1 多处理器启动流程

1.1 多处理器支持

为了支持多处理器，首先需要知道多处理器的配置，这个配置通常是存储在 BIOS 里面。BIOS 需要传递配置信息给多个处理器，同时需要能复原多处理器及其相关组件，多处理器的 BIOS 也要扩展功能，增加 MP 配置。

SMP 是指所有处理器都是平等的，包括内存对称和 IO 对称。内存对称指所有处理器都共享同样的内存地址空间，访问相同的内存地址。而 IO 对称则是所有处理器共享相同的 IO 子系统(包括 IO 端口和中断控制器)，任一处理器可以接收任何源的中断。虽然处理器都平等的，但是可以分为 BSP(启动处理器)和 AP(应用处理器)，BSP 负责初始化其他处理器，至于哪个处理器是 BSP 则是由 BIOS 配置决定的。

APIC（Advanced Programmable Interrupt Controller）基于分布式结构，分为两个单元，一个是处理器内部的 Local APIC 单元(LAPIC)，另一个是 IO APIC 单元，它们两个通过 Interrupt Controller Communications (ICC) 总线连接。APIC 作用一是减轻了内存总线中关于中断相关流量，二是可以在多处理器里面分担中断处理的负载。

LAPIC 提供了 interprocessor interrupts (IPIs),它允许任意处理器中断其他处理器或者设置其他处理器，有好几种类型的 IPIs，如 INIT IPIs 和 STARTUP IPIs。每个 LAPIC 都有一个本地 ID 寄存器，每个 IO APIC 都有一个 IO ID 寄存器，这个 ID 是每个 APIC 单元的物理名称，它可以用于指定 IO 中断和 interprocess 中断的目的地址。因为 APIC 的分布式结构，LAPIC 和 IO APIC 可以是独立的芯片，也可以将 LAPIC 和 CPU 集成在一个芯片，如英特尔奔腾处理器（735\90, 815\100），而 IO APIC 集成在 IO 芯片，如英特尔 82430 PCI-EISA 网桥芯片。集成式 APIC 和分离式 APIC 编程接口大体是一样的，不同之处是集成式 APIC 多了一个 STARTUP 的 IPI。

在 SMP 系统中，每个 CPU 都伴随有一个 LAPIC 单元，LAPIC 用于传递和响应系统中断，LAPIC 也为与它连接的 CPU 提供了一个唯一 ID，在 lab4 中，我们只用到 LAPIC 的一些基本功能：

- 读取 LAPIC 标识来告诉我们当前代码运行在哪个 CPU 上(见 cpunum()）。
- 从 BSP 发送 STARTUP IPI 到 AP，用于启动 AP，见（lapic_startup())。
- 在 part C，我们变成 LAPIC 内置的计时器来触发时钟中断支持抢占式多任务(见 apic_init())。

处理器访问它的 LAPIC 使用的是 MMIO，在 MMIO 里，一部分内存硬连线到了 IO 设备的寄存器，因此用于访问内存的 load/store 指令可以用于访问 IO 设备的寄存器。比如我们在实验 1 中用到 0xA0000 开始的一段内存作为 VGA 显示缓存。LAPIC 所在物理地址开始于 0xFE000000(从 Intel 的文档和测试看这个地址应该是 0xFEE00000)，在 JOS 里面内核的虚拟地址映射从 KERNBASE(0xf00000000)来说，这个地址太高了，于是在 JOS 里面在 MMIOBASE(0xef800000）地址处留了 4MB 空间用于 MMIO，后面实验会用到更多的 MMIO 区域，为此我们要映射好设备内存到 MMIOBASE 这块区域，这个过程有点像 `boot_alloc`，注意映射范围判断。接下来完成作业 1，见代码。

```
 *    MMIOLIM ------>  +------------------------------+ 0xefc00000      --+
 *                    |      Memory-mapped I/O       | RW/--  PTSIZE
 * ULIM, MMIOBASE -->  +------------------------------+ 0xef800000
```

1.2 AP 启动流程

在启动 AP 前，BSP 首先要收集多处理器系统的信息，比如 CPU 数目，CPU 的 APIC ID 和 LAPIC 单元的 MMIO 地址。`kern/mpconfig.c` 的 mp_init()函数通过读取驻留在 BIOS 内存区域中的 MP 配置表来获取这些信息。

boot_aps()函数驱动 AP 启动进程。AP 以实模式启动，很像 `boot/boot.S` 中那样，boot_aps()将 AP entry 代码拷贝到实模式可寻址的一个地址，与 bootloader 不同的是，我们可以控制 AP 开始执行代码的位置，我们将 AP entry 代码拷贝到 0x7000(MPENTRY_PADDR)，当然其实你拷贝到 640KB 之下的任何可用的按页对齐的物理地址都是可以的。

