

电子科技大学计算机科学与工程学院

实 验 报 告

(实验) 课程名称 计算机操作系统

电子科技大学

实 验 报 告

学生姓名：岳子豪 学 号：2018051404015 指导教师：刘杰彦

实验地点：清水河校区主楼 A2-412 实验时间：2020.11.8

一、实验室名称：主楼 A2-412

二、实验项目名称：进程与资源管理器设计

三、实验学时： 6 学时

四、实验原理：

利用高级程序语言的数据结构模拟操作系统中的进程和资源信息，设计指令输入与结果输出功能，用于模拟接收用户命令、反馈系统执行结果，并通过设计一定的算法，对操作系统进程与资源的管理进行模拟。

五、实验目的：

设计和实现进程与资源管理，并完成 Test shell 的编写，以建立系统的进程管理、调度、资源管理和分配的知识体系，从而加深对操作系统进程调度和资源管理功能的宏观理解和微观实现技术的掌握。

六、实验内容：

在实验室提供的软硬件环境中，设计并实现一个基本的进程与资源管理器。该管理器能够完成进程的控制，如进程创建与撤销、进程

的状态转换；能够基于优先级调度算法完成进程的调度，模拟时钟中断，完成对时钟中断的处理，在同优先级进程中采用时间片轮转调度算法进行调度；能够完成资源的分配与释放，并完成进程之间的同步。该管理器同时也能完成从用户终端或者指定文件读取用户命令，通过 Test shell 模块完成对用户命令的解释，将用户命令转化为对进程与资源控制的具体操作，并将执行结果输出到终端或指定文件中。

七、实验环境：

设计平台：Windows 10

设计语言：python 3.7

八、实验步骤：

1. 系统功能需求分析

该实验需设计模拟进程与资源管理系统，并实现驱动程序，通过用户输入的命令调用内核模拟函数，实现对模拟的进程与资源的管理。根据该系统需实现的功能，设计如下需求：

(1) 指令输入

包括从终端或者测试文件读取命令，并作为系统的输入。

(2) 进程调度

包括进程的创建、运行、就绪、阻塞、抢占、以及时间片轮转调度等功能。

(3) 资源管理

包括资源的初始化、申请、占有等功能。

(4) 结果输出

包括系统运行状态的反馈、进程和资源信息的输出等功能。

2. 总体框架设计

(1) 总体设计

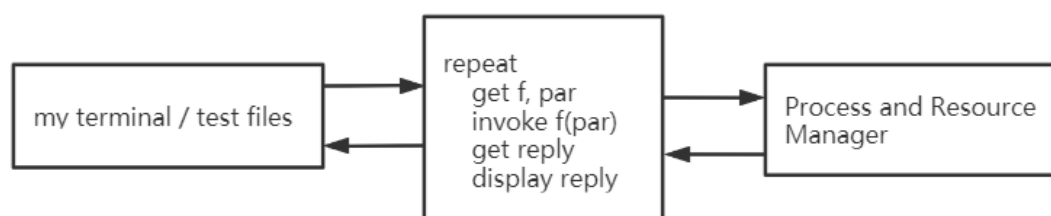


图 1 系统总体结构

系统总体架构如图 1 所示，左边部分为用户输入指令，通过终端（如键盘输入）或者测试文件来给出相应的用户命令，以及模拟硬件引起的中断。

中间为驱动程序 **test shell**，将调度所设计的进程与资源管理器来完成测试，并完成人机交互，解析用户指令、反馈内核运行结果。**Test shell** 应具有的功能有：

- 从终端或者测试文件读取命令；
- 将用户需求转换成调度模拟的内核函数；
- 在终端或输出文件中显示结果。

右边部分为进程与资源管理器，属于操作系统内核的功能。该管理器具有如下功能：

- 完成进程创建、撤销和进程调度；
- 完成多单元 (**multi_unit**)资源的管理，包括资源的申请和释放；
- 完成错误检测和定时器中断功能。

该系统的总体工作流程如图 2 所示。

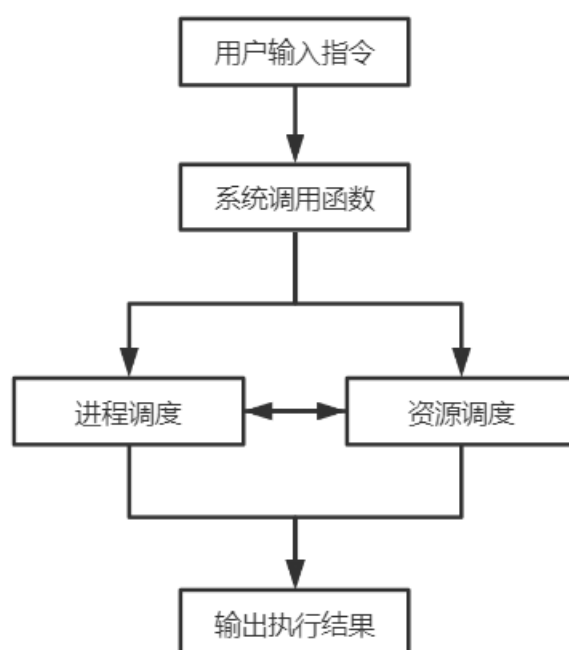


图 2 总体工作流程

(2) 模块设计

在该系统的具体设计与实现中，通过输入输出处理实现指令的输入和系统反馈的输出，通过各种函数模拟操作系统调度进程与资源的过程，根据实验要求，设计如下模块：

- (a) 指令获取与解析 (Input)
- (b) 进程创建 (Create)
- (c) 进程释放 (Release)
- (d) 进程抢占 (Preempt)
- (e) 进程撤销 (Destroy)
- (f) 超时 (Time out)

(g) 资源设置 (Set Resource)

(h) 资源请求 (Request)

(i) 插入就绪队列 (Insert Ready List)

(j) 插入阻塞队列 (Insert Block List)

(k) 状态与反馈信息输出 (Output)

通过各模块的紧密合作，实现对操作系统进程与资源管理的模拟，以上各个模块的详细介绍如下。

(3) 具体模块设计

(a) 指令获取与解析 (Input)

该模块包括用户指令的读取和解析两部分，由程序主函数完成。

其中，指令读取功能的实现包括两种方式：从终端中读取用户输入是命令和从指令文件中读取。按照实验要求，用户输入字符串形式的命令，因此，利用一个 `command` 列表存储。终端输入方法，首先输出 “`shell>`” 提示用户输入指令，而后将用户输入的指令存入列表中。读取文件方法，用户预先将指令写好存进 “`input.txt`” 文件里，每条命令之间以换行分隔，程序读取该文件并解析执行。终端输入方法与读取文件方法的代码如下：

```
command = []

# =====终端输入方法=====
# while(True):
#     command = input("shell>").split()

# =====读取文件方法=====
with open('input.txt') as f:
    instruction_list = []
    instruction = f.readline()
```

```

while instruction:
    instruction_list.append(instruction)
    instruction = f.readline()
for instruction in instruction_list:
    command = instruction.split()

```

指令的解析由预先设计好的指令匹配程序完成，通过主要通过 if-elif-else 结构来实现，在每一种指令的识别中，通过多级判断检测指令长度、参数合法性等确保接收正确的输入并对无效的输入反馈错误信息。有效输入将存入 command 列表作为输入控制程序运行。指令解析部分的代码如下：

```

# 跳过空白指令
if len(command)==0:
    continue
# 读取 init 指令
elif command[0]=='init' and len(command)==1:
    Create('init',0)
    print("{} process is running".format(Running))
# 读取 cr 指令
elif command[0]=='cr' and len(command)==3:
    if command[2] in ['0','1','2']:
        Create(command[1],command[2])
        new_PCB=get_PCB(command[1])
        print("process {} is running".format(Running))
# 读取 de 指令
elif command[0]=='de' and len(command)==2:
    if get_PCB(command[1])!="null":
        Destroy(command[1])
    else:
        print('Invalid requirement: {} is not in process
list.'.format(command[1]))
# 读取 req 指令
elif command[0]=='req' and len(command)==3:
    if command[1] in ['R1','R2','R3','R4']:
        if command[2].isalnum():
            amount=int(command[2])
            if amount>=0:
                RID_index = int(command[1][-1:]) - 1
                Request(RID_index,amount)

```

```

        else:
            print('Invalid requirement: resource requirement
amount must be natural number.')
        else:
            print('invalid requirement: resource requirement amount
must be integers.')
        else:
            print('Invalid requirement: {} is not a valid resource
name.'.format(command[1]))
# 读取 rel 指令
elif command[0]=='rel' and len(command)==3:
    if command[1] in ['R1','R2','R3','R4']:
        if command[2].isalnum():
            amount=int(command[2])
            if amount>=0:
                RID_index = int(command[1][-1:]) - 1
                release_running(RID_index,amount)
            else:
                print('Invalid requirement: resource release amount
must be natural number.')
        else:
            print('invalid requirement: resource release amount must
be integers.')
    else:
        print('Invalid requirement: {} is not a valid resource
name.'.format(command[1]))
# 读取 to 指令
elif command[0]=='to' and len(command)==1:
    time_out()
# 读取 list ready / list res / list block 指令
elif command[0]=='list' and len(command)==2:
    if command[1]=='ready':
        print_ready_list()
    elif command[1]=='res':
        print_available_resource_list()
    elif command[1]=='block':
        print_list_block()
# 读取 pr 指令
elif command[0]=='pr' and len(command)==2:
    print_pcb_info(command[1])
# 无效指令反馈
else:
    print('Invalid command.')

