OS 实践项目 2 设计与实现文档

4p 小组
10 年 12 月 20 日

内容目录

1 小组人员组成与分工

1.1 小组成员

4p 小组由 4 名 08cs 成员组成，列表如下

陈歌，C，<olivia200705@126.com>

李梦阳，L，<mayli.he@gmail.com>

王盛，W，<wstnap@gmail.com>

肖天骏，X，<xiaotj1990327@gmail.com>

开发代号是每人姓氏的首字母

1.2 任务分工

任务分工由一系列表格组成，并且有较强的松散自由度。

日期	内容	人员	结果
11.27	阅读需求文档，初步规划任务	**CLWX**	0
12.6	提出并实现第一题解法	**CW**	延期 12.9
12.6	提出第二题解法	**X**	延期 12.9
12.6	设计第三题	**L**	完成
12.9	初步实现 Syscall	**L**	完成
12.10	代码上传合并	**WXL**	完成
12.12	调试测试样例	**X**	延期 12.13
12.13	整理代码并提交	**W**	完成
12.18	文档清理上传	**CLWX**	完成
12.20	文档合并	**CL**	完成

1.3 版本控制

由于多人同时贡献代码，所以决定在项目中采用版本控制。

代码托管于 googlecode,采用 SVN 进行控制，可以在以下页面找到项目信息：

http://pintosof4p.googlecode.com

2 额外的参考资料

在作业完成的过程中除了压缩包中的资料，还参考了
pintos 官方文档： http://www.scs.stanford.edu/10wics140/pintos
minix3 源码 ：http://www.minix3.org/source.html

3 运行用户程序

这一部分主要由 LX 构思，W 实现，L 做出测试提出修改意见。X 整理了这部分文档。

pintos 操作系统的用户程序是要基于文件系统才能运行的，我们需要先做一些文件系统的操作，其步骤为：1.创建 disk;2.格式化 disk; 3.将可执行文件 elf 装入 disk;

pintos 操作系统的用户程序运行流程：

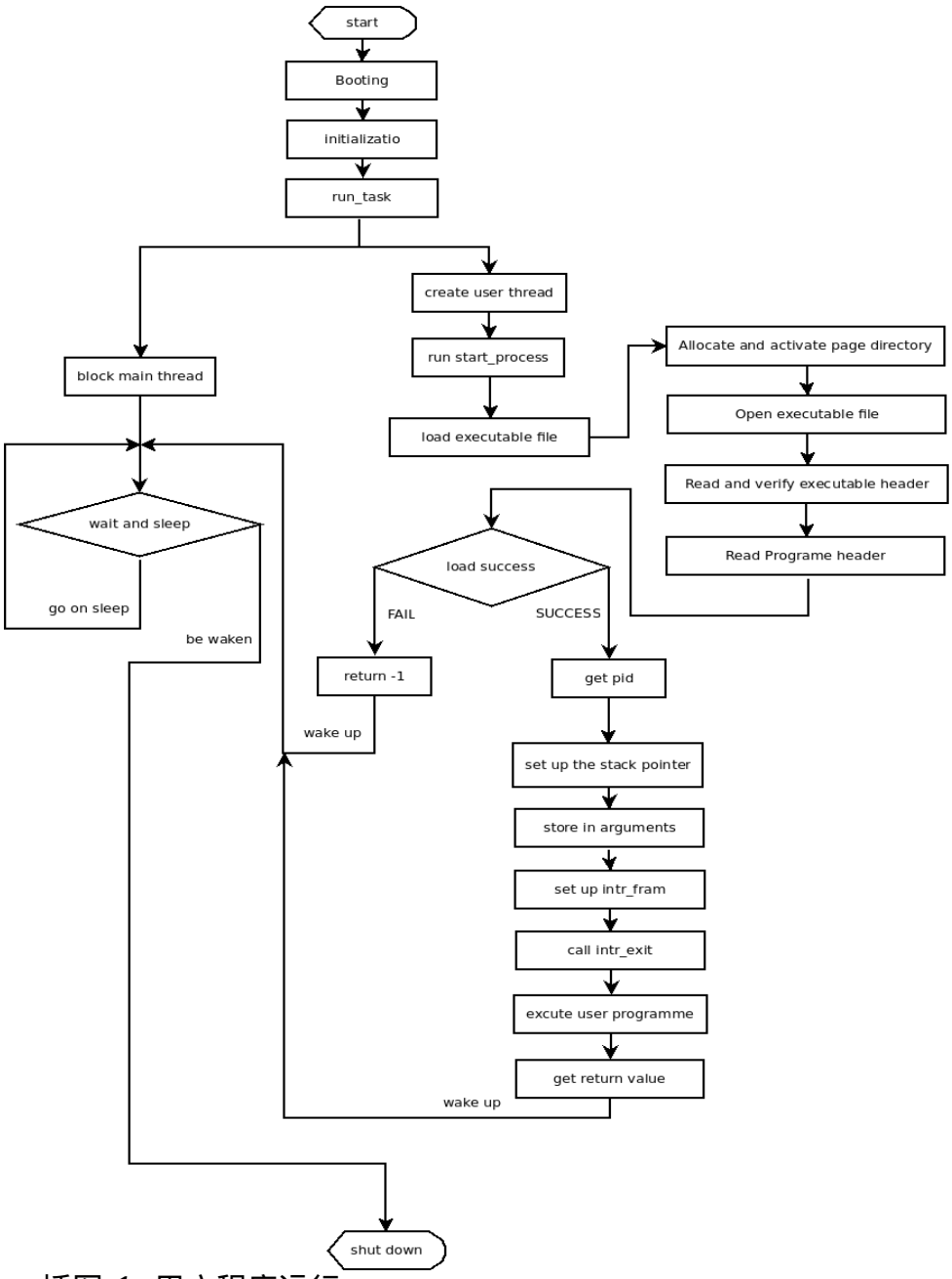

插图 *1:* 用户程序运行

如图，用户程序的运行遵循以上几个步骤：

1.操作系统启动与初始化，其中包括内核程序的存储空间分配；

2.加载用户可执行文件，为用户程序分配内存地址空间，由于目前操作系统下没有安装编译器，我们只能加载 elf 可执行文件；

3.我们在载入 elf 文件时需要做一系列检查，主要是为了确认 elf 文件的存储空间占用情况，代码段入口等。这些信息一般都保存在 elf 文件的 executable header 和 program header 里。在这里，我们遇到了一个小问题导致载入 elf 文件失败，需要载入的 elf 文件的 header 中标明，第一个 loader 部分占用虚存从 0x000000 开始，然而 pintos 操作系统默认虚存的低 0x1000 位置不被映射，即第 0 页不被映射，但是我们注意到，首个 loader 段实际用到的地址是从 offset 开始的，正好是 0x1000，那么函数 validate_segment ( ) 的判别机制要有所修改，将 if (phdr->p_vaddr < PGSIZE)改为 if (phdr->p_offset < PGSIZE)。这样做的话可以成功载入 elf 文件，不过这么做的后果是分配用户内存空间时，仍就会映射到虚存的第 0 页，这是 pintos 所不允许的，为了解决这个问题，我们在函数 load()里做一下修改，如果我们发现读出的 mem_page==0，我们需要手动将其设置为 0x1000,这样做实际上是将一个条件判断所作的工作手动实现了，因为我们无法控制由 make 得到的 elf 文件，我们只能从 load 函数中做修改。