之后，boot_aps()通过向 AP 的 LAPIC 发送 STARTUP IPIs 依次激活 AP，并带上 AP 要执行的初始入口地址 CS:IP（MPENTRY_PADDR)。入口代码在 kern/mpentry.S，跟 `boot/boot.S` 非常相似。在简单的设置后，它将 AP 设置为保护模式，并开启分页，然后调用 mp_main()里面的 C 设置代码。boot_aps()会等待 AP 在其 CpuInfo 中的 cpu_status 字段发出 CPU_STARTED 标志，然后继续唤醒下一个 AP。为此需要将 MPENTRY_PADDR 这一页内存空出来。

接下来我们要分析下加入多处理器支持后 JOS 的启动流程，新加的几个相关函数是 `mp_init()`, `lapic_init()`以及 `boot_aps()`。

多处理器配置搜索和初始化

mp_init()主要是搜索多处理器配置信息，要怎么找呢，首先是按下面的顺序找 `MP Floating Pointer Structure`(简写为 MPFPS)。

- 1 去 Extended BIOS Data Area (EBDA)的前 1KB 处
- 2 去系统 base memory 的最后 1KB 找
- 3 去 BIOS 的只读内存空间： 0xE0000 和 0xFFFFF 之间找(代码里面用的是 0xF0000 到 0xFFFFF 位置)。

```
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

而 EBDA 的起始位置可以从 BIOS 的 40:0Eh 处找到，也就是 0x40 << 4 + 0x0Eh = 0x40Eh 处找 EBDA 的起始位置。在测试中，我的测试机里面显示该值为 `0x9fc0`，故而会先在 EBDA 的 `0x9fc00`(左移 4 位得到物理地址) 到 0xA0000 之间找。在 BIOS 的 40:13h 处可以找到 base memory 大小值减 1KB 的值，这个值是 以 KB 为单位的，比如我的测试环境显示该值为 0x9fc00，则 base memory 为 0x9fc00 + 1K = 0xA0000 也就是 640KB。由此我们这里 EBDA 的前 1KB 和 base memory 的最后 1KB 其实是同一个内存区域。如果前面两个位置找不到，则会去 0xE0000h 到 0xFFFFFh 区域找。在测试环境中在位置 3 找到了 MPFPS，这里的校验方式是 **mp->signature == "MP" 且 mp 结构体的所有字段之和为 0**。

找到了 MPFPS 后，我们要根据它的配置去找 `MP Configuration Table`(MPCT)，发现 MPFPS 中的 `physical address` 值为 0xf64d0，即表示 MPCT 地址在 0xf64d0 开始的一段 BIOS ROM 里面。可以调试发现我们测试机里面的 MPCT 的版本为 1.4，LAPIC 的基地址为 `0xfee00000`，配置项有 20 个，而这里的配置项又分为 `Processor`，`Bus`，`IO APIC`，`IO Interrupt Assignment` 以及 `Local Interrupt Assignment` 这五种类型。对于处理器类型，这里有几个比较重要的字段，其中有 cpu 的几个标识，其中一个 BP 如果设置为 1，表示这个处理器是启动处理器 BSP。另一个是 EN，为 1 表示启用，为 0 表示禁用。还有一个 LAPIC ID 字段，用于标识该处理器里面的 LAPIC 的 ID，ID 是从 0 开始编号的。JOS 里面最多支持 8 个 CPU，多余的 CPU 不会启用。

在我们测试的时候 `make qemu CPUS=n`，其中的 n 就是指定的模拟的 CPU 的数目，指定了几个我们就能找到几个 CPU 的 MPCT 配置项。为维护 CPU 状态，JOS 内核中维护了一个 cpus 的数组和 CpuInfo 结构体。

```
// Per-CPU state
struct CpuInfo {
    uint8_t cpu_id;                 // Local APIC ID; index into cpus[] below
    volatile unsigned cpu_status;   // The status of the CPU
    struct Env *cpu_env;            // The currently-running environment.
    struct Taskstate cpu_ts;        // Used by x86 to find stack for interrupt
};

