، اطمینان حاصل کنید.

همزمانی و همگام سازی: بهینه سازی های مربوط به همزمانی و همگام سازی را در نظر بگیرید. مقاله Liedtke ممکن است تکنیک هایی را برای به حداقل رساندن اختلاف و اطمینان از ارتباط کارآمد بین فرآیندها پیشنهاد کند.

تست و ارزیابی: مکانیسم IPC اصلاح شده را به طور کامل با بارهای کاری و سناریوهای مختلف آزمایش کنید تا مطمئن شوید که تغییرات منجر به بهبود عملکرد می شود. سیستم را برای تعیین کمیت سودهای کارایی محک بزنید.

نتیجه نهایی و خروجی make grade **:**