**一、思考题**

**Thinking 4.1 思考并回答下面的问题：**

* **内核在保存现场的时候是如何避免破坏通用寄存器的？**

调用SAVE\_ALL宏定义，将数据以Trapframe结构体形式存在sp为开头的空间中，即压入内核栈。

* **系统陷入内核调用后可以直接从当时的$a0-$a3 参数寄存器中得到用户调用msyscall 留下的信息吗？**

可以，这几个寄存器为通用寄存器，从存入到使用并没有发生变化

在handle\_sys中维护了这几个寄存器的值，使得内核调用中这几个寄存器以及其他参数的传值符合MIPS的约定。

* **我们是怎么做到让sys 开头的函数“认为”我们提供了和用户调用msyscall 时同样的参数的？**

以汇编的方式显式地把函数地参数值“转移”到内核空间，在当下内核栈上分配能够存储这6个参数的空间并将6个参数存入栈中，同时维护好a0-a3寄存器的值

* **内核处理系统调用的过程对Trapframe 做了哪些更改？这种修改对应的用户态的变化是？**

EPC寄存器值+1：执行下一条指令

将sys\_\*函数的返回值，也就是v0寄存器的值存入Trapframe对应第2个寄存器的位置，在用户态下就可以使用这个内核态中的返回值。

**Thinking 4.2 思考下面的问题，并对这两个问题谈谈你的理解：**

* **子进程完全按照fork() 之后父进程的代码执行，说明了什么？**

父进程与子进程的fork之后所执行的代码段是相同的

* **但是子进程却没有执行fork() 之前父进程的代码，又说明了什么？**

fork（）只将其之后的代码段赋给了子进程让其执行

**Thinking 4.3** **关于fork 函数的两个返回值，下面说法正确的是：**

C

A. fork 在父进程中被调用两次，产生两个返回值

B. fork 在两个进程中分别被调用一次，产生两个不同的返回值

C. fork 只在父进程中被调用了一次，在两个进程中各产生一个返回值

D. fork 只在子进程中被调用了一次，在两个进程中各产生一个返回值

**Thinking 4.4** **如果仔细阅读上述这一段话, 你应该可以发现, 我们并不是对所有的用户空间页都使用duppage 进行了保护。那么究竟哪些用户空间页可以保护，哪些不可以呢，请结合include/mmu.h 里的内存布局图谈谈你的看法。**

UTOP一下的用户空间页可以保护

UTOP以上的用户空间，包括UENVS,UPAGES,UVPT,ULIM等，用户程序均不可写，也就不必进行保护。

**Thinking 4.5** **在遍历地址空间存取页表项时你需要使用到vpd 和vpt 这两个“指针的指针”，请思考并回答这几个问题：**

* **vpt 和vpd 的作用是什么？怎样使用它们？**

在课下查找资料时我发现有学长的博客里称这两个数组的作用为回环搜索；

即给定一个虚拟地址，构造出其页目录项和页表表项

假设查询的虚拟地址为va=PDX|PTX|OFFSET

要得到对应的页目录项:vaddr=UVPT[31:22]|UVPT[31:22]|PDX|00

要得到对应的页表项:vaddr=UVPT[31:22]|PDX|PTX|00

而我们的vpt与vpd就发挥了这样的作用，

vpd记载对应的页目录项，vpt记载对应的页表项

* **从实现的角度谈一下为什么能够通过这种方式来存取进程自身页表？**

在实现中我们使用指针\*vpt与\*vpd来找对应的页表项与页目录项

在entry.S中有如下定义

.globl vpt

vpt:

.word UVPT

.globl vpd

vpd:

.word (UVPT+(UVPT>>12)\*4)

