华东师范大学数据科学与工程学院实验报告

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**：池欣宁	**学号**：10185501409
上机实践名称：chrt 系统调用和 EDF 近似实时调度		**上机实践日期**：
上机实践编号：	**组号**：	**上机实践时间**：

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. 增加系统调用 chrt
2. 修改 MINIX3.3 **的调度算法**

三、 **使用环境**

MINIX R3.3.0

四、 **实验过程**

整体思路:

类比于其它系统调用(fork)增加一个名为 chrt 的系统调用,这个系统调用同时为调用 chrt 的进程设置一个闹钟,同时这个系统调用通过_sys_call 和_kernel_call 向内核传递 deadline 信息;

同时修改 MINIX 的调度算法,将调用 chrt 的进程的优先级设置为一个较高的合适的优先级,但不能是 0 优先级等,因为这样会和系统进程产生调度冲突;当这些调用 chrt 进程的优先级被设置较高时,与其余未调用 chrt 的进程或 deadline=0 的普通进程比,因为享有较高优先级,所以被优先调度;

在设置了较高优先级队列的实时进程队列中,遍历整个队列,将原先的时间片轮转调度修改为离 deadline 最近的进程被置于队首被优先调度;

同时需要注意的是对离“deadline 最近”这一概念的表达:

设置 deadtime=nowtime(调用 chrt 的时刻)+deadline(实际秒数);

将 deadtime 作为参数,实际参与从**应用层**的传递;

deadtime 的理解是:deadtime 反映了从调用 chrt 的时刻起,进程的结束调用时刻,所以只需要比较各个进程间的进程结束调用时刻,当调用时刻越早,即从数值上来看越小时,说明该进程越需要被优先调度;同时,即使是在后续 deadtime 时间内同一进程再次调用 chrt 修改了进程的 deadtime 时间,也是**实时**修改了进程的结束调用时刻;

1. 增加系统调用 chrt

int chrt(long deadline);

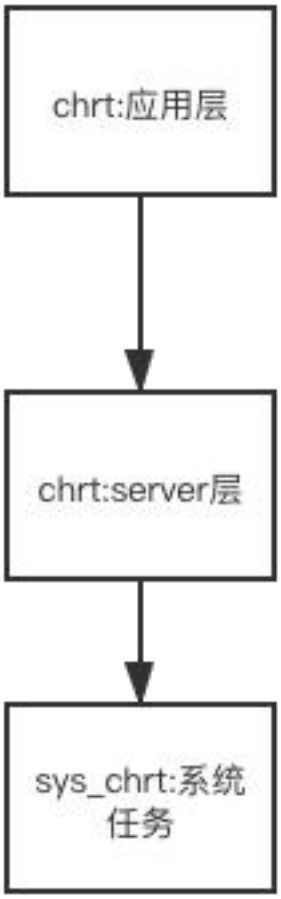
该系统调用实现的功能:

a. **Deadline=0,为普通进程,priority 比实时进程的优先级低;**

b. Deadline>0,为实时进程,即该进程需要在在 deadline 秒内结束.如果到达进程结束时间点时该进程仍然没有结束,则超过时间强制停止该进程;

支持在进程的 deadline 时间内进行多次调用 chrt 修改 deadline;

c. Deadline<0;直接返回不成功;



在 MINIX3 中,用户进程发出的系统调用将被转换为发往服务器进程的消息;由服务器进程给系统任务进程发消息,由系统任务进程完成剩下的工作;

