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：**2019 级　　**上机实践成绩：**

指导教师：翁楚良　　**姓名：**周辛娜

上机实践名称： 进程管理　　**学号：**10195501442　　**上机实践日期：**4.27

上机实践编号：2

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

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。例如：
chrt 的定义：
int chrt(long deadline);
/*deadline 是最后期限值(秒)，返回值 1 表示成功，返回值 0 表示该调用出错 */
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

VMware Workstation Pro
MobaXterm_Personal_20.6
Source Insight 4.0

四、 实验过程

1. 应用层：

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

```
int chrt(long deadline); /*chrt函数定义*/
```

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。chrt 函数调用_syscall(PM_PROC_NR, PM_CHRT, &m)，通过消息结构体 m 进行 IPC 通信，传递 deadline
具体函数实现：

```
int chrt(long deadline)
{
  struct timespec time;
  message m;
  memset(&m, 0, sizeof(m));
  alarm((unsigned int)deadline);
  if(deadline<=0)
  return 0;
  if(deadline>0){
    clock_gettime(CLOCK_REALTIME, &time);  //获
    deadline = time.tv_sec + deadline;
  }
  m.m2_l1 = deadline;

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

可以通过 int clock_gettime(clockid_t clk_id, struct timespec *tp)来获取系统时间，其中用到 clk_id 参数 CLOCK_REALTIME，#define CLOCK_REALTIME 0 表示系统实时时间。

chrt(dealine)最多可以运行的时间是 deadline 秒，设置 alarm，强制终止，表示当超过 deadine 以后进程要被强制终止。但是传递给内核的参数必须要看剩余时间，所以传递给内核的参数应该是进程的终止时间：当前时间+终止时间（chrt 传进的参数）（初始 deadline)
所以 m.m2_l1=deadline，这里 Ipc.h 中关于 long 的定义（传递 3 个 int，2 个 long，1 个指针时使用 m_m2）
最后_syscall(PM_PROC_NR, PM_CHRT, &m) 进程号，调用的服务，消息结构体（将 deadline 放入）

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

```
sync.c syscall.c sysuname.c truncate.c umask.c unlink.c write.c \
utimensat.c utimes.c futimes.c lutimes.c futimens.c \
_exit.c _ucontext.c environ.c __getcwd.c vfork.c sizeup.c init.c \
getrusage.c setrlimit.c setpgid.c chrt.c
```

2. 服务层：
查找系统调用中是否有 PM_CHRT，若有则调用映射表中其对应的 do_chrt 函数，do_chrt 函数调用 sys_chrt，sys_chrt 函数调用_kernel_call(SYS_CHRT, &m)
1) 在/usr/src/minix/servers/pm/proto.h 中添加 chrt 函数定义。

```
/* chrt.c */
int do_chrt(void);
```

2) 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);
}
```

do_chrt 函数调用 sys_chrt(who_p,消息结构体)。

3) 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 */
```

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

```
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
```

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

```
  CALL(PM_CHRT) = do_chrt
```

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

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

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

```
int sys_chrt(who, deadline)
endpoint_t who;
long deadline;
{
  message m;
  m.m2_i1 = who;
  m.m2_l1 = deadline;
  return _kernel_call(SYS_CHRT, &m);
}
```

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

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

```
sys_clear.c \
sys_chrt.c\
sys_cprof.c \
sys_diagctl.c \
sys_endsig.c \
```

3. 内核层：

查找映射表中是否有 SYS_CHRT,若有则调用其对应的 do_chrt 函数，do_chrt 函数找到内核中进程地址，并修改进程内容

1) 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
```

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

```c
int do_chrt(struct proc *caller, message *m_ptr)
{
  struct proc *rp;
  rp = proc_addr(m_ptr->m2_i1);//get the process
  rp->p_deadline =m_ptr->m2_l1;//set the deadline
  return OK;
}
```

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

```
do_statectl.c \
do_chrt.c
```

4) 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 */
```

5) 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);
```

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

```
  { "CHRT", SYS_CHRT},
  { NULL,   0 }
};
```

4.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:

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

```
long  p_deadline; /* deadline of process */
```

2) 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，在这里可以设置为 5。enqueue() 按优先级将进程加入列队尾，同样将优先级设置为 5

