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**：叶秋雨

上机实践名称：进程管理　　**学号**：10184102103　　**上机实践日期**：

上机实践编号：

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他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

物理机：Windows10

虚拟机：Minix3

虚拟机软件：Vmware

终端控制软件：MobaXterm

代码编辑：Source insight & VScode

物理机与虚拟机文件传输：FileZilla

四、 实验过程

1、应用层

实现 chrt 函数，调用_syscall(PM_PROC_NR,PM_CHRT,&m)，通过消息结构体 m，

将进程的 deadline 传递给服务层。

（1）在/usr/src/minix/lib/libc/sys/chrt.c 添加 chrt 函数的实现。

• 利用 alarm((unsigned int) deadline)函数来实现超时强制终止。在经过 deadline 的时间后，会传递信号 SIGALRM 给目前的进程。

• 利用 source insight 查看消息结构的定义，将 deadline 存放在结构体中 long 类型的一个相应变量即可。

• 结构体 m 中保存的 deadline 值要加上当前的时间。否则，就可能出现一个进程的 deadline 值更大但其先被调度，它的剩余运行时间小于一个后被调度但是 deadline 更小的进程。

message 结构体定义：

```
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 m1_i1  m_m1.m1i1
#define m1_i2  m_m1.m1i2
```

chrt 函数实现：

```
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));
}
```

（2）在对应的头文件/usr/src/include/unistd.h 中添加函数定义。

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

（3）/usr/src/minix/lib/libc/sys/Makefile.inc 文件中添加 chrt.c 条目。

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对应的 do_chrt 函数将进程号和 deadline 传递给 sys_chrt 函数，sys_chrt 函数将其放入消息结构体中并调用 _kernel_call(SYS_CHRT, &m)传递给内核层。

（1）在/usr/src/minix/include/minix/callnr.h 中定义 PM_CHRT 编号。在定义 PM_CHRT 编号时，系统调用总数也要进行相应修改。

```
#define PM_GETPROCNR        (PM_BASE + 46)
#define PM_GETSYSINFO       (PM_BASE + 47)

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

（2）在/usr/src/minix/servers/pm/table.c 中调用映射表。参照 fork、exit 定义进行添加。

```
   CALL(PM_EXIT)       = do_exit,
   CALL(PM_FORK)       = do_fork,
    CALL(PM_CHRT)           = do_chrt
```

（3）在对应头文件/usr/src/minix/servers/pm/proto.h 添加 chrt 函数的定义。

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

（4）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);
}
```

who_p 表示发起调用者的进程号，m_in.m2_l1 表示其 deadline。

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

（6）在/usr/src/minix/include/minix/syslib.h 中添加 sys_ chrt () 定义。通过查看同文件其他函数中用到进程号的定义可知，进程号的是 endpoint_t 类型的。

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

（7）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。

```
int sys_chrt(proc_ep,deadline)
```

```
endpoint_t proc_ep;
long deadline;
{
int r;
message m;
//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;
}
```

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

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_fork(struct proc * caller, message *m_ptr);
#if ! USE_FORK
#define do_fork NULL
#endif

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。

查看 proc_addr(n)的定义

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

利用消息结构体中的进程号通过函数 proc_addr 定位进程在内核中的地址，然后将消息结构中的 deadline 赋值给该进程的 p_deadline。

```
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);
}
```

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

（4）在/usr/src/minix/kernel/proc.h 中添加 deadline 成员。

```
    long long p_deadline;
```

（5）在/usr/src/minix/include/minix/com.h 中定义 SYS_CHRT 编号。定义系统调用编号后也要修改系统调用总数。

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

（6）在/usr/src/minix/kernel/system.c 中添加 SYS_CHRT 编号到 do_chrt 的映射。参考 do_fork、do_exec 按照相应格式添加。

```
map(SYS_FORK, do_fork);
map(SYS_EXEC, do_exec);
map(SYS_CHRT,do_chrt);
```

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

```
 system_tab[]=
{
    { "PRIVCTL",        SYS_PRIVCTL },
   { "TRACE",      SYS_TRACE },
   ...
    { "CHRT",    SYS_CHRT },
    { NULL,     0 }
};
```

（8）在/usr/src/minix/kernel/config.h 中添加需要的系统调用。

```
#define USE_FORK            1    /* fork a new process */
#define USE_CHRT            1
#define USE_NEWMAP          1    /* set a new memory map */
#define USE_EXEC            1    /* update process after execute */
```

4、进程调度

在/usr/src/minix/kernel/proc.h 中，struct proc 维护每个进程的信息，用于决策调度。向其中添加 deadline 成员。long long p_deadline。

在/usr/src/minix/kernel/proc.c 中，enqueue()和 enqueue_head()函数中添加代码，将 deadline>0 的实时进程添加到合适的优先级队列中。如果优先级队列不合适，则无法达到执行效果不理想。（测试出来优先级队列 5、6 都满足）

```
if (rp->p_deadline >0)
{
    rp->p_priority = 6;
}
```

在 pick_proc()函数中添加代码，在队列内部将时间片轮转调度改为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。

实现思路：遍历优先级队列，temp 记录下一个就绪的进程，如果当前进程结束或者 temp 进程剩余时间比当前进程进程更少并且 temp 进程可以运行，则用 temp 进程替换 rp，保证 rp 是当前剩余时间最少的进程。

```
rdy_head = get_cpulocal_var(run_q_head);
for (q=0; q < NR_SCHED_QUEUES; q++) {
  //优先级队列为空时
  if(!(rp = rdy_head[q])){
      TRACE(VF_PICKPROC, printf("cpu %d queue %d empty\n", cpuid, q););
      continue;
  }
  //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;
  if(q==6){
      //遍历链表
      //选择剩余时间最少的进程,并放到队首
      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;
    }
}
```

5、测试代码运行结果

在测试代码中，子进程顺序创建，最后创建的是进程 3，进程 3 先执行一次。其中进程 1 和进程 2 是实时进程 deadline 分别是 20s 和 15s，进程 3 为普通进程。因此优先级从高到低以此为 P2、P1、P3。所以进程按照 2、1、3 的顺序执行。5s 后，P1 的 deadline 修改为 5s，此时 P2 的 deadline 为 10s，P3 仍为普通进程，优先级变为 P1、P2、P3，进程按照 1、2、3 的顺序执行。在第 10s 时，只剩下 P2 和 P3 进程，同时设置了 P3 的 deadline 为 3s，此时 P2 的 deadline 为 5s，优先级顺序为 P3、P2。因此按照 3、2 的顺序执行。结果如图：

```
# ./test
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 minix 的不同服务模块和内核都是运行在不同进程中，熟悉了基于消息的进程间系统调用，进一步熟悉了 minix 的调度算法。

• 熟悉了利用 source insight 查看函数间的调用关系、变量的定义等等。

• 熟悉了利用 FileZilla 功能，连接虚拟机，在虚拟机与物理机之间进行文件的传递。

• 熟悉了利用 git diff 检查代码的修改。由于涉及到的文件比较多，可更直观地查看修改内容，避免一些错误。

• 本次实验遇到的问题：

1、在应用层传递 deadline 时应该加上当前时间，否则可能出现一个先运行的 deadline 时间更

长的进程其剩余时间小于一个后运行的 deadline 时间更短的进程。

2、由于修改的文件较多，按照应用层、服务层、内核层三层将文件整理好，便于查看每一层修改了哪些文件。在修改时要有耐心并且仔细。每一层修改完之后，及时进行编译，检查错误。

3、修改头文件和 Makefile 文件时要注意格式，可以参考 fork 函数的实现，否则可能多一个空格就会导致编译错误。