// Initialized in mpconfig.c
extern struct CpuInfo cpus[NCPU];
```

找到配置后，接着会设置 BSP 为 falg 为 BP 的处理器，并将其状态设置为 CPU_STARTED。接下来开始初始化 LAPIC。

lapic_init()

因为 LAPIC 的起始地址默认是在物理地址 0XFEE00000，为了方便访问，JOS 将这地址通过 MMIO 映射到了虚拟地址 MMIOBASE。映射完成后，我们就可以用 lapic 这个虚拟地址来访问和设置 LAPIC 了。lapic_init()主要对 LAPIC 的一些寄存器进行设置，包括设置 ID，version，以及禁止所有 CPU 的 NMI(LINT1)，BSP 的 LAPIC 要以 Virtual Wire Mode 运行，开启 BSP 的 LINT0，以用于接收 8259A 芯片的中断等。

pic_init()

pic_init()用于初始化 8259A 芯片的中断控制器。8259A 芯片是一个中断管理芯片，中断来源除了来自硬件本身的 NMI 中断以及软件的 INT n 指令造成的软件中断外，还有来自外部硬件设备的中断(INTR)，这些外部中断时可以屏蔽的。而这些中断的管理都是通过 PIC（可编程中断控制器）来控制并决定是否传递给 CPU，JOS 中开启的 INTR 中断号有 1 和 2。

boot_aps()

接下来是启动 AP 了。首先通过 memmove 将 mpentry 的代码拷贝到 MPENTRY_PADDR (0x7000)处(其中习题 2 要将 0x7000 对应的一页设置为已用，不要加入到空闲链表)，设置好对应该 cpu 的堆栈栈顶指针，然后调用 `kern/lapic.c` 中的 `lapic_startap()`开始启动 AP。

lapic_startap()完成 lapic 的设置，包括设置 warm reset vector 指向 mpentry 代码起始位置，发送 STARTUP IPI 以触发 AP 开始运行 mpentry 代码，并等待 AP 启动完成，一个 AP 启动完成后再启动下一个。

那么 mpentry 代码就是在 `kern/mpentry.S` 中了，它的作用类似 bootloader，最后是跳转到 mp_main()函数执行初始化。mpentry.S 在加载 GDT 和跳转时用到了 MPBOOTPHYS 宏定义，因为 mpentry.S 代码是加载在 KERNBASE 之上的，在 CPU 实模式下是无法寻址到这些高地址的，所以需要转换为物理地址。而 boot.S 代码不用转换，是因为它们本身就加载在实模式可以寻址的 `0x7c00-0x7dff`。后面的流程跟 boot.S 类似，也是开启保护模式和分页。因为 mpentry 的代码加载到了 0x7000，需要在 `page_init()` 中将这一页从 page 空闲链表去除，见作业 2.

```
#define MPBOOTPHYS(s) ((s) - mpentry_start + MPENTRY_PADDR)
```

此时用的页目录跟 `kern/entry.S` 时一样，用的也是 entry_pgdir，因为此时的运行指令在低地址，并没有在 `kern_pgdir` 建立映射。 最后通过 `call` 指令跳转到 mp_main()函数执行，注意下这里用了间接 call，为什么不是直接 `call $mp_main` 呢？这里之所以不直接 call，是因为直接 call 用的是相对地址，即将 EIP 设置为 `EIP + call` 后跟的一个相对地址，比如这里我们的 call 指令的地址为 `0x7050`，然后 EIP 会指向下一条地址 0x7055，call 地址会被设置为 `0x7050 + 5 + 0xffffa609 = 0x10000165e`，地址溢出后变成 `0x165e`，而这个地址内容是 `0x80050044`，可知 0x165e 处对应的指令是 0x44，也就是 `inc %esp`，当然这一步不会报错，接着下一条指令在 0x165f，指令对应的是 `00 05 80 44 00 05`，即 `add %al, 0x05004480`，则此时访问地址 `0x05004480` 会报错，因为此时用的是 `entry_pgdir`，还没有建立该地址的页面映射。

```
## 正确方式
mov $mp_main, %eax;
call *%eax;

## 错误方式
call mp_main
f0105bc8:       e8 09 a6 ff ff          call   f01001d6 <mp_main>

(gdb) x /16x 0x165e
0x165e:   0x80050044        0x90050044        0xa0050044        0xb0050044
```

mp_main()函数先是加载了 kern_pgdir 到 CR3 中，然后调用下面几个方法，包括前面提过的 lapic_init(),以及为每个 CPU 初始化进程相关内容和中断相关内容，最后设置 cpu 状态为 CPU_STARTED 让 BSP 去启动下一个 CPU。注意到这里用到了 xchg 函数来设置 cpu 状态，该函数用到 xchgl 来交换 addr 存储的值和 newval，并将 addr 原来存储的值存到 result 变量中返回，指令中的 `lock;`用于保证多处理器操作的原子性。

