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：**2018 级　　**上机实践成绩：**

指导教师：翁楚良　　**姓名：**郑佳辰

上机实践名称：进程管理　　**学号：**10182100359　　**上机实践日期：**2020/4/14

上机实践编号：　　~5/24

一、 目的

熟悉安装和调试 MINIX3.3.0 操作系统，巩固操作系统通过串口输出信息。巩固操作系统的进程调度机制和策略，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。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。

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。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。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 15.5
Minix 3.3.0
MobaXterm Personal Edition v20.1
FileZilla 3.48.2.1
Dev-C++ 5.11
SourceInsight 4.0

四、 实验过程

1.增加系统调用 chrt

MINIX3 中的系统调用结构分成三个层次：应用层，服务层，内核层。我们需要在这三层中分别进行代码修改，实现系统调用 chrt 的信息传递。从应用层用_syscall 将信息传递到服务层，在服务层用_kernel_call 将信息传递到内核层，在内核层对进程结构体增加 deadline 成员。

1.1 应用层

在用户调用 chrt 系统调用之后，首先进入应用层。在应用层，我们应该设置 alarm，记录进程终止时间赋值给 deadline，并将 deadline 通过_syscall 传到服务层。

首先在/usr/src/include/unistd.h 中添加 chrt 函数定义。

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

然后在/usr/src/minix/lib/libc/sys/chrt.c 中添加 chrt 函数实现。用 alarm 函数实现超时强制终止。在实现中通过_syscall(调用号)向系统服务传递消息。进入函数之后要判断 deadline 的值是否合法，若 deadline 小于或等于 0 则直接返回不成功

0，不必将该进程设置为实时进程。通过消息结构体 m 进行进程间通信，传递 deadline。

```
int chrt(deadline)
const long deadline;
{
  message m;
  memset(&m, 0, sizeof(m));
  m.m_m2.m2l1 = deadline;

  if (deadline <= 0)
        return 0;

  alarm(deadline);

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

查询课本和使用全局搜索功能可知在/usr/src/minix/include/minix/ipc.h 中有 message 的定义。由于我们要传递一个 long 类型的数据，所以选择了 m_m2 传递 deadline。该结构体定义如图。

```
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;
_ASSERT_MSG_SIZE(mess_2);
```

最后，由于我们添加了 C 文件，需在同目录下的 Makefile/Makefile.inc 中添加条目。这里我们应该在/usr/src/minix/lib/libc/sys 中 Makefile.inc 文件添加 chrt.c 条目。

1.2 服务层

在服务层，我们需要注册 chrt 服务，将 deadline 传入到内核层。服务层会查找系统调用中是否有 PM_CHRT，如果有则调用映射表中其对应的 do_chrt 函数。do_chrt 函数调用 sys_chrt 函数，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。

在/usr/src/minix/include/minix/callnr.h 中定义 PM_CHRT 编号。这里同时需要修改编号的取值范围限制。

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

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

```
CALL(PM_CHRT) = do_chrt /* chrt(2) */
```

在/usr/src/minix/servers/pm/proto.h 中添加 do_chrt 函数定义。由于其他函数的参数均为 void，所以这个系统调用也应没有参数。

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

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

do_chrt 函数需要传递消息的进程号和 deadline。参考同文件夹下的系统调用文件，使用查看调用关系功能可以查找得，mp 是指向当前进程结构体的指针。

```
/* Global variables. */
EXTERN struct mproc *mp;     /* ptr to 'mproc' slot of current process */
EXTERN int procs_in_use;     /* how many processes are marked as IN_USE */
EXTERN char monitor_params[MULTIBOOT_PARAM_BUF_SIZE];
```

同样使用调用关系功能可以查得，m_in 是输入的结构体，也是刚刚在应用层携带 deadline 的 message 结构体。

```
/* Allocate space for the global variables. */
static message m_in;        /* the input message itself */
static message m_out;       /* the output message used for reply */
static endpoint_t who_e;    /* caller's proc number */
static int callnr;      /* system call number */
```

mp->endpoint 是进程号对应的 endpoint，可以根据 endpoint 重新定位到该进程。m_in.m_m2.m2l1 是在应用层存储的 deadline，将这几个参数传入 sys_chrt。

```
int do_chrt(void)
{
  return (sys_chrt(mp->mp_endpoint,m_in.m_m2.m2l1));
}
```

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

在/usr/src/minix/include/minix/syslib.h 中添加 sys_ chrt () 定义。由于 proc_ep 是刚刚在 do_chrt 中传递的 mp->endpoint，数据类型是 endpoint_t。deadline 类型为 long，所以函数的参数如下。

