一、 目的

1. 巩固操作系统的进程调度机制和策略
2. 熟悉MINIX系统调用和MINIX调度器的实现

二、 内容与设计思想

由于minix微内核的特征，需要依次修改应用层，服务层，内核层来实现系统调用。应用层用户调用 chrt 系统调用，将 deadline传入到服务层。服务层注册 chrt 服务，将deadline 传入到内核层。最后由内核层修改内核信息来实现chrt系统调用。

在内核中修改proc.c和proc.h中相关的调度代码，实现最早deadline的用户进程相对于其它用户进程具有更高的优先级，从而被优先调度运行。通过在入队时将当前deadline大于0的进程添加到合适的优先级队列，实现实时调度。在该队列内部按剩余时间最少优先调度，将剩余时间最小的进程移到队列首部。

三、 使用环境

Xcode

VMware fusion 虚拟机

本地终端terminal

Sublime Text

四、 实验过程

增加系统调用chrt

1.应用层

chrt函数 通知进程管理器PM处理CHRT函数，并把deadline放到message

记录当前时间，利用deadline=nowtime+deadline来计算deadline

alarm定义为unsigned int alarm(unsigned int);

故用alarm（（unsigned int) deadline）；实现超时强制终止

alarm()函数用来设置一个定时器，当时间超时时，会产生SIGALRM信号，该信号默认是终止该进程；

#define PM_PROC_NR ((endpoint_t) 0) /* process manager */

```
int chrt(long deadline){
    //struct timespec time={0,0};
    struct  timeval    tv;
    struct  timezone   tz;
message m;
memset(&m,0,sizeof(m));
    //设置alarm
    alarm((unsigned int)deadline);
    //将当前时间记录下来 算deadline
    if(deadline>0){
        gettimeofday(&tv,&tz);
        //clock_gettime(CLOCK_REALTIME, &time);
        //deadline=nowtime+deadline
        deadline = tv.tv_sec + deadline;
    }
    //存deadline
m.m2_l1=deadline;
return(_syscall(PM_PROC_NR,PM_CHRT,&m));
}
```

memset(&m, 0, sizeof(m));将已开辟内存空间 &m 的首 sizeof(m) 个字节的值设为值 0。

查找minix/include/minix/ipc.h的消息结构

```
typedef struct {
  int64_t m2ll1;
  int m2i1, m2i2, m2i3;
  long m2l1, m2l2;
  char *m2p1;
  sigset_t sigset;
  short m2s1;
  uint8_t padding[6];
} mess_2;
```

```
#define m2_l1  m_m2.m2l1
#define m2_l2  m_m2.m2l2
```

2.服务层

在/usr/src/minix/servers/pm/proto.h中添加chrt函数定义。

在/usr/src/minix/servers/pm/chrt.c中添加chrt函数实现，调用sys_chrt()

类似其他参数中用到进程号的进程的定义endpoint_t proc_ep

可知进程号是endpoint_t类型的

who_p标示发起调用者的进程号

```
#include"pm.h"
#include<minix/syslib.h>
#include<minix/callnr.h>
#include<sys/wait.h>
#include<minix/com.h>
#include <minix/vm.h>
#include "mproc.h"
#include <sys/ptrace.h>
#include <sys/resource.h>
#include <signal.h>
#include <stdio.h>
#include<minix/sched.h>
#include <assert.h>
```

```
int do_chrt()
{
  sys_chrt(who_p, m_in.m2_l1);
  return OK;
}
```

在/usr/src/minix/include/minix/callnr.h中定义PM_CHRT编号。

```
#define PM_GETSYSINFO   (PM_BASE + 47)
#define PM_CHRT   (PM_BASE + 48)
#define NR_PM_CALLS   49  /* highest number from base plus
one */
```

在/usr/src/minix/servers/pm/Makefile中添加chrt.c条目。

/usr/src/minix/servers/pm/table.c 中调用映射表

CALL(PM_GETSYSINFO) = do_getsysinfo末尾加,

添加CALL(PM_CHRT) = do_chrt

/usr/src/minix/include/minix/syslib.h 中添加sys_ chrt () 定义

在/usr/src/minix/lib/libsys/sys_chrt.c 中添加sys_chrt () 实现。

```
#include "syslib.h"
int sys_chrt( proc_ep, deadline)
int  proc_ep;
long deadline;
{
int r;
 message m;
 m.m2_i1 =  proc_ep;
 m.m2_l1 = deadline;
 r = _kernel_call(SYS_CHRT, &m);
 return r;
}
```

在/usr/src/minix/lib/libsys 中的Makefile中添加sys_chrt.c条目。

3.内核层

在/usr/src/minix/kernel/system.h中添加do_chrt函数定义。添加

```
int do_chrt(struct proc * caller, message *m_ptr);
#if ! USE_CHRT
#define do_chrt NULL
#endif
```

在/usr/src/minix/kernel/system/do_ chrt.c中添加do_chrt函数实现。

```
int do_chrt(struct proc *caller, message *m_ptr)
{
  struct proc *rp;
  long exp_time;
  exp_time = m_ptr->m2_l1;
//通过 proc_addr 定位内核中进程地址
  rp = proc_addr(m_ptr->m2_i1);
//将消息结构体中的deadline 赋值给该进程的 p_deadline
  rp->p_deadline = exp_time;
  return (OK);
}
```