4.在用户内存空间指定之后，我们就可以向用户内存的堆栈传递参数了。具体的实现方法将在下一章进行介绍。

5.在用户进程载入完毕之后，通过中断返回的方式进入用户程序的代码段进行执行。但实际上我们发现操作系统很快地关机了，经过分析，由于用户进程与操作系统 main 线程的优先级相同，造成的运行效果就是两者交替运行，而 main 函数执行 process_wait()迅速返回之后，就调用 shutdown 关机了，所以初期的做法是将 process_wait()中加入一个循环，不断的去检查该用户线程是否执行结束，如果结束就跳出循环，这是该 project 前期的实现方法，该方法会造成忙等待，更好地实现我们会在后面的几章进行说明。

6 .用户程序运行之后会传回返回值 ret_status，将其作为 process_wait()的返回值传回，main 线程关机，执行结束。

4 进程退出信息

4.1 需求分析

在运行用户程序结束时，我们需要得到进程退出信息，以反映给用户进程的运行情况，这些进程退出信息有时是用户需要计算机来运算的结果，有时是用户文件操作是否成功的标志，当用户进程因为不当操作被终止时，我们需要通知用户来修改自己的程序，可见，进程退出信息是十分必要的。

4.2 设计与实现

该功能具体实现涉及 exit system call,以及错误处理机制。exit 对应于大多数运行正常的用户进程，而当用户进程发生错误时，我们也要有相应的机制来返回执行信息。

4.2.1 数据结构

在 thread.h 中为 struct thread 添加成员变量:int ret_status，该成员变量用来保存该线程的退出信息，以供别的代码段寻访。

在 thread.h 中为 struct thread 添加成员变量: struct list child_list,该成员变量使得一个用户进程的推出信息在该进程执行完毕之后仍然可以被得到，这个成员变量主要被利用在 wait 这个 system_call 中，将在后文详细介绍。

4.2.2 处理方法

当用户进程执行到 return 的语句时，系统调用 exit system call，并将 return value 压到堆栈中，所以我们可以通过指针访问堆栈得到正常退出信息，通过 thread_name()函数，

得到对应进程的进程名。而对于被内核终止的用户进程，我们手动地将该进程的 ret_status 设置为-1,然后输出。

5 参数传递

5.1 需求分析

执行一个可执行文件，常常需要传入参数来控制程序运行，比如用哪种模式运行，需要用到的文件名称与路径是什么，所以，pintos 操作系统在执行用户进程的时候需要有参数传递的能力。

C 程序的风格是为 main 函数设置两个默认的参数 int argc 与 char* argv[]，argc 代表参数个数，而 argv 则是用字符串形式传入的指针数组。参数的摆放位置应该是用户堆栈的栈顶,拜访的格式应该为

Adress	Name	Data	Type
0xbffffffc	argv[3][...]	‘bar\0’	char[4]
0xbffffff8	Argv[2][...]	'foo\0'	char[4]
0xbffffff5	argv[1][...]	‘-l\0’	char[3]
0xbfffffed	argv[0][...]	‘/bin/ls\0’	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbfffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

需要注意几点：

- 字对齐，这样做的好处是访存速度加快；
- argv[argc]要为 0；
- 堆栈向低地址空间扩展。

5.2 数据结构

解决该问题的方法是字符串处理，涉及的数据结构有：

- process_exec 的参数字符串 parstr；
- 局部变量 int Stop(代表 Stack top)，用来记录参数传递过程中，用户堆栈顶端的地址；
- 局部变量 int argc,用来记录字符串解析出的子串个数，即参数个数；

- 局部变量 char* argv[],用来记录各个参数字符串在内存中的起始位置。

5.3 算法与实现

5.3.1 预处理

我们先要提取出可执行文件的路径与文件名，采用的做法是从字符串头部开始，找到第一个空格或者字符串结束标志即可，该文件名用来寻找 elf 文件以及为进程命名；

5.3.2 参数传递的时机

我们知道用户地址空间是 0～PHYS_BASE，但是 pintos 在初始化时并不会为用户地址空间建立实存与虚存的映射，在操作系统做这个工作之前，我们访问用户地址空间会产生 page_fault。

由上一章的说明可知，process_exec 进行用户线程创建时，用户程序并不是立刻开始执行的，当运转到该线程的时间片时，进入的是 start_process()函数，在这个函数中，调用了 load()函数，之后，通过一个中断返回，进入用户的 elf 文件入口，而操作系统为用户分配地址空间是在 load()函数中，所以在 start_process()调用 load()结束之后，我们就可以往栈顶写值了。

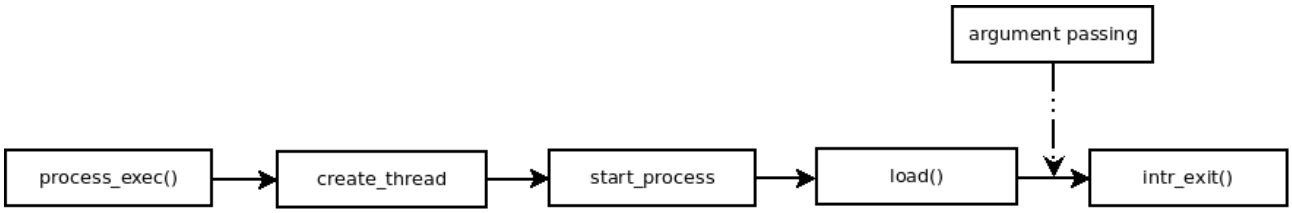

插图 *2:* 参数传递

5.3.3 实现

我们这里利用源代码中已有函数 strtok_r()来进行处理，以防止代码冗余。该函数的功能是将目标字符串分解为前缀串与剩下的串，前缀串即使从字符串头部读到第一个空格或者字符串终结符的串。在此处加一个循环，当前缀串为空时得到字符串解析成功。

我们每得到一个前缀串，即将该串放入用户堆栈的栈顶，这里有两种方法来做这样一个操作：1.内联汇编：asm volatile ("movl %0,%%ah;pushl %%ah " : : "g" (&*(token+offset)) : "memory");2.C 语言风格直接读写内存：*((char*)(Stop))=*(token+i)；这里 Stop 是用户堆栈顶端地址，token 是前缀串，i 表示该前缀串传到第 i 个字符。为了增加程序的可读性，我们使用第二种方法。

5.3.4 esp 指针设置

在参数传递实现前，我们将 esp 值设置为 PHYS_BASE-12,这个 12 代表着返回地址，

argc,argv 分别 4 个字节，而在参数传递实现之后，我们就需要根据参数的个数与长短来确定堆栈顶指针，其实我们在解析字符串，向内存空间中传值的过程中，就已经保存了堆栈顶的信息，于是，在参数放置完毕后，将 esp 设置为 Stop 的值，整个参数传递功能就是实现了。

6 Syscall 设计与实现

这部分由 L 设计，X 提出修改意见，L 整理书写文档。

6.1 数据结构

数据结构是由在 process 和 thread 中添加的。主要有：

```
Thread.h 中
   struct list fd_list;
   int ret_status;
   struct semaphore tsem,wsem;
   struct list child_list;
   struct file* elffile;
   bool alwaited;
   /*L: fd_elem */
   struct file_desc{
    int fd;  /* L:file descriptor */
    struct file *file;
    struct list_elem elem;
   };
```

