**浙江大学实验报告**

课程名称： 计操作系统 实验类型： 综合

实验项目名称： 实现fork机制

学生姓名： 专业： 计算机科学与技术 学号：

实验日期： 2023 年 12 月 13 日

1. **实验内容**

1.1整体背景

在本实验中，我们主要需要实现fork()函数，即进程创建子进程的功能。fork的关键在于进程状态及相应内存的复制。我们不仅需要完整地拷贝一份父进程页表以及VMA结构中记录的用户态内存，还需要复制内核态的寄存器状态和内核态的内存，并在最后，将fork的task“伪装”成是因为调度才进入时间片轮转Ready Queue。

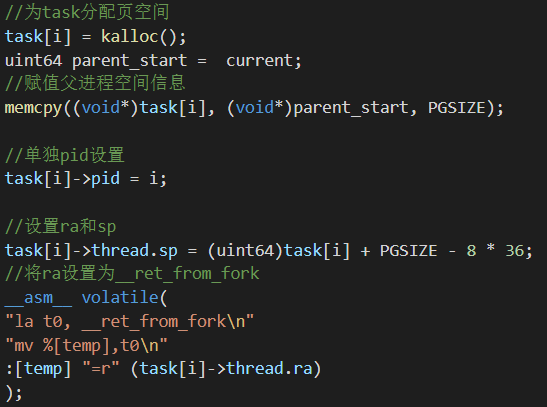
对于拷贝页表而言，我们需要遍历页表中映射到父进程用户地址空间的页表项，这些应该由父进程专有的页，如果已经分配并映射到父进程的地址空间中，我们就需要额外分配空间，并从原来的内存中拷贝相应的数据到这页新分配的空间之中，并将这一新页映射到子进程的地址空间中。

另外，在fork新进程时，我们要在task数组中寻找一个空闲的位置。我们可以使用最简单的管理方式，将原本的task数组的大小直接设置为16，考虑IDLE进程和初始化时新建进程各占一个，并将剩余的14个task全部赋值为NULL。在后续的实现中判断，如果task[pid] == NULL，说明这个pid对映的task还没有被使用，因而可以作为新进程的pid；此时再将task[pid]赋值为新的struct task\_struct\*结构。

