华东师范大学数据学院上机实践报告

课程名称： 操作系统　　**年级：**大二　　**上机实践成绩：**

指导教师：　　**姓名：**沈小奇

上机实践名称：　　**学号：** 10185501401　　**上机实践日期：**

上机实践编号：

一、 **目的**

Minix 添加实时进程功能

二、 **内容与设计思想**

Minix 调度的设计

三、 **使用环境**

VMware，Moba

四、 **实验过程**

下载源码

```
cd /usr
git clone git://git.minix3.org/minix src
git branch –a  # 查看代码版本
git checkout R3.3.0   # 将代码版本切换为 3.3.0
```

编译

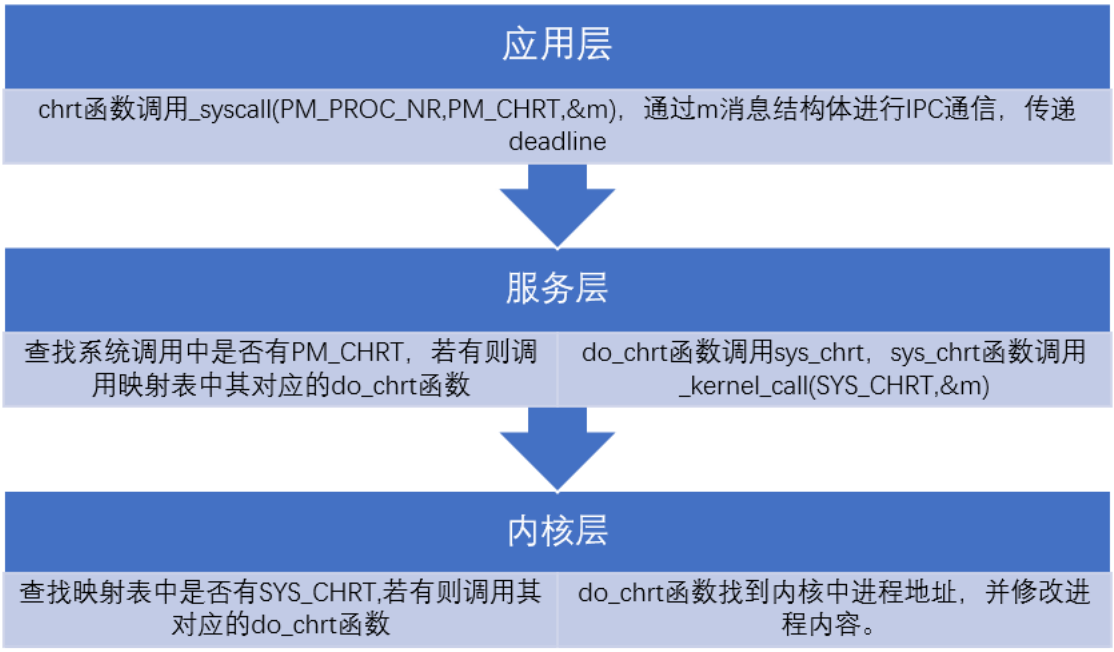
```
cd /usr/src
make build   #首次编译以及修改了头文件
make build MKUPDATE  #增量式编译
```

总体思想：chrt 调用涉及到 minix 的三层：应用层，服务层，内核层，层层递进。

要创造一个 chrt 函数实则要在三层里修改代码。

助教将三层的关系已经展示的很清楚了：

chrt 系统调用过程：



一． 应用层

添加 chrt 函数的定义和实现

/usr/src/include/unistd.h

```
int chrt(long);
```

添加 chrt.c，使用_syscall(编号)向下层（服务层）传递 /usr/src/minix/lib/libc/sys/chrt.c

```
#include <sys/cdefs.h>
#include "namespace.h"
#include <lib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <time.h>

int chrt(long deadline)
{
   struct timespec time;
   message m;
   memset(&m, 0, sizeof(m));

   alarm((unsigned int)deadline);/* set the alarm */

   if (deadline > 0)
   {
      clock_gettime(CLOCK_REALTIME, &time);
      deadline = time.tv_sec + deadline;
   }
   m.m2_l1 = deadline; /* store the deadline */
```

```
        return (_syscall(PM_PROC_NR, PM_CHRT, &m));
    }

/* #define PM_PROC_NR      ((endpoint_t) 0)
```

在/usr/src/minix/lib/libc/sys 中 Makefile.inc 文件添加 chrt.c 条目

二. 服务层

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

```
 int do_chrt(void);
```

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

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

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

```
      /末尾添加
CALL(PM_CHRT)           = do_chrt           /*chrt(2)*/
```

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

```
 int sys_chrt(endpoint_t proc_ep, long deadline);
```

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)
endpoint_t proc_ep;
long deadline;
{
   message m;
   int r;
   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

三. 内核层

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

```
 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

```
#include "kernel/system.h"
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <lib.h>
#include <minix/endpoint.h>

#if USE_CHRT

/*===========================================================*
 *                                do_chrt                                *
*===========================================================*/
int do_chrt(struct proc *caller, message *m_ptr)
{

   struct proc *rp;
   long exp_time;

   exp_time = m_ptr->m2_l1;
```

```
        rp = proc_addr(m_ptr->m2_i1);

        rp->p_deadline = exp_time;

        return (OK);
    }

#endif /* USE_CHRT */
```

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

```
    #   define SYS_CHRT      (KERNEL_CALL + 58)        /* sys_chrt() */

    /* 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 }
```

四.　进程调度

MINIX3 中的进程调度：进程调度模块位于/usr/src/minix/kernel/下的 proc.h 和 proc.c，修 改影响进程调度顺序的部分。

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

switch_to_user() 选择进程进行切换。

enqueue_head()按优先级将进程加入列队首。实验中需要将实时进程的优 先级设置成合适的优先级。
enqueue() 按优先级将进程加入列队尾。
同上。

pick_proc()从队列中返回一个可调度的进程。遍历设置的优先级队列，返回剩余时间最小并可运行的进程。

在 proc.h 中增加 p_deadline 项
Long p_deadline;

在 proc.c 中修改 enqueue、enqueue_head、pick_proc 函数

将 deadline>0 的函数设置较高的优先级 4

在 enqueue 和 enqu_head 中增加：

```
    if (rp->p_deadline > 0)
      {
 rp->p_priority = 4;     }
```

在 pick_proc 函数中遍历优先级为 4 的进程队列，比较 deadline 大小，越近的 deadline 就把他放在要调度的队列前面。

```
    if (q == 4)//遍历优先级为 4 的进程
      {
            rp = rdy_head[q];//从头开始
            tmp = rp->p_nextready;//头的下一个
            while (tmp != NULL)
            {
                  if (tmp->p_deadline > 0)
                  {
                        /* if rp is end or tmp is has less exp_time*/
                        if (rp->p_deadline == 0 || (tmp->p_deadline < rp->p_deadline))
                        {
                              if (proc_is_runnable(tmp))
                                    rp = tmp; /* replace the current process */
                        }
                  }
                  tmp = tmp->p_nextready;
            }
      }
```

最后的结果：

```
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

添加内核调用和系统调用必需的一些机制，如注册系统调用、内核调用编号，消息传递，建立消息与函数库的关联（map()）等。

修改 proc.c 中进程调度有关算法。 修改 enqueue,enqueue_head,pick_proc 函数，每次入队时比较时钟到期时间，选择插入优先级队列合适的位置，使优先级队列排序为到期时间由近到远。

实验难度不小，涉及很多底层知识，需要在懂得整个架构的基础上入手，在改写的过程中可以阅读其他函数（如 fork）的框架，进行模仿。