6.2 机制设计

系统调用是由系统提供的一组完成底层操作的函数集合，由用户程序通过中断调用，系统根据中断向量表和中断服务号确定函数调用，调用相应的函数完成相应的服务。

6.2.1 Pintos 的 syscall 流程

pintos 中的 syscall 的中断向量号是 30h，一个典型 syscall 的过程 SYS_HALT 调用如下（通过反汇编 echo 得到）：

```
void
halt (void)
{
 8049ba0:      55                      push   %ebp
 8049ba1:      89 e5                   mov    %esp,%ebp
 8049ba3:      83 ec 18                sub    $0x18,%esp
```

```
    syscall0 (SYS_HALT);
   8049ba6:       6a 00                   push   $0x0
   8049ba8:       cd 30                   int    $0x30
   8049baa:       83 c4 04                add    $0x4,%esp
    NOT_REACHED ();
   8049bad:       c7 44 24 0c e4 a1 04    movl   $0x804a1e4,0xc(%esp)
   8049bb4:       08
   8049bb5:       c7 44 24 08 05 a0 04    movl   $0x804a005,0x8(%esp)
   8049bbc:       08
   8049bbd:       c7 44 24 04 44 00 00    movl   $0x44,0x4(%esp)
   8049bc4:       00
   8049bc5:       c7 04 24 be a1 04 08    movl   $0x804a1be,(%esp)
   8049bcc:       e8 ff fd ff ff          call   80499d0 <debug_panic>
   8049bd1:       90                      nop
   8049bd2:       90                      nop
   8049bd3:       90                      nop

  08049bd4 <flush>:
  }
```

分析上面可以知道 在执行 SYS_HALT 这个系统调用时，实际是把 SYS_HALT 的值 0 压入堆栈，然后就进行了中断 30h 的调用，接下来系统的中断服务函数从堆栈里获取相关信息（如 SYS_HALT），接管剩余的过程，并完成相应的功能。

一个较完整的中断调用 SYS_WRITE，以由 printf()产生为例*:

```
  ……
  Kernel PANIC at ../../userprog/syscall.c:215 in syscall_handler(): [KP
HERE]
  Call stack: 0xc0028899 0xc002c058 0xc0021d26 0xc0021e63 0x8049ad4
0x8049c20 0x8049c6c 0x8049c95 0x8048c27 0x804810e 0x8048160
  ……
  backtrace:
  In kernel.o:
  0xc0028899: debug_panic (.../../lib/kernel/debug.c:38)
  0xc002c058: syscall_handler (...../userprog/syscall.c:222)
  0xc0021d26: intr_handler (..../threads/interrupt.c:367)
  0xc0021e63: intr_entry (threads/intr-stubs.S:38)
  In ../../examples/echo:
  0x08049ad4: write (...../lib/user/syscall.c:123)
  0x08049c20: flush (.../../lib/user/console.c:93)
  0x08049c6c: vhprintf (.../../lib/user/console.c:73)
  0x08049c95: vprintf (.../../lib/user/console.c:12)
  0x08048c27: printf (...xamples/../lib/stdio.c:89)
  0x0804810e: main (...tos/src/examples/echo.c:8)
  0x08048160: _start (...les/../lib/user/entry.c:9)
```

*这个例子由人为在 *SYS_WRITE* 的 *STD_OUT* 产生一次 *Kernel PANIC* 生成。

用户程序和内核交互的流程图如下:

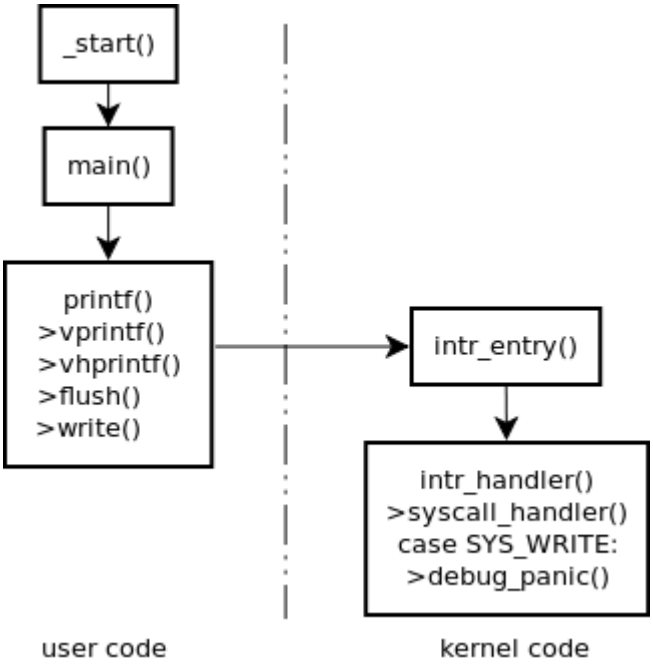

插图 *3: printf* 的流程

可以清晰的看出用户空间的代码和内核空间的代码的交互是通过 printf 调用库函数，库函数和中断服务进行绑定调用来完成最后的输出(对于 STD_OUT，最终调用 console 里提供的 putbuf 实现输出)。由于我们人为的加入了 kernel panic，所以这里并没有显示中断返回值，实际上约定返回值是放入 eax 寄存器的。

6.3 *Syscall* 的用户需求

本次需求的 syscall 有 12 个，分别如下（lib/user/syscall.h）

```
void halt (void) NO_RETURN;
void exit (int status) NO_RETURN;
pid_t exec (const char *file);
int wait (pid_t);
bool create (const char *file, unsigned initial_size);
bool remove (const char *file);
int open (const char *file);
int filesize (int fd);
int read (int fd, void *buffer, unsigned length);
int write (int fd, const void *buffer, unsigned length);
void seek (int fd, unsigned position);
unsigned tell (int fd);
void close (int fd);
```

对应着服务为（lib/syscall-nr.h）

```
    /* Projects 2 and later. */
    SYS_HALT,                   /* Halt the operating system. */
    SYS_EXIT,                   /* Terminate this process. */
    SYS_EXEC,                   /* Start another process. */
    SYS_WAIT,                   /* Wait for a child process to die. */
```

```
    SYS_CREATE,                 /* Create a file. */
    SYS_REMOVE,                 /* Delete a file. */
    SYS_OPEN,                   /* Open a file. */
    SYS_FILESIZE,               /* Obtain a file's size. */
    SYS_READ,                   /* Read from a file. */
    SYS_WRITE,                  /* Write to a file. */
    SYS_SEEK,                   /* Change position in a file. */
    SYS_TELL,                   /* Report current position in a file. */
    SYS_CLOSE,                  /* Close a file. */
```

在具体调用中按照参数数目分为了 4 组 syscall[0-3]* (lib/user/syscall.c)
下面以 syscall2 为例说明用户空间内的系统调用如何产生

```
/* Invokes syscall NUMBER, passing arguments ARG0 and ARG1, and
   returns the return value as an `int'. */
#define syscall2(NUMBER, ARG0, ARG1)                                \
      ({                                                            \
       int retval;                                                  \
       asm volatile                                                 \
         ("pushl %[arg1]; pushl %[arg0]; "                          \
          "pushl %[number]; int $0x30; addl $12, %%esp"             \
          : "=a" (retval)                                           \
          : [number] "i" (NUMBER),                                  \
            [arg0] "g" (ARG0),                                      \
            [arg1] "g" (ARG1)                                       \
          : "memory");                                              \
       retval;                                                      \
      })
```