```


其中，复杂的嵌套 if-else 结构是为了保证系统能有效应对各种不合理的输入，使用户得以在输入错误的指令后得到反馈与引导，同时也避免因不合法指令而产生报错等，尽可能在实现实验需求的同时保证用户体验。

(b) 进程创建 (Create)

在本系统中，进程通过一个对象来表示，由定义的进程控制块 (PCB) 类创建。定义 PCB 类，其属性包括进程名 PID、占用资源 occupied_resources、进程状态 status、继承关系 creation_tree、优先级 priority、因请求何种资源而阻塞 blocked_RID_index、因请求多大数量的资源而阻塞 blocked_request_amount，类定义代码如下，注释中注明了部分属性的应用示例。

```
# 进程控制块类
class PCB(object):
    def __init__(self):
        # process name
        self.PID=''
        # occupied resource
        self.occupied_resources=[0,0,0,0]
        # type: ready, block, running
        self.status=''
        # creation_tree[0]:parent, creation_tree[1,2,...]:childs
        self.creation_tree=['null']
        # 0,1,2 (Init,User,System)
        self.priority=0 # 0,1,2 (Init,User,System)
        # blocked by which kind of resource. index=0 -> R1, index=1 -> R2...
        self.blocked_RID_index=-1
        # how much resource it needs to return ready status
        self.blocked_request_amount=0
```

创建进程通过函数 Create 实现。Create 的设计思想为：首先判断用户输入的进程名是否与进程列表中存在的进程重名，重名验证通过

之后利用 PCB 类定义一个对象 new，将用户输入的进程名和优先级存入 new 的相应属性，并将新进程插入进程列表。之后，若创建该进程时没有进程运行（如用户输入 init 指令），则新进程的父进程为自身，否则为创建时正运行的进程。对应地，将当前新建的进程添加至其父进程（当下运行进程）的子进程队列中，最后，判断是否抢占当前运行进程。代码如下，其中，get_PCB(PID)和 preempt_or_not(PID)分别为通过 PID 获取对应进程对象和通过 PID 决定该进程执行抢占或不抢占，之后会有详细介绍。

```
# 创建一个新进程
def Create(name, priority):
    global Process_List
    global Ready_List
    global Running
    global Creation_Tree
    # 同名判断
    for each in Process_List:
        if each.PID==name:
            print("Failed to create a new process. {} is already
created!".format(name))
            return
    new=PCB()
    new.PID=name
    new.priority=int(priority)
    Process_List.append(new) #添加至进程列表

    # 连接父进程
    if Running=="null":
        new.creation_tree[0]=new.PID
    else:
        new.creation_tree[0]=Running
    # 为父进程添加子进程
    running=get_PCB(Running) # 获取当前正在运行的进程
    running.creation_tree.append(new.PID) # 将新进程添加进其父进程的子树
中
```

```
preempt_or_not(new.PID) # 执行抢占或不抢占
```

(c) 进程释放（Release）

进程释放有两种实现，首先是实验所要求的释放当前进程所占用的部分资源，通过函数 `release_running(RID_index,amount)` 实现，用户输入释放资源的种类与数量，调用该函数释放进程占用资源。函数先获取当前进程，检查输入的合法性（资源名称是否合法）和合理性（所释放资源的量是否符合要求），通过之后更新可用资源表和当前进程占用资源表，最后检查阻塞队列，判断在新的资源条件下是否有进程能被唤醒。其代码实现如下：

```
# 释放当前进程占用的部分资源
def release_running(RID_index,amount):
    global Running
    global Resource_Vector
    global Process_List
    # 获取当前进程
    process=get_PCB(Running)

    if RID_index not in [0,1,2,3]:
        print('Invalid requirement: there is no resource named
        {}'.format(RID_index+1))
        return
    if process.occupied_resources[RID_index]<amount:
        print('Invalid requirement: process {} only occupies {}
        R{}'.format(process.PID,process.occupied_resources[RID_index],RID_index+
        1))
        return
    # 更新可用资源表与当前进程占用资源表
    Resource_Vector[RID_index].available+=amount
    process.occupied_resources[RID_index]-=amount
    # 检查阻塞队列，尝试唤醒进程
    check_block_list()
```

第二种实现是在第一种的基础上的补充，为后续进程撤销模块提

供便利。通过函数 `release_all_by_pid(PID)`，传入 `PID` 调用该函数释放 `PID` 对应进程所占用的所有资源。首先判断是否存在 `PID` 对应进程，如不存在则 `return`，否则更新资源向量表，将进程占用的所有资源添加到可用资源向量表，并将占用资源向量表清零。最后调用 `check_block_list()`并返回其返回值。`check_block_list()`的返回值是检查阻塞队列之后唤醒的所有进程，以存储对象的列表形式返回。`release_all_by_pid(PID)`返回该列表用于将唤醒进程传入撤销进程的函数中进一步使用，稍后将对这一点进行介绍。`release_all_by_pid(PID)`对应代码如下：

```
# [子函数]释放进程占用的所有资源
def release_all_by_pid(PID):
    global Process_List
    global Ready_List
    global Block_List
    global Resource_Vector
    global Running

    process=get_PCB(PID)
    # 判断是否存在该进程
    if process=="null":
        print('Invalid requirement: there is no process which is named {} in
the process list'.format(PID))
        return
    # 释放资源，更新资源向量表
    for i in range(4):
        Resource_Vector[i].available=Resource_Vector[i].available+
process.occupied_resources[i]
        process.occupied_resources[i]=0

    return check_block_list()
```

`check_block_list()`用于检查阻塞队列，判断在当下资源情况下是否可以唤醒阻塞进程，按照阻塞队列中的顺序依次对阻塞进程进行判

断，决定是否唤醒阻塞进程。若当前资源能满足阻塞进程的请求，则将其唤醒，唤醒流程包括：移出阻塞队列、判断是否需要抢占当前进程、满足进程需求并更新资源表、更新 PCB 信息、将其添加至唤醒列表用于将相关信息反馈给用户。对应代码如下：

```
# 检查阻塞队列，尝试唤醒进程
def check_block_list():
    global Block_List
    global Resource_Vector

    wakeup=[]
    for each in Block_List:
        RID_index=each.blocked_RID_index
        if each.blocked_request_amount >
Resource_Vector[RID_index].available:
            continue
        else:
            Block_List.remove(each) # 移出阻塞队列
            preempt_or_not(each.PID) # 判断是否抢占
            Resource_Vector[RID_index].available-=
each.blocked_request_amount # 满足其请求，更新资源表
            each.blocked_RID_index=-1 # 重置 PCB 中的阻塞资源种类
            each.blocked_request_amount=0 # 重置 PCB 中的阻塞资源需求量
            wakeup.append(each) # 添加至唤醒列表，以便反馈信息给用户
    return wakeup
```

(d) 进程抢占 (Preempt)

进程抢占包括是否抢占的判断和抢占动作的执行，通过函数 `preempt_or_not(PID)` 函数来实现。先判断当下是否有进程正在运行，若无，则运行进程，修改相关状态参量；否则根据优先级判断是否抢占，若需抢占，则将当前运行的进程更改为该进程，该进程状态改为运行中，将之前正在运行的进程的状态改为就绪，并插入就绪队列；否则，将该进程改为就绪态插入就绪队列。对应代码如下：

```

# 执行抢占或不抢占
def preempt_or_not(PID):
    global Running
    global Process_List
    # 决定进程状态

    process=get_PCB(PID)
    running=get_PCB(Running)
    # 若当前没有进程正在运行，则运行该进程
    if Running=='null':
        process.status='running'
        Running=process.PID
    # 根据优先级判断是否抢占
    else:
        running=get_PCB(Running)
        if running.priority<process.priority:
            process.status='running'
            running.status='ready'
            Running=process.PID
            insert_RL(running) # 将当前进程插入就绪队列
        else:
            process.status='ready'
            insert_RL(process) # 将新建进程插入就绪队列