```
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。在实现中通过 _kernel_call(调用号)向内核传递。将接收的进程号和截止时间打包到 m 消息结构体中，然后将该结构体作为参数传入内核层。

```
int sys_chrt(proc_ep, deadline)
endpoint_t proc_ep;
long deadline;
{
  message m;
  int r;

  m.m_lsys_krn_sys_trace.endpt = proc_ep;
  m.m_lsys_krn_sys_trace.data = deadline;

  r = _kernel_call(SYS_CHRT, &m);
  return r;
}
```

从 ipc.h 中继续查找 message 类型，选取 m_lsys_krn_sys_trace 来传递消息进行进程间通信 IPC，该消息定义如图。

```
typedef struct {
    int request;
    endpoint_t endpt;
    vir_bytes address;
    long int data;

    uint8_t padding[40];
} mess_lsys_krn_sys_trace;
_ASSERT_MSG_SIZE(mess_lsys_krn_sys_trace);
```

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

1.3 内核层

内核层会查找映射表中是否有 SYS_CHRT，如果有则调用映射表中其对应的 do_chrt 函数。do_chrt 函数找到内核中的进程地址，并修改进程内容。

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

```
{ "CHRT",           SYS_CHRT },
```

在/usr/src/minix/include/minix/com.h 中定义 SYS_CHRT 编号，同时修改编号的取值范围限制。

```
#  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);  /* change process deadline */
```

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);
```

在/usr/src/minix/kernel/system/do_ chrt.c 中添加 do_chrt 函数实现。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。通过查看调用的功能可以得知，isokendpt 函数是将传入的进程的 endpoint 转换进程号 p_proc。然后利用 proc_addr 函数可以将进程号 p_proc 转换为一个 proc 指针。最后更改这个 proc 指针所指的 proc 结构体中的 p_deadline 成员变量，用来完成下面的的实时调度功能。

```
int do_chrt(struct proc * caller, message * m_ptr)
{
  struct proc *rpp;            /* process pointer */
  int p_proc;

  if(!isokendpt(m_ptr->m_lsys_krn_sys_trace.endpt, &p_proc))
  return EINVAL;

  rpp = proc_addr(p_proc);
  rpp->p_deadline = m_ptr->m_lsys_krn_sys_trace.data;

  return OK;
}
```

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

2.实现 Earliest-Deadline-First 近似实时调度功能

MINIX3 使用一种多级调度算法。进程优先级数字越小，优先级越高，根据优先级不同分成了 16 个可运行进程队列。每个队列内部采用时间片轮转调度，找到最高非空优先级队列，选取队列首部可运行的进程。当用完了时间片，则移到当前队列的队尾。

将 EDF 添加到多级调度算法中，可控制入队实现实时调度。入队是将当前剩余时间大于 0 的进程添加到某个优先级队列，即设置进程优先级。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，即在实时进程的队列中选取截止时间最近的进程。

2.1 进程调度实现

进程调度模块位于/usr/src/minix/kernel/下的 proc.h 和 proc.c，我们需要修改影响进程调度顺序的部分。在/usr/src/minix/kernel/proc.h 中的结构体 struct proc 中添加以下记录截止时间 deadline 的成员变量。

```
long p_deadline;
```

在/usr/src/minix/kernel/proc.c 中的 enqueue 和 enqueue_head 两个函数的开头添加如下语句。将实时进程的优先级设置为 5，也就是全部放入第 5 个队列中进行调度。这样实时进程的优先级高于非实时的用户进程。

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

在 pick_proc 函数的循环体内添加如下代码段。当我们从优先级为 5 的队列中选取进程时，要选取截止时间最近的实时进程.如果没有实施进程，就选取队列头部的非实时进程。具体的做法如下。遍历第五个队列中的每个进程，如果该进程可以运行，并且是实时进程，那么就与当前记录的将要被调度的进程 rp 作比较。如果 rp 指向的是非实时进程，或 rp 是实时进程，但 tp 的截止时间比 rp 要接近。那么我们就调度 tp，让截止时间小的进程先运行。