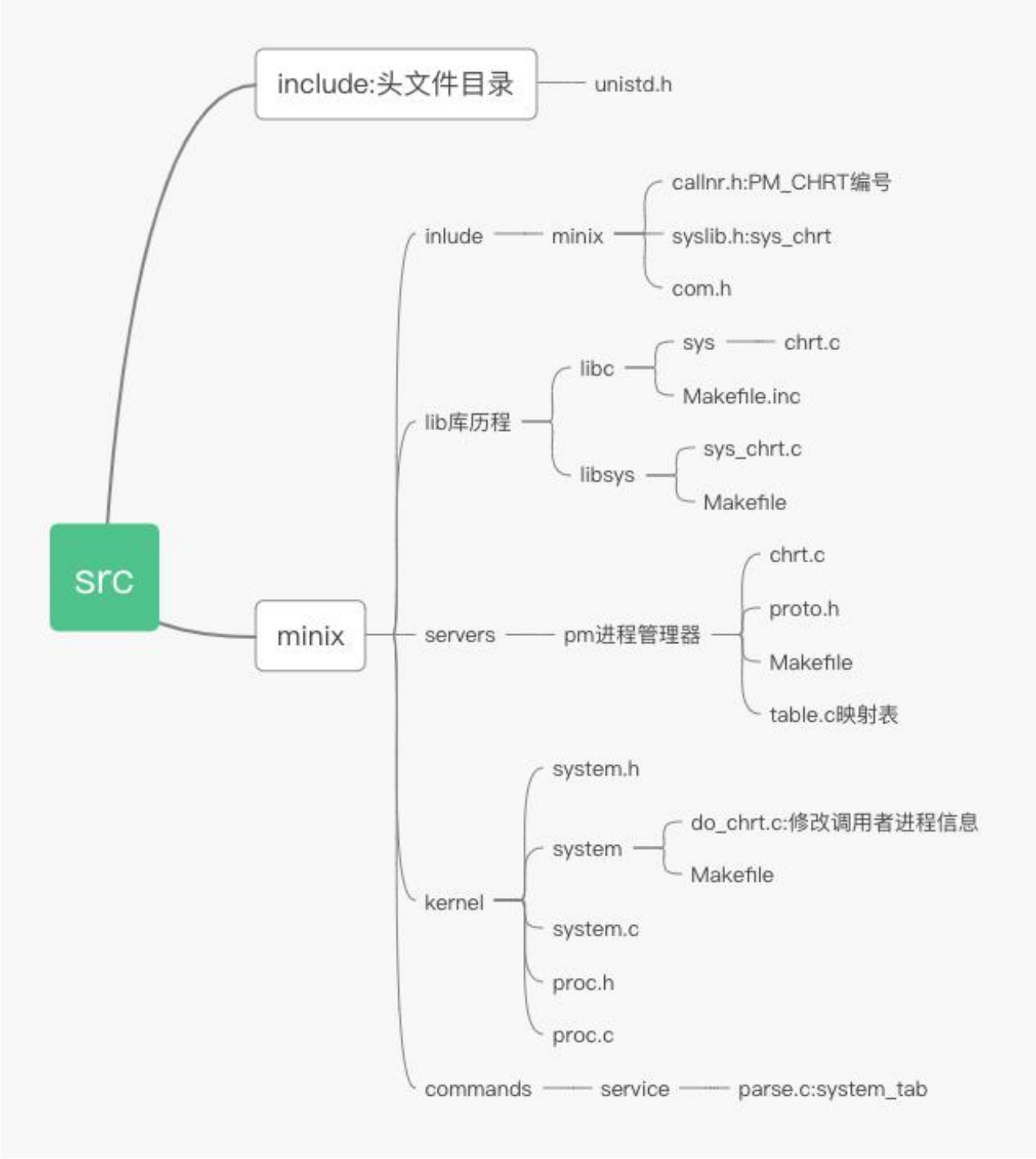
由系统任务接受的消息类型:

这里 sys_chrt 对应 POSIX 系统调用,属于进程管理类;

系统函数库:在系统任务主程序 system.c 中每一个名字为 do_xyz 形式的函数的源码在 kernel/system/do_xyz.c 文件中;在 system.c 中也添加响应编号的映射;

MINIX3 操作系统本身是由许多进程组成,这些系统进程之间通过消息传送机制相互通信,用户进程也可以通过消息传送机制与操作系统进程之间通信,从而请求操作系统的服务;

2. 在 MINIX 操作系统下增加系统调用的流程:



如上述流程图所示,分别从应用层,服务层,内核层进行系统调用;

A. 内核层:

usr/src/minix/kernel/system.h,system.c:系统任务的总框架;

usr/src/minix/kernel/system/:包含了每个响应函数的源文件;

内核层需要实现的内容如下:

改内核信息，例如进程的截至时间。

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

_ 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));
}
```

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

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

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

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

内核层 do_chrt.c 的具体实现:

```
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);/*获得调用 chrt 的进程的 proc 结构
*/
```

```
rp->p_deadline = deadline;
```

```
return (OK);
}
```

注意这里的消息结构体:m2_l1 表示进程的 deadline,m2_i1 表示进程是哪个进程;
消息结构体在各层逻辑相同;

B. 服务层:

PM 服务实现进程的创建,启动与终止等系统调用;

- 服务层：需要向 MINIX 系统的进程管理服务中注册 chrt，使得

chrt 服务可以向应用层提供 。

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

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

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

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

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

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

_ 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);
}

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

这里 chrt.c 调用 sys_chrt.c,
Sys_chrt.c 在服务层需要像内核传递信息,信息是 m2_i1 和 m2_l1;

```
//声明风格模仿 sys_fork.c
int sys_chrt(who, deadline)
long deadline;
endpoint_t who;
{
message m;
m.m2_l1=deadline;
m.m2_i1=who;
return _kernel_call(SYS_CHRT,&m);
}
```

C. 应用层:

应用层：需要添加的系统调用 chrt 可以定义在 unistd 头文件中，并在 libc 中添加 chrt 函数体实现 。

- 在/usr/src/include/unistd.h 中添加 chrt 函数定义。
- 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。例如：

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

- 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）。

