西北工业大学 操作系统实验 实验报告

班号：10012006　　姓名：夏卓　　学号：2020303245

实验日期：2022/11/20　　实验名称：运行用户态程序

一、实验目的

掌握在 GeekOS 系统用户态模式下加载并运行可执行程序的方法。

二、实验要求

1. 按照实验讲义 P127 页中的设计要求，实现在用户态模式下加载并运行可执行程序的代码，给出关键函数的代码以及实验结果。

2.回答问题：用户程序的参数是如何传递给程序的？第九章第 3 题。

三、实验过程及结果

首先在 main 函数中的 Spawn_Init_Process 中调用 spawn 函数生成新的用户级进程；

在 spawn 函数中依次读取可执行文件、按 ELF 格式解析、调用 Load_User_Program 函数加载用户程序，最后通过 Start_User_Thread 创建一个进程并使该进程进入准备队列。

修改 Load_User_Program()函数，加载可执行文件镜像并通过调用 Create_User_Context 函数创建新进程的 User_Context 结构，这里需要注意用户态进程除了需要数据段和代码段用于装入可执行文件和数据外，还需要加载进程的堆栈以及参数块数据结构：

```
/* 程序参数数目 */
unsigned int numArgs;
/* 获取参数块的大小 */
ulong_t argBlockSize;
Get_Argument_Block_Size(command, &numArgs, &argBlockSize);
/* 用户进程大小 = 参数块总大小 + 进程堆栈大小(8192) */
ulong_t size = Round_Up_To_Page(maxva) + DEFAULT_USER_STACK_SIZE;
/* 参数块地址 */
ulong_t argBlockAddr = size;
size += argBlockSize;
/* 按相应大小创建一个进程 */
userContext = Create_User_Context(size);
```

在 Create_User_Context()函数中，除了创建用户上下文并进行初始化以外，还需要为进程创建相应的 LDT：

```
/* 初始化段描述
Init_LDT_Desc                                    erContext->ldt, NUM_USER_LDT_ENTRIES);
/* 新建一个 LD
userContext->ldtSelector = Selector(KERNEL_PRIVILEGE, true,
                                    Get_Descriptor_Index(userContext->ldtDescriptor));
/* 新建一个代码段描述符 */
Init_Code_Segment_Descriptor(&userContext->ldt[0], (ulong_t)userContext->memory,
                             size / PAGE_SIZE, USER_PRIVILEGE);
/* 新建一个数据段描述符 */
Init_Data_Segment_Descriptor(&userContext->ldt[1], (ulong_t)userContext->memory,
                             size / PAGE_SIZE, USER_PRIVILEGE);
/* 新建数据段和代码段选择子 */
userContext->csSelector = Selector(USER_PRIVILEGE, false, 0);
userContext->dsSelector = Selector(USER_PRIVILEGE, false, 1);
/* 将引用数清零 */
userContext->refCount = 0;
```

ushort_t User_Context::ldtSelector
Selector for the LDT's descriptor in the GDT

Start_User_Thread()函数中需要调用 Setup_User_Thread()函数，为进程初始化内核栈，栈中是为进程首次进入用户态运行时设置处理器状态要使用的数据：

```
/* 初始化用户态进程堆栈，使之看上去像刚被中断运行一样 */
/* 分别调用 Push 函数将以下数据压入堆栈 */
Push(kthread, dsSelector);  /* DS 选择子 */
Push(kthread, userContext->stackPointerAddr);  /* 堆栈指针 */
Push(kthread, eflags);  /* Eflags */
Push(kthread, csSelector);  /* CS 选择子 */
Push(kthread, userContext->entryAddr);  /* 程序计数器 */
Push(kthread, 0);  /* 错误代码(0) */
Push(kthread, 0);  /* 中断号(0) */
```

```
/* 初始化通用寄存单元，向 esi 传递参数块地址 */
Push(kthread, 0);  /* eax */
Push(kthread, 0);  /* ebx */
Push(kthread, 0);  /* ecx */
Push(kthread, 0);  /* edx */
Push(kthread, userContext->argBlockAddr);  /* esi */
Push(kthread, 0);  /* edi */
Push(kthread, 0);  /* ebp */
```

```
/* 初始化数据段寄存单元 */
Push(kthread, dsSelector);  /* ds */
Push(kthread, dsSelector);  /* es */
Push(kthread, dsSelector);  /* fs */
Push(kthread, dsSelector);  /* gs */
```

在 user.c 文件中补全函数 Switch_To_User_Context()，它用于在执行一个新的进程前切换用户地址空间，即通过更改 LDTR 切换到新的 LDT：

```
//之前最近使用过的 userContxt
static struct User_Context *s_currentUserContext;
//指向User_Conetxt的指针，并初始化为准备切换的进程
struct User_Context *userContext = kthread->userContext;
KASSERT(!Interrupts_Enabled());
// userContext为0表示此进程为核心态进程就不用切换地址空间
if (userContext == 0)
    return;
if (userContext != s_currentUserContext)
{
    //为用户态进程时则切换地址空间
    Switch_To_Address_Space(userContext);
    //新进程的核心栈指针
    ulong_t esp0 = ((ulong_t)kthread->stackPage) + PAGE_SIZE;
    //设置内核堆栈指针
    Set_Kernel_Stack_Pointer(esp0);
    //保存新的 userContxt
    s_currentUserContext = userContext;
}
```

最后补全一些向用户程序提供的系统调用函数，以及内核空间和用户空间之间的数据复制等等。

最终程序运行结果如下所示：

```
Welcome to GeekOS!
Spawning init process (/c/shell.exe)
$ pid
6
$ b 1 2
I am the b program
Arg 0 is b
Arg 1 is 1
Arg 2 is 2
$ c
I am the c program
Illegal system call -1 by process 8
$ long
Start Long
End Long
$ exit
DONE!
Init process exited with code 0
```