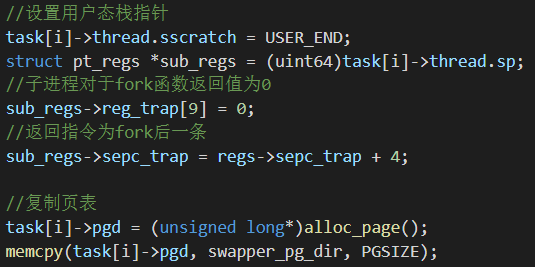
1.2 sys\_clone函数的实现

首先，我们需要遍历task数组，寻找空闲的task位置，若task均已被使用，则返回错误信息。

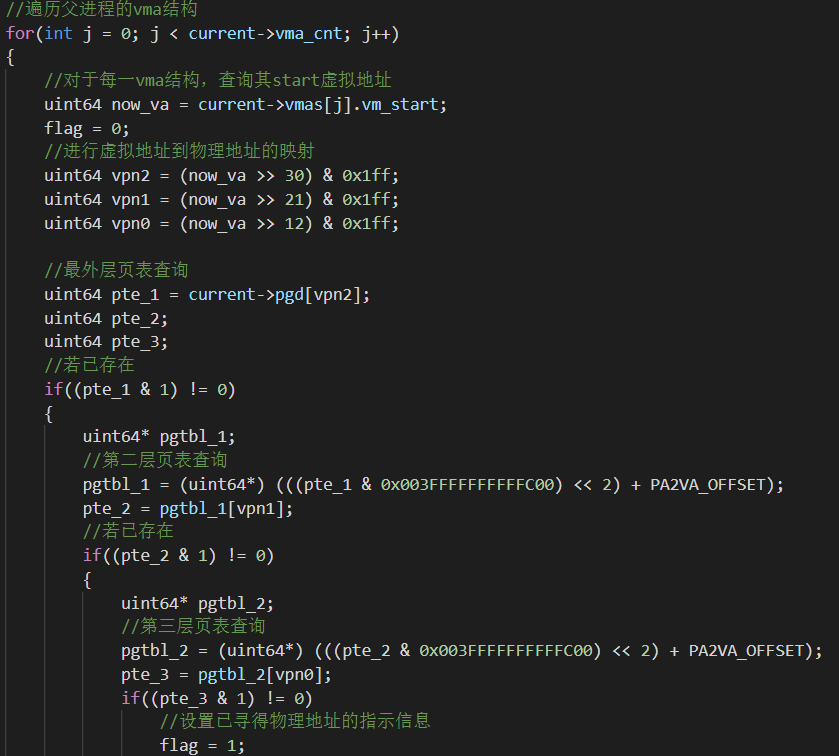
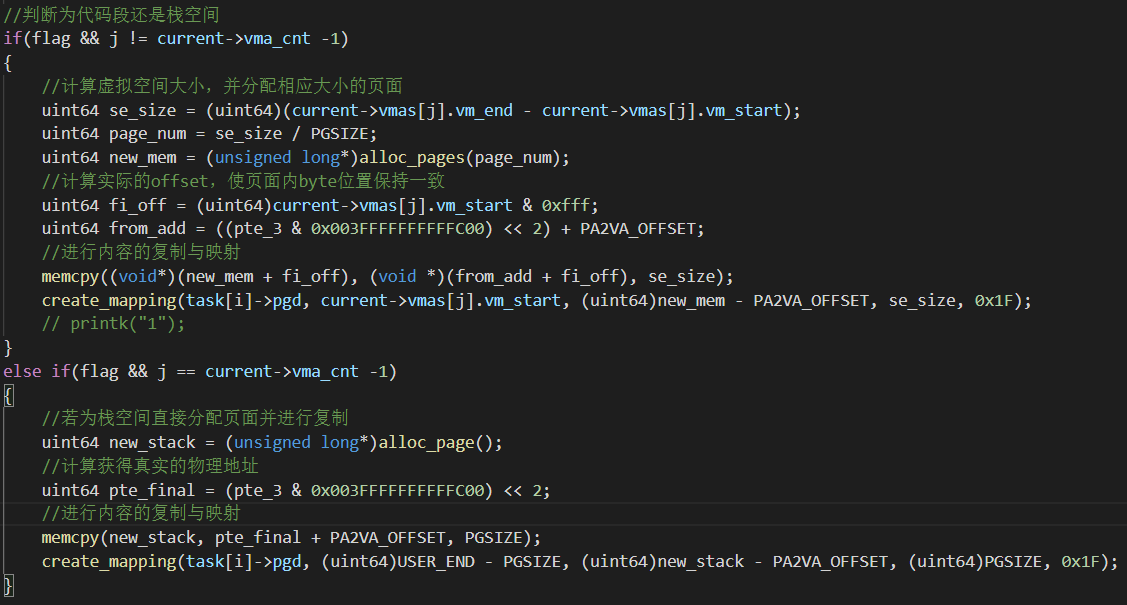
接着，我们需要为选中的task分配页面，此行为等价于新进程的建立；并使用memcpy函数将父进程的task页面所有信息复制到新分配的页面中（包括task\_struct结构体数据与内核态栈数据）。在此过程中，我们需要修改该页面中的部分数据，以表示子进程的独特性。首先，我们需要把该进程task\_struct结构中的pid信号设置为task对映的序号，并将其ra寄存器设置为\_\_ret\_from\_fork标签，使子进程第一次被调度时可以进行正常内核态栈指针操作；另外，需要将内核态栈指针同样设置至正确位置（含有一定的偏移量）。



接着，我们需要找到子进程对映的pt\_regs结构体，将其中的a0寄存器设置为0，表示子进程对于fork函数的返回值是0.另外，我们需要将该子进程的用户态栈指针指向user\_end位置，并将其返回位置设置为fork函数之后（sepc寄存器相关）；接着，复制idle线程最初建立的页表。