用消息结构体中的进程号，通过 proc_addr 定位内核中进程地址，然后将消息结构体中的deadline 赋值给该进程的 p_deadline

```
#define proc_addr(n)      (&(proc[NR_TASKS + (n)]))
```

在 proc.h 头文件中添加struct proc中添加**long** long p_deadline;

在/usr/src/minix/kernel/system/ 中Makefile.inc文件添加do_chrt.c条目。do_chrt.c \

在/usr/src/minix/include/minix/com.h中定义SYS_CHRT编号。

```
#  define SYS_CHRT (KERNEL_CALL + 58)
/* Total */
#define NR_SYS_CALLS  59  /* number of kernel calls */
```

在/usr/src/minix/kernel/system.c 中添加SYS_CHRT编号到do_chrt的

映射。map(SYS_CHRT, do_chrt);

在/usr/src/minix/commands/service/parse.c的system_tab中添加名称编

号对。{ "CHRT",　SYS_CHRT },

进程调度

struct proc 维护每个进程的信息，用于调度决策。添加deadline成员。

在proc.c中

enqueue和enqueue_head中添加代码将当前剩余时间大于0的进程添加到合适优先级的队列，如果优先级不合适，无法达到实时进程优先的效果。需要通过测试，找到合适的队列。

```
if (rp->p_deadline > 0)
  {
   rp->p_priority = 5;
  }
```

在该队列内部将时间片轮转调度改成剩余时间最少优先调度，即将剩

余时间最小的进程移到队列首部。

get_cpulocal_var(run_q_head)包括Process scheduling information and the kernel reentry count

在config.h中，看到

/* Scheduling priorities. Values must start at zero (highest priority) and increment./
#define NR_SCHED_QUEUES　16 /* MUST equal minimum priority + 1 */

Pick_proc中遍历优先级队列选择剩余时间最少的进程。

通过链表实现。如果当前进程结束或者temp进程剩余时间比当前进程更少，并且temp进程可以运行，替换当前进程来保证rp是剩余时间最少的进程。

此处我们可以不需要读取当前时间，由于测试用例，剩余时间短的,dealine也小，通过deadline判断即可。普通进程deadline没有赋值。

添加代码如下

```
  //遍历优先级队列
     //将剩余时间最小的进程移到队列首部
          rp=rdy_head[q];
          //temp记录下一个就绪的进程
          temp=rp->p_nextready;
      //保证rp是剩余时间最少的进程
       if(q==5){
          //遍历链表
          //选择剩余时间最少的进程
          while(temp!=NULL){
               if (temp->p_deadline > 0)
               {
               //如果当前进程结束或者temp进程剩余时间比当前进程更少
                    if (rp->p_deadline == 0|| (temp-
>p_deadline < rp->p_deadline))
                    {
                         //并且temp进程可以运行
                         if (proc_is_runnable(temp))
                              //替换当前进程
                              rp = temp;
                    }
               }
               temp = temp->p_nextready;
          }
      }
```

至此完成了EDF的实现。

在测试代码中

子进程是顺序创建，最后一个创建的是进程3，进程3先执行一次。此时的进程3时普通进程，进程1和2是实时进程。由于采取了deadline，而此时进程2的deadline最近，进程1其次。所以就是2，1抢先执行，抢占进程3，进程按2，1，3顺序执行。5s后进程1的deadline改变，此时进程1变为剩余时间最短的进程故抢先执行，按123顺序执行，后面进程3deadline调整的结果同理。

测试结果

```
proc1 set success
proc2 set success
proc3 set success
# prc3 heart beat 1
prc2 heart beat 1
prc1 heart beat 1
prc3 heart beat 2
prc2 heart beat 2
prc1 heart beat 2
prc3 heart beat 3
prc2 heart beat 3
prc1 heart beat 3
prc3 heart beat 4
prc2 heart beat 4
prc1 heart beat 4
prc3 heart beat 5
Change proc1 deadline to 5s
prc1 heart beat 5
prc2 heart beat 5
prc3 heart beat 6
prc1 heart beat 6
prc2 heart beat 6
prc3 heart beat 7
prc1 heart beat 7
prc2 heart beat 7
prc3 heart beat 8
prc1 heart beat 8
prc2 heart beat 8
prc3 heart beat 9
Change proc3 deadline to 3s
prc3 heart beat 10
prc2 heart beat 9
prc3 heart beat 11
prc2 heart beat 10
prc2 heart beat 11
prc2 heart beat 12
prc2 heart beat 13
```

五、 总结

本次实验中MINIX的不同服务模块和内核都是运行在不同进程中，熟悉了使

用基于消息的进程间系统调用/内核调用。进一步熟悉了minix调度算法。

熟悉了利用sublimeText进行跳转，明晰调用关系和全局搜索的应用。

熟悉了使用git diff 检查代码修改。修改涉及文件较多，避免引入无意的错误。

熟悉了利用suberduck连接虚拟机，拉取需修改的文件，修改后上

传到虚拟机。

熟悉了虚拟机的内核编译过程，ssh本地运行和串口调试。