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：**龚敬洋

上机实践名称：进程调度 EDF　　**学号：**10195501436　　**上机实践日期：**2021/4/27

上机实践编号：

一、 目的

修改 MINIX3 系统内核，增加一个系统调用 chrt，并在其中实现 EDF(Earlist-Deadline-First)进程调度算法。

二、 内容与设计思想

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：

```
#include<unistd.h>
...
chrt(10); /* 该程序将可以运行的最长时间为 10 秒，若没有运行结束，则强制结束 */
...
```

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

开发环境：Clion 2020.3

宿主机系统环境：Mac OS Big Sur 11.2.2

虚拟机应用：VirtualBox 6.1.18

虚拟机环境：Minix 3.3.0

四、 实验过程

1. MINIX 系统构架

作为一个微内核构架系统，Minix 将系统进程分为了 4 层：内核层、驱动管理层、服务器进程层、用户进程层，其中内核层运行在系统内核态，而后三层均运行在用户态。

在 Minix 中，层与层之间的消息传递通过系统调用来完成，而这又分为了 System Call 和 Kernel Call。System Call 用于应用层向服务层的消息传递，Kernel Call 则用于服务层向内核层的消息传递。

消息传递本质上是进程间通讯，从内存角度看即为内存地址间的内容拷贝。幸运的是，Minix 已经帮我们封装好了这些调用的底层实现，我们只需要传入正确的参数，系统会自动托管底层内存拷贝的相关事务：

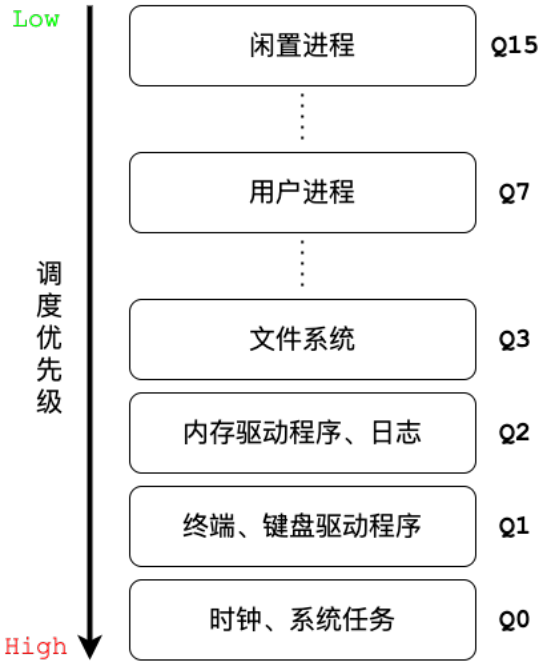
（minix/lib/libc/sys/syscall.c）

```
1.  int _syscall(endpoint_t who, int syscallnr, message *msgptr)
2.  {
3.    int status;
4.
5.    msgptr->m_type = syscallnr;
6.    status = ipc_sendrec(who, msgptr);
7.    if (status != 0) {
8.      /* 'ipc_sendrec' itself failed. */
9.      /* XXX - strerror doesn't know all the codes */
10.     msgptr->m_type = status;
11.   }
12.   if (msgptr->m_type < 0) {
13.     errno = -msgptr->m_type;
14.     return(-1);
15.   }
16.   return(msgptr->m_type);
17. }
```

（minix/lib/libsys/kernel_call.c）

```
1.  int _kernel_call(int syscallnr, message *msgptr)
2.  {
3.    msgptr->m_type = syscallnr;
4.    do_kernel_call(msgptr);
5.    return(msgptr->m_type);
6.  }
```

Minix 系统采用了一种多级调度算法，通过维护 16 个进程队列并赋予其不同的队列优先级来实现进程的分级。其中，0 号队列用于放置时钟及系统任务，系统会允许其持续运行直到阻塞（但如果其运行时间过长，系统会设置一个罚时将其暂时移出队列以防止其他进程发生饥饿），7 号队列用于放置用户进程，15 号队列用于放置闲置进程。在每个进程队列内部，系统采用了时间片轮转的方式使得进程可以公平的分配到运行时间。



2. EDF 调度实现

要实现 EDF 调度算法，需要记录每个进程的截止时间。为此，我们在进程控制块（Process Control Block）的结构定义中新增一个 p_deadline 项：（minix/kernel/proc.h）

```
1.  struct proc {
2.      //Unrelated codes
3.
4.      long p_deadline;  /* Deadline of the process */
5.
6.      //Unrelated codes
7.  };
```