完成上述操作后，我们还需要处理用户栈以及用户态程序的拷贝与映射，我们可以借助vma结构链表来处理这一部分的内容。我们需要遍历这一vma结构链表，并借助每一vma结构体中的vm\_start虚拟地址，搜索其在父进程页表中的记录状况。若该虚拟地址未被页表记录，说明该页尚未被父进程使用因而尚未分配，子进程可以暂时不加以拷贝与映射。若该地址已经被页表记录，说明该页已被父进程使用；这时，若改页为用户态栈信息，则子进程直接拷贝映射即可；若改页为代码段信息，子进程则需要判断页面数量，vm\_start偏移字节数等信息，介导当前页面从父进程至子进程的正确拷贝与映射。具体代码实现如下：

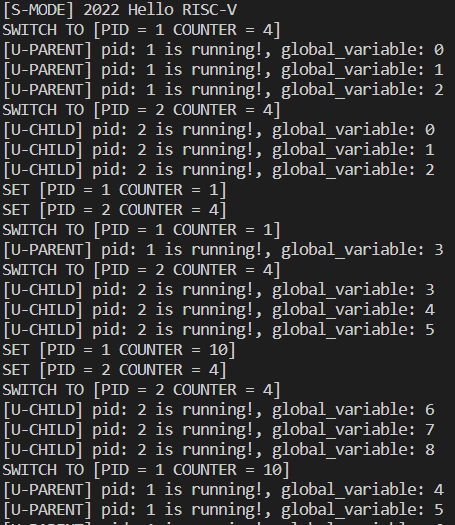


1.3实验测试结果

1.3.1 main(1)测试

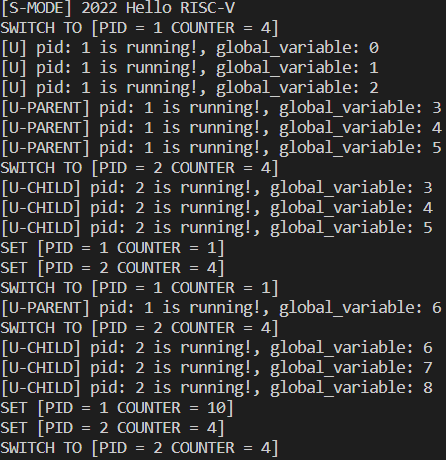
Main(1)测试为fork函数最基础测试，仅测试父进程调用fork函数后的正确返回地址与返回值的设置正确性，并判断子进程能否返回正确运行起始地址与返回值的正确性。

测试结果如下：



由结果可见，经过fork函数，测试运行了两个进程，且父进程与子进程之间可以正常的进行时间片轮转运算；且父进程与子进程均维护统计了自己的运行变量（均从0开始输出）。

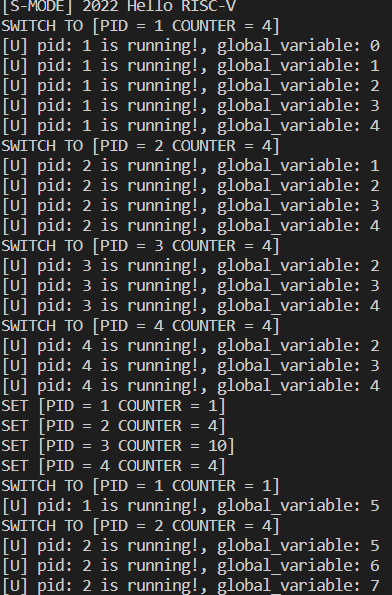
1.3.2 main(2)测试

Main(2)测试在main(1)测试的基础上，使父进程优先运行一段时间后，再进行fork操作；以此判断子进程是否正确复制了父进程中的用户栈数据。测试结果如下：

由结果可以看出，虽然fork延后执行，但父进程与子进程之间仍然可以正常的进行时间片轮转运算。且对于父进程而言，其管理增加的变量依旧从0开始输出；但对于子进程而言，因为其拷贝了父进程的用户态栈信息，所以其继承了父进程的输出变量数据，因而输出从3开始。

1.3.1 main(3)测试

Main(3)测试在main(1)/main(2)测试的基础上，测试子进程是否可以正确的调用fork函数，由此来判断子进程的sp信息是否设置正确。因为main(3)调用了两次fork函数，因而最终会产生4个进程调度运行。测试结果如下：



由结果可以看到，我们最终确实只生成了四个进程，且这四个进程间可以相互调度与运行。其中进程1为父进程，其fork得到了进程2与进程3，进程2又fork得到了进程4。从测试结果中我们也可以看出，在进程1优先执行一次输出的情况下，进程2输出从1开始；在进程1输出两次后再fork的进程3，输出从2开始；在进程2被创建与输出一次后再fork得到的进程4，输出也从2开始，这从一定程度上说明了我们fork函数设计的正确性。

1. **思考题**
2. 参考task\_init创建一个新的task，将的parent task的整个页复制到新创建的task\_struct页上。这一步复制了哪些东西?