* **它们是如何体现自映射设计的？**

在对于vpd地址的定义中，使用了(UVPT+(UVPT>>12)\*4)的设计，即计算出UVPT以上的地址中映射到UVPT本身的偏移，这体现了自映射的设计。

* **进程能够通过这种存取的方式来修改自己的页表项吗？**

不能，进程只能访问相应的页表项，没有修改页表项的权限。

**Thinking 4.6 page\_fault\_handler 函数中，你可能注意到了一个向异常处理栈复制Trapframe 运行现场的过程，请思考并回答这几个问题：**

* **这里实现了一个支持类似于“中断重入”的机制，而在什么时候会出现这种“中断重入”？**

发生写时复制特性下的缺页异常后。

（1）正常的程序运行中出现缺页异常对应程序中else情况

此时tf中保存的栈顶指针(即tf->regs[29])应该是在USTACKTOP之下,即normal user stack区域

在该情况下要进入缺页异常处理应该设置栈顶指针为异常处理栈指针,即tf->regs[29] = curenv->env\_xstacktop - sizeof(struct Trapframe)

（2）在缺页异常处过程中触发缺页异常对应程序中if情况

此时tf中保存的栈顶指针(即tf->regs[29])应该是在UXSTACKTOP之下,即user exception stack区域

在该情况下要进入缺页异常处理应该将异常处理栈的栈指针下移即可,即tf->regs[29] = tf->regs[29] - sizeof(struct Trapframe)

* **内核为什么需要将异常的现场Trapframe 复制到用户空间？**

鉴于微内核的设计理念，对于写时复制的缺页中断处理，真正的处理过程是由用户进程自身去完成的，而这个异常的抛出处于内核态，故需要将现场复制到用户空间。并在最后将异常处理程序入口设置为tf中的epc，使得异常处理完成恢复现场时重新开始运行的地方是处理缺页异常的代码，即env\_pgfault\_handler。

**Thinking 4.7** **到这里我们大概知道了这是一个由用户程序处理并由用户程序自身来恢复运行现场的过程，请思考并回答以下几个问题：**

* **用户处理相比于在内核处理写时复制的缺页中断有什么优势？**

微内核优势，将系统服务层分离出来，提供了更好的可拓展性和更加有效的应用环境。

若需要将这个缺页中断的过程做修改，只需要改动用户程序中的模块，不需要改变整个操作系统。

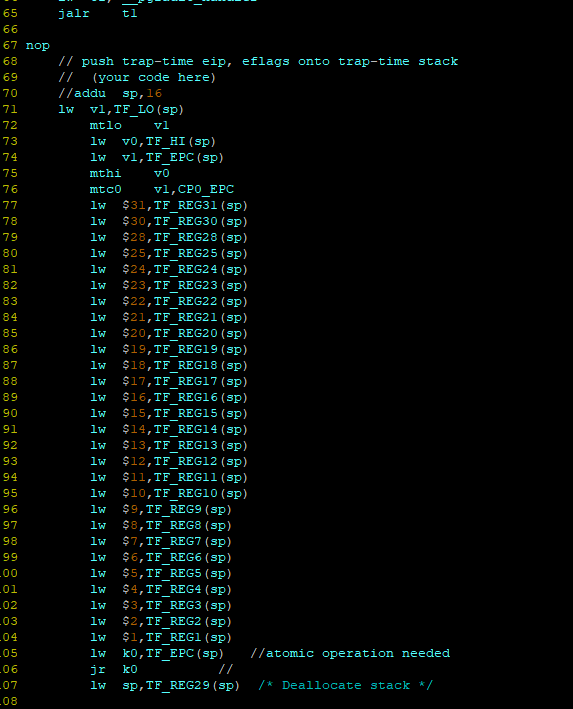
各个服务的地址空间独立，不影响其他模块，一个服务组件的失效不会导致整个系统的崩溃。

可移植性强。

减小内核所占的内存，精简内核。

* **从通用寄存器的用途角度讨论用户空间下进行现场的恢复是如何做到不破坏通用寄存器的？**

在\_\_asm\_pgfault\_handler中调用pgfault函数进行异常处理，结束后返回，在汇编中显示地进行了恢复现场操作，如下图



从通用寄存器的用途考虑，恢复现场时，在从上下文环境中恢复31个寄存器的值之前，汇编只用到了sp、v0、v1三个寄存器，v0-v1用于存放表达式的值或函数的整形、指针类型返回值，sp寄存器存放栈指针，并没有对其进行修改。接下来使用lw之类从保存的上下文环境中恢复原本的31个寄存器的值。

