操作系统 实验报告2

温兆和 10205501432

实验背景

在本次实验中，我们要在MINIX 3中实现Earliest-Deadline-First进程调度。也就是说，用户可以通过 `chrt(x)` 系统调用把进程的deadline修改为调用后的x秒，如果deadline到了以后进程还没有 运行完，系统就会强制结束该进程。

实验过程

I.在MINIX操作系统中添加 `chrt` 系统调用

这一步，我们会从应用层到服务层再深入到内核层，在MINIX系统中添加一个可以修改进程最长运行时间的系统调用。

应用层

应用层中， `chrt` 函数会调用 `_syscall` ，通过消息结构体 `m` 把进程的deadline传递到服务层。

首先，在 `/usr/src/include/unistd.h` 中添加 `chrt` 函数定义。其中， `chrt` 函数的定义被添加在第91行，需要传入一个长整型的deadline：

In [ ]:
```
__BEGIN_DECLS
__dead void _exit(int);
int access(const char *, int);
unsigned int alarm(unsigned int);
int chdir(const char *);
int chrt(long);
#if defined(_POSIX_C_SOURCE) || defined(_XOPEN_SOURCE)
```

在 `/usr/src/minix/lib/libc/sys/chrt.c` 中添加 `chrt` 函数实现。它将获取调用 `chrt` 时的时间，把这个时间与deadline相加获得进程最晚应该停止运行的时间，将这个时间存入消息结构体 `m` 中并通过系统调用 `_syscall()` 传递到服务层。同时，调用 `alarm()` ，使该进程在到达deadline后被强制结束。

In [ ]:
```
#include <lib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/time.h>
```

```
int chrt(long deadline)
{
    struct timeval tv;
    struct timezone tz;
    message m;
    memset(&m,0,sizeof(m));
    unsigned int us_ddl = (unsigned int)deadline;
    alarm(us_ddl);//set alarm
    if(deadline>0)//record present time and calculate deadline
    {
        gettimeofday(&tv,&tz);
        deadline = tv.tv_sec+deadline;
    }
    //save and send deadline to service layer
    m.m2_l1=deadline;
    return(_syscall(PM_PROC_NR,PM_CHRT,&m));
}
```

在 `/usr/src/minix/lib/libc/sys/Makefile.inc` 文件中添加 `chrt.c` 条目。值得注意的是，最好不要使用Windows自带的写字板或者记事本来修改这个文件，否则会破坏这个文件的格式，导致无法编译。

In [ ]:
```
.PATH:  ${NETBSDSRCDIR}/minix/lib/libc/sys

# added
SRCS+=  accept.c access.c adjtime.c bind.c brk.c sbrk.c m_closefrom.c getsid.c \
    chdir.c chmod.c fchmod.c chown.c fchown.c chroot.c close.c \
    clock_getres.c clock_gettime.c clock_settime.c \
    connect.c dup.c dup2.c execve.c fcntl.c flock.c fpathconf.c chrt.c fork.c \
    fstatfs.c fstatvfs.c fsync.c ftruncate.c gcov_flush_sys.c getdents.c \
```

服务层

在服务层中， `do_chrt` 函数将调用 `sys_chrt` 函数，将deadline传递到内核。

在 `/usr/src/minix/servers/pm/proto.h` 中添加 `do_chrt` 函数定义。这个函数的定义被添加在 `proto.h` 文件的最后。

In [ ]:
```
/* utility.c */
pid_t get_free_pid(void);
char *find_param(const char *key);
struct mproc *find_proc(pid_t lpid);
int nice_to_priority(int nice, unsigned *new_q);
int pm_isokendpt(int ep, int *proc);
void tell_vfs(struct mproc *rmp, message *m_ptr);

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

在 `/usr/src/minix/servers/pm/chrt.c` 中添加 `do_chrt` 函数实现，把进程号和消息结构体中的deadline传递给 `sys_chrt()` 并调用 `sys_chrt()` ：

In [ ]:
```
#include "pm.h"
#include <minix/syslib.h>
#include <minix/callnr.h>
#include <sys/wait.h>
#include <minix/com.h>
#include <minix/vm.h>
#include "mproc.h"
#include <sys/ptrace.h>
#include <sys/resource.h>
#include <signal.h>
#include <stdio.h>
#include <minix/sched.h>
#include <assert.h>
int do_chrt()
{
    sys_chrt(who_p,m_in.m2_l1);//get process information and call sys_chrt
    return OK;
}
```

在 `/usr/src/minix/include/minix/callnr.h` 中定义 `PM_CHRT` 编号：

In [ ]:
```
#define PM_GETSYSINFO (PM_BASE + 47)
#define PM_CHRT (PM_BASE + 48)