因为对于每一个task而言，我们只是为其分配了一个page，并从地址低处开始存放该task对映的task\_struct结构，其中包括进程state、进程剩余时间片、进程对映vma结构数量与记录等信息；而从该页的地址高处开始，我们存放了该进程内核态的栈记录信息，这些信息将在\_trap标签处被写入。

因而在复制整个页的过程中，我们复制了以上的所有信息；因为复制发生于trap\_handler函数内且trap\_handler函数在\_trap标签内被调用，因而内核态的栈信息也已经被记录而产生复制。

1. 将 thread.ra 设置为\_\_ret\_from\_fork，并正确设置thread.sp。仔细想想，这个应该设置成什么值？可以根据 child task 的返回路径来倒推。

Thread.sp的数值应该被设置为“对映task进程的地址 + Page size大小 - sp固有偏移量（pt\_regs结构体大小，即8 \* 36 bytes）”。

首先，我们可以判断，thread.sp对映的是当前task在内核态的栈指针，根据我们一般的设计，内核态栈指针应该指向分配于task的page的顶端。但依照fork进程的首次调度返回顺序，我们可以发现，进程会返回\_\_ret\_from\_fork标签处，而这个标签位于\_trap标签的后半段，最后伴有sp += 8 \* 36 bytes的操作，而此时进行该操作的sp指针即为thread.sp指针（该指针于\_\_switch\_to标签处，进行进程调度上下文切换时取出），因而thread.sp需要被设置为“task地址 + Page size - pt\_regs结构体大小”。

3.利用参数 regs 来计算出 child task 的对应的 pt\_regs 的地址，并将其中的 a0, sp, sepc 设置成正确的值。为什么还要设置 sp?

因为此处设置的sp与思考题2中所述的sp并非一致。在思考题2中，我们设置的sp为内核态sp，即在fork子进程首次调度至\_\_ret\_from\_fork标签处时，可以正确的获取（拷贝父进程而得到的）内核态栈空间中的寄存器存储信息；而在此段的最后，我们需要将sp寄存器与sscratch寄存器的数值进行互换，表示返回用户态程序时用户态栈指针与内核态栈指针的互换；因而在设置内核态栈指针之外，我们还需要设置用户态栈指针，对映于内核态中sscratch寄存器。

**三、心得**

操作系统实验完结撒花！一直认为操作系统相关实验难度较大，又接近于底层，也不太方便进行debug的工作；至此自己一点一点完成了操作系统的全部实验任务，成就感满满也受益匪浅。

但在本实验的实现过程中，我也还是遇到了些许问题，困扰良久。首先是遗留的立即数取用问题，在先前的实验中，因为虚拟化的开启与boot\_stack\_top等标签初始化使用的抵牾，我选择直接使用立即数来控制这一标签的具体物理位置；这在程序较小，标签较少时并不产生问题，但随着标签数量的增加以及程序规模的变大，boot\_stack\_top这一具体的标签位置，可能会发生一定的偏移，因而导致实验崩溃。

另外，对于fork得到的进程，在首次调度时，我们需要小心的修改trap\_handler函数中部分的内容，使其与直接返回\_\_ret\_from\_fork标签处的汇编代码行为保持一致。首先，我们需要明确，在返回\_\_ret\_from\_fork标签后，我们要从内核态栈中读取相应数据，存入当前进程使用的寄存器中；而这些寄存器的数据来源，是复制父进程的内核态栈存入数据，因而对于sepc寄存器（ecall返回地址）而言，我们需要在trap\_handler进行专门的设置。 另外，当前进程对映的sp寄存器的数值也需要进行特别的处理，不能再选择从内核态栈中读出，这会导致栈指针的指向错误（错误指向父进程的内核态栈空间），从而在连续调用两个fork()函数时产生进程不断fork的情况。