华东师范大学数据科学与工程学院实验报告

课程名称：计算机网络与编程	**年级**：2021	**上机实践成绩**：
指导教师：翁楚良	**姓名**：唐小卉	**学号**：10215501437
上机实践名称：		**上机实践日期**：2023.4.13
上机实践编号：	**组号**：	**上机实践时间**：2023.4.13

一、实验目的

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、实验任务

在 MINIX3 中实现 Earliest-Deadline-First 近似实时调度功能：

1. 提供设置进程执行期限的系统调度 chrt (long deadline)，用于将调用该系统调用的进程设为实时进程，其执行的期限为：从调用处开始 deadline 秒。
2. 在内核进程表中需要增加一个条目，用于表示进程的实时属性；修改相关代码，新增一个系统调用 chrt，用于设置其进程表中的实时属性。
3. 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。
4. 在用户程序中，可以在不同位置调用多次 chrt 系统调用，在未到 deadline 之前，调用 chrt 将会改变该程序的 deadline。
5. 未调用 chrt 的程序将以普通的用户进程(非实时进程)在系统中运行。

三、使用环境

物理机：Windows10
虚拟机：Minix3
虚拟机软件：Vmware
代码编辑：VScode
物理机与虚拟机文件传输：FileZilla

四、实验过程

在 usr/src/minix/include/minix/ipc.h 文件中查找到 message 结构体

```
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;
```

应用层：

1.在/usr/src/include/unistd.h 中添加 chrt 函数定义。

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

2.在/usr/src/minix/lib/libc/sys/chrt.c 中添加 chrt 函数实现。可用 alarm 函数实现超时强制终止。参照该文件夹下 fork.c 文件，在实现中通过 _syscall (调用号)向系统服务传递。

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

int chrt(long deadline){
    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);
        deadline = tv.tv_sec + deadline;
    }
    //存deadline
m.m2_l1=deadline;
return(_syscall(PM_PROC_NR,PM_CHRT,&m));
}
```

Message 是和操作系统通信的结构体。最后返回的时候调用_syscall 函数，请求实时调度机制。前两个参数分别指定了一个进程号和调度机制，&m 指向 message，包含了请求的详细信息。

3.在/usr/src/minix/lib/libc/sys 中 Makefile.inc 文件添加 chrt.c 条目（添加 C 文件后，需在同目录下的 Makefile/Makefile.inc 中添加条目）。

```
sigprocmask.c socket.c socketpair.c stat.c statvfs.
symlink.c \
sync.c syscall.c sysuname.c truncate.c umask.c unli
utimensat.c utimes.c futimes.c lutimes.c futimens.c
_exit.c _ucontext.c environ.c __getcwd.c vfork.c si
getrusage.c setrlimit.c setpgid.c chrt.c
```

服务层：

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

```
int do_chrt(void)#
```

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

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

这个函数设置了实时调度的相关参数，第一个参数指定了调度的进程，第二个参数制定了进程设置的截止时间。

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

callhr.h

```
#define PM_CHRT (PM_BASE + 48)
#define NR_PM_CALLS 49
```

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

```
PROG= pm
SRCS= main.c forkexit.c exec.c time.c alarm.c \
  signal.c utility.c table.c trace.c getset.c misc.c \
  profile.c mcontext.c schedule.c chrt.c

DPADD+= ${LIBSYS} ${LIBTIMERS}
LDADD+= -lsys -ltimers
```

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

```
#define CALL(n) [((n) - PM_BASE)]

int (* const call_vec[NR_PM_CALLS])(void) = {
  CALL(PM_CHRT) = do_chrt  /*参照 fork、exit 定义进行添加*/
  CALL(PM_EXIT)    = do_exit,     /* _exit(2) */
```

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

```
/*新加入*/
int sys_chrt(endpoint_t proc_ep,long deadline);
```

10.在/usr/src/minix/lib/libsys/sys_chrt.c 中添加 sys_chrt () 实现。可参照该文件夹下的 sys_fork 文件，在实现中通过_kernel_call (调用号)向内核传递。例如：

int sys_fork(parent, child, child_endpoint, flags, msgaddr)
{
_kernel_call(SYS_FORK, &m);
}

* sys_chrt.c　sys_fork.c

