Lab 03

实验提交

截止时间: 2016/04/28 23:59:59 (如无特殊原因，迟交的作业将损失 50%的成绩(即使迟了 1 秒)，请大家合理分配时间)

请大家在提交的实验报告中注明你的邮箱，方便我们及时给你一些反馈信息。

学术诚信: 如果你确实无法完成实验，你可以选择不提交，作为学术诚信的奖励，你将会获得 10%的分数；但若发现抄袭现象，抄袭双方(或团体)在本次实验中得 0 分。

提交地址：http://cslabcms.nju.edu.cn/

提交格式: 你需要将整个工程打包上传，特别地，我们会清除中间结果重新编译，若编译不通过，你将损失相应的分数（请在报告中注明你实验所使用的 gcc 的版本，以便助教处理一些 gcc 版本带来的问题）. 我们会使用脚本进行批量解压缩. 压缩包的命名只能包含你的学号。另外为了防止编码问题，压缩包中的所有文件都不要包含中文.如果你需要多次提交，请先手动删除旧的提交记录(提交网站允许下载，删除自己的提交记录)，否则若脚本解压时出现多次提交相互覆盖的现象，后果自负. 我们只接受以下格式的压缩包：

- tar.gz
- tar.bz2
- zip

若提交的压缩包因格式原因无法被脚本识别，后果自负。

请你在实验截止前务必确认你提交的内容符合要求(格式、相关内容等)，你可以下载你提交的内容进行确认。如果由于你的原因给我们造成了不必要的麻烦，视情况而定，在本次实验中你将会被扣除一定的分数，最高可达 50% 。

git 版本控制：我们建议你使用 git 管理你的项目，如果你提交的实验中包含均匀合理的 git 记录，你将会获得 10% 的分数奖励（请注意，本实验的 Makefile 是由你自己准备的，你可以选择像 PA 中一样在每一次 make 后增加新的 git 记录作为备份，但是请注意，这样生成的 git log 一般是无意义的，所以不能作为加分项）。为此，请你确认提交的压缩包中包含一个名为 .git 的文件夹。

实验报告要求：仅接受 pdf 格式的实验报告，不超过 3 页 A4 纸，字号不能小于五号，尽可能表现出你实验过程的心得，你攻克的难题，你踩的不同寻常的坑。

分数分布： - 实验主体：80% - 实验报告：20%

解释：

1. 每次实验最多获得满分；
2. git 的分数奖励是在实验主体基础上计算的
3. git 记录是否“均匀合理”由助教判定；
4. 迟交扣除整个实验分数的 50% ；
5. 作弊扣除整个实验分数的 100% ；
6. 提交格式不合理扣除整个实验分数的一定比例；
7. 实验批改将用随机分配的方式进行；
8. 保留未解释细节的最终解释权；
9. 答辩时未能答对问题会扣掉总体 5%~30%的分数。

进程的组织和调度

进程数据结构

如果你在 lab2 中用“散乱在全局”的方式组织你的用户进程的各种信息（比如页目录表）， 那么在 lab3 开始时，你便需要构造一个数据结构用于管理你的用户进程。这个数据结构可以包括以下信息：

- 进程在内核状态下的栈
- 进程的标识符
- 进程的父进程的标识符
- 进程的状态（正在执行、可以执行、正在休眠、不可执行等）
- 进程已经运行的时间片数量
- 进程的页目录表的地址

上述的信息和你后面的实现相关，因为你可以选择自己的调度策略，不同的调度策略会需要不同的信息。

进程数据结构的组织

现在你已经定义好了你的进程数据结构，为了方便起见我们将其称为 PCB 首先，你应该准备好 PCB 池和进程标识符池，这二者不是一一对应的，进程标识符应该体现出父进程 与子进程创建的先后关系。PCB 是可以回收的：当进程出现致命错