由于 Minix 采用多级进程队列，我们可以选择其中的一个进程队列，并在其中使用 EDF 算法进行调度。由于要保证所有调用 chrt 系统调用的进程都使用该调度规则，所选择的进程队列的整体优先级要高于用户进程所在的队列，但同时又不能影响系统进程的运作。这里我们选择优先级为 5 的队列作为 EDF 调度队列（事实上，4 号队列也可以作为要替换的目标队列，但为了防止驱动或系统进程临时调度到这一队列，在此我们将其留出作为缓冲）。

对于一般的进程，我们在进程初始化时将 p_deadline 置为 0。这样在进程调度时，若检测到 p_deadline>0，即可得知其为调用了 chrt 系统调用的进程，我们便将其加入优先级为 5 的队列中：（minix/kernel/proc.h）

```
1.  void enqueue(
2.  register struct proc *rp  /* this process is now runnable */
3.  )
4.  {
5.      //Unrelated codes
6.      if(rp->p_deadline > 0){
7.          rp->p_priority = 5;
8.      }
9.      //Unrelated codes
10. }
11.
```

```
static void enqueue_head(struct proc *rp)
{
    //Unrelated codes
    if(rp->p_deadline > 0){
        rp->p_priority = 5;
    }
    //Unrelated codes
}
```

在调度时，我们要找出队列中 p_deadline 最小的进程并返回。一个可行的办法是维护一个优先队列，按照 p_deadline 对进程控制结构建立小根堆，其可以在$O(lgn)$的时间内返回目标进程，但这样做需要修改整个进程队列的数据结构，操作起来过于复杂，也不符合 Minix3 的原始设计风格。由于进入该队列的进程是由用户指定的，其规模通常较小，因此我们可以直接遍历整个队列，其效率仍然是可以接受的。实现代码如下：
（minix/kernel/proc.c）

```
static struct proc * pick_proc(void)
{
    //Unrelated codes
    for (q=0; q < NR_SCHED_QUEUES; q++) {
        //Unrelated codes
        //EDF algorithm
        if(q == 5){
            rp = rdy_head[q];
            struct proc *cur = rp->p_nextready;
            //Traverse the queue
            while(cur != NULL) {
                if(proc_is_runnable(cur) && (cur->p_deadline > 0)) {
                    if (rp->p_deadline > cur->p_deadline) {
                        rp = cur;
                    } else if (rp->p_deadline == 0){
                        rp = cur;
                    }
                }
                cur = cur->p_nextready;
            }
        }
        //Unrelated codes
        return rp;
    }
    return NULL;
}
```

3. 应用层实现

在应用层中，我们需要实现面向用户的 chrt 函数，并将用户指定的进程和截止时间传入服务层。

首先我们在 POSIX 规定的操作系统 API 头文件中定义 chrt 的函数原型：
（include/unistd.h）

```
int chrt(long deadline); // 0 - Normal process; >0 - Realtime process; <0 - Unsuccessful
```

用户指定的截止时间是一个相对时间，即从该语句执行时刻向后 deadline 秒，因此我们需要将其转为绝对时间（事实上是相对系统时钟当前时刻的时间）。Minix 系统提供了一个 clock_gettime 函数用于获取系统的时间戳，因此我们可以直接调用该函数来算出当前进程所指定 deadline 对应的绝对时刻：(minix/lib/libc/sys/chrt.c)

```c
#include <sys/cdefs.h>
#include "namespace.h"
#include <lib.h>

#include <string.h>
#include <unistd.h>
#include <time.h>

int chrt(long deadline){
    message m;
    struct timespec now;
    memset(&m, 0, sizeof(m));
    //Unrelated codes
    if(deadline < 0) return 0;
    else if(deadline > 0){
        clock_gettime(CLOCK_REALTIME, &now);
        deadline = now.tv_sec + deadline;
    }
    //Unrelated codes
}
```

此外，我们还需要对传入的 deadline 参数做一些边界处理，并通过 alarm 系统调用将超时响应的应用提前结束。随后我们便可以将其放入一个消息结构体中，并通过 System Call（_syscall 函数）将消息传入服务层中：(minix/lib/libc/sys/chrt.c)

```c
int chrt(long deadline){
    //Unrelated codes
    alarm((unsigned int) deadline);
    //Unrelated codes
    m.m2_l1 = deadline;
    return _syscall(PM_PROC_NR, PM_CHRT, &m);
}
```

4. 服务层实现

对于要实现的 chrt 系统调用来说，服务层起到了消息传递的作用。在 Minix3 系统中，这需要两步来完成，先接受应用层传来的消息，再将消息重新打包并通过 Kernel Call 传入内核中。

首先我们来实现消息接受的功能。在应用层中，我们通过调用标识符为 PM_CHRT 的 System Call 将消息发到了服务层中，于是我们需要在服务层中申明这一 System Call 并将其与消息接收函数相关联：（minix/include/minix/callnr.h）

```c
#define PM_CHRT     (PM_BASE + 48)

#define NR_PM_CALLS     49  /* highest number from base plus one */
```