```
void
mp_main(void)
{
	// We are in high EIP now, safe to switch to kern_pgdir
	lcr3(PADDR(kern_pgdir));
	lapic_init();
	env_init_percpu();
	trap_init_percpu();
	xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up
	for (;;);
}

static inline uint32_t
xchg(volatile uint32_t *addr, uint32_t newval)
{
    uint32_t result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1"
             : "+m" (*addr), "=a" (result)
             : "1" (newval)
             : "cc");
    return result;
}
```

1.3 CPU 初始化

多核 CPU 需要各自优化，每个 CPU 都有自己的一些初始化变量，如下：

内核栈

每个 cpu 都要有一个内核栈，以免互相干扰。`percpu_kstacks[NCPU][KSTKSIZE]`用于保存栈空间。

TSS 和 TSS 描述符

每个 CPU 都要有自己的 TSS(任务状态段)和 TSS 描述符。CPU i 的 TSS 存储在 `cpus[i].cpu_ts`，而 TSS 描述符在 GDT 中的索引是 `gdt[(GD_TSS0 >> 3) + i]`，之前实验用到的全局变量 ts 不再需要了。

当前进程指针

每个 CPU 都要有自己运行的当前 CPU 运行的当前进程(Env)的指针 curenv，存储在 `cpus[cpunum()].cpu_env` 或 `thiscpu->cpu_env`。

系统寄存器

所有寄存器，包括系统寄存器都是每个 CPU 独有的。因此 lcr3，ltr，lgdt，lidt 这些指令在每个 CPU 上都要执行一次，其中 `env_init_per_cpu()`和 `trap_init_per_cpu()`就是用于这个目的。

具体实现见作业 3-4。

1.4 内核锁

在 mp_main 中初始化 AP 后，我们开始 spin 循环。在 AP 进一步操作前，我们需要解决多个 CPU 同时运行内核代码时的资源竞争问题，因为多进程同时运行内核代码，会影响内核中的数据正确性。最简单的方式是使用 `big kernel lock`(大内核锁)，进程在进入内核时获取大内核锁，回到用户态时释放锁。在该模式下，进程可以并发的运行在空闲的 CPU 上，但是同时只能有一个进程运行在内核态，其他进程想进入内核态必须等待。

大内核锁在 `kern/spinlock.h` 中定义，可以通过 `locker_kernel()`和 `unlock_kernel()`来进行加锁和解锁。我们需要在下面几处位置加锁和释放锁。

- 在 `i386_init()`中，在 BSP 唤醒 AP 前加锁。
- 在 `mp_main()`中，初始化 AP 后加锁，并调用 `sched_yield()`在该 AP 上运行进程。
- 在 `trap()`中，进程从用户态陷入时加锁。
- 在 `env_run()`中，进程切换到用户态之前释放锁。

这样，我们在 BSP 启动 AP 前，先加了锁。AP 经过 mp_main()初始化后，因为此时 BSP 持有锁，所以 AP 的 `sched_yield()`需要等待，而当 BSP 执行调度运行进程后，会释放锁，此时等待锁的 AP 便会获取到锁并执行其他进程。

1.5 轮转调度

轮转调度(round-robin)在 `sched_yield()`中完成，核心思想就是从进程列表中找到一个状态为 ENV_RUNNABLE 的进程运行。注意，不能同时有两个 CPU 运行同一个进程，这个可以根据进程状态进行判断，已经运行的进程状态为 ENV_RUNNING 。如果在列表中找不到 ENV_RUNNABLE 的进程，而之前运行的进程又处于 ENV_RUNNING 状态，则可以继续运行之前的进程。

修改了 `kern/init.c` 运行 3 个 `user_yield` 进程，可以看到输出如下：

```
# make qemu CPUS=2
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Back in environment 00001000, iteration 0.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
Back in environment 00001000, iteration 1.
```