```
#include"syslib.h"
int sys_chrt(proc_ep,deadline)
endpoint_t proc_ep;
long deadline;
{
  message m;
  int r;
  //将进程号和 deadline 放入消息结构体
  m.m2_i1=proc_ep;
  m.m2_l1=deadline;
  //通过_kernel_call 传递到内核层
  r=_kernel_call(SYS_CHRT,&m);
  return r;
}
```

sys_chrt 函数将进程号和 deadline 放入消息结构体，通过_kernel_call 传递到内核层。

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

```
SRCS+=  \
  sys_chrt.c \
  alloc_util.c \
  assert.c \
```

内核层：

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

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

13.在/usr/src/minix/kernel/system/do_ chrt.c 中添加 do_chrt 函数实现。参考该文件下的 do_fork 文件，修改调用者进程信息。例如：

```
pid_t fork(void)
{
return(_syscall(PM_PROC_NR, PM_FORK, &m));
}
```

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

```
* do_chrt.c    do_fork.c
#include "kernel/system.h"
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <lib.h>
#include <minix/endpoint.h>
#include <string.h>

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 exp_time 赋值给该进程的 p_deadline
  rp->p_deadline = exp_time;
  return (OK);
}
```

proc_addr 函数可确定进程在内核中地址。M_ptr->m2_l1 存储 deadline，m_pte->m2_i1 存储的是 m 中包含的进程号。

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

```
.PATH:  ${.CURDIR}/system
SRCS+=  \
        do_fork.c \
        do_exec.c \
        do_chrt.c \
        do_clear.c \
        do_exit.c \
        do_trace.c \
        do_runctl.c \
```

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

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

16.在/usr/src/minix/kernel/system.c 中添加 SYS_CHRT 编号到 do_chrt 的映射

```
191
192   /* Process management. */
193   /*新加入*/
194   map(SYS_CHRT,do_chrt);//参考 do_fork、do_exec
195
```

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

```
823 struct
824 {
825   char *label;
826   int call_nr;
827 } system_tab[]=
828 {
829   { "PRIVCTL",     SYS_PRIVCTL },
830   { "TRACE",     SYS_TRACE },
831   { "KILL",    SYS_KILL },
832   { "UMAP",    SYS_UMAP },
833   { "VIRCOPY",     SYS_VIRCOPY },
834   { "PHYSCOPY",    SYS_PHYSCOPY },
835   { "UMAP_REMOTE",   SYS_UMAP_REMOTE },
836   { "VUMAP",     SYS_VUMAP },
837   { "IRQCTL",    SYS_IRQCTL },
838   { "INT86",     SYS_INT86 },
839   { "DEVIO",     SYS_DEVIO },
840   { "SDEVIO",    SYS_SDEVIO },
841   { "VDEVIO",    SYS_VDEVIO },
842   { "ABORT",     SYS_ABORT },
843   { "IOPENABLE",     SYS_IOPENABLE },
844   { "READBIOS",    SYS_READBIOS },
845   { "STIME",     SYS_STIME },
846   { "VMCTL",     SYS_VMCTL },
847   { "MEMSET",    SYS_MEMSET },
848   { "PADCONF",     SYS_PADCONF },
849   { "CHRT",     SYS_CHRT },
850   { NULL,    0 }
851 };
852
```

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

proc.h 修改

```
21 /*新加入*/
22 long long p_deadline;//设置 deadline
23
```

proc.c 修改

```
1538 //将当前dealine大于0的进程添加到最高优先级的队列
1539     if (rp→p_deadline >0)
1540     {
1541           rp→p_priority = 5;
1542     }
```

```
1742    rp=rdy_head[q];
1743    temp=rp→p_nextready;
1744    if(q==5){
1745      while(temp≠NULL){
1746        if (temp→p_deadline > 0)
1747        {
1748          if (rp→p_deadline == 0 || (temp→p_deadline < rp→p_deadlin
1749          {
1750            if (proc_is_runnable(temp))
1751            rp = temp;
1752          }
1753        }
1754          temp = temp→p_nextready;
1755           }
1756      }
```

此处的 rp 指的是正在进行的进程。遍历队列中所有的进程，如果进程的截止时间大于 0，并且该进程的截止时间比当前正在运行的进程更截止时间更短，则将其设置为下一个要运行的进程。如果有多个满足条件的进程，则选择其中优先级最高的一个。

实验结果：

```
# cd /usr/src
# ./test_code.o
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
```

```
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**