可以清楚的看到当 syscall2 被调用时（例如 create(file,size)），参数按照由右至左的顺序被压入堆栈，然后放入 *NUMBER* (这里是 SYS_CREATE)，最后产生 30h 的中断调用。印证了前文提到的 syscall 流程中的参数传递和中断调用流程。

*注：*syscall[0-3]*只是一组宏，将每个调用替换为不会被编译器优化的 *AT&T* 风格汇编代码。

6.4 *Syscall* 的功能性设计与实现

Syscall 的架构与功能性主要由 userprog/syscall.c 中对 static void syscall_handler (struct intr_frame *f) 函数的填充而实现。这里对其中部分函数进行解释和说明。

整个函数由一个很庞大的 switch 来进行不同的服务调用，由前文知道这时 intr_frame 的 esp 栈顶里保存着中断服务号 *NUMBER*，并且在栈顶的后面若干个字节保存了调用的其他参数。把 syscall 设计为一个调用其他函数的流程控制器，保持这个函数的过程清晰。

```
uint32_t *p=f->esp;
switch(*p){
   case SYS_HALT:{                  /* Halt the operating system. */
     /*L: we just power-off here, right?
      * no return, just an one-way call */
```

```
      shutdown();
    }
  /* 这是第一个也是最简单的系统调用，我们直接关机即可，这个服务无须返回，到这里就关机了。
*/
```

```
    case SYS_EXIT:{                    /* Terminate this process. */
      /*L: here i need a place to store "return code"
       * please add a new member(int) in struct thread
       * still an one-way call, no break is needed */
      thread_current ()->ret_status = *((int*)(p+1));
      if(p+1>=PHYS_BASE)
       thread_current ()->ret_status=-1;
      printf ("%s: exit(%d)\n", thread_name(), thread_current()->ret_status);
      thread_exit ();
    }
  /* 这个系统调用在程序 return 或者其他退出情况时调用，我们保存它的退出码在一个新的添加的变
量里，然后打印退出码退出 */
```

```
    case SYS_EXEC:{                    /* Start another process. */
      /*L: IN: filename in (p+1)
       * OUT : return a pid_t(in eax) or fail
       * CALL: process_execute()
       * this is a return-needed intr-handler that is a "break;"
       * from this one, almost handler below need a break */
      if(*(p+1)==NULL||*(p+1)>=PHYS_BASE||pagedir_mapped(thread_current()-
>pagedir,*(p+1))==0)
       {
     printf ("%s: exit(%d)\n", thread_name(),-1);
     thread_current()->ret_status=-1;
     thread_exit();
     break;
     }
      f->eax = process_execute(*(p+1));
      break;
    }
  /* sys_exec 是一个很复杂的系统调用，由于设计时所有的系统调用服务过程被设计成为调用其他函
数的过程，这样来保持整个服务函数的流程清晰。 这里根据这个原则，只进行 process_execute 函数
调用，这个函数的实现将在后文具体描述 */
```

```
    case SYS_WAIT:{                    /* Wait for a child process to die. */
      /*L: IN: tid in (p+1)
       * OUT : -1(killed) or child_status
       * CALL: process_wait()
       * process_wait need more code work
       * */
      f->eax = process_wait(*(p+1));
      break;
    }
  /* 根据刚才叙述的原则，这里相同的方法只调用 process_wait，这个函数的实现也在后文具体描述
*/
```

```
    case SYS_CREATE:{                  /* Create a file. */
      /*L: IN: filename(p+1), initial_size(p+2)
       * OUT : f->eax = some result;
       * */
      if(*(p+1)==NULL||*(p+1)>=PHYS_BASE||pagedir_mapped(thread_current()-
```

```
>pagedir,*(p+1))==0)
      {
     printf ("%s: exit(%d)\n", thread_name(),-1);
     thread_current()->ret_status=-1;
     thread_exit();
     break;
     }
      f->eax = filesys_create (*(p+1), *(p+2));
      break;
     }
  /* SYS_CREATE 是一个典型的文件类型的调用，而实际上这里也只是调用位于 filesys.c 的
filesys_create 函数完成操作，这个函数由 pintos 源码提供，后面的其他文件操作也采用了类似的实
现，这里只举这个函数为例其他的实现请参加源码 */
```

```
  ......
    default:{
     printf("Unhandled SYSCALL(%d)\n",*p);
     thread_exit ();
    }
  }
```

由 syscall 设计思想的要求，实际上在实现 syscall_handler 时产生的就是一堆对其他函数的调用，所有整个函数没有什么可以欣赏的，参考源码和注释就可以完全理解我们到底做了什么，故这里不再赘述，详情请参阅 userproc/syscall.c l25-304 。

6.5 Syscall 样例

这里通过几个典型的调用来表明我们的设计思想和几个关键问题的解决。

6.5.1 syscall 样例 1：exec 的工作流程

原始代码的 process_execute 实现了初步的进程创建和内存分配填充的工作，但是这里并不保证可以保证进程创建失败时可以稳定的产生-1，并且这里的进程创建流程并不能满足我们用户需求，即在子进程成功或失败创建前，父进程必须被阻塞。