```
Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001002, iteration 1.
Back in environment 00001001, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001002, iteration 2.
Back in environment 00001001, iteration 3.
Back in environment 00001000, iteration 4.
Back in environment 00001002, iteration 3.
All done in environment 00001000.
[00001000] exiting gracefully
[00001000] free env 00001000
Back in environment 00001001, iteration 4.
Back in environment 00001002, iteration 4.
All done in environment 00001001.
All done in environment 00001002.
[00001001] exiting gracefully
[00001001] free env 00001001
[00001002] exiting gracefully
[00001002] free env 00001002
```

流程如下：

- BSP 先加载 3 个进程，并设置为 ENV_RUNNBALE 状态。
- BSP 先唤醒 AP，由于 BSP 先在 i386_init 时持有内核锁，所以 BSP 先运行进程 1 0x1000，运行进程时 env_run() 切换到用户态前会释放内核锁，此时等待锁的 AP 开始运行 `sched_yield`，这样 AP 会开始运行进程 2 0x1001，开始运行后释放内核锁。
- BSP 打印出进程号后调用了 `sys_yield()`，陷入到内核的 trap()里面会加内核锁，所以等到 AP 开始运行进程 2 且打印了进程号后，BSP 此时运行进程 3。此后两个 CPU 轮流调度可运行的三个进程。

1.6 创建进程的系统调用

Unix 提供了 fork()系统调用创建进程，Unix 拷贝了调用进程(父进程）的整个地址空间用于创建新进程(子进程），在用户空间看来他们的唯一区别就是进程 ID 不同。在父进程中，fork()返回子进程 ID，而在子进程中，fork()返回 0。默认情况下父子进程都有自己的私有地址空间，且它们对内存修改互不影响。

在 JOS 中我们要提供几个不同的系统调用用于创建进程，这也是 Unix 早期实现 fork()的方式，下一节会讨论使用 COW 技术实现的新的 fork()。

sys_exofork

这个系统调用创建了一个几乎空白的新的进程，它没有任何东西映射到其地址空间的用户部分，且它不可运行。这个新的进程与父进程有意义的寄存器状态，在父进程中，它返回子进程的 envid，而在子进程中，它返回 0。由于 sys_exofork 初始化将子进程标记为 ENV_NOT_RUNNABLE，因此 sys_exofork 不会返回到子进程，只有父进程用 sys_env_set_status 将其状态设置 ENV_RUNNABLE 后，子进程才能运行。

sys_env_set_status

设置指定的进程状态为 ENV_NOT_RUNNABLE 或者 ENV_RUNNABLE，用于标记进程可以开始运行。

sys_page_alloc

用于分配一页物理内存并将其映射到指定的虚拟地址。不同于 page_alloc，sys_page_alloc 不仅分配了物理页，而且要通过 page_insert()将分配的物理页映射到虚拟地址 va。

sys_page_map

从一个进程的地址空间拷贝一个页面映射(**注意，不是拷贝页的内容**）到另一个进程的地址空间。其实就是用于将父进程的某个临时地址空间如 UTEMP 映射到子进程的新分配的物理页，方便父进程访问子进程新分配的内存以拷贝数据。

sys_page_unmap

取消指定进程的指定虚拟地址处的页面映射以下次重复使用。

所有上面的系统调用都接收进程 ID 参数，如果传 0 表示指当前进程。通过进程 ID 得到进程 env 对象可以通过函数 `kern/env.c` 中的 envidenv() 实现。

在 user/dumbfork.c 中有一个类似 unix 的 fork()的实现，它使用了上面这几个系统调用运行了子进程，子进程拷贝了父进程的地址空间。父子进程交替切换，最后父进程在循环 10 次后退出，而子进程则是循环 20 次后退出。

```
void
duppage(envid_t dstenv, void *addr)
{
    int r;

