+ -------------------------- +

| CS 140 |

| 项目2：用户程序|

| 设计文件|

+ -------------------------- +

---- GROUP ----

>>填写小组成员的姓名和电子邮件地址。

王子正<wangzz0316@foxmail.com> 贡献百分比90%

王小雨<869833423@qq.com> 贡献百分比10%

查克能<caklen@163.com>

姚树基<871035123@qq.com>

----前提----

>>如果您对提交的内容有任何初步意见，请注意该

>>助教，或额外的信用，请在这里给他们。

>>请引用您在咨询期间参考的所有离线或在线资源

>>准备您的提交内容，而不是Pintos文档，课程

>>文本，讲义和课程人员。

参数传递

===============

---- ----数据结构----

怎么写请见：[http://web.stanford.edu/~ouster/cgi-bin/cs140-spring20/pintos/pintos\_1.html#SEC8](http://web.stanford.edu/~ouster/cgi-bin/cs140-spring20/pintos/pintos_1.html" \l "SEC8)

http://web.stanford.edu/~ouster/cgi-bin/cs140-spring20/pintos/pintos\_9.html#SEC142

>> A1：在此处复制每个新的或更改的`struct'或

>>`struct'的声明成员，全局或静态变量，“ typedef”或

>>枚举。用25个字以内的单词来确定每个单词的目的。

----process.c----

static struct list read\_list; //该列表保存所有准备阅读的元素

static struct list wait\_list; //保存所有读取请求的列表

struct read\_elem{

int pid; // 进程id

enum action action;

struct list\_elem elem; //列表元素

int value; //列表的值，用来判断是否需要创建读取请求

};

----process.h----

/\*\*file descriptor\*/struct file\_descriptor{ int fd; struct list\_elem elem; //描述符列表的列表元素

struct file\* file; //打开文件};struct process{ struct list\_elem elem; // 同上 int thread; // 进程id};

----thread.h----

struct thread //这里只有参数传递部分用到的数据 { /\* Owned by thread.c. \*/ tid\_t tid; /\* 进程id \*/ enum thread\_status status; /\* 表示进程状态 \*/ uint8\_t \*stack; /\* 保存堆栈指针。 \*/

/\* Shared between thread.c and synch.c. \*/ struct list\_elem elem; /\* 列表元素\*/ #ifdef USERPROG

tid\_t parent\_id; /\* 变量parent pid (tid) \*/ uint32\_t \*pagedir; /\* 页面目录 \*/ struct list children; /\* 子进程\*/ int exit\_status; struct file \*executable; /\* 为线程的可执行文件\*/ #endif

};

----算法----

>> A2：简要描述如何实现参数解析。

>>您如何安排argv []的元素以正确的顺序排列？

>>如何避免堆栈页面溢出？

如何实现参数传递：在process.c的start\_process()函数中使用结构体intr\_frame，用堆栈的指针实现。

argv[]的顺序：栈顶指针esp是不断减小的，也就是从后往前获取。用这种方法就能依次获取argv,argc和返回地址。

避免堆栈页面溢出：esp指针不合法直接退出进程。

----理由----

>> A3：为什么Pintos实现strtok\_r（）但不实现strtok（）？

strtok()函数将每次调用后的字符串位置保存在一个函数内部的静态局部变量中，如果有多个线程调用这个函数，有可能出现冲突。

而strtok\_r()使用自定义指针指向切好的字符串，是线程安全的

>> A4：在Pintos中，内核将命令分为可执行文件名

>>和参数。在类似Unix的系统中，shell进行

>>分离。至少确定Unix方法的两个优点。

优点：1.分离命令可以用管道 2.可以检查该文件是否存在（用文件名）

系统调用

===========

syscall.c

---- ----数据结构----

>> B1：在此复制每个新的或更改的`struct'或

>>'struct'成员的声明，全局或静态变量`typedef'或

>>枚举。用25个字以内的单词来确定每个单词的目的。

----syscall.c----

struct fd\_entry{

int fd; /\*文件描述符\*/

struct file \*file;

struct list\_elem elem; /\*同上\*/

struct list\_elem thread\_elem; /\*进程拥有的文件描述符列表的列表元素\*/

};