根据用户需求，添加的数据结构如下：

```
在 thread.h 中为 struct thread 添加
struct semaphore tsem;
```

下面以流程图的形式说明同步机制在其中起的作用。

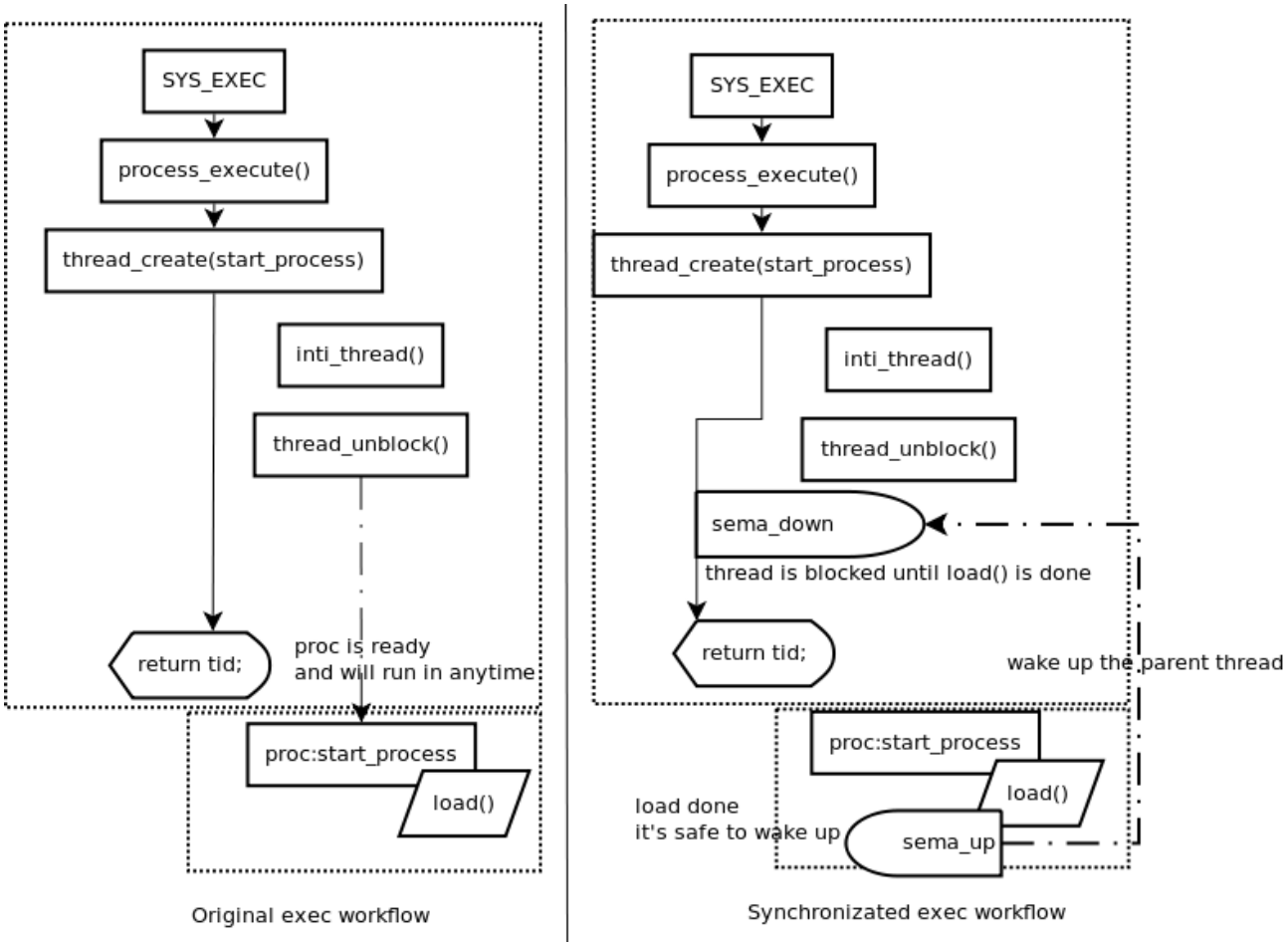

插图 *4:* 带有同步机制的流程

新的机制使用 semaphore tsem 来实现进程创建时的同步。原始的线程创建机在创建一个内容为 start_process 的进程后，并 unblock 后便返回一个临时的 *tid*（*tid*!=-1）。父进程到这里便和子进程并行跑，没有任何机制保证在子线程 load 成功或失败后可以令父线程可以得知。并且当子进程 load 失败时，把自己 *tid* 改为-1 也是不安全的。

由此我们设计了一种基于信号量同步机制的 exec 流程，如图*。这种机制下，在创建 start_process 后，会进行 sema_down。这里的信号量直到 load()完毕后才会 sema_up，此时可由 load()的返回值信息直接判断是否进程是否创建成功，这样可以返回一个确定的 *tid*(*tid*>0 或 tid=-1)。至此完成了进程创建工作，并且父进程获得了正确的 *tid*。

*注：流程只表现了 *exec* 的主要函数调用关系，一些控制判断语句（例如 *load()*是否成功等）没有画入该图。

6.5.2 syscall 样例 2：wait 的工作流程

用户需求中涉及的 wait 要求比较多，包括了主进程在各种上下文环境下调用 wait 时的情形。

必须马上返回**-1** 的情形：

- 不是直系孩子
- 第二次 wait 某一 tid

需要处理的情形：

- 子进程正在运行 → 阻塞父进程
- 子进程已经运行完毕 → 取回退出代码
- 异常退出 → 取回 -1 或其他异常

分析用户需求，需要添加的数据结构有：

```
thread.h 的 struct thread 中添加的有
   /*[X]sema to avoid busy wait*/
   struct semaphore wsem;
   /*[X]store child threads' information*/
   struct list child_list;
   /*[X] whether the thread has already been waited*/
   bool alwaited;
```

我们设计的 wait 同步机制可以描述为使用信号量 wsem sema_down 阻塞发起等待的父进程，直到子进程退出时 sema_up 唤醒父进程。为了实现马上返回-1 的两种情况，添加了 struct list child_list 来实现对子进程的辨识；添加 bool alwaited 实现对第二次访问的辨识。为了处理取回退出代码的情形，改动了进程销毁机制，在当前进程退出时，只释放子进程的资源，保留自己的 thread 结构直到父进程退出时再释放。

使用流程图表现如下：

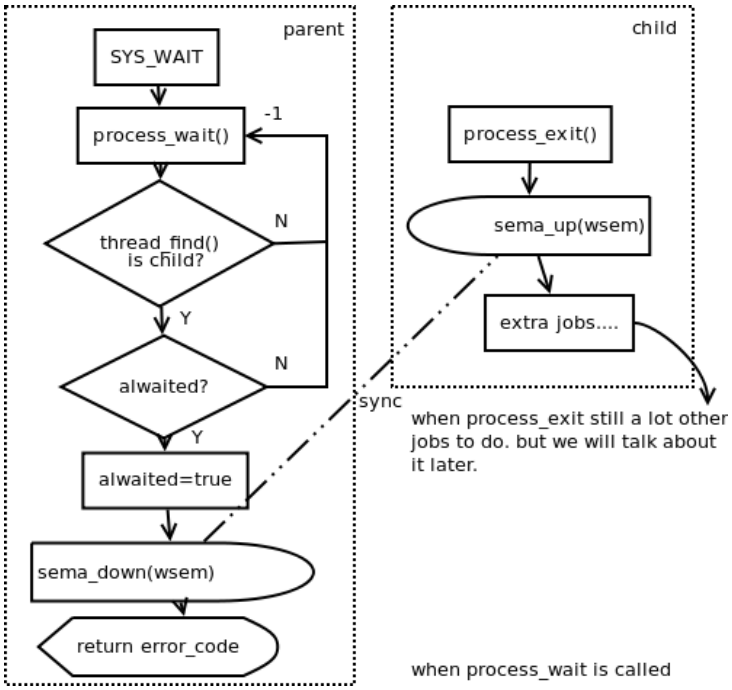

插图 *5:* 等待机制

上图便是 process_wait 的实现流程，在这种机制下 wait 发起者会一直被阻塞直到子进程退出 sema_up 为止，这时已经获得 *exit_code*(或 error_code)。要求与之配套的 kill 实现中会把-1 写入 *error_code*。

*注：图中只表现了信号量相关的操作，资源释放的操作被略去。
*注 2：我们探索过另外一种使用忙等待的解法，但是由于在例如 *oom* 的情形下下会引发严重的系统资源浪费，于是我们抛弃了这种简单的实现。

6.5.3 syscall 样例 3：open 的工作流程

文件操作是操作系统提供的重要系统调用之一，这里由于实现文件操作的过程也是基于对 pintos 已经实现的函数的调用，并不包含复杂的机制策略，故仅以 open 和 write 为例说明。

需要添加的数据结构有

在 thread.h 中添加
```
struct file_desc
{
int fd;     /* L:file descriptor */
struct file *file;
struct list_elem elem;
};
```
在 struct thread 中添加
```
  struct list_elem child_elem;        /*[X] List element for child threads */
```