    // This is NOT what you should do in your fork.
    if ((r = sys_page_alloc(dstenv, addr, PTE_P|PTE_U|PTE_W)) < 0)
        panic("sys_page_alloc: %e", r);
    if ((r = sys_page_map(dstenv, addr, 0, UTEMP, PTE_P|PTE_U|PTE_W)) < 0)
        panic("sys_page_map: %e", r);
    memmove(UTEMP, addr, PGSIZE);
    if ((r = sys_page_unmap(0, UTEMP)) < 0)
        panic("sys_page_unmap: %e", r);
}
```

user/dumbfork.c 中的 dumbfork()具体实现流程是这样的：

- 1）先通过 sys_exofork() 系统调用创建一个新的空白进程。
- 2）然后通过 duppage 拷贝父进程的地址空间到子进程中。用户进程地址空间开始位置是 UTEXT（0x00800000），结束位置是 end。duppage 是一页页拷贝的，它将父进程的 addr 开始的一页物理内存内容拷贝到子进程 dstenv 的对应的页中。
-

iii. 完成父进程到子进程内存数据的拷贝。

  - 3.1）先通过 sys_page_alloc 为子进程 addr 开始的一页内容分配一个物理页并完成映射，此时，分配的物理页还是空的，没有数据。然后通过 sys_page_map 将子进程 va 开始的这分配好的物理页映射到父进程的 UTEMP 地址处(0x00400000)，这么做的目的就是为了在父进程中访问到子进程新分配的物理页。
  - 3.2)接下来，通过 memmove 函数将父进程 addr 处的一页数据拷贝到了 UTEMP 中，而因为前面看到 UTEMP 已经映射到了子进程的那页内存，所以最终效果就是将父进程的 addr 处的一页内存数据拷贝到子进程的 addr 对应的那页内存完成数据的复制。
  - 3.3)最后通过 sys_page_unmap 取消父进程在 UTEMP 的映射以下次使用，当然还有个重要目的是预防父进程误操作到子进程的内存数据。

2 写时复制(Copy On Write)

前面实现 fork 是直接将父进程的数据拷贝到了子进程，这是 Unix 系统最初采用的方式，但是这样有个问题就是会造成资源浪费，很多时候我们 fork 一个子进程，接着是直接 exec 替换子进程的内存直接执行另一个程序，子进程在 exec 之前用到父进程的内存数据很少。

于是后续的 Unix 版本优化了 fork，利用了虚拟内存硬件支持的方式，fork 时拷贝的是地址空间而不是物理内存数据，这样，父子进程各自的地址空间都映射到同样的内存数据，共享的内存页会被标记为只读。当父子进程有一方要修改共享内存时，此时会报 `page fault` 错误，此时 Unix 内核会为报错的进程分配一个新的物理页，并拷共享内存页的数据到新分配的物理页中。执行 exec 时，只需要拷贝堆栈这一个页面即可。

2.1 用户程序页面错误处理

为了实现写时复制，首先要实现用户程序页面错误处理功能。基本流程是：

- 1）用户进程通过 set_pgfault_handler(handler) 设置页面错误处理函数。
- 2）函数 set_pgfault_handler 中为用户程序分配异常栈，通过系统调用 sys_env_set_pgfault_upcall 设置通用的页面错误处理调用入口。
- 3）**当用户进程发生页面错误时，陷入内核**。内核先判断该进程是否设置了 env_pgfault_upcall，如果没有设置，则报错。如果设置了，则切换用户进程栈到异常栈，设置异常栈内容，然后设置 EIP 为 env_pgfault_upcall 地址，**切回用户态执行 env_pgfault_upcall 函数(即_pgfault_upcall)**。
- 4）**env_pgfault_upcall 作为页面错误处理函数的入口函数，它在用户态运行**。先调用步骤 1 中注册的页面错误处理函数，然后再恢复进程在页面错误之前的栈内容，并切回常规栈，跳转到页面错误之前的地方继续运行。

设置用户级页面错误处理函数

前面提到，新的 fork 并不直接拷贝内存数据，而是先对共享的内存页设置一个特殊标记，然后在父子进程的一方写共享内存发生页面错误时，内核捕获异常并分配新的页和拷贝数据。这里首先要实现的是对用户级的页面错误的捕获和处理。

COW 只是用户级页面错误处理的许多可能用途之一。大多数 Unix 内核最初只映射新进程的堆栈，随着堆栈消耗增加，访问尚未映射的堆栈地址会导致页面错误，内核捕获错误后会分配并映射附加的堆栈页面。典型的 Unix 内核必须跟踪进程空间的每个区域发生页面错误时要采取的操作。例如，堆栈区域中的错误通常会分配并映射新的物理内存页面，程序的 BSS 区域中的错误通常会分配一个新页面，填充零并映射它。而可执行代码中导致的页面错误将触发内核从磁盘读取可执行文件的相应页面，然后映射它。

为了处理用户进程页面错误，用户进程需要设置一个页面错误处理函数，新增加一个系统调用 `sys_env_set_pgfault_call` 来设置 Env 结构体的 env_pgfault_upcall 字段即可。

用户进程异常栈和常规栈

而为了处理用户级页面错误，JOS 采用了一个用户异常栈 UXSTACKTOP(0xeec00000)，注意用户进程的常规栈用的是 USTACKTOP(0xeebfe000)。当用户进程发生页面错误时，内核会切换到异常栈，异常栈大小也是 PGSIZE。从用户常规栈切换到异常栈的过程有点像发生中断/异常时从用户态进入内核时的堆栈切换。

当运行在异常栈时，用户级页面错误处理函数可以调用 JOS 的常规系统调用去映射新的页面，以期修复导致页面错误的问题。当用户级页面错误处理函数处理完成后，再通过一段汇编代码返回到常规堆栈存储的发生页面错误的地址处继续运行。

需要支持用户级页面错误处理的用户进程都需要为它的异常栈分配内存，可以使用前面用过的 sys_page_alloc 分配内存。

调用用户页面错误处理函数

修改 `kern/trap.c` 中的页面错误处理代码以支持用户进程的页面错误处理。如果用户进程没有注册页面错误处理函数，则跟之前一样返回错误即可。而如果设置了页面错误处理函数，则需要在异常栈中压入下面内容以记录出错状态，这些内容正好构成了一个 UTrapframe 结构体，方便统一处理，接着设置 EIP 为 env_pgfault_upcall 函数地址，并将进程的堆栈切换到异常栈，然后开始运行页面错误处理函数。

页面错误处理函数是在 `lib/pfentry.S` 中定义的，它首先要执行用户程序中定义的 pgfault_handler 函数，然后再回到程序出错位置继续运行。

需要注意的是，如果用户进程已经运行在异常栈了，此时又发生嵌套页面错误，则需要在 `tf->tf_esp` 而不是从 UXSTACKTOP 压入异常数据，而且这种情况下，你要保留一个空的 4 字节，再压入 UTrapframe。要检查用户进程是否运行在异常栈，可以检查 tf->tf_esp 是否在区间 [UXSTACKTOP-PGSIZE, UXSTACKTOP-1]。