----thread.h----

struct thread { /\* Owned by thread.c. \*/ tid\_t tid; /\* 进程id \*/ enum thread\_status status; /\* 表示进程状态 \*/ char name[16]; /\* 无特殊意义，用途为debug \*/ uint8\_t \*stack; /\* 保存堆栈指针。 \*/ int priority;

struct list\_elem allelem; /\* 所有线程列表的列表元素。\*/ /\* Shared between thread.c and synch.c. \*/ struct list\_elem elem; /\* 列表元素\*/ #ifdef USERPROG /\* Owned by userprog/process.c. \*/ tid\_t parent\_id; /\* 变量parent pid (tid) \*/ uint32\_t \*pagedir; /\* 页面目录 \*/ struct list children; /\* 子进程\*/ struct list fd\_list; /\* 所有file\_descriptor的列表\*/ int exit\_status; struct file \*executable; /\* 为线程的可执行文件\*/ #endif /\* Owned by thread.c. \*/ unsigned magic; /\* 检测堆栈溢出 \*/ };

>> B2：描述文件描述符如何与打开的文件关联。

>>用syscall.c打开的文件的文件描述符都是一一对应的。

>>文件描述符在整个操作系统中还是在单个进程中是唯一的？

文件描述符在整个操作系统中是唯一的。在syscall.c中建立了一个文件链表file\_list，而不是在单独的进程中。

----算法----

>> B3：描述用于从

>>内核读取和写入用户数据的代码。

READ：先检查指针fd,buffer,size是否有效（无效直接退出）。获取锁并调用read()函数。在read()函数中，如果是 (fd == STDIN)就是键盘输入，要不就是用fd编号获取文件，判断文件指针是否为空，非空就调用file\_read()并返回状态。最后释放锁。

WRITE：先检查指针fd,buffer,size是否有效（无效直接退出），再检查buffer指针是否有效。获取锁并调用write()函数。在write()函数中，如果是 (fd == STDOUT)就是写入控制台，要不就是用fd编号获取文件，判断文件指针是否为空，非空调用file\_write()从缓冲区写入文件并返回状态。最后释放锁。

>> B4：假设系统调用导致整页（4,096字节）的数据从用户空间复制到内核。 可能导致的对页表的检查次数最少（例如最大）（例如，对pagedir\_get\_page（）的调用）？ 对于仅复制2个字节的数据的系统调用该怎么办？ 这些数字是否还有改进的余地，还有多少？

整页数据：次数最少为1。不连续数据最大为4096.

2字节：最少为1，最大为2

>> B5：简要描述“ wait”系统调用的实现

>>以及它如何与流程终止交互。

直接调用process\_wait()。

需要判断是不是调用者的孩子，如果不是就终止。使用is\_child()函数判断。

>> B6：

由于错误的指针值，对用户指定地址处的用户程序存储器的任何访问都可能失败。 这样的访问必须导致进程终止。 系统调用中充斥着此类访问，例如 “写入”系统调用需要从用户堆栈中读取系统调用号，然后是该调用的三个参数中的每个参数，然后是任意数量的用户内存，任何这些都可能在任何时候失败。 这带来了设计和错误处理问题：如何最好地避免在错误处理的混乱中模糊代码的主要功能？ 此外，当检测到错误时，如何确保释放所有临时分配的资源（锁，缓冲区等）？ 在几段中，描述您用于管理这些问题的策略。 举个例子。

如何避免：

例如：read操作。

函数会先检查是否为无效指针（调用检查函数），然后检查缓冲区的开始和结束指针是否无效。检测到错误时，调用exit()函数直接关闭fd对应文件并清空文件链表。

---- SYNCHRONIZATION同步 ----

>> B7：如果加载新的可执行文件

>>失败，“ exec”系统调用将返回-1 ，因此在新的可执行文件完成

>>加载之前，它无法返回。您的代码如何确保这一点？负载

>>成功/失败状态如何传递回调用“ exec”的线程？

检查缓冲区指针和文件名。

>> B8：考虑将父进程P与子进程C一起使用。当P在C退出之前调用wait（C）时，如何确保正确的同步并避免争用情况？ 在C退出之后？ 如何确保在每种情况下都释放所有资源？ 在C退出之前P何时不等待终止而如何呢？ 在C退出之后？ 有什么特殊情况吗？

---- RATIONALE 理由----

>> B9：为什么选择以

这种方式从>>内核实现对用户内存的访问？

>> B10：

对于文件描述符，您可以看到设计的哪些优缺点？

>> B11：默认的tid\_t到pid\_t映射是身份映射。

>>如果您进行了更改，您的方法有什么优势？