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

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

In [ ]:
```
.include <bsd.own.mk>

# Makefile for Process Manager (PM)
PROG=   pm
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
```

在 `/usr/src/minix/servers/pm/table.c` 中调用映射表。定义好以后，应用层的 `chrt` 函数调用 `_syscall(PM_PROC_NR,PM_CHRT,&m)` 系统调用后，服务层就会调用 `do_chrt` 函数。

In [ ]:
```
CALL(PM_GETSYSINFO) = do_getsysinfo,     /* getsysinfo(2) */
    CALL(PM_CHRT) = do_chrt
};
```

在 `/usr/src/minix/include/minix/syslib.h` 中添加 `sys_chrt()` 定义：

In [ ]:
```
int copyfd(endpoint_t endpt, int fd, int what);
int sys_chrt(endpoint_t proc_ep,long deadline);
#define COPYFD_FROM 0   /* copy file descriptor from remote process */
```

在 `/usr/src/minix/lib/libsys/sys_chrt.c` 中添加 `sys_chrt()` 实现。这个函数传入进程号和deadline，把它们存入消息结构体 `m` 中并通过 `_kernel_call` 把消息结构体 `m` 传进内

核。

In [ ]:
```
#include "syslib.h"
int sys_chrt(endpoint_t proc_ep,long deadline)
{
    int r;
    message m;
    //save process id and deadline into m
    m.m2_i1=proc_ep;
    m.m2_l1=deadline;
    //conduct kernal call
    r=_kernel_call(SYS_CHRT,&m);
    return r;
}
```

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

In [ ]:
```
SRCS+=  \
    alloc_util.c \
    ...
    sys_clear.c \
    sys_chrt.c \
    sys_cprof.c \
    ...
    vm_update.c
```

内核层

在内核层中，我们将通过 `do_chrt` 函数，用消息结构体中的进程号定位到相应进程，在进程结构体中添加 `deadline` 项目并调用 `chrt` 的进程的 `deadline` 修改为消息结构体中传来的deadline。

在 `/usr/src/minix/kernel/system.h` 中添加 `do_chrt` 函数定义。这个定义被加在文件的最后。

In [ ]:
```
int do_chrt(struct proc * caller, message *m_ptr);
#if ! USE_CHRT
#define do_chrt NULL
#endif

#endif /* SYSTEM_H */
```

在 `/usr/src/minix/kernel/system/do_chrt.c` 中添加 `do_chrt` 函数实现。这个函数通过进程号定位到相应进程，并将该进程结构体中的 `deadline` 条目修改为函数传入的消息结构体中存放的deadline。

In [ ]:
```
#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 exp_time;
    exp_time = m_ptr->m2_l1;
    //find the address of process in kernal
    rp = proc_addr(m_ptr->m2_i1);
    //set the process's deadline
    rp->deadline = exp_time;
    return (OK);
}

#endif /* USE_CHRT */
```

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

In [ ]:
```
SRCS+=  \
    ...
    do_statectl.c \
    do_chrt.c
```

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

In [ ]:
```
#  define SYS_PADCONF (KERNEL_CALL + 57)   /* sys_padconf() */

#  define SYS_CHRT (KERNEL_CALL + 58)
/* 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。定义好以后，服务层的 `sys_chrt` 函数调用的 `_kernel_call(SYS_CHRT,&m)` 会调用内核层的 `do_chrt` 函数，并把消息结构体 `m` 作为 `do_chrt` 函数的输入之一。

In [ ]:
```
/* Process management. */
  ...
  map(SYS_STATECTL, do_statectl);/* let a process control its state */
  map(SYS_CHRT,do_chrt);
```

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

In [ ]:
```
system_tab[]=
{
    ...
    { "CHRT",        SYS_CHRT},
    { NULL,    0 }
};
```

最后，在 `/usr/src/minix/kernel/proc.h` 中定义的进程结构体中添加 `deadline` 条目：

In [ ]:
```
struct proc {
  ...
  long deadline;/* deadline of the process */
  ...
}
```

II.将MINIX中的进程调度算法修改为EDF算法

该部分旨在将MINIX操作系统中的进程调度算法从时间片轮换改为EDF算法。所有的修改均在 `/usr/src/minix/kernel/proc.c` 中进行。

MINIX操作系统采用多级调度算法。所有进程按照其优先级被放进十六个进程队列，队列号越小，进程优先级越高。操作系统总是优先执行优先级较高的进程；在队列内部则采用时间片轮换。我们希望我们设置了deadline的进程优先级比较高，否则测试实验结果时效果不好。又由于第零层到第四层都是系统进程，所以我们把调用了 `chrt` 的进程全部放在第五个进程队列里。所以，在 `proc.c` 中按照优先级向队首队尾加进程的两个函数 `enqueue_head()` 和 `enqueue()` 中，我们需要将所有调用过 `chrt` 的实时进程放进第五个进程队列：

In [ ]:
```
/*===========================================================================*
 *                        enqueue                                    *
 *===========================================================================*/
void enqueue(
  register struct proc *rp/* this process is now runnable */
)
{
/* Add 'rp' to one of the queues of runnable processes.  This function is
 * responsible for inserting a process into one of the scheduling queues.
 * The mechanism is implemented here.   The actual scheduling policy is
 * defined in sched() and pick_proc().
 *
 * This function can be used x-cpu as it always uses the queues of the cpu the
 * process is assigned to.
 */
  if (rp->deadline>0)
  {
    rp->p_priority=5;
  }