```
                    <-- UXSTACKTOP
trap-time esp   // 页面错误时用户栈的地址
trap-time eflags
```

```
trap-time eip
trap-time eax       start of struct PushRegs
trap-time ecx
trap-time edx
trap-time ebx
trap-time esp
trap-time ebp
trap-time esi
trap-time edi       end of struct PushRegs
tf_err (error code)
fault_va            <-- %esp when handler is run
```

回到页面错误处继续执行

在执行完页面错误处理函数后，需要回到用户进程之前出错的位置继续执行，这里需要完成 `lib/pfentry.S` 中的`_pgfault_upcall` 函数。

这个函数做的工作就是将异常栈切换到常规栈，重新设置 EIP，注意之前页面出错地址存储在 fault_va 中。这里要添加代码如下，此时 esp 指向的是前一节的 UTrapframe 的地址，这里做的工作主要是：

- 将用户进程的常规栈当前位置减去 4 字节，然后将用户进程页面错误时的 EIP 存储到该位置。这样恢复常规栈的时候，栈顶存储的是出错时的 EIP。
- 然后将异常栈中存储的用户进程页面错误时的通用寄存器和 eflags 寄存器的值还原。
- 然后将异常栈中存储的 esp 的值还原到 esp 寄存器。
- 最后通过 ret 指令返回到用户进程出错时的地址继续执行。(ret 指令执行的操作就是将弹出栈顶元素，并将 EIP 设置为该值，此时正好栈顶是我们在之前设置的出错时的 EIP 的值）
- **现在可以看到如果发生嵌套页错误为什么多保留 4 个字节了，这是因为发生嵌套页错误时，此时我们的 trap-time esp 存储的是异常栈，此时会将 trap-time 的 EIP 的值会被设置到 esp-4 处，如果不空出 4 字节，则会覆盖原来的 esp 值了。**

```
movl 0x28(%esp), %ebx # trap-time 时的 eip，注意 UTrapframe 结构
subl $0x4, 0x30(%esp)
movl 0x30(%esp), %eax
movl %ebx, (%eax)    # 将 trap-time 的 eip 拷贝到 trap-time esp-4 处
addl $0x8, %esp

popal

addl $0x4, %esp # 设置 eflags
popfl

popl %esp     # 将栈顶地址弹出到 esp，此时栈顶值是用户进程出错时的 eip 值
ret
```

最后还要完成 `lib/pgfault.c` 中的 `set_pgfault_handler` 函数，用于为用户进程分配异常栈以及页面错误处理函数 env_pgfault_upcall 的初始化设置。

2.2 实现写时复制 fork

完成上一节准备工作后，开始实现 COW 的 fork，fork 实现的流程如下：

- 1）父进程设置 pgfault()函数为页面错误处理函数，用到前面的 set_pgfault_handler 函数。
- 2）父进程调用 sys_exofork() 创建一个空白子进程。
- 3）对父进程在 UTOP 之下的可写或者 COW 的物理页，父进程调用 duppage，duppage 会将这些页面设置为 COW 映射到子进程的地址空间，同时，也要将父进程本身的页面重新映射，将页面权限设置为 COW(注：子进程的 COW 设置要在父进程之前)。duppage 将父子进程相关页面权限设置为不可写，且在 avail 字段设置为 COW，用于区分只读页面和 COW 页面。异常栈不以这种方式重新映射，需要在子进程分配一个新的页面给异常栈用。fork()还要处理那些不是可写的且不是 COW 的页面。
- 4）父进程设置子进程的页面错误处理函数。
- 5）父进程标识子进程状态为可运行。

当父子进程中任意一个试图修改一个还没有写过的 COW 页面，会触发页面错误，开始下面流程：

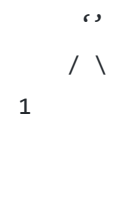
- 1）内核发现用户程序页面错误后，转至_pgfault_upcall 处理，而_pgfault_upcall 会调用 pgfault()。
- 2）pgfault()检查这是一个写错误(错误码中的 FEC_WR)且页面权限是 COW 的，如果不是则报错。
- 3）pgfault()分配一个新的物理页，并映射到一个临时位置，然后将出错页面的内容拷贝到新的物理页中，然后将新的页设置为用户可读写权限，并映射到对应位置。

fork()，pgfault(),duppage()三个函数的具体实现见作业 12。完成后 `make run-forktree`，正常应该输出下面的内容(顺序可能不同)：