（minix/servers/pm/table.c）

```c
int (* const call_vec[NR_PM_CALLS])(void) = {
    CALL(PM_CHRT)       = do_chrt,      /* chrt */
};
```

（minix/servers/pm/proto.h）

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

至于消息接收函数的实现，我们只需要将发来消息的进程号和发来的消息传递给承接服务层向内核进行消息传递的函数即可：（minix/servers/pm/chrt.c）

```
#include "pm.h"
#include <signal.h>
#include <sys/time.h>
#include <minix/com.h>
#include <minix/callnr.h>
#include "mproc.h"

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

随后我们来实现服务层向内核层的消息传递。首先我们定义消息传递函数的原型：（minix/include/minix/syslib.h）

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

对于该函数我们只需要将传入的函数重新打包为一个新的消息，并通过 Kernel Call（_kernel_call 函数）将其传入内核即可：（minix/lib/libsys/sys_chrt.c）

```
#include "syslib.h"

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

5. 内核层实现

首先我们定义服务层中调用的 SYS_CHRT 内核调用，并将其与内核实现函数相关联：（minix/include/minix/com.h）

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

/* Total */
#define NR_SYS_CALLS    59  /* number of kernel calls */
```

(minix/kernel/system.c)

```
map(SYS_CHRT, do_chrt);       /* chrt */
```

随后我们定义实现函数的原型，并在内核中默认启用它：（minix/kernel/config.h）

```
#define USE_CHRT           1    /* chrt */
```

（minix/kernel/system.h）

```
int do_chrt(struct proc * caller, message *m_ptr);
#if ! USE_CHRT
```

```
#define do_chrt NULL
#endif
```

内核的任务就是把上层传递下来的消息解包，并将目标进程的截止时间设置为用户所指定的时间：（minix/kernel/system/do_chrt.c）

```
#include "kernel/system.h"
#include "kernel/vm.h"
#include <signal.h>
#include <string.h>
#include <assert.h>

#include <minix/endpoint.h>
#include <minix/u64.h>

#if USE_CHRT

int do_chrt(struct proc *caller, message *m_ptr){
    struct proc *rp;
    long deadline;
    deadline = m_ptr->m2_l1;
    rp = proc_addr(m_ptr->m2_i1);
    rp->p_deadline = deadline;
    return OK;
}

#endif /* USE_FORK */
```

6. 功能测试

我们使用以下代码来对实现的功能进行测试：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <lib.h>
#include <time.h>

void proc(int id);
int main(void)
{
  //创建三个子进程，并赋予子进程 id
  for (int i = 1; i < 4; i++)
  {
    if (fork() == 0)
    {
      proc(i);
    }
  }
  return 0;
}
void proc(int id)
{
  int loop;
  switch (id)
  {
  case 1: //子进程 1，设置 deadline=20
```

```
30.     chrt(20);
31.     printf("proc1 set success\n");
32.     //sleep(1);
33.     break;
34.   case 2: //子进程 2，设置 deadline=15
35.     chrt(15);
36.     printf("proc2 set success\n");
37.     //sleep(1);
38.     break;
39.   case 3: //子进程 3，普通进程
40.     chrt(0);
41.     printf("proc3 set success\n");
42.     break;
43.   }
44.   for (loop = 1; loop < 40; loop++)
45.   {
46.     //子进程 1 在 5s 后设置 deadline=5
47.     if (id == 1 && loop == 5)
48.     {
49.       long tmp;
50.       tmp = chrt(5);
51.       printf("Status of CHRT: %d\n", tmp);
52.       printf("Change proc1 deadline to 5s\n");
53.     }
54.     //子进程 3 在 10s 后设置 deadline=3
55.     if (id == 3 && loop == 10)
56.     {
57.       chrt(3);
58.       printf("Change proc3 deadline to 3s\n");
59.     }
60.     sleep(1); //睡眠，否则会打印很多信息
61.     printf("prc%d heart beat %d\n", id, loop);
62.   }
63.   exit(0);
64. }
```

该程序创建了 3 个子进程，并对其分别设置了不同的截止时间，其运行结果如下：

```
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
Status of CHRT: 0
Change proc1 deadline to 5s
prc3 heart beat 4
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
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
```

可以看到，程序中 chrt 系统调用的返回值为 0，表明其成功将消息传入了内核，且程序行为与预期相符，表明了实现的正确性。

五、总结

在本实验中，我们通过修改 Minix3 的系统源码，实现了一个完整的系统调用，并在进程调度中实现了 EDF 算法，极大的加深了我们对一个微内核操作系统的系统调用、消息传递及进程调度机制的理解。