**Thinking 4.8 请思考并回答以下几个问题：**

* **为什么需要将set\_pgfault\_handler 的调用放置在syscall\_env\_alloc 之前？**

在子进程运行过程中不需要再注册缺页处理函数。

* **如果放置在写时复制保护机制完成之后会有怎样的效果？**

会出错。在set\_pgfault\_handler中分配的是异常处理栈，而duppage是处理写时复制的缺页中断的，需要用到这个异常处理栈。

* **子进程需不需要对在entry.S 定义的字\_\_pgfault\_handler 赋值？**

不需要，这个字被定义为一个全局的word，所有进程的缺页处理函数都可以用到这个字。

**二、实验难点**

**(1)系统调用部分**

系统调用在mips中会被作为一种异常处理，发生系统调用后，会触发一系列的异常处理程序。

首先，硬件发挥作用，CPU进行一下行为

设置epc

禁止中断

将异常产生原因填入cause寄存器

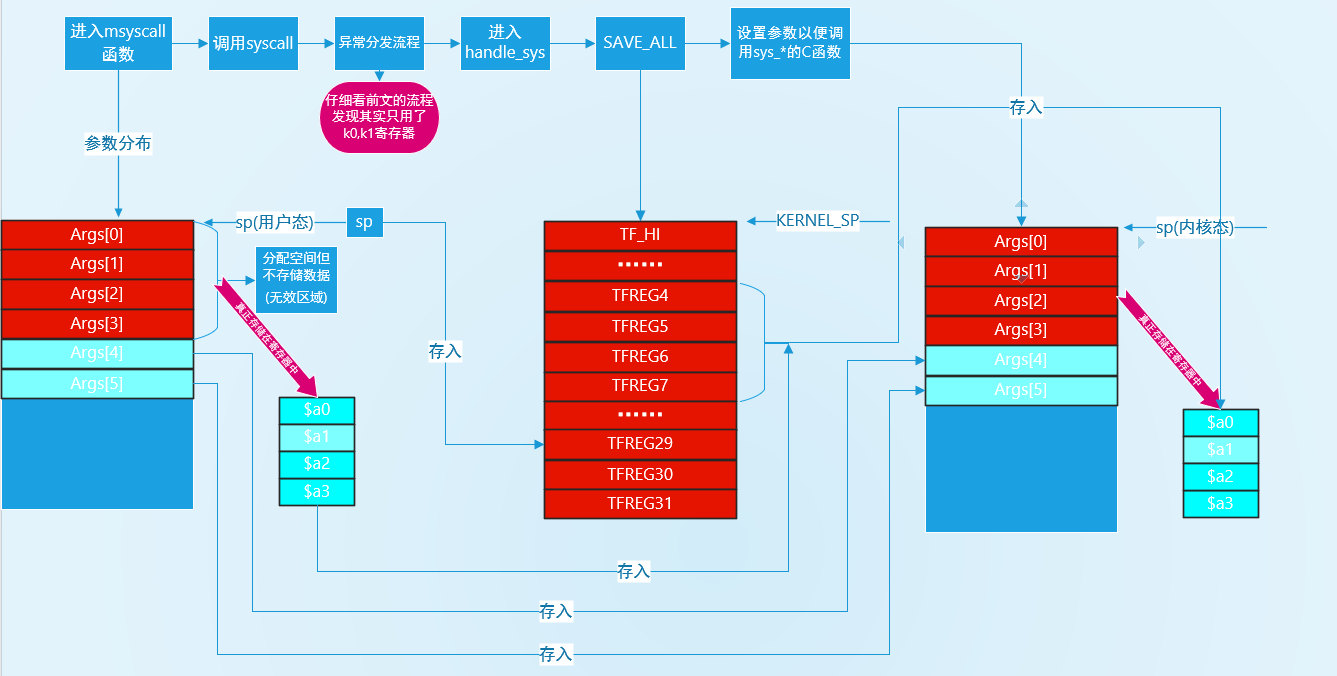
设置pc值为异常处理程序入口，(0x80000080)