```
1000: I am ''
1001: I am '0'
2000: I am '00'
2001: I am '000'
1002: I am '1'
3000: I am '11'
3001: I am '10'
4000: I am '100'
1003: I am '01'
5000: I am '010'
4001: I am '011'
2002: I am '110'
```

```
1004: I am '001'
1005: I am '111'
1006: I am '101'
```

forktree 这个程序比较有意思，它先创建两个子进程打印第一层 0， 1，然后子进程再分别创建子进程打印一棵树出来，比如两层是这样的，打印结果是 `''`, `0`, `1`, `00`, `01`, `10`, `11`。



3 抢占式调度和进程间通信

3.1 时钟中断

最后一部分是通过时钟中断来完成抢占式调度。运行 `make run-spin` 可以看到子进程死循环占用了 CPU，没法切换到其他进程了，现在需要通过时钟中断来强制调度。**时钟中断属于可屏蔽中断，可以通过 eflags 寄存器的 IF 位来控制，注意由 int 指令触发的软件中断不受 eflags 寄存器的控制，它是不可屏蔽中断，此外 NMI 也属于不可屏蔽中断。**

外部中断通常称之为 IRQ，IRQ 到中断描述符表的入口不是固定的。不过在 pic_init 中我们将 IRQ 的 0-15 映射到了 IDT 的[IRQ_OFFSET, IRQ_OFFSET+15]。其中 IRQ_OFFSET 为 32，所以 IRQ 在 IDT 中范围为[32, 47]，共 16 个。JOS 中对中断做了简化处理，在内核态时外部中断是禁止的，在用户态时才会开启。中断开启和禁止是通过 eflags 寄存器的 FL_IF 位来控制，为 1 表示开启中断，为 0 则禁止中断。

接下来类似实验 3 那样，设置中断号和中断处理程序。**注意在实验 3 中我将 istrap 基本都设置为 1 了，虽然那时候不影响实验结果，在实验 4 这里必须要全部将 istrap 值设为 0。因为 JOS 中的这个 istrap 设为 1 就会在开始处理中断时将 FL_IF 置为 1，而设为 0 则保持 FL_IF 不变，设为 0 才能通过 trap()中对 FL_IF 的检查。**最后在 trap() 函数中处理 IRQ_TIMER 中断，调用 `lapic_eio()`和 `sched_yield()`即可。

3.2 进程间通信(IPC)

最后要完成进程间通信，常见的一个 IPC 例子就是管道。实现 IPC 有很多方式，哪种方式最好至今仍有争论，JOS 中会实现一种简单的 IPC 机制。需要完成 sys_ipc_try_send() 和 sys_ipc_recv() 两个系统调用，以及封装了这两个系统调用的库函数实现。

JOS IPC 中的消息包括两个部分：一个 32 位的值以及一个可选的页面映射。消息中包含这个页面映射是为了传输更多的数据以及实现进程间共享内存。

进程调用 sys_ipc_recv() 接收消息，调用 sys_ipc_try_send() 发送消息。如果要发送页面映射，则调用时设置 srcva 参数，表示要将 srcva 处的页面映射共享给接收进程。而接收进程的 sys_ipc_try_recv() 如果希望接收页面映射，则会提供一个 dstva 参数。如果发送进

程和接收进程都没有设置参数表示希望传输页面映射，则不传输。内核会在接收进程的 env_ipc_perm 字段设置接收的页面映射的权限。

任何进程都可以发送消息给其他进程，不需要它们是父子进程。这里的安全由 IPC 相关系统调用保障，一个进程不能通过发送消息导致另一个进程奔溃，除非接收消息的进程本身存在 BUG。

4 一些注意点

- 完成作业 15 后，可以发现 stressched 通不过测试，这个有个坑，检查了很久才发现，原来要在 `kern/sched.c` 的 `sched_halt(void)` 中去掉 `//sti` 的注释，因为在 AP 启动完成且获得锁且第一次调用 sched_yield()时，如果发现没有可运行进程，会执行 sched_halt()导致 CPU 处于 HALT 状态。因为我们在 bootloader 中通过 cli 关闭了中断的，所以此时需要开启中断，不然 AP 就一直处于 HALT 状态而不参与调度了。
- 另外，spin 测试不要多加参数如 `CPUS=2`，否则会测试失败，因为当父子进程在不同的 CPU 运行时，此时父进程去销毁子进程会先将子进程设置为 ENV_DYING 状态，而后等子进程调度的时候再自己销毁自己，这会跟要求输出不一样导致通不过测试。
- 一些调试语句要注意输出位置，可能会干扰测试结果，因为作业是根据输出来判定的，最好去掉多余的调试语句来测试。