用户需要由 SYS_OPEN 获得一个文件描述符 *fd*（file descriptor ）或者当失败时返回-1，这里的特殊情况是 *fd 0* (STDIN_FILENO) 和 *fd 1* (STDOUT_FILENO) 已经被保留做 console 的输入输出，所以所有的分配的 *fd* 从 2 开始。每个进程拥有自己的 *fd_list* 来记录已经打开的文件。Struct file 中已经包含 *inode*，解决 *pos* 的问题，所以每个打开的 *fd* 拥有自己的写入点也是由系统自动提供的。

```
case SYS_OPEN:{                    /* Open a file. */
    /*L: IN: filename
     * OUT : Returns the new file* if successful or a null pointer
    /* L:handle the null filename */
    struct file *file = filesys_open (*(p+1));
    /*L: check if file is opened successfully or return -1 */
    if (file == NULL){
     f->eax = -1;
     return;
    }
    /*L:the file desc thing */
    struct list_elem *e;
    struct list* fd_list = &thread_current()->fd_list;
    struct file_desc *file_d = (struct file_desc *)malloc(sizeof(struct file_desc));

    /*L: 0,1 are stdios,
     * here only one fd is supported, more file need more code work */
    int maxfd;
```

```
      if(list_empty(fd_list)){
       maxfd=1;
      }
      else{
       e = list_begin (fd_list);
       maxfd = list_entry (e, struct file_desc, elem)->fd;
      }
       file_d->fd = maxfd + 1;
       file_d->file = file;
       list_push_front (fd_list, &file_d->elem);

       f->eax = file_d->fd;
       break;
      }
   /* 首先会检查文件是否被成功打开，否则返回-1.在成功打开的情况下分配 fd_list 中最大的 fd+1
为新文件的 fd，并且放入列表。 */
```

注：这里为了更快（复杂度为 *O(1)*），*fd* 可能是不连续的。

6.5.4 syscall 样例 4：write 的工作流程

SYS_WRITE 是另一个有代表性的系统调用，最基础的 printf()也是最终由 write 实现向屏幕的输出。

这里的核心也是通过调用 file_write()来实现。

```
      case SYS_WRITE:{
       /* Write to a file. */
        /*L:IN :fd(p+1),buf(p+2),size(p+3)
         * OUT :eax=count
         * NOTICE the diff between stdout and a normal file */
        if (*(int*)(p+3) <= 0)
        {
         f->eax = 0;
         return;
        }
        if(*(p+1)==STDOUT_FILENO)
        /* STDOUT_FILE is 1, defined in lib/stdio.h */
        {
         /* putbuf is in lib/kernel/console */
         putbuf ((char*)*(p+2), *(int*)(p+3));
         /* putbuf have no return, so eax=size */
         f->eax=*(p+3);
        }

        /* check fd in fd_list */
        struct list_elem *e;
        struct file_desc *file_d;
```

```
    struct list* fd_list = &thread_current()->fd_list;
    for (e = list_begin (fd_list); e != list_end (fd_list);
       e = list_next (e)){
      file_d = list_entry (e, struct file_desc, elem);
      if (file_d->fd == *(p+1))
       {
        int count = 0;
        count = file_write (file_d->file, *(p+2), *(p+3));
        f->eax = count;
        break;
       }
     }
    break;
  }
/* 判断需要写入的大小，判断是否写入 STD_OUT，判断是否是已经打开 fd，调用 file_write */
```

这里没有更多需要描述的细节，一个简单的流程图如下：

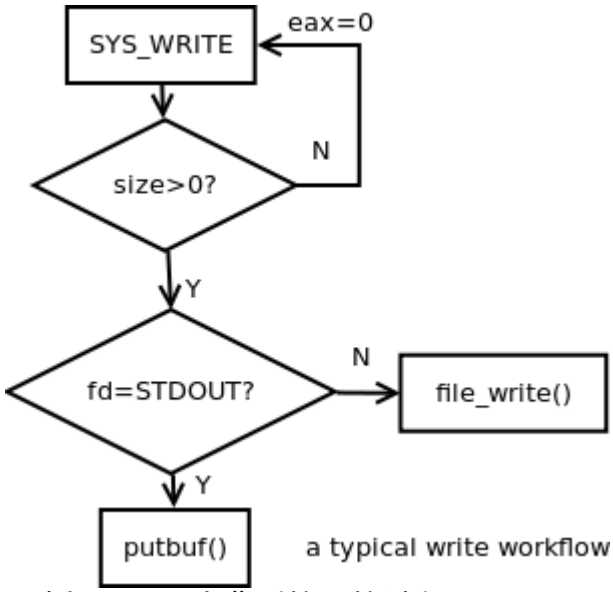

插图 *6:* 一个典型的写的过程

6.6 权衡

6.6.1 访问用户空间内存

pintos 文档中提供了访问用户空间内存的方式，一种简单的，一种利用 MMU 的复杂快速的。这里实际采用较为简单的判断 (X 实现)。

```
//[X]防止读到 code 段以下的 bad
 if(p<=0x08084000-64*1024*1024||pagedir_mapped(thread_current()->pagedir,p)==0||
```

```
(*p)==NULL||(*p)>=PHYS_BASE)
    {
    printf ("%s: exit(%d)\n", thread_name(),-1);
    thread_current()->ret_status=-1;
    thread_exit();
    }
```

我们不确定 pintos 文档中的提供的两个函数能够引起什么情况，这里 X 书写了一个简单的判断来实现这个功能，不过这只是一个比较弱智的实现，L 并不十分赞同这种行为。

6.6.2 文件描述符的分配

在文件描述符(file descriptor)的分配中，如果保持新分配的总是最大的 *fd*+1（时间复杂度 O(1)），那么 *fd* 列表就会产生空洞。最坏的情形下，空洞（已经关闭但又无法使用的 fd）的大小为最大整数-1，这时再申请新的 *fd* 时，分配就会产生越界错误，而实际上可能只打开了两个文件，*fd* 列表中只有两个很大的 *fd*，其他的都被浪费掉了。而唯一寻找空洞的方式就是遍历（时间复杂度 O(n)），小组讨论决定采用牺牲空间代价换取时间，即第一种方法。

7 错误处理

7.1 需求分析

操作系统是一个特殊的程序，它控制着整个计算机的运行，在用户程序符合要求的情况下，稳定和正确是操作系统运行的基本要求。但是仅仅做到这一点还不够，我们不能对用户的行为做出任何假设，用户很可能会作出不当操作，这些不当操作有可能是用户粗心大意，也可能是有意为之，无论何种情况引发的不当操作造成操作系统的崩溃都有可能造成用户的损失。因此，系统的健壮性是评价一个操作系统优劣的重要指标。

此次 project 之中，在实现了基础的 system call 之后，我们有必要采取一系列措施，在用户程序产生不当操作时，终止用户进程，而不是任其执行，造成操作系统崩溃。

7.2 错误类型与处理机制

首先，在这里说明遇到可能造成 kernel panic 的行为时，我们所做的操作是将该进程的结束状态改为-1，调用 thread_exit()终止进程，由 thread_exit()调用 process_exit()，释放进程所占用的资源。

接下来，我将会造成 kernel panic 的原因进行了一个简单的分类，下面逐一分析造成的原因:

7.2.1 用户访问系统内存

造成后果：

系统内存区域是操作系统运行时使用的区域，内部存放的信息与系统运行相关，对该区域的写操作有可能导致操作系统异常。

解决方法：

用户程序访问内核内存时，系统会调用 page_fault 函数，根据 page_fault 函数中确定产生原因的代码段功能（/* Determine cause. */），我们可以通过加一个 if(user)判断解析出是否是由用户程序造成的错误，如果是，按照确定的错误处理机制，调用 thread_exit()终止进程，返回失败信息-1.