  ...
}
```

对 `enqueue_head()` 的修改也是一样的。

下面，我们来着重看一看 `pick_proc()` 函数。它从队列中返回一个可调度的进程。

In [ ]:
```
/*===========================================================================*
 *                        pick_proc                                  *
 *===========================================================================*/
static struct proc * pick_proc(void)
```

```
{
/* Decide who to run now.  A new process is selected an returned.
 * When a billable process is selected, record it in 'bill_ptr', so that the
 * clock task can tell who to bill for system time.
 *
 * This function always uses the run queues of the local cpu!
 */
  register struct proc *rp;/* process to run */
  register struct proc *next_process;/* process to run */
  struct proc **rdy_head;
  int q;/* iterate over queues */


  /* Check each of the scheduling queues for ready processes. The number of
   * queues is defined in proc.h, and priorities are set in the task table.
   * If there are no processes ready to run, return NULL.
   */
  rdy_head = get_cpulocal_var(run_q_head);
  for (q=0; q < NR_SCHED_QUEUES; q++) {
    if(!(rp = rdy_head[q])) {
        TRACE(VF_PICKPROC, printf("cpu %d queue %d empty\n", cpuid, q););
        continue;
    }
    //if we've set a deadline for this process
    if (q==5)
    {
        rp = rdy_head[q];
        next_process = rp->p_nextready;
        //search the whole queue for the process with the earliest deadline
        while(next_process!=NULL)
        {
            if(next_process->deadline>0)
            {
                if(rp->deadline==0||(rp->deadline>next_process->deadline))
                {
                    if(proc_is_runnable(next_process))
                    {
                        rp=next_process;
                    }
                }
            }
            next_process = next_process->p_nextready;
        }
    }
    assert(proc_is_runnable(rp));
    if (priv(rp)->s_flags & BILLABLE)
        get_cpulocal_var(bill_ptr) = rp; /* bill for system time */
    return rp;
  }
  return NULL;
}
```

这里， `rp` 就是那个会被 `pick_proc()` 函数返回的、将要被调度的进程。我们只关心第五个进程列表中的调度算法。首先， `rp` 被赋值为第五个进程列表首部的进程。然后，进程结构体变量 `next_process` 通过 `while` 循环遍历整个进程列表。在每一次迭代中，如果

`next_process` 的deadline比 `rp` 的更早（也就是说， `next_process` 优先级更高），那么 `rp` 的值就会被赋为 `next_process` 。当整个进程列表都被遍历完以后， `rp` 一定是整个进程列表中deadline最早的进程。它将会被 `pick_proc()` 返回，并被操作系统调度。

实验结果

在运行了 `test_code.c` 中的测试代码以后，结果如下所示：

In [ ]:
```
# ./test_lab2
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
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
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
#
```

在一开始，进程一、进程二的deadline分别被设置成开始时刻$+25$、开始时刻$+15$，进程三不是实时进程，它只是普通进程。所以，三个进程的优先级从大到小应该是进程二、进程一、进程三。第五秒时进程一调用 `chrt(5)` ，其deadline被修改为第五秒的时刻再加五秒钟（也就是开始时刻$+10$）。所以，这时三个进程的优先级从大到小应该是进程一、进程二、进程

三。第十秒时，进程一因为到达其deadline而被强制结束，而进程三调用了`chrt(3)`，其deadline被修改为第十秒的时刻再加三秒钟（也就是开始时刻+13）。所以，从第十秒开始，仅剩的两个进程的优先级从大到小应该是进程三、进程二。所以，上面的运行结果应该是正确的。