```

(e) 进程撤销（Destroy）

进程撤销包括撤销该进程和其子孙进程，通过 Destroy(PID)函数来实现。撤销步骤包括：获取该进程，释放其占有的资源、将其从进程列表中移除、根据该进程状态将其从对应列表中移除（如果是正在运行的进程则用另一进程替换它）、将进程唤醒的情况反馈给用户、递归调用撤销其所有子进程。对应代码如下。

```

# 撤销进程
def Destroy(PID):
    global Process_List
    global Ready_List
    global Block_List
    global Resource_Vector

```

```

global Running

process=get_PCB(PID) # 获取当前进程
# 两级 return: check_block_list() -> release_all_by_pid() -> wakeup
wakeup=release_all_by_pid(PID) #释放 PID 占用资源并获取唤醒进程

Process_List.remove(process) # 从进程队列中移除
# 若撤销正在运行的进程，则就绪队列首个进程换入
if process.status=='running':
    Running=Ready_List[0].PID
# 若撤销阻塞进程，则将其从阻塞队列中移除
elif process.status=='blocked':
    Block_List.remove(process)
# 若撤销就绪队列，则将其从就绪队列中移除
elif process.status=='ready':
    Ready_List.remove(process)

#递归撤销所有子孙进程
for each in process.creation_tree[1:]:
    if(each=='null'):
        return
    else:
        Destroy(each)

# 唤醒每一个可唤醒的进程（考虑一次唤醒多个进程的情况）
for each in wakeup:
    if each in Process_List:
        print("release R{}. wake up process
{}".format(each.blocked_RID_index+1,each.PID))
        each.blocked_RID_index = -1 # 重置 PCB 中的阻塞资源种类

```

(f) 超时 (Time out)

超时通过 `time_out()` 来实现，通过用户指令模拟系统时钟，对当前进程进行中断。超时的流程包括：将当前进程状态更改为就绪态、降低进程优先级、从就绪队列中选择一个进程替换当前进程、将调度情况反馈给用户。

```
# 超时
```

```

def time_out():
    global Running
    global Ready_List
    # 将当前运行的进程移至就绪队列
    running=get_PCB(Running)
    running.status='ready'
    # 降低当前程序优先级（最低降到1）
    if running.priority>1:
        running.priority-=1
    last_running_PID=Running
    if len(Ready_List)!=0:
        insert_RL(running)
        # 将就绪队列的第一个进程调整为运行状态
        Running=Ready_List[0].PID
        Ready_List[0].status="running"
        # 将替换超时进程从就绪队列中弹出
        Ready_List.pop(0)
        print("process {} is running. ".format(Running),end='')
        if last_running_PID!=Running:
            print("process {} is ready. ".format(last_running_PID),end='')
        print('')

```

(g) 资源设置（Set Resource）

在本系统中，各类资源均通过对象来表示，由定义的资源（Resource）类创建。定义 Resource 类，其属性包括资源总量 total、可用资源量 available，类定义代码如下，注释中注明了部分属性的应用示例。

```

# 资源类
class Resource(object):
    def __init__(self):
        self.total=0
        self.available=0

```

资源初始化通过函数 set_resource_vector()实现。分别为每一种资源创建一个对象之后，将这些对象存入列表，并对其相应属性进行初

始化。按照实验要求，第 i 种资源 R_i 的总量为 i 。对应代码如下：

```
# 初始化资源表
def set_resource_vector():
    global Resource_Vector
    R1=Resource()
    R2=Resource()
    R3=Resource()
    R4=Resource()
    Resource_Vector=[R1,R2,R3,R4]
    for i in range(4):
        Resource_Vector[i].total=Resource_Vector[i].available=i+1
```

(h) 资源请求 (Request)

资源请求模块中，针对可用资源是否足够分配的两种情况提供了不同的处理方案，通过 Request(RID_index,amount)函数实现。首先检查用户输入的指令是否合理，输入合理的指令后，若进程所请求资源的量大于可用资源的量，则阻塞该进程，将其插入阻塞队列并更新其 PCB；若请求资源的量小于可用资源的量，则执行资源分配，更新资源向量表和 PCB。对应代码如下：

```
# 运行中的进程请求资源
def Request(RID_index,amount):
    global Process_List
    global Ready_List
    global Block_List
    global Resource_Vector
    global Running
    # 检查指令是否合理
    if(Running=="null"):
        print('Invalid requirement: there is no running process')
        return
    if RID_index not in [0,1,2,3]:
        print('Invalid requirement: there is no resource named
R{}'.format(RID_index+1))
        return
```

```

running=get_PCB(Running)
# 如果可用资源无法满足请求, 则阻塞
if(Resource_Vector[RID_index].available < amount):
    # print("Failed to request: there is no enough available R{}
resource".format(RID_index+1))
    # 阻塞当前进程
    running.status='blocked'
    insert_BL(running)
    running.blocked_RID_index=RID_index
    running.blocked_request_amount=amount
# 从就绪队列中选择一个进程运行
last_running_PID=Running
if len(Ready_List)==0:
    Running="null"
else:
    Running=Ready_List[0].PID
    Ready_List[0].status='running'
    Ready_List.pop(0)
    print('process {} is running. process {} is
blocked'.format(Running,last_running_PID))
# 如果可用资源可以满足请求, 则分配
else:
    running.occupied_resources[RID_index]+=amount
    Resource_Vector[RID_index].available-=amount
    print('process {} requests {}
R{}'.format(Running,amount,RID_index+1))

```

(i)插入就绪队列（Insert Ready List）

插入就绪队列模块的功能是将某一进程插入就绪队列，通过insert_RL(process)函数来实现，其参数为进程对象。由于就绪队列本来就是按进程的优先级排序的有序列表，因此可以采用从前往后逐个比对的方法确定插入的位置，将进程插入就绪队列。对应代码如下：

```

# 将进程插入就绪队列
def insert_RL(process):
    global Ready_List
    # 往有序数列表中插入元素的算法
    insert_index=len(Ready_List)

```

```
for i in range(len(Ready_List)):
    if Ready_List[i].priority < process.priority:
        insert_index=i
        break
Ready_List.insert(insert_index,process)
```

(j) 插入阻塞队列（Insert Block List）

插入阻塞队列模块的功能是将某一进程插入阻塞队列，通过 insert_BL(process)函数来实现，其参数为进程对象。其实现方法与 insert_RL(process)相似。对应代码如下：

```
# 将进程插入阻塞队列
def insert_BL(process):
    global Block_List
    insert_index= len(Block_List)
    for i in range(len(Block_List)):
        if Block_List[i].priority < process.priority:
            insert_index=i
            break
    Block_List.insert(insert_index,process)
```

(k) 状态与反馈信息输出（Output）

状态与反馈信息模块主要负责将系统的状态、反馈等信息输出，展示给用户。该模块除了穿插在各个模块和主函数之中的输出信息外，还包括若干种指定的输出内容，包括就绪队列、阻塞队列、各资源下的阻塞情况、进程的 PCB 信息、可用资源向量表信息的输出，分别通过 print_ready_list()、print_block_list()、print_list_block()、print_pcb_info()、print_available_resource_list()来实现。各函数对应代码及输出结果示例如下：

print_ready_list()

```
# 打印就绪队列
def print_ready_list():
    running=get_PCB(Running)
    print_running_flag=0
    # 按优先级依次降低的顺序输出就绪队列中的进程
    for p in range(2,-1,-1):
        print(p,end=':')
        # （按实验要求）将正在运行的进程作为就绪队列同等优先级的首位打印出来
        if running.priority==p:
            print('{:}'.format(running.PID), end='')
            print_running_flag=1
        for i in range(len(Ready_List)):
            # 莫名其妙的 if-else 问就是为了保证输出格式！
            if Ready_List[i].priority==p:
                if print_running_flag and running.priority==p:
                    print('-',end='')
                    print_running_flag=0
                print(Ready_List[i].PID,end='')
                if i==len(Ready_List)-1:
                    break
                elif Ready_List[i+1].priority!=p:
                    break
            if Ready_List[i].priority==p:
                print('-',end='')
        print(end='\n')
```

```
2:
1:x-p-q-r
0:init
```

图 3 print_ready_list()输出结果示例

print_block_list()

```
# 打印阻塞队列
def print_block_list():
```

```

for p in range(2,-1,-1):
    print(p,end=':')
    for i in range(len(Block_List)):
        if Block_List[i].priority==p:
            print(Block_List[i].PID,end='')
            if i==len(Block_List)-1:
                break
            elif Block_List[i+1].priority!=p:
                break
        if Block_List[i].priority==p:
            print('-',end='')
    print(end='\n')