涉及测试样例：

tests/userprog/read-bad-ptr

tests/userprog/bad-write2

tests/userprog/bad-jump2

7.2.2 用户访问尚未被映射或不属于自己的用户虚存地址空间

造成后果：

目前的 pintos 有一个简单的虚存映射机制，大致的流程如下：

1)规定将实存一分为二，低空间分配给内核，高空间分配给用户；

2)系统初始化时，将指定 PHYS_BASE，高于 PHYS_BASE 的虚存地址空间分配给内核；

3)在初始化用户进程时，指定一块虚存地址空间给该进程，具体的页表信息保存在该进程对应线程的结构体 struct thread 中，其成员变量名为 pagedir；

如果访问尚未被映射的地址空间，会造成 page_fault，引发系统崩溃。如果一个用户程序可以任意访问别的用户程序的内存，极易造成用户程序运行的相互影响，程序执行的正确性得不到保证。

解决办法：

将这个问题细分，有三种情况：1.用户进程直接访问错误的地址；2.用户通过 system call 机制访问到不该访问的地址。

第一种错误引发 page_fault，我们只要判断出它是由用户进程产生的 not_present 错误即可按照错误处理机制解决，但是对于后面一个问题，我们仅通过在 page_fault 函数里加判断是难以解决的。

我们注意到用户调用 system call 时，实际上运行的是内核的代码，所以当由于 system call 参数不合法产生 page fault 时，Determine Cause 代码段认为它是内核造成的错误，从而引发 kernel panic，系统崩溃。所以，我们在 system call 的调用机制里要加上错误检查，在检查出错误时直接终止该用户进程。具体的做法是利用 pagedir.c 中的 bool lookup_page(uint32_t *pd, const void *vaddr, bool create)这个函数，第一个参数指的是页表索引，第二个参数代表需要访问的虚存地址，函数的功能是判断该虚存地址是否在该页表索引指向的页中，第三个参数是一个执行选项，当 create 为 true 时，激活该虚存地址所在页，

否则只做判断。我们利用这个函数，以及 struct thr－ ead 的成员变量 pagedir，改写 lookup_page 函数，只利用其判断功能得到 pagedir_mapped 函数，通过 pagedir_mapped(thread_current()->pagedir,*(参数指针))调用，可以得知当前线程是否访问合法的地址，这样两个问题的处理方法得到了统一，也能处理一个用户进程访问其它用户进程内存的错误。

另外需要特殊说明两点：1.当初在载入可执行文件时，我们跳过了 vaddr<PGSIZE 的检查，所以如果在后面的操作中不加任何处理，访问第 0 页的地址空间不会引发 page fault，然而这是不合理的，所以我们需要像第一章介绍的发现读出的 mem_page==0，我们需要手动将其设置为 0x1000 。2.访问地址空间时，有一段坏地址（bad address）,大概在用户存储区代码段一下 64M 左右，我们需要额外加一个控制条件以处理这种错误。

涉及测试样例：

tests/userprog/sc-bad-arg

tests/userprog/sc-bad-sp

tests/userprog/sc-bad-arg

tests/userprog/open-bad-ptr

tests/userprog/read-bad-ptr

tests/userprog/write-bad-ptr

tests/userprog/exec-bad-ptr

tests/userprog/bad-read

tests/userprog/bad-write

tests/userprog/bad-read2

tests/userprog/bad-write2

tests/userprog/bad-jump

tests/userprog/bad-jump2

7.2.3 参数无效导致文件加载失败

造成后果：

system call 需要用户传入参数来决定执行的对象，如果参数无效或者缺失，虽然不会造成系统崩溃，但是用户程序也无法正常运行。

解决办法：

系统内部程序针对错误参数，特别是涉及到文件操作时，往往会返回功能调用失败，即 return -1,。在具体测试时，并未在这个问题上出现 fail 的情况，所以我们无须加入复杂机制，在内部调用之前检查一下参数有效性即可，比如判断字符串是否为空等，根据判断和内部函数调用的返回值，我们适时地终止用户进程。需要特别注意的是，exec-missing 这个测试情形，虽然同样可以由 open()函数返回-1 得到，但是考虑到 process_exec()函数的具体功能，在函数执行过程中新的线程在运行过程进行到 load()这一步才能得到文件是否载入成功的信息，我们不能立刻返回进程号，这里，根据上一章所讲的 process_exec()利用信号量的具体实现便可以很好地解决这一问题，创建线程时不

立刻返回线程号，而是在 thread_create()返回之前 sema_down 目标线程的信号量 tsem,在 load()函数返回有效信息式 sema_up 该信号量，然后再返回进程号，作为 process_exec()的返回值，具体实现见上一章。

涉及测试样例：

tests/userprog/create-empty

tests/userprog/open-missing

tests/userprog/open-empty

tests/userprog/exec-missing

tests/userprog/wait-bad-pid

7.2.4 向可执行文件写入

造成后果：

程序在执行过程中被写入，可能导致指令执行混乱等一系列严重后果，如果发生这种情形，我们需要制止，知道该段程序运行完毕才允许对其重新进行写入。

解决办法：

pintos 的基本文件读写机制已经帮我们实现了控制文件读写的函数 file_deny_write(), file_allow_write()，我们通过找到合适的时机加入这两个函数，可以做到制止向可执行文件中写入的操作。添加 file_deny_write()的时机在这段可执行文件具体执行之前，具体地说，在 start_pro cess()函数执行到内联汇编调用 intr_exit 之前。而 file_allow_write()可以选则在进程销毁的过程之中，即 process_exit()之中。

涉及测试样例：

tests/userprog/rox-simple

tests/userprog/rox-child

tests/userprog/rox-multichild

8 性能压力测试

8.1 需求分析

操作系统在保证运行正确的基础上，需要有较高的性能。

其性能指标主要有：

- 就绪队列的线程数量，或系统平均负载
- 进程切换速率
- 资源利用率，包括 CPU 利用率，内存利用率等

- 系统的稳定性

在操作系统能够运行用户程序之后，我们就要开始考虑这些问题。

8.2 测试方法

测试样例中提供了这样一个测试 multi-oom，该测试通过用户进程不断地做递归调用，并且不断地进行打开文件操作，并随机地终止某一个线程。该测试考察到了系统平均负载，资源利用率，以及系统的稳定性，具体考察过程如图。