```c
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);
deadline =time.tv_sec+deadline;//nowtime+deadline;
}
m.m2_l1=deadline;

return(_syscall(PM_PROC_NR, PM_CHRT, &m));//在消息结构体中将deadline 放入
}
```

3. 修改系统调度算法,模拟 EDF:early-deadline-first 的近似实时调度:

A. MINIX 系统的进程调度方式:

◆ 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,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(终止时间-运行时间)大于 0 的进程添加到某个优先级队列，即设置进程优先级(需要选择合适的优先级否则执行效果不理想）。

◆ 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，即将剩余时间最小的进程移到队列首部。

具体代码修改思路:
1.修改 proc.h 为进程结构体增加 deadline(deadtime)信息;
2.修改 enqueue()将进程加入至队尾函数,对于 deadtime>0(即实时进程,设置优先级为 5);
3.修改 enqueue_head()将进程加入至队首函数,对于 deadtime>0(即实时进程,设置优先级为 5);
4.修改 pick_proc()函数,对于之前设置的优先级 5 的队列,在该队列中进行进程调度时,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(deadtime)**并**可运行**的进程.
Proc.c 中代码如下

```
if(q==5){//对于特殊的优先级队列
rp=rdy_head[q];//rp 是当前队首的第一个
tmp=rp->p_nextready;//待调队列队首的下一个,实质承载遍历
while(tmp!=NULL){//tmp 非空才有遍历的价值
if(tmp->p_deadline>0){//当下一个不为 0 的时候:
if(rp->p_deadline==0)&&proc_is_runnable(tmp))//如果队首进程是普通进程,注意这里不意味着优先级为 5 一定是实时进程;
rp=tmp;
else if(rp->deadline>tmp-
>deadline)&&proc_is_runnable(tmp))
rp=tmp;//deadtime 早的享有优先调度的权利;
}
}
tmp=tmp->p_nextready;
}
}
```

4. 测试

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。
P1 和 P2 为实时进程，deadline 分别设为 20s 和 15s。
三个子进程会打印出子进程 id 和循环次数。
第 0s 时：优先级 P2 > P1 > P3;
第 5s 时：P1 设置 deadline 为 5s，P1 调用 chrt(5);
第 5s 后：优先级 P1 > P2 > P3;
第 10s 时：P3 设置 deadline 为 3s，P3 调用 chrt(3);
第 10s 后：优先级 P3 > P2;

测试代码如下:

```
void proc(int id);
int main(void)
{
//创建三个子进程，并赋予子进程 id
for (int i = 0; i < 3; i++)
{
if (fork() == 0)
{
proc(i);
}
}
return 0;
}
void proc(int id)
{
int loop;
switch (id)
{
case 0: //子进程 1，设置 deadline=20
chrt(20);
printf("proc1 set success\n");
sleep(1);
break;
case 1: //子进程 2，设置 deadline=15
chrt(15);
```

```
printf("proc2 set success\n");
sleep(1);
break;
case 2: //子进程 3，普通进程
chrt(0);
printf("proc3 set success\n");
break;
}
for (loop = 1; loop < 40; loop++)
{
sleep(1); //睡眠，否则会打印很多信息
printf("prc%d heart beat %d\n", id+1, loop);
}
exit(0);
}
```

```
# ./final
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
```

利用测试样例测试结果如上图所示;

通过 git diff 工具可得,最终修改的 minix 系统 src 文件如下,被修改的文件在上述思维导图结构中已经给出:

1.内核层:

Src/minix/kernel/proc.h
Src/minix/kernel/proc.c

Src/minix/kernel/system/system.h
Src/minix/kernel/system/do_chrt.c
Src/minix/kernel/system/Makefile.inc

Src/minix/kernel/commands/service/parse.c

2.服务层

Src/minix/servers/pm/proto.h
Src/minix/servers/pm/table.c
Src/minix/servers/pm/chrt.c
Src/minix/servers/pm/Makefile.inc
Src/minix/lib/libsys/sys_chrt.c
Src/minix/lib/libsys/Makefile.inc
Src/minix/include/minix/callnr.h
Src/minix/include/minix/syslib.h
Src/minix/include/minix/com.h

3.应用层

Src/include/unistd.h
Src/minix/lib/libc/sys/chrt.c
Src/minix/lib/libc/sys/Makefile.inc

五、总结

这次第一次真正意义上的接触了操作系统和系统调用的任务,在实验过程中加深了对于课本第二章理论知识的认识,同时也是第一次接触到了操作系统本身的编译;

对于 **MINIX3** 的微内核也有了初步感知,才发现从理论到实践落地是一个不断迭代试错的过程.

希望自己可以更多的加深对理论概念的理解和感性认识,从线性学习完成到工科学习的转变.