```

print_list_block()

```

# 打印各种资源下的阻塞情况
def print_list_block():
    global Block_List
    for i in range(4):
        print('R{} '.format(i+1),end='')
        print_line_flag=0
        for j in range(len(Block_List)):
            if Block_List[j].blocked_RID_index==i:
                if print_line_flag==1:
                    print('-',end='')
                    print_line_flag=0
                print(Block_List[j].PID,end='')
                if j==len(Block_List)-1:
                    break
                if print_line_flag==0:
                    print_line_flag=1
        print(end='\n')

```

print_list_block()与 print_block_list()的区别在于，前者以阻塞原因（因何种资源而阻塞）为依据划分，后者以进程优先级为依据划分，两者的输出效果对比如下：

print_list_block()

```
R1
R2 r
R3 p
R4 q
```

print_block_list()

```
2:
1:r-q-p
0:
```

图 4 print_list_block()与 print_block_list()输出结果对比

print_pcb_info()

```
# 打印进程信息
def print_pcb_info(PID):
    global Process_List

    process=get_PCB(PID)
    if process!="null":
        print('PCB information about {}'.format(PID))
        print('* PID:\t\t\t\t\t{}'.format(PID))
        print('* priority:\t\t\t\t\t{}'.format(process.priority))
        print('* status:\t\t\t\t\t{}'.format(process.status))
        print('* parent:\t\t\t\t\t{}'.format(process.creation_tree[0]))
        print('* child:\t\t\t\t\t{}'.format(process.creation_tree[1]))
        print('* Occupied
Resource:\t{}'.format(process.occupied_resources))
    else:
        print('Invalid requirement: there is no process which is named {} in
the process list'.format(PID))
```

```
PCB information about q:
* PID:                q
* priority:           1
* status:             blocked
* parent:             x
* child:              []
* Occupied Resource:  [0, 0, 3, 0]
```

图 5 print_pcb_info()输出结果示例

`print_available_resource_list()`

```
# 打印可用资源表
def print_available_resource_list():
    global Resource_Vector
    for i in range(4):
        print('R{} {}'.format(i+1,Resource_Vector[i].available))
```

```
R1 1
R2 1
R3 0
R4 1
```

图 6 `print_available_resource_list()`输出结果示例

九、实验数据及结果分析：

使用跟实验指导书中一致的指令通过指令文件读取和终端输入两种方式测试系统功能的有效性和正确性。

1. 指令文件方式

首先创建“input.txt”文件，将测试命令输入并保存，然后运行程序，读取该文件并执行指令。输入与输出结果如下：

输入：

```
cr x 1
cr p 1
cr q 1
cr r 1
list ready

to
req R2 1
```

```
to
req R3 3
to
req R4 3
list res
```

```
to
to
req R3 1
req R4 2
req R2 2
list block
```

```
to
de q
to
to
```

输出结果:

```
init process is running
process x is running
process x is running
process x is running
process x is running
2:
1:x-p-q-r
0:init
process p is running. process x is ready.
process p requests 1 R2
process q is running. process p is ready.
process q requests 3 R3
process r is running. process q is ready.
process r requests 3 R4
R1 1
R2 1
R3 0
R4 1
process x is running. process r is ready.
process p is running. process x is ready.
```



```

process q is running. process p is blocked.
process r is running. process q is blocked.
process x is running. process r is blocked.
R1
R2 r
R3 p
R4 q
process x is running.
release R0. wake up process p
process p is running. process x is ready.
process x is running. process p is ready.

```

图 7 展示了该程序的运行结果与实验指导书中的参考运行结果的对比，(a)为程序的运行结果，(b)为参考结果。可以看出，在同样的输出下，运行结果与参考结果完全一致，效果较好，整体上能满足实验要求。

<pre> input.txt - 记事本 文件(F) 编辑(E) 格式(O) init cr x 1 cr p 1 cr q 1 cr r 1 list ready to req R2 1 to req R3 3 to req R4 3 list res to to req R3 1 req R4 2 req R2 2 list block to de q to to </pre>	<pre> Run: Exp1-Scheduler-1 (1) D:\newdesktop\项目\py\OS-experiments-environment init process is running process x is running process x is running process x is running process x is running 2: 1:x-p-q-r 0:init process p is running. process x is ready. process p requests 1 R2 process q is running. process p is ready. process q requests 3 R3 process r is running. process q is ready. process r requests 3 R4 R1 1 R2 1 R3 0 R4 1 process x is running. process r is ready. process p is running. process x is ready. process q is running. process p is blocked. process r is running. process q is blocked. process x is running. process r is blocked. R1 R2 r R3 p R4 q process x is running. release R0. wake up process p process p is running. process x is ready. process x is running. process p is ready. </pre>	<pre> init cr x 1 cr p 1 cr q 1 cr r 1 list ready 2: 1:x-p-q-r 0:init process p is running. process x is ready process p requests 1 R2 process q is running. process p is ready process q requests 3 R3 process r is running. process q is ready process r requests 3 R4 R1 1 R2 1 R3 0 R4 1 process x is running. process r is ready process p is running. process x is ready process q is running. process p is blocked. process r is running. process q is blocked. process x is running. process r is blocked. R1 R2 r R3 p R4 q process x is running. release R3. wake up process p process p is running. process x is ready. process x is running. process p is ready. </pre>
--	---	---

(a) 运行结果

(b) 参考结果

图 7 测试文件运行结果与指导书参考结果对比

2. 终端命令方式

采取终端输入的方式对系统进行测试。

点击运行程序后，控制台中显示“shell>”，表示已就绪，准备接收指令，并提示用户输入指令，如图 8 所示。

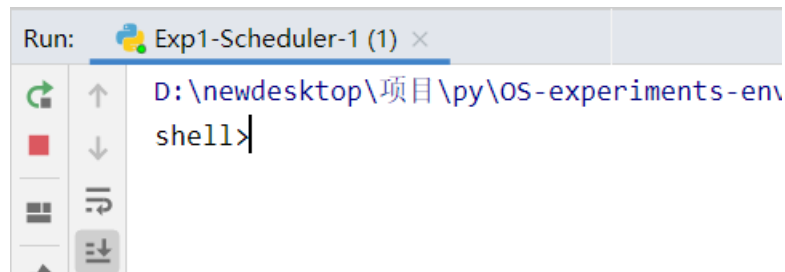


图 8 运行结果图示（1）

用户输入'init'指令，控制台输出“init process is running”，并再次输出“shell>”，准备接受下一条指令。该过程对应程序调用函数 Create('init',0)，创建了一个名为 init 的初始化进程，其优先级为 0。

```
shell>init
init process is running
shell>
```

图 9 运行结果图示（2）

再输入“cr x 1”指令，控制台输出“process x is running”，并再次输出“shell>”，准备接受下一条指令。该过程对应程序调用函数 Create('init',0)，创建了一个名为 init 的初始化进程，其优先级为 0。

```
shell>init
init process is running
shell>cr x 1
process x is running
shell>
```

图 10 运行结果图示（3）

输入“cr p 1”指令，控制台输出“process x is running”。由于 p 与 x 优先级相同，故 p 进程进入就绪队列等待被执行，当下运行进程

依然为 x。

```
shell>init
init process is running
shell>cr x 1
process x is running
shell>cr p 1
process x is running
shell>|
```

图 11 运行结果图示（4）

输入“cr q 1”指令，控制台输出“process x is running”，同理。

```
shell>init
init process is running
shell>cr x 1
process x is running
shell>cr p 1
process x is running
shell>cr q 1
process x is running
shell>|
```

图 12 运行结果图示（5）

输入“cr r 1”指令，控制台输出“process r is running”，同理。

```
shell>init
init process is running
shell>cr x 1
process x is running
shell>cr p 1
process x is running
shell>cr q 1
process x is running
shell>cr r 1
process x is running
shell>|
```

图 13 运行结果图示（6）

输入“list ready”指令，控制台分别输出优先级为 2、1、0 的进

程的就绪队列，在每个队列中按照进程的先后次序进行排序。

```
shell>list ready
2:
1:x-p-q-r
0:init
shell>|
```

图 14 运行结果图示（7）

输入“to”指令，控制台输出“process p is running. process x is ready”，表示之前正在运行的进程 x 因超时被放进就绪队列，就绪队列中最高优先级队列的第一个进程转为运行态。

```
shell>to
process p is running. process x is ready.
shell>|
```

图 15 运行结果图示（8）

输入“req R2 1”指令，控制台输出“process p requests 1 R2”，表示当下正在运行的进程 p 请求了 1 个 R2 资源。

```
shell>to
process p is running. process x is ready.
shell>req R2 1
process p requests 1 R2
shell>|
```