误或者正常执行退出时， 你应该回收进程的 PCB；而进程标识符一般都被实现为不可回收的，所以你可以单纯的用一个变量 来表示当前的标识符分配了多少了。

组织 PCB 应该使用什么数据结构？这个你可以自己选择： - 一种比较暴力但是简单的方式是用单纯的数组来管理 PCB， 因为 PCB 里面记录了进程的状态，所以即使他们在同一个数组中，你也可以区分出哪些进程是 不可以执行的，哪些是可以执行的。 - 优雅一点的办法是用几条链表（循环队列）来表示不同状态的进程，这样管理链表也会花费不少精力 - 进阶：为你的进程设计不同的优先级，采用多条队列来管理不同优先级的进程 不管什么样的管理方式，实现才是最终的目标，能实现的策略都是好策略。

进程管理的函数

回忆一下你在 lab2 中做的工作：如果你采用分页，那么你启动的时候静态或是动态的初始化了一个 启动时用的页表，然后你为了获得对整个物理内存的访问权，你在初始化了一个与物理页面 一一映射的页描述符数组。你还初始化了串口输出，中断和各个中断的处理函数等。以上工作都可以 看作是对系统的初始化。

然后你为用户进程分配了一个页目录表，根据它的 elf 信息将它加载到虚存空间中，然后为它 初始化运行时所需的环境（比如正常执行的栈，陷入内核时使用的栈，还有他的 Trapframe）， 最后将 Trapframe 中的寄存器、状态字等信息 pop 到寄存器，用 iret 开始执行用户程序。 但是以后我们面对的是多个进程，你不可能手动对每一个进程重复以上操作， 所以现在你需要对以上过程进行封装，下面举一个封装的例子（但是这并不是限定你们这样做， 只是提供一个示例，而且这个示例可能并不适合你的架构）：

1. 因为现在我们需要用 PCB 来存储进程的信息，所以第一件事肯定是分配一个 PCB

2. PCB 中包含用户的内核栈（陷入内核态的时候使用的栈），而用户最初的 Trapframe 便指向这里， 你现在应该初始化一些 Trapframe 中的信息：在这里将 4 个段寄存器的 DPL 初始化为用户特权级， 将 esp 初始化为你设定的用户栈顶的虚拟地址，别的字段无需关心

3. 为进程分配进程标识符并存到 PCB 中；将程序的父进程的标识符存入 PCB 中（手动 在内核中创建的进程的父进程的标识符可以看做 0，如果你的进程标识符从 1 开始的话）； 如果你不是采用链表来管理不同状态的进程，在这里你应该初始化进程的状态；如果你 后面希望用时间片轮转法来进行调度，那么现在你应该将进程的时间片占用数初始化为 0； 类似的进程的通用属性的初始化都可以在这一步完成。

4. 为进程分配一个页目录表，然后根据 elf 信息将程序装载到虚拟空间。另外，我们上面还说到了用户栈，现在你应该为用户栈所在的虚拟空间至少分配一个物理页

5. 如果你使用链表管理进程，最后你还应该把它插入到可执行的进程的链表中

6. 万事俱备之后，你就可以运行进程了，运行进程仍然是用内联汇编将各个寄存器 pop 出来。为了管理方便，建议你在此之前将进程的状态改为“正在运行”；如果你使用时间片 轮转法调度，那么你应该将进程的“已使用时间片”加 1.

你可以按照以上步骤实现，为每一个步骤写一个函数，甚至可以为其中某一步 封装多个函数；你也可以按照自己的思路封装一整套自己的函数.

tips:

- 为了对付助教的检查，你只需要实现上述的功能就可以了。如果考虑到程序运行结束或是异常终止， 你还应该实现一些函数来回收 PCB 和为进程分配的页面，如果你完成了这些工作且在实验报告中 描述你实现的功能和效果，将会获得一定的加分。