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

3) 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。

```
    if(q==5)
    {
      rp=rdy_head[q];
      tmp=rp->p_nextready;
      while(tmp!=NULL)
      {
          if (tmp->p_deadline > 0) //去优先级队列中找
          {
              if ((rp->p_deadline == 0) && proc_is_runnable(tmp))
                  //如果rp不是实时进程
                  rp = tmp;
              else if ((rp->p_deadline > tmp->p_deadline) && proc_is_runnable(tmp))
                  //如果rp是实时进程，其对应的deadline大与tmp对应的deadline
                  rp = tmp;
          }
          tmp = tmp->p_nextready;
      }
  }
```

rp 的初始值是 head，tmp 是 next，将正在运行的进程切换到 tmp 的情况是：rp 不是实时进程，而 tmp 是实时进程；rp,tmp 是实时进程，rp 对应的终止时间大于 tmp 对应的终止时间

5.实验结果

```
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

本次实验是在 MINIX3 中实现 Earliest-Deadline-First 近似实时调度功能，具体来说是增加一个系统调用以及修改在内核层关于进程调度的模块。

1）对于系统调用：

一个普通的进程调用这个系统调用之后就变成了一个实时进程了，即 chrt(long deadline)，这个进程最多可以运行 deadline 时间，如果它在 deadline 之前正常结束没问题，如果 deadline 时还没有结束，就必须强制终止它，所以这里就要用到 alarm 函数，所以在应用层来说就要用这个函数把进程终止。从应用层用_syscall 将 deadline 信息传递到服务层，在服务层用_kernel_call 将进程位置和 deadline 信息传递到内核层，在内核层对进程结构体增加 deadline 成员。系统调用是传递信息的作用。

2) 修改在内核层关于进程调度的模块:

在内核层修改调度，在内核里面通过多级调度算法，一个进程开始，其初始优先级确定，随着进程执行，优先级可能发生变化。用户创建的初始进程都在一个优先级队列里面，那么实现实时便需要把调用了系统调用的进程的优先级提高，提高了之后就优先执行。在优先级队列中选择最小的终止时间去优先执行。

注意的问题：

1. chrt 传递给内核的参数应该是进程的终止时间：当前时间+输入的 deadline。当消息传到内核，要根据消息结构体中的进程号，找到对应的它是修改的是内核中哪一个进程结构体，即进程的结构体的地址，然后把 deadline 传递到进程结构体的成员变量里面来。

2. 编译非常耗时，make build 需要大约 20 分钟，当仅仅是修改少量 C 文件时使用增量编译大约 2 钟。

3. 在实验过程中一直被这个问题困扰，在编译完成后输出的结果为：

```
proc1 set success
proc3 set success
proc2 set success
# prc3 heart beat 1
prc1 heart beat 1
prc2 heart beat 1
prc3 heart beat 2
prc1 heart beat 2
prc2 heart beat 2
prc3 heart beat 3
prc1 heart beat 3
prc2 heart beat 3
prc3 heart beat 4
prc1 heart beat 4
prc2 heart beat 4
prc3 heart beat 5
```

```
Change proc1 deadline to 5s
prc2 heart beat 5
prc3 heart beat 6
prc1 heart beat 5
prc2 heart beat 6
prc3 heart beat 7
prc1 heart beat 6
prc2 heart beat 7
prc3 heart beat 8
prc1 heart beat 7
prc2 heart beat 8
prc3 heart beat 9
prc1 heart beat 8
Change proc3 deadline to 3s
prc2 heart beat 9
prc3 heart beat 10
prc2 heart beat 10
prc3 heart beat 11
prc2 heart beat 11
prc2 heart beat 12
prc2 heart beat 13
```

与正确的输出结果不同，后来发现 chrt 没有执行_system_call，chrt 返回值为-1，所以系统调用出现了问题。由于修改的文件数量太多，该错误难以解决，故重新装 minix 源码，重新修改，最后才运行正确。

4. 在解决 3 中出现的问题时，发现在编译的时候开头出现了如图白色边框中的提示，误以为 3 中的问题可能是这个造成的，实际上这个并不影响。

```
# make build MKUPDATE=yes
Build started at: Fri Apr 16 17:09:53 GMT 2021
check-tools ===> .
params ===> .
     create  params
clean_METALOG ===> .
clean_METALOG ===> distrib/sets
do-distrib-dirs ===> .
distrib-dirs ===> etc   (with: DESTDIR=/)
proc:   permissions (0755, 0555, not modified: Function not implemented)
sys:    permissions (0755, 0444, not modified: Operation not permitted)
includes ===> .
includes ===> include
includes ===> include/../minix/include
includes ===> include/../minix/include/arch
includes ===> include/../minix/include/arch/i386
includes ===> include/../minix/include/arch/i386/include
includes ===> include/../minix/include/ddekit
```

support MobaXterm by subscribing to the professional edition here: https://mobaxterm.mobatek.net