图 16 运行结果图示（9）

重复上述过程，运行结果如下。

```

shell>to
process p is running. process x is ready.
shell>req R2 1
process p requests 1 R2
shell>to
process q is running. process p is ready.
shell>req R3 3
process q requests 3 R3
shell>to
process r is running. process q is ready.
shell>req R4 3
process r requests 3 R4
shell>|

```

图 17 运行结果图示（10）

输入“list res”指令，控制台输出 4 种资源各自的可用量。

```

shell>list res
R1 1
R2 1
R3 0
R4 1
shell>|

```

图 18 运行结果图示（10）

再通过“to”指令，分别令 r 和 x 超时，使得运行进程为 p。

```

shell>to
process x is running. process r is ready.
shell>to
process p is running. process x is ready.
shell>req R3 1
process q is running. process p is blocked.
shell>|

```

图 19 运行结果图示（11）

连续使用“req”指令请求资源，先后将 p、q 和 r 阻塞，x 处于运行状态。

```

shell>to
process x is running. process r is ready.
shell>to
process p is running. process x is ready.
shell>req R3 1
process q is running. process p is blocked.
shell>req R4 2
process r is running. process q is blocked.
shell>req R2 2
process x is running. process r is blocked.
shell>|

```

图 20 运行结果图示（12）

使用“list block”指令，输出阻塞队列，可以看到 R2、R3 和 R4 资源下分别由 r、p、q 三个进程阻塞，与预期相符。

```

shell>list block
R1
R2 r
R3 p
R4 q
shell>|

```

图 21 运行结果图示（13）

使用“to”指令使 x 超时，但由于就绪队列中没有其它进程，故 x 继续执行。

```

process x is running.
shell>to
process x is running.
shell>de q
release R0. wake up process p
shell>|

```

图 22 运行结果图示（14）

使用“de q”指令撤销进程 q，阻塞进程 q 占有资源 R0，撤销后 R0 被释放，进程 p 被唤醒。

```

shell>to
process x is running.
shell>to
process x is running.
shell>de q
release R0. wake up process p
shell>to
process p is running. process x is ready.
shell>to
process x is running. process p is ready.
shell>|

```

图 23 运行结果图示（15）

接下来是附加功能的测试。分别输入指令“pr x”、“pr p”、“pr r”，控制台输出进程 x、p、r 的 PCB 信息，包括 PID、priority、status、parent、child、Occupied Resource。

```

shell>pr x
PCB information about x:
* PID:                x
* priority:            1
* status:              running
* parent:              init
* child:               ['p', 'q', 'r']
* Occupied Resource:   [0, 0, 0, 0]
shell>pr p
PCB information about p:
* PID:                p
* priority:            1
* status:              ready
* parent:              x
* child:               []
* Occupied Resource:   [0, 1, 0, 0]
shell>pr r
PCB information about r:
* PID:                r
* priority:            1
* status:              blocked
* parent:              x
* child:               []
* Occupied Resource:   [0, 0, 0, 3]
shell>|

```

图 24 运行结果图示（16）

经过逐步测试与分析，该程序能正确运行，得到预期结果。

3. 其它测试

上述两种测试方法都是在完成整个程序之后对程序的整体运行情况进行测试，在实验的过程中，我也针对每一个模块进行了单独的测试，以验证其正确性。如插入就绪队列模块的 `insert_RL(PID)` 函数。测试时，先初始化一些进程，并创建新的进程调用 `insert_RL(PID)`。若输出的就绪队列信息与实际分析结果一致，则说明该函数正常运行且功能正确。以下图所示的输入为例：

```
shell>init
init process is running
shell>cr p1 1
process p1 is running
shell>cr p2 1
process p1 is running
shell>cr p3 2
process p3 is running
shell>cr p4 1
process p3 is running
shell>cr p5 2
process p3 is running
shell>list ready
2:p3-p5
1:p2-p1-p4
0:init
shell>
```

图 25 运行结果图示（2）

该过程一共创建了 6 个进程：init、p1、p2、p3、p4、p5，其中，init 为初始化进程，优先级最低；p1 创建后，init 移至就绪队列的 0 优先级队列，p2 与 p1 优先级相同，创建后移至就绪队列的 1 优先级

队列；p3 优先级为 2，创建后抢占 CPU，系统将 p1 移至就绪队列的末端；p4 创建后，加入优先级为 1 的就绪队列；p5 优先级为 2，加入优先级为 2 的就绪队列。因此，就绪队列中应为 2:p5; 1:p2-p1-p4; 0:init。按照实验要求，就绪队列的输出包括正在运行的进程，因此 2 中应为 p3-p5。可见，通过调用函数，程序的运行结果与预期一致，可认为该模块能正确工作。与之类似的测试贯穿在整个实验过程中，在此不一一列举。

十、实验结论：

结合计算机操作系统进程管理与调度、资源管理与分配的知识体系，利用 python 实现了一个用于模拟进程和资源管理的管理器，能接收并执行用户指令，对虚拟进程和资源进行管理，并输出反馈信息。通过测试与分析，该系统在测试范围内能正常工作、得到正确结果，符合实验要求。

十一、总结及心得体会：

作为本科期间为数不多的几个百行代码大实验之一，本次实验让我颇有心得，受益匪浅。从最开始的毫无头绪，到从最简单的功能开始着手，不断添砖加瓦，一点一点将所有功能（包括部分附加功能）全部实现，再经过不断地调试、修改，寻找可能存在的 bug，完善输出样式等细节，直到最后功能符合预期，测试结果与样例结果完全相同，并通过老师检阅，整个过程虽然漫长而艰辛，但结果令人充满成就感。尤其是解决最后一个已知的 bug，把实验进度 todo-list 里的最后一个 TODO 变成 DONE，合上电脑那一瞬间的轻松与愉悦，让我

觉得自己的认真对待是完全值得的。通过自己编码实现模拟进程与资源调度，不但加深了我对操作系统进程与资源调度相关知识的掌握，让我更加深刻地理解了计算机调度进程与资源的基本方式，也对我的设计和编码能力有较大的提升，此外，还让我在不断地寻找问题、思考解决方案的过程中锻炼了自己解决问题的能力。

十二、对本实验过程及方法、手段的改进建议：

本次实验尽管成功实现了实验要求，整体效果较好，但依然存在诸多不足之处，如：

1. 缺乏预先规划，功能逻辑混乱。

虽然在实现系统功能的过程中全部采用函数思想，极大得简化了编码的工作量，提高了代码的可读性，但函数的设计以及函数之间的互相调用拖泥带水，一定程度上增大了修改、维护的复杂度和读者理解代码的难度，并且容易出错。下图是本次实验中定义的所有函数名。

```
30 # 初始化资源表
31 def set_resource_vector():...
40 # 创建一个新进程
41 def Create(name, priority):...
66 # 执行抢占或不抢占
67 def preempt_or_not(PID):...
89 # 将进程插入就绪队列
90 def insert_RL(process):...
99 # 将进程插入阻塞队列
100 def insert_BL(process):...
108 # [子函数] 释放进程占用的所有资源
109 def release_all_by_pid(PID):...
127 # 释放当前进程占用的部分资源
128 def release_running(RID_index, amount):...
146 # 检查阻塞队列，尝试唤醒进程
147 def check_block_list():...
164 # 撤销进程

165 def Destroy(PID):...
197 # 运行中的进程请求资源
198 def Request(RID_index, amount):...
234 # 超时
235 def time_out():...
256 # 通过PID获取进程对象
257 def get_PCB(PID):...
263 # 打印可用资源表
264 def print_available_resource_list():...
268 # 打印就绪队列
269 def print_ready_list():...
293 # 打印阻塞队列
294 def print_block_list():...
307 # 打印各种资源下的阻塞情况
308 def print_list_block():...
324 # 打印进程信息
325 def print_pcb_info(PID):...
```

图 26 函数名列表

以阻塞进程为例，阻塞进程包含两部分内容：更改进程属性、插入阻塞队列，本实验中，两部分分别通过分散在各个函数里的更改进

程属性的语句和 `insert_BL(process)` 来实现，但更合理的做法应该是将两部分封装进同一个函数，如 `block_process(process)`，一方面保证操作的原子性，使得每次阻塞进程时不会有操作的冗余或者遗漏，另一方面也让编码过程更加便捷、程序更加精练、代码可读性更好。类似的例子还有不少，系统结构和功能逻辑有待优化。

导致这种问题的原因在于：（1）编码过程是添砖加瓦式地逐步完善功能，缺乏统一的规划，导致后期新增功能和维护程序的工作繁杂；（2）设计功能的时候以实验要求的功能为导向，更偏向于使一个函数直接实现一个完整的功能，尽管有些功能从逻辑上讲更适合由多个有序操作组合完成。这也导致了代码结构的混乱。