```
if(q==5){
  for(tp = rp; tp!=NULL; tp=tp->p_nextready)
        if(proc_is_runnable(tp) && tp->p_deadline!=0 && (rp->p_deadlin
e==0 || tp->p_deadline < rp->p_deadline))
              rp = tp;
}
```

2.2 进程调度分析

在 proc.c 的函数中，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。enqueue()按优先级将进程加入列队尾。实验中需要将实时进程的优 先级设置成合适的优先级。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。switch_to_user 函数会调用 pick_proc 选择进程，并进行上下文切换等操作。

在内核层 do_chrt 函数中，设置 p_deadline 之后，继续设置实时进程的优先级 p_priority 是无效的。因为在程序的其他地方还有可能改变实时进程的优先级，从而使得我们设置的优先级无效。在每次进入队列时设置优先级，可以保证实时进程进入到合适的队列中。

我们进行了以上更改之后，不同的队列中仍然按照优先级调度进程，每个队列中按照时间片来调度进程。但截止时间最近的实时进程的时间会优先被调度，即最先获得时间片，这样我们就实现了近似实时调度。

3.编译及测试

3.1 编译 MINIX3

使用 make build 指令编译修改过源文件之后的 MINIX3，首次编译，修改过头文件或 Makefile 文件时必须采用这种编译方式。使用增量式编译应输入 make build MKUPDATE=yes 指令，这种编译方式较快。此外，首次编译时需要下载部分文件，这时虚拟机应保持 NAT 模式，并保持网络畅通。编译成功时如下图所示。

```
install -N /usr/src/etc -c -p -r ../sbin/init/ini
rm /dev/c0d3p0s0:/boot/minix/3.3.0r1
Done.
Build started at:  Fri May 22 18:28:22 GMT 2020
Build finished at: Fri May 22 18:35:16 GMT 2020
# 
```

3.2 测试

测试程序主要代码如下。

```
int main(void)
{
  for (int i = 1; i < 4; i++)
  {
    if (fork() == 0)
    {
      proc(i);
    }
  }
  return 0;
}
void proc(int id)
{
  int loop;
  switch (id)
  {
  case 1:
    chrt(20);
    printf("proc1 set success\n");
    break;
  case 2:
    chrt(15);
    printf("proc2 set success\n");
    break;
  case 3:
    chrt(0);
    printf("proc3 set success\n");
    break;
  }
  for (loop = 1; loop < 40; loop++)
  {
    if (id == 1 && loop == 5)
    {
      chrt(5);
      printf("Change proc1 deadline to 5s\n");
    }
    if (id == 3 && loop == 10)
    {
      chrt(3);
      printf("Change proc3 deadline to 3s\n");
    }
    sleep(1);
    printf("prc%d heart beat %d\n", id, loop);
  }
  exit(0);
}
```

在测试中，在 main 函数中 fork 三个子进程(P1, P2, P3)，并为每个子进程设置 id。开始时设置 P1 和 P2 为实时进程，deadline 分别设为 20s 和 15s。P3 为非实时进程。前 5s 内，根据截止时间，优先级为 P2 > P1 > P3。

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

第 5s 时 P1 调用 chrt(5)，将 P1 的 deadline 设置为 5s。所以第 5s 后至第 10s，优先级为 P1 > P2 > P3。

```
Change proc1 deadline to 5s
prc1 heart beat 5
prc2 heart beat 5
prc3 heart beat 5
prc1 heart beat 6
prc2 heart beat 6
prc3 heart beat 6
prc1 heart beat 7
prc2 heart beat 7
prc3 heart beat 7
prc1 heart beat 8
prc2 heart beat 8
prc3 heart beat 8
prc2 heart beat 9
prc3 heart beat 9
```

第 10s 时 P3 调用 chrt(3)，将 P3 的 deadline 设置为 3s。此时 P1 的截止时间已到，结束运行。所以第 10s 后，优先级为 P3 > P2。由于进程休眠时间并不精确，有一定时间差。

```
Change proc3 deadline to 3s
prc3 heart beat 10
prc2 heart beat 10
prc3 heart beat 11
prc2 heart beat 11
prc2 heart beat 12
prc2 heart beat 13
prc2 heart beat 14
```

五、 总结

这次实验的主要工作是在 MINIX3 系统中添加系统调用和修改进程调度部分。添加系统调用 chrt 时需要修改应用层，服务层和内核层的代码，从而设置 alarm，修改内核中的截止时间变量。修改进程调度分需要为实时进程选择合适的优先级，并且根据截止时间选择被调度的进程。

实验中我使用了很多有用的工具帮助我分析并调试代码。如 source insight 高级功能中的调用关系，全局搜索等功能，有效地帮我查找每个变量或函数的定义和用途。修改涉及文件较多，git diff 可直观看到修改内容，避免引入无意的错误。FileZilla 连接虚拟机，拉取需修改的文件，修改后上传到虚拟机。通过串口可以将虚拟机内的运行日志输出到外部物理机上查看，并追踪程序运行逻辑。善用这些工具可以有效地减少无用功。

通过这次进程管理的实验，我对进程表和进程控制块有了更深入的理解，学会使用 message 进行进程间通信，更加明确了调度算法及其实现。总体上我对进程这个重要概念有了更直观的印象，同时的借助 MINIX3 更好地理解进程的实现。