- 我们还需要实现 fork 系统调用，你实现以上功能的时候最好留好相关接口，比如父进程的进程号； 如果你要实现优先级调度，还应该留下进程优先级相关的接口

进程调度

在一个小型的 OS 里面，调度策略的实现是很简单的，下面举几个例子：

1. 最最简单的时间片轮转法实现：PCB 用纯数组管理，调度的时候从当前进程的下一个进程开始遍历数组， 如果进程的状态是可以运行的，那么就将执行它（不要忘记把上一个进程的状态从“正在运行”改变为 “可执行”）。什么时候进行调度呢？你可以在每一个时钟中断到来的时候将进程的“已使用时间片”加一， 当到达一定数量之后，则执行重新调度；另外，你还可以实现一个系统调用 yield 用于进程主动放弃 自己的本次执行机会，这个系统调用看上去没什么 luan 用，不过他可以方便你展示你的调度效果。

2. 队列式的时间片轮转：创建进程的时候，将它的 PCB 加入可执行的队列。每次重新调度的时候， 你只需要执行当前进程的下一个进程即可。这样做需要注意一点：如果你需要实现 sleep 系统调用，则你还需要增加一条“休眠”队列，在每一个时钟中断到来时，将休眠队列上的进程 “已休眠时间”加一，如果完成了“休眠”，则将它加入“可执行”队列。

3. 优先级调度：创建多个优先级队列，创建进程的时候需要指明优先级，以便于将它们插入到指定的 队列。重新调度时，依优先级次序查看不同的队列，搜索下一个进程。如果你使用优先级调度， 那么你还可以实现一个系统调用 nice：用于某个进程降低自己的优先级。 对，这又是一个没什么用的系统调用，不过 Linux 确实有这样的系统调用。

4. 这不是小型 OS 的调度策略，纯属拓展知识： 链接 摘要：

- Implement fully O(1) scheduling. Every algorithm in the new scheduler completes in constant-time, regardless of the number of running processes or any other input.
- Provide good interactive performance. Even during considerable system load, the system should react and schedule interactive tasks immediately. ...

Fork

我们的实验现在已经能够从内核中加载一个用户程序并运行了， 但是现实的内核是支持多个进程并发执行的， 那么我们如何执行第二个、第三个进程呢？ 经典的方法是，内核只主动加载执行名为 init 的用户态程序， 这个程序根据配置好的启动脚本，进一步地创建新的进程，加载系统运行所需要的其他程序。 最终到达提供用户一个可以交互的 shell （终端或者桌面环境）。 此后便主要由用户输入来决定何时创建新的进程，以及加载哪个程序。

由于内核只会在启动时主动创建一个用户进程，而之后创建进程的行为由用户程序来进行决断， 所以我们需要在内核中提供创建进程的系统调用，这便是 fork。

下面我们来具体说下 fork 干了什么事。 粗略地来讲，fork 会创建一个新的进程，并将调用 fork 的进程（下面称作**父进程**）的资源拷贝过去。 具体哪些资源需要拷贝，哪些资源不需要 / 不能拷贝，这要看操作系统的具体策略，但是就目前的实验进度而言， 我们的进程拥有的下面这些资源是要拷贝的：

- 中断现场
- 地址空间映射（页目录，页表）
- 物理页的**内容**

这里涉及到浅拷贝和深拷贝的问题。简单地说，浅拷贝只复制引用关系，深拷贝复制所有内容， 下面这段代码是一个示例：

```
char a[] = "Hello, World!";       // 一个隐含的深拷贝
const char *b = "Hello, World";   // 浅拷贝

char *c = a;                      // 浅拷贝，对 c 中元素的写操作会即对 a 中元素的写操作

char *d = malloc(sizeof(a));
strcpy(d, a);                     // 深拷贝
```

对于 fork，文件描述符、信号量这种本身属于引用型的资源不需要进一步的深拷贝，但是我们现在还没有这些设施，可以说所有的内容都需要深拷贝， 内核栈、页目录、页表、物理页都需要开辟新的空间进行拷贝，而不能只改写指针。 特别地，对于页目录项和页表项，要填写新分配的页框号。 一个例外是只读页，由于它不