2. 属性标识不统一

以 `insert_BL(process)` 和 `Destroy(PID)` 为例，在设计函数的过程中，先完成的部分代码统一使用 `process` 作为函数的入口参数，类型是 `PCB` 类；而后完成的代码统一以 `PID` 为函数的入口参数，类型是整型。这是由于我在编码的过程中逐渐发现使用 `PID` 作为参数在大部分情况下更为便捷，因此逐渐弃用 `process` 作为函数参数。与之类似的还有 `RID_index`。

此外，还有一些同样不影响功能、但是从设计与实现的角度来看还有待优化的其它问题，在此不一一列举。而我之所以没有对整个代码进行大幅的修整以解决这些存在的问题，最主要的阻力在于时间与精力有限，作为大三刚转到新专业的学生，一学期上两学期的课，课业压力巨大，也恳请老师与助教学长理解。此外，上述问题也并非一

无是处，如果封装过于干脆利落，一定程度上将对中间过程值的读取造成困难，需要额外的解决方案。

总之，本实验过程中确实存在一些值得改进的地方，这些问题将为我后续的学习与工作提供重要参考，成为我在未来的成长道路中的宝贵经验。

报告评分：

指导教师签字：

电子科技大学计算机科学与工程学院

实 验 报 告

(实验) 课程名称 计算机操作系统

电子科技大学

实验报告

学生姓名：岳子豪 学号：2018051404015 指导教师：刘杰彦

实验地点：清水河校区主楼 A2-412 实验时间：2020.12.05

一、实验室名称：主楼 A2-412

二、实验项目名称：内存地址转换实验

三、实验学时：2 学时

四、实验原理：

1. 逻辑地址到线性地址的转换

逻辑地址：Intel 段式管理中：“一个逻辑地址，是由一个段标识符加上一个指定段内相对地址的偏移量，表示为 [段标识符：段内偏移量]。”

段标识符：也称为段选择符，属于逻辑地址的构成部分，段标识符是由一个 16 位长的字段组成，其中前 13 位是一个索引号。后面 3 位包含一些硬件细节：



图 1 段选择符

索引号：可以看作是段的编号，也可以看做是相关段描述符在段表中

的索引位置。系统中的段表有两类：GDT 和 LDT。

GDT：全局段描述符表，整个系统一个，GDT 表中存放了共享段的描述符，以及 LDT 的描述符（每个 LDT 本身被看作一个段）。

LDT：局部段描述符表，每个进程一个，进程内部的各个段的描述符，就放在 LDT 中。

TI 字段：Intel 设计思想是：一些全局的段描述符，就放在“全局段描述符表(GDT)”中，一些局部的，例如每个进程自己的，就放在所谓的“局部段描述符表(LDT)”中。那究竟什么时候该用 GDT，什么时候该用 LDT 呢？这是由段选择符中的 TI 字段表示的，TI=0，表示相应的段描述符在 GDT 中，TI=1 表示表示相应的段描述符在 LDT 中。

段描述符(即段表项)：具体描述了一个段。在段表中，存放了很多段描述符。我们可以通过段标识符的前 13 位，直接在段描述符表中找到一个具体的段描述符，也就是说，段标识符的前 13 位是相关段描述符在段表中的索引位置。

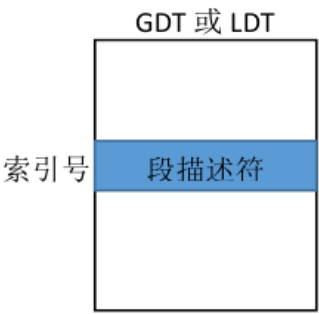
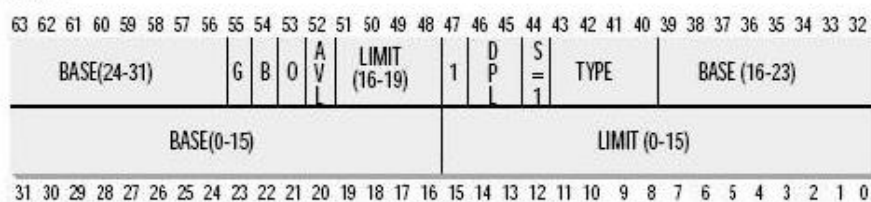


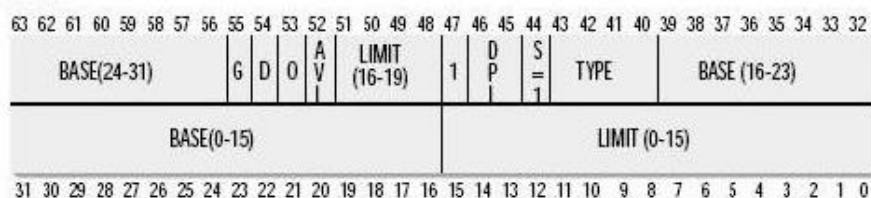
图 2 GDT 或 LDT

示例每一个段描述符由 8 个字节组成，如图 3：

数据段描述符



代码段描述符



系统段描述符

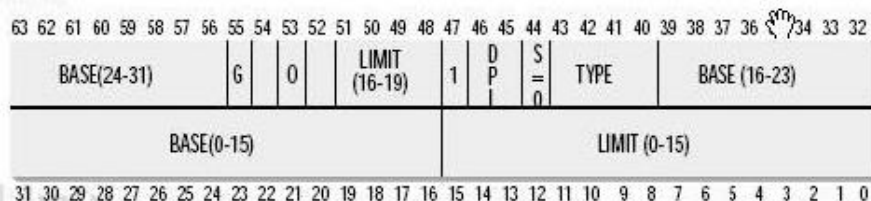


图 3 段描述符

Base 字段：它描述了一个段的开始位置：段基址。Base(24-31)：基地址的高 8 位，Base(16-23)：基地址的中间 8 位，Base(0-15)：基地址的低 16 位。（这里的段基址，不是相应的段在内存中的起始地址，而是程序编译链接以后，这个段在程序逻辑(虚拟)地址空间里的起始位置。）

相关寄存器：

GDTR：存放 GDT 在内存中的起始地址和大小 **LDTR：**分两种情况：

- (1) 当段选择符中的 TI=1 时，表示段描述符存放在 LDT 中，如何找到 LDT 呢，LDT 本身也被看作一个段，LDT 的起始地址存放在 GDT 中，此时 LDTR 存放的就是 LDT 在 GDT 中的索引。这也是本实验关注的情况。

(2) 当段选择符中的 $TI=0$ 时，表示段描述符存放在 GDT 中，通过 GDTR 找到 GDT，当 $TI=0$ 时，不涉及对 LDT 和 LDTR 的使用。

段选择符：如在 DS, SS 等寄存器内存储，取高 13 位作为在相应段表（如上例中的 DS 的高 13 位为对应段在 LDT）中的索引。

线性地址：段标识符用来标明一个段的编号，具体的，我们需要通过段的编号，查找段表，来获得这个段的起始地址，即段基址。如前所述，这里的段基址，不是相应的段在内存中的起始地址，而是程序编译链接以后，这个段在逻辑地址空间里的起始位置。进一步的，段基址+段内偏移量，就得到线性地址（即要访问的数据在整个程序逻辑(虚拟)地址空间中的位置）。

从逻辑地址到线性地址的转换过程，如图 4 所示（以 $TI=1$ 为例，此时从段选择符 DS 中分离出段索引号（高 13 位）和 TI 字段， $TI=1$ ，表明段描述符存放在 LDT 中）；

- (1) 从 GDTR 中获得 GDT 的地址，从 LDTR 中获得 LDT 在 GDT 中的偏移量，查找 GDT，从中获取 LDT 的起始地址；
- (2) 从 DS 中的高 13 位获取 DS 段在 LDT 中索引位置，查找 LDT，获取 DS 段的段描述符，从而获取 DS 段的基址；
- (3) 根据 DS 段的基址+段内偏移量，获取所需单元的线性地址。

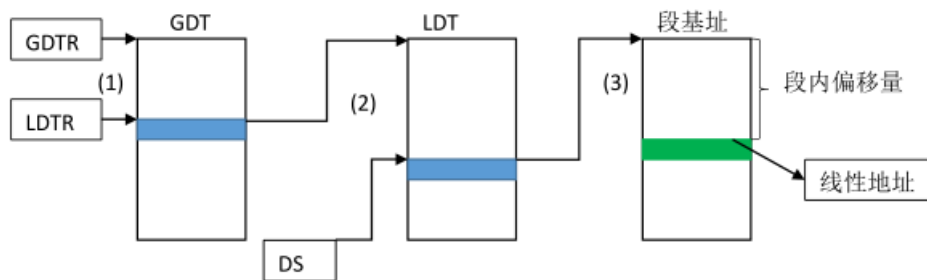


图 4 逻辑地址到线性地址的转换

2. 线性地址到物理地址的转换

物理地址：分段是面向用户，而分页则是面向系统，以提高内存的利用率，简言之，内存空间是按照分页来管理的。一个 32 位的机器，支持的内存空间是 4G，在页面大小为 4KB 的情况下，如果采用二级分页管理方式，线性地址结构如图 5 所示。