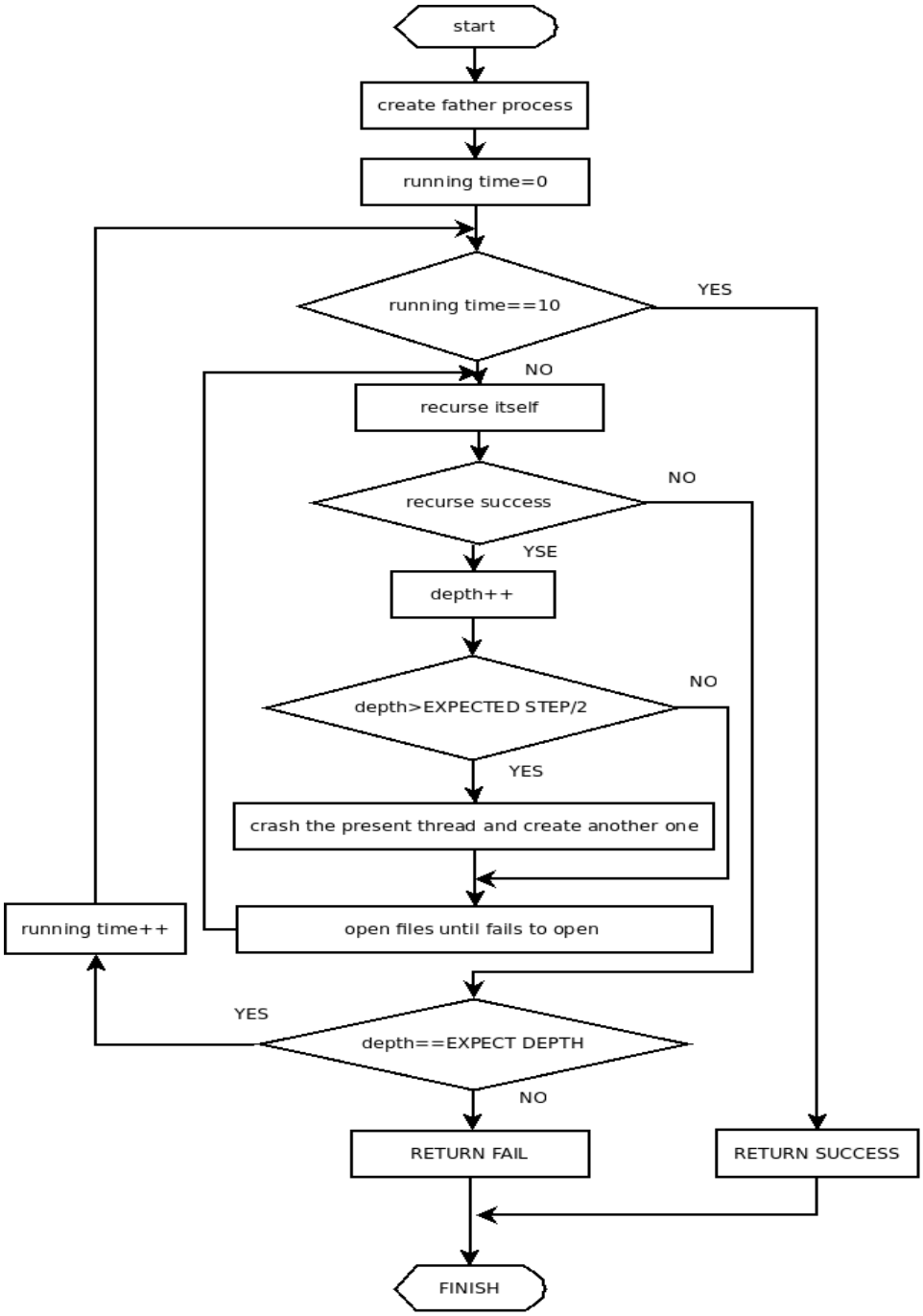

插图 7: 考察

8.3 测试过程

8.3.1 忙等待造成超时处理

在之前设计 process_wait 函数中，我们设置了一个 while(1)死循环，利用 get_thread_ by_tid (child_tid)函数得到目标进程，然后检查该进程的运行状态,发现状态是 THREAD_DYIN G， 即可得到返回值退出循环。

但是在实际运行过程中，随着递归层数的加深，进程创建的速度越来越慢，测试程序有一个时间限制：360 秒，然而采取刚才所说的方法不能按时跑完。

我在调试过程中，注意检查创建线程所调用的函数，发现没有任何操作的耗时会随着线程数量的增加而增加，加入调试信息发现，创建线程的每一步操作耗时都是线性增长的，由此判断出程序调度过程中发生了忙等待。此时，我们设计出利用信号量的方法实现 process_wait()函数，解决了测试超时的问题。当程序递归调用时，process_wait()在子线程运行完毕之前会通过 sema_down()进入子线程 wsem 成员变量的 wait_list 里，在子线程运行完毕进入 process_exit()阶段 sema_up()该信号量唤醒父线程。这样一来，在递归调用过程中，实际上只有一个线程在 ready_list 之中，递归速度大大加快。

8.3.2 内存资源释放操作

在 multi-oom 运行时，进程会不停地打开文件，并且不去关闭，如果不在进程退出时关闭这些文件，释放这些文件的文件描述符(fd)，会造成内存泄漏，影响系统效率。

解决办法首先是修改 struct thread 数据结构，向前文描述的，加入 fd_list,用来保存由该进程打开文件的文件描述符，在进程退出函数 process_exit()之中，遍历 fd_list 的每一个结点，通过 list_entry 函数解析出文件指针，用 file_close()函数来关闭所有文件，之后，调用 free 函数释放文件描述符 fd 的内存空间，这样便可以防止内存资源的泄漏，在这个方面上保证 pintos 操作系统的性能。

8.3.3 系统平均负载能力测试

系统平均负载能力反映在同时处于就绪状态的线程数，在本测试中，对应的就是递归调用的层数，层数越多，并发能力越强，而就绪程序越多，对于系统的压力就越大。我们需要测试出 pintos 操作系统的并发能力上限，对用户创建线程做出一定限制，以保证 pintos 系统正常运行。

我们采用二分法进行测试，确定系统上限，测试结果如下。

最大线程数	结果	最大线程数	结果
20	正常	100	出错
60	出错	40	正常
50	正常	55	出错

Multi-oom 程序要求的递归层数为 30,加上系统主线程和用户主线程共 32 个线程，经测试，我们修改后的 pintos 负载能力在要求之上。最后，我们参考 linux 操作系统的做法，采取限制线程数的方式来控制系统稳定地运行。

具体做法是在线程创建函数 thread_create()中，利用 list_size(&all_list)检测 all_list 的结点个数，若超过 50,则返回创建线程失败信息。这样做的好处是：1.避免线程数接近系统极限；2.针对该测试，可以保证每次测试系统的表现稳定，反映出来就是递归调用的层数相同。

当然，也可以不采用这种硬性限定的办法，在分配页失败的时候返回线程创建失败消息，不过由于无法预知分配页失败会造成的后果，且无法保证递归层数的确定，我们没有采取这种方式。

9 开发日志

这里记录了开发过程中遇到的困难与解决进度

时间表	项目进展
11 月 23 日	开始文档的阅读，任务的分配。
12 月 2 日	研究用户程序运行流程，出现 load fail 问题。
12 月 3 日	通过修改判断条件跳过 load 问题，可以运行用户程序，开始研究参数传递与 system call 的运行流程。
12 月 4 日	实现参数传递，开始设计 exit system call 和 write system call（写命令行）。
12 月 5 日-9 日	System call 基本功能的实现。
12 月 10 日	涉及内存操作以及文件读写出错的错误处理机制实现，出现 read twice，syn_read，syn_write 测试样例出错。
12 月 11 日	完善 read 与 write 的 system call 代码，exec_missing 测试样例出错
12 月 12 日	利用信号量实现 process_exec,解决上一日出现问题。由于前期修改 load 判断条件，read_bad 出错，手工修改 load()函数解决该问题。通过 file_deny_write()通过 rox_multiple 相关测试
12 月 13 日	运行 multi_oom 测试超时，通过采用信号量方法重新实现 process_wait()解决忙等待问题，通过二分法确定影响递归层数性能的参数。通过所有测试。

10 版权信息

我们的修改后的源码使用和 pintos 代码相同的 Lisense，并且保留署名的权利。

文档使用 CC 协议发布

注：我们的文档及代码以开放方式托管于互联网，已经确认被其他 (多数) 小组参考，由于我们的授权方式，并不能阻止这样的事情发生。

pintos 源码采用以下 License 发布

11 修订版本

10 年 12 月 20 日	文档合并整理	mayli he
10 年 12 月 23 日	添加版权信息	mayli he