四、实验分析

与上一次实验加载内核线程相比，加载用户态线程的过程要复杂得多，这主要是因为内核级线程和用户级线程有很多不同之处。

首先它们的进程结构不同：内核进程没有 User_Context 结构体，只有 Kernel_Thread 结构体。而用户进程既拥有 Kernel_Thread 结构体，也拥有 User_Context 结构体。User_Context 结构体保存了用户栈地址、段描述符、段选择子、用户程序入口地址等内容。

其次它们的栈空间不同，内核进程只有有 1 个栈，就是内核栈；而用户有两个栈空间，内核栈和用户栈空间：当进程在用户态运行时，CPU 栈指针寄存器指向的是用户栈地址。当进程运行在内核态时，CPU 栈指针寄存器指向的是内核栈空间地址。其中用户栈地址保存在 User_Context 结构体中，内核栈地址保存在 Kernel_Thread 结构体中。

另外可以看到运行结果中，PID 的值为 6，这是因为在系统初始化之初已经运行了 5 个进程，因此我们的第一个用户程序的进程 ID 是从 6 开始的。这五个进程分别为：Idle、Reaper、Init_Floppy()、 Init_IDE()和 Main。

五、所遇问题及解决方法

本实验难度较大，需要对加载用户态线程的整个流程有一个十分清晰的把握，理解用户线程调度的基本过程，需要掌握 LDT 与 GDT 的联系和区别。代码量较大，所幸书上已给出参考代码的实现，需要仔细加以阅读理解，理清整个程序的脉络和各个函数的功能以及相互之间的调用关系。

在理解 LDT 与 GDT、用户上下文以及进程加载流程的过程中，我感到有些费劲，在网上查阅了相关的资料加以研读，并结合所给出的示例代码对比分析，才对这些概念有了个较为清晰的认知。

六、思考与练习

1. 用户程序的参数是如何传递给程序的？

在用户态进程首次被调用前，系统必须初始化用户态进程的堆栈，使之看上去像进程刚被中断运行一样。在 Setup_User_Thread 函数中，使用了 push 操作将所需数据都压入了堆栈中，初始化通用寄存器和数据段寄存器单元。其中使 esi 指向参数块地址：

```
/* 初始化通用寄存单元，向 esi 传递参数块地址 */
Push(kthread, 0);  /* eax */
Push(kthread, 0);  /* ebx */
Push(kthread, 0);  /* ecx */
Push(kthread, 0);  /* edx */
Push(kthread, userContext->argBlockAddr);  /* esi */
Push(kthread, 0);  /* edi */
Push(kthread, 0);  /* ebp */
```

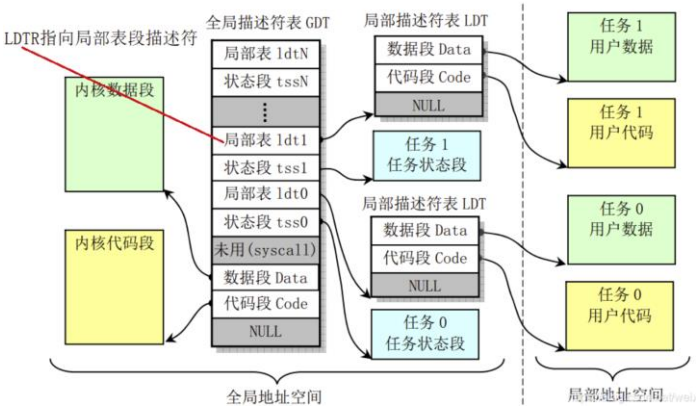
而用户参数块地址是在 Load_User_Program 函数中通过解析传递给 spawn 函数的 command 参数得到的：

```
/* 格式化参数块 */
Format_Argument_Block(userContext->memory + argBlockAddr, numArgs, argBlockAddr, command);
/* 初始化数据段、堆栈段及代码段信息 */
userContext->entryAddr = exeFormat->entryAddr;
userContext->argBlockAddr = argBlockAddr;
userContext->stackPointerAddr = argBlockAddr;
```

综上所述，用户程序的参数首先传递给 spawn 函数，接着 spawn 函数在调用 Load_User_Program 函数时会将命令行参数传递给它进行初始化用户内存空间中的参数块，在得到用户上下文后，spawn 函数继续调用 Start_User_Thread 函数创建一个新进程并使其进入就绪队列，其中需要为进程初始化内核堆栈，使得 esi 指向参数块地址，这样，用户程序就可以通过访问栈的方式得到传递给它的用户参数了。

2. 简要说明 LDT 与 GDT 的区别与联系。

每个用户态进程都拥有属于自己的内存段空间，如：代码段、数据段、栈段等，每个段有一个段描述符，并且每个进程有一个段描述符表（LDT），用于保存该进程的所有段描述符。操作系统中还设置一个全局描述符表（GDT），用于记录了系统中所有进程的 LDT 描述符。它们之间的关系可以用下图表示：



从图中我们可以清楚的看到，LDT 和 GDT 基本结构其实是类似的，LDT 内部存储的是进程相关的段的基址内容,GDT 存储的是内核代码的段的基址。GDT 只有一张且全局可见，可以通过 GDTR 寄存器找到；而 LDT 在系统中可以存在多个，且只对引用它们的任务可见，每个任务最多可以拥有一个 LDT。每一个 LDT 自身作为一个段存在，它们的段描述符被放在 GDT 中。因此我们可以把它们之间的关系看作：GDT 为一级描述符表，LDT 为二级描述符表。