每一个 32 位的线性地址被划分为三部份，页目录索引(10 位)：页表索引(10 位)：偏移(12 位，因为页面大小为 4K)。最终，我们需要根据线性地址，来获得物理地址。

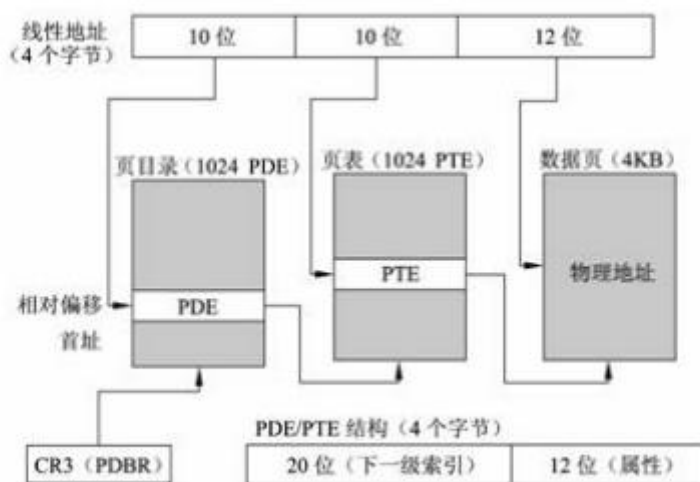


图 5 线性地址结构

将线性地址转换成物理地址的步骤：

- (1) 因为页目录表的地址放在 CPU 的 cr3 寄存器中，因此首先从 cr3 中取出进程的页目录表（第一级页表）的起始地址（操作系统负责在调度进程的时候，已经把这个地址装入对应寄存器）；
- (2) 根据线性地址前十位，在页目录表（第一级页表）中，找到对应的索引项，因为引入了二级管理模式，线性地址的前十位，是第一级页表中的索引值，根据该索引，查找页目录表中对应的项，该项即保存了一个第二级页表的起始地址。
- (3) 查找第二级页表，根据线性地址的中间十位，在该页表中找到数据页的起始地址；
- (4) 将页的起始地址与页内偏移量（即线性地址中最后 12 位）相加，得到最终我们想要的物理地址。

五、实验目的：

- (1) 掌握计算机的寻址过程
- (2) 掌握页式地址地址转换过程
- (3) 掌握计算机各种寄存器的用法

六、实验内容：

本实验运行一个设置了全局变量的循环程序，通过查看段寄存器，LDT 表，GDT 表等信息，经过一系列段、页地址转换，找到程序中该全局变量的物理地址。

七、实验环境：

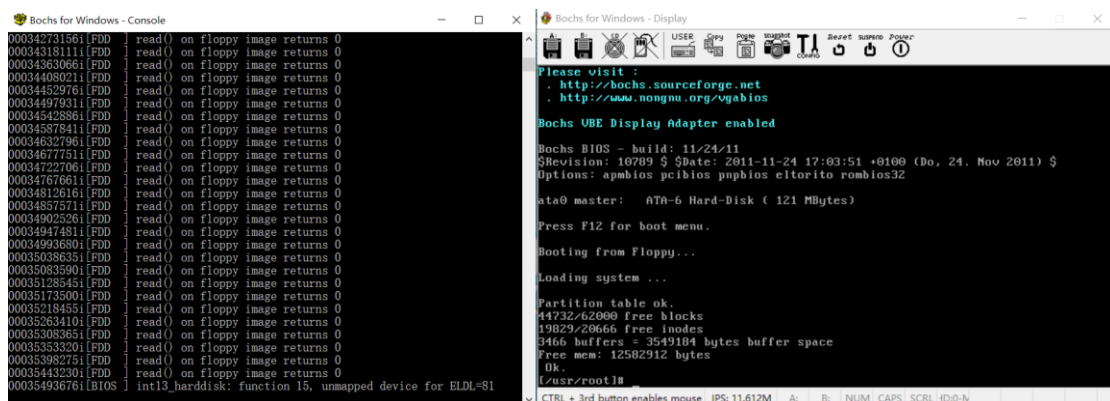
Linux 内核（0.11）+Bochs 虚拟机

八、实验步骤及结果分析：

1. 运行“bochs.exe”文件安装 bochs。
2. 拷贝 bootimage-0.11-hd、diska.img、hdc-0.11-new.img、mybochsrc-hd.bxrc 至安装目录。
3. 在安装目录中找到 bochsdbg.exe 程序，并运行。
4. 在弹出的界面中，点击“Load”加载配置文件“mybochsrc-hd.bxrc”。

随后，点击“Start”启动 Bochs 虚拟机。虚拟机启动后，出现两个窗口，一个为 Bochs 控制窗口，另一个为 Linux 操作系统运行窗口（主显示窗口）。

5. 在控制窗口输入“c”后回车，加载 Linux 操作系统。

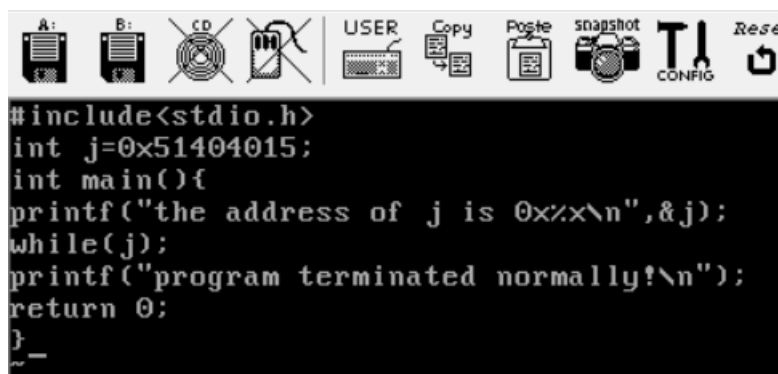


6. 在 Linux 操作系统中，使用 vi 工具编写 yzhtest.c 源文件。

先输入如下指令：

```
vi yuezihstest.c
```

然后按 Enter 键，进入代码编辑。输出如下代码：

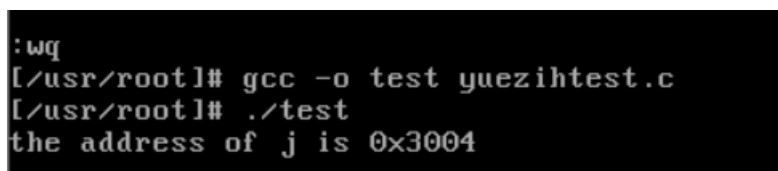


```
#include<stdio.h>
int j=0x51404015;
int main(){
printf("the address of j is 0x%x\n",&j);
while(j);
printf("program terminated normally!\n");
return 0;
}
~_
```

然后按 Esc 键， 输入如下指令， 保存文件并退出。

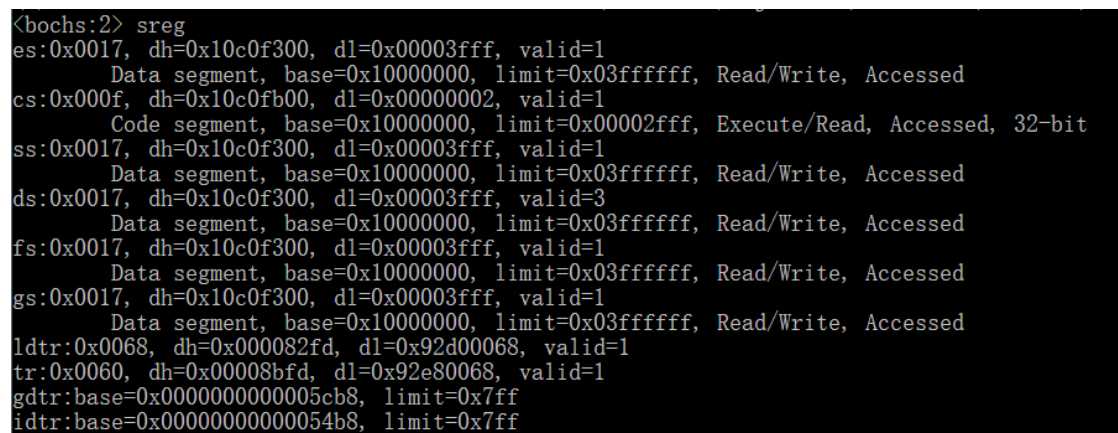
```
:wq
```

7. 随后输入 “gcc -o test yuezihtest.c”， 编译并生成 “test” 可执行文件。
8. 在 Linux 操作系统中， 输入 “./test” 指令运行该可执行文件， 运行结果如下图所示。



```
:wq
[/usr/root]# gcc -o test yuezihtest.c
[/usr/root]# ./test
the address of j is 0x3004
```