会被写，多个进程使用同一个只读页不会产生竞争问题， 所以可以不为它分配新的物理页进行拷贝，而是直接使用原来的页框号填到新分配的页表里。

Fork 的另一个问题是内核栈。 由于新的进程有新的内核栈，所以如果你在内核栈存放了局部变量的地址，进行深拷贝后就会有引用问题。 不过这个问题有一些前置条件：

1. 发生了嵌套中断，这样在两个中断现场之间才会存在临时变量；
2. 你真的将临时变量的地址放到了栈上，这常见于将局部变量地址作为函数参数的场合。

一般来说，发生嵌套中断是可能的，但这往往是保护（General Protection）异常，当然，还有在内核态开中断； 对于前者，我们没理由在出错的情况下进行 fork. 对于后者，虽然在内核态合适的场合开中断是现代操作系统的基本素养， 但是出于 KISS 原则的考虑，我们目前还是建议在整个内核态关闭硬件中断（这样的坏处是并发粒度太粗，响应延迟大）。

基于以上简化，其实我们没必要拷贝整个内核栈，只需要将父进程陷阱帧的内容放到子进程内核栈的顶部就可以了。

以上是 fork 的一些基本信息，下面明确一下为了实现 fork, 你需要做哪些事情：

1. 分配一个新的 PCB 并分配 PID。 怎么分配？可以创建一个固定大小的 PCB 数组，然后用下标做 PID。也可以像管理物理页那样，使用空闲链表进行优化。你应该在实验二中设计说 PCB 的结构体，并且将内核栈以某种形式和 PCB 关联起来。
2. 分配新的页目录，并遍历父进程的页目录，将存在的页目录项复制，但是使用新分配的页表的物理地址来（所以其实只复制了属性……）。 对新分配的页表，递归地执行类似的操作。
3. 在子进程的内核栈顶复制父进程的陷阱帧，伪造中断现场。

Fork 并不要求先从父进程返回还是先从子进程返回。如何从进程 A 切换到进程 B，则是进程调度相关的内容。

延伸阅读

- 2012 级操作系统实验关于 fork 等系统调用的说明见这里。
- 关于 Linux 下的 fork 系统调用，通过 `man fork` 了解。
- 查询 Wikipedia 和 OSDev 关于 fork 的词条。
- JOS 关于写时拷贝（Copy On Write, COW） fork 的介绍见这里。

下面是一个关于 fork 的趣味题（来源），可以帮助你进一步理解什么叫拷贝父进程的资源。

```
#include <stdio.h>
#include <unistd.h>

int main()
{
  int i;
  for (i = 0; i < 2; i++) {
    fork();
    printf("X");
  }
}
```

- 请问上面的代码会输出几个 X？
- 如果给 "X" 加上换行符呢？
- 如果加上换行符并将输出重定向到文件呢？（这个和 fork 没关系了）

Exit

Exit 系统调用用于销毁进程，它需要考虑的内容不多，只需要在代码层面实现物理页（相关的物理页信息，包含页表和页目录的）、 内核栈空间、PCB 以及其它一些资源的释放。PCB 的释放同时意味着将它从可调度进程集合中移除。

更多信息可参见 `man exit` 和 2012 级的相关内容（见上）。

Sleep

我们已经可以创建和销毁进程了，相信上过理论课之后你一定知道，进程的管理和控制还包括阻塞、唤醒（还有挂起和激活，但是我们并不打算在我们的实验中引入）。 你需要实现一个函数：

```
void sleep(int);
```

这个函数的作用相当于一个阻塞原语，进程通过调用这个函数来阻塞自己，然后等待一个事件被唤醒。 进程阻塞的基本步骤感谢我这个课本搬运工： 1.停止进程执行 2.保存现场信息到 PCB 3.修改进程 PCB 中有关的状态内容，如将状态由运行态改为等待态等 4.将进程移入相应事件的等待队列 5.转向进程调度

简单来说，你可以往 PCB 结构中加些内容，用以保存进程状态；或者说你也可以实现一个阻塞队列，将运行态和等待态的进程区别开来。当进程调用 sleep 函数时，你应该根据自己选择的实现方式进行相应的操作，完成进程的状态切换。

为了让想要睡眠的进程尽快地调用 sleep 函数把自己放入阻塞队列，你需要实现一个系统调用，参数为阻塞的时钟周期，调用后用户程序就能马上地让自己睡过去了。