然后进入异常分发程序，根据cause寄存器的值判断异常类型并跳转到对应的处理程序

在lab4的系统调用中，系统调用异常码为8，用户态的syscall\_\*，会调用msyscall，触发系统调用异常，陷入内核态，跳转到handle\_sys函数。

在handle\_sys这个系统调用异常处理程序中就主要进行一些保存现场，设置pc值之类的与其他异常处理类似的处理。还要通过系统调用号来判断跳转到哪一个内核态函数。其中比较重要的，是对于传入参数的处理。syscall\_\*函数与内核态下的sys\_\*函数是一一对应的，在异常处理程序中就需要对传入的参数进行维护，具体要求是，前4个参数存储在$a0-$a3四个寄存器中，剩下的参数存在从偏移量16开始的栈区空间中，至于前4个参数的空间，则只是预留，不做改动。由于我对于这四个寄存器以及栈区的维护过程的不熟悉，在lab4-1的课上光荣牺牲了……

在课下我对mips函数调用传参的处理做了更多的了解，贴上一张博客里看到的，对我理解有很大帮助的图，时刻提醒自己。



**（2）fork部分**

这一部分涵盖了很多的内容，包括缺页中断处理、用户态与内核态的切换、异常栈的分配等。由于涵盖的范围广，对于先前的代码的正确性、以及对于当下代码的调试，都提出了极高的要求。

对缺页中断Page Fault的处理，要填写的代码量不多，重要的是对缺页中断处理过程的熟悉与理解。Lab4-2课上测试的Extra题就考察了这个。

难点在于父进程需要在唤醒子进程之前遍历进程的大部分用户空间页，并使用duppage函数加以合适的PTE\_COW标志位。在遍历的过程中，需要大量进行系统调用以及针对内存的插入操作，这就要求了先前几个实验内容的正确性。在我第一次写lab4第二部分实验时我就遇到了page\_insert中报告异常的问题，耗时十几个小时寻找bug未果后我从lab3开始重写代码，这个问题才消失，估计应该是lab2之后pullLab3代码时取错了版本发生的问题。而对于duppage函数中权限位的判断，需要我们对写时复制机制非常熟悉。

fork中还需要处理写时复制机制导致的缺页中断。硬件上，CPU会在根据虚拟地址查询页表时发现对应得物理页框被COW位保护，则记录异常类型并跳转到异常分发代码。接下来时软件得工作，在异常分发代码判断次为缺页异常，跳转到写时复制缺页异常处理代码，具体的缺页重填代码在用户态实现。也就是fork.c文件中的pgfault函数。

**（3）进程间通信部分**

我们的小操作系统中通过进程共享的内核空间中的进程控制块结构envs[]来实现进程间的通信。

与进程通信有关的就是系统调用函数中的ipc\_recv与ipc\_can\_send函数，在Lab4中对进程间通信的要求较为简单，代码量也不大，个人认为这一部分只要理解两个进程间共享内存的机制就成。

**三、体会与思考**

Lab4是迄今为止我花时间最长的一次操作系统实验。五一假期一半以上的时间都处于写bug与debug的过程中。在lab2与lab3留下来的祖传bug让我痛不欲生。但在一次又一次地检查先前代码地过程中我对整个操作系统的架构也更为熟悉。对包括各种栈区的设置，用户态与内核态的转化、进程上下文环境的保存等的了解也更为深入。

**四、指导书建议**

可以对于一些重点的模块，比如系统调用、中断处理等，添上一些类似于架构图的思维导图，帮助我们更好地理解操作系统运作的流程。