9. 控制窗口按 Ctrl+C， 中断当前运行， 进入调试状态。
10. 在控制窗口中输入 sreg 命令， 查看段的具体信息。 结果如下图所示。



```
<bochs:2> sreg
es:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
    Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
cs:0x000f, dh=0x10c0fb00, dl=0x00000002, valid=1
    Code segment, base=0x10000000, limit=0x00002fff, Execute/Read, Accessed, 32-bit
ss:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
    Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ds:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=3
    Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
fs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
    Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
gs:0x0017, dh=0x10c0f300, dl=0x00003fff, valid=1
    Data segment, base=0x10000000, limit=0x03ffffff, Read/Write, Accessed
ldtr:0x0068, dh=0x000082fd, dl=0x92d00068, valid=1
tr:0x0060, dh=0x00008bfd, dl=0x92e80068, valid=1
gdt:base=0x00000000000005cb8, limit=0x7ff
idt:base=0x000000000000054b8, limit=0x7ff
```

ds 段的段标识符信息为 0x0017， 转化为二进制为 0000 0000

0001 0111, 其中, 索引号后一位 (倒数第三位) 为 TI 位, 可得 TI=1, 表明段描述符存放在 LDT 中, 右移 3 位之后为 0x02, 因此局部描述符表 LDT 的偏移量为 2。

11. 查看 LDTR 寄存器信息, 其中存放了 LDT 在 GDT 中的位置。由上图可得 0x0068, 即 0000 0000 0110 1000, 右移 3 位之后为 0000 0000 0000 1101, 转化为 16 进制为 0x000D, 故在 GDT 中的索引为 13。

12. GDTR 中存放了 GDT 的起始地址, 输入“xp/2w 0x00005cb8+13*8”指令, 查看 GDT 中对应表项, 得到 LDT 段描述符, 如图所示。其中, LDT 基址的 24-31 位用 LDT 段描述符高 32 位的最高 8 位表示, 低 24 位用 LDT 段描述符的低 32 位的 24-31 位和高 32 位的 0-7 位表示。经过拼接可得 LDT 的基址为 0x00fd92d0。

```
<bochs:3> xp/2w 0x00005cb8+13*8
[bochs]:
0x00000000000005d20 <bogus+      0>:   0x92d00068   0x000082fd
<bochs:4> _
```

13. 输入 “xp /2w 0x00fd92d0+2*8”, 查看 LDT 中第 2 项段描述符, 结果如图所示。可以发现, 得到的 ds 段的段描述符信息与与 ds 寄存器 (dl、dh) 中的结果完全相同。

```
<bochs:4> xp /2w 0x00fd92d0+2*8
[bochs]:
0x00000000000fd92e0 <bogus+      0>:   0x00003fff   0x10c0f300
<bochs:5> _
```

14. 根据上述结果, 可以得到 ds 段的基地址为 0x10000000。

15. 根据程序运行结果 “the address of j is 0x3004”, 可以计算线性地址为 0x10000000+0x00003004=0x10003004。转化为二进制为 0001

0000 0000 0000 0011 0000 0000 0100, 其中, 高 10 位为页目录索引, 为 00 0100 0000, 即 0x40; 低 12 位为偏移, 为 0000 0000 0100, 即 0x40; 中间 10 位为页表索引, 为 00 0000 0011, 即 0x03。因此, 页目录索引为 0x40, 页表索引为 0x03, 偏移量为 0x04。

16. 输入 “creg” 指令, 结果如图所示。寄存器 CR3 的值为 0, 即页目录表的起始地址为 0。

```
<bochs:5> creg
CR0=0x8000001b: PG cd nw ac wp ne ET TS em MP PE
CR2=page fault laddr=0x0000000010002fa8
CR3=0x0000000000000000
  PCD=page-level cache disable=0
  PWT=page-level write-through=0
CR4=0x00000000: smep osxsave pcid fsgsbase smx vmx osxmmexcpt osfxsr pce pge mce pae pse de tsd pvi vme
EFER=0x00000000: ffxsr nxe lma lme sce
<bochs:6> _
```

17. 由于页目录表的起始地址为 0, 故 PDE 的地址为 $0+64*4$, 因此, 输入 “xp /w 64*4” 指令, 结果如图所示。可以得到, PDE 的值为 0x00fa6027, 低 12 位置零作为下一级的索引, 即 0x00fa6000。则 PTE 的地址为 $0x00fa6000+3*4$ 。

```
xp /w 64*4
[bochs]:
0x00000000000000100 <bogus+      0>:    0x00fa6027    0x00000000
<bochs:7> _
```

18. 输入 “xp /w 0x00fa6000+3*4” 指令, 结果如图所示。得到 PTE 的值为 0x00fa3067, 低 12 位置零作为下一级的索引, 即 0x00fa3000。因此, 物理地址为 $0x00fa3000+4$ 。

```
xp /w 0x00fa6000+3*4
[bochs]:
0x0000000000fa600c <bogus+      0>:    0x00fa3067
```

19. 输入 “xp /w 0x00fa3000+4” 指令, 结果如图所示。得到的结果与 yuezihetest.c 中定义的 j 的值相同, 说明正确找到了 j 的物理地址。

```
<bochs:8> xp /w 0x00fa3000+4
[bochs]:
0x0000000000fa3004 <bogus+      0>:    0x51404015
<bochs:9> _
```

20. 接下来对 `j` 的值进行修改，输入 “`setpmem 0x00fa3004 4 0`” 指令，将该物理地址开始的 4 个字节的值全部清零。清零之后，可以再次使用 “`xp /w 0x00fa3000+4`” 指令查看该物理地址存放的当前值，发现已经成功清零。

```
<bochs:8> xp /w 0x00fa3000+4
[bochs]:
0x0000000000fa3004 <bogus+      0>:    0x51404015
<bochs:9> setpmem 0x00fa3004 4 0
<bochs:10> xp /w 0x00fa3000+4
[bochs]:
0x0000000000fa3004 <bogus+      0>:    0x00000000
<bochs:11> _
```

21. 输入命令 “`c`”，继续运行。回到 `Display` 窗口，显示程序已正常结束。表明本次通过物理地址对 `j` 值的修改有效，成功完成实验。

```
:wq
[/usr/root]# gcc -o test yuezihetest.c
[/usr/root]# ./test
the address of j is 0x3004
program terminated normally!
x[/usr/root]#
```

九、实验结论：

本次实验通过 Linux 内核和 Bochs 虚拟机，在一个 .c 文件中定义了一个变量，通过查看段寄存器，LDT 表、GDT 表等信息，经过一系列段、页地址转换，先后完成了逻辑地址到线性地址的转换和线性地址到物理地址的转换，找到程序中该全局变量的物理地址，并通过物理地址的访问对其进行修改，使得程序能按照预期顺利退出。

十、总结及心得体会：

本实验通过 Linux 内核和 Bochs 虚拟机，让我第一次在实践中接触到计算机的寻址过程和页式地址的地址转换过程，也对计算机中的部分寄存器有了初步的了解。在经过课堂理论知识的学习之后，该实

验不但帮助我巩固了寻址过程、地址转换等操作系统基本知识，也让我通过实际操作，加深了对计算机工作过程的理解和认识。在实验的过程中，由于对虚拟机操作不够熟悉，中途遇到过很多问题，走了很多弯路，这也是对我解决问题的能力锻炼和耐心的培养。在此也非常感谢在我遇到问题时帮助我排忧解难的同学们。经过本次实验，我也对 Linux 内核和 Bochs 虚拟机有了一个初步的了解，第一次使用虚拟机，用 vi 编辑 C 代码，在一定程度上开拓了视野。总之，本次实验让我收获很大，不论是对理论知识有了更扎实的掌握，也对实际过程和实践操作有了一定的了解。

十二、对本实验过程及方法、手段的改进建议：

本实验由于涉及的实验技能较为陌生，因此提供了非常详细的指导，整个实验完成起来比较简单。我觉得在引导学生完成本实验内容、对相关理论与实践方法形成基本掌握之后，可以增设一些挑战性内容，帮助学生更深入地了解计算机寻址过程、地址转换过程等理论，通过自己的思考在解决问题的过程中强化对于相关知识的理解和掌握。

报告评分：

指导教师签字：