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：** 2021 级	**上机实践成绩：**
指导教师： 翁楚良	**姓名：** 杨茜雅	**学号：** 10215501435
上机实践名称： EDF 近似实时调度		**上机实践日期：** 2023.4
上机实践编号： project2	**组号：**	**上机实践时间：** 2023.4

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. 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。

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

物理机：Windows10
虚拟机：Minix3
虚拟机软件：Vmware
代码编辑：VScode
物理机与虚拟机文件传输：FileZilla

四、 注意事项

1. MINIX 的不同服务模块和内核都是运行在不同进程中，只能使用基于消息的进程间系统调用/内核调用，不能使用直接调用普通 C 函数。
2. 添加调用编号，需要修改取值范围限制。
3. 以源码为准（博客等资料版本落后）。
4. 善用 source insight 高级功能（调用关系，全局搜索）。
5. 善用 git diff 检查代码修改。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。
6. 善用 FileZilla 功能。连接虚拟机，拉取需修改的文件，修改后上传到虚拟机。

五、 实验过程

1、 虚拟机中下载 minix3.3.0 源码

```
cd /usr
git clone git://git.minix3.org/minix src #联机下载代码
cd src#下载的代码里有 src 文件夹
git branch -a # 查看代码版本
git checkout R3.3.0 # 将代码版本切换为 3.3.0
```

得到虚拟机上有以下文件

文件名	文件大小	文件类型	最近修改	权限
..				
.git		文件夹	2023-04-1...	drwxr
bin		文件夹	2023-04-1...	drwxr
common		文件夹	2023-04-1...	drwxr
distrib		文件夹	2023-04-1...	drwxr
docs		文件夹	2023-04-1...	drwxr
etc		文件夹	2023-04-1...	drwxr
external		文件夹	2023-04-1...	drwxr
games		文件夹	2023-04-1...	drwxr
gnu		文件夹	2023-04-1...	drwxr
include		文件夹	2023-04-1...	drwxr
lib		文件夹	2023-04-1...	drwxr
libexec		文件夹	2023-04-1...	drwxr
minix		文件夹	2023-04-1...	drwxr
releasetools		文件夹	2023-04-1...	drwxr
sbin		文件夹	2023-04-1...	drwxr
share		文件夹	2023-04-1...	drwxr
sys		文件夹	2023-04-1...	drwxr
tests		文件夹	2023-04-1...	drwxr
tools		文件夹	2023-04-1...	drwxr
usr.bin		文件夹	2023-04-1...	drwxr
usr.sbin		文件夹	2023-04-1...	drwxr
.gitignore	428	文本文档	2023-04-1...	-rw-r-
.gitreview	547	GITREVI...	2023-04-1...	-rw-r-
build.sh	62,484	Shell Sc...	2023-04-1...	-rwxr
change.txt	9,233	文本文档	2023-04-1...	-rw-r-
LICENSE	2,669	文件	2023-04-1...	-rw-r-
Makefile	16,950	文件	2023-04-1...	-rw-r-
Makefile.inc	352	Include...	2023-04-1...	-rw-r-
params	3,013	文件	2023-04-1...	-rw-r-

2、 编译源码并安装

cd/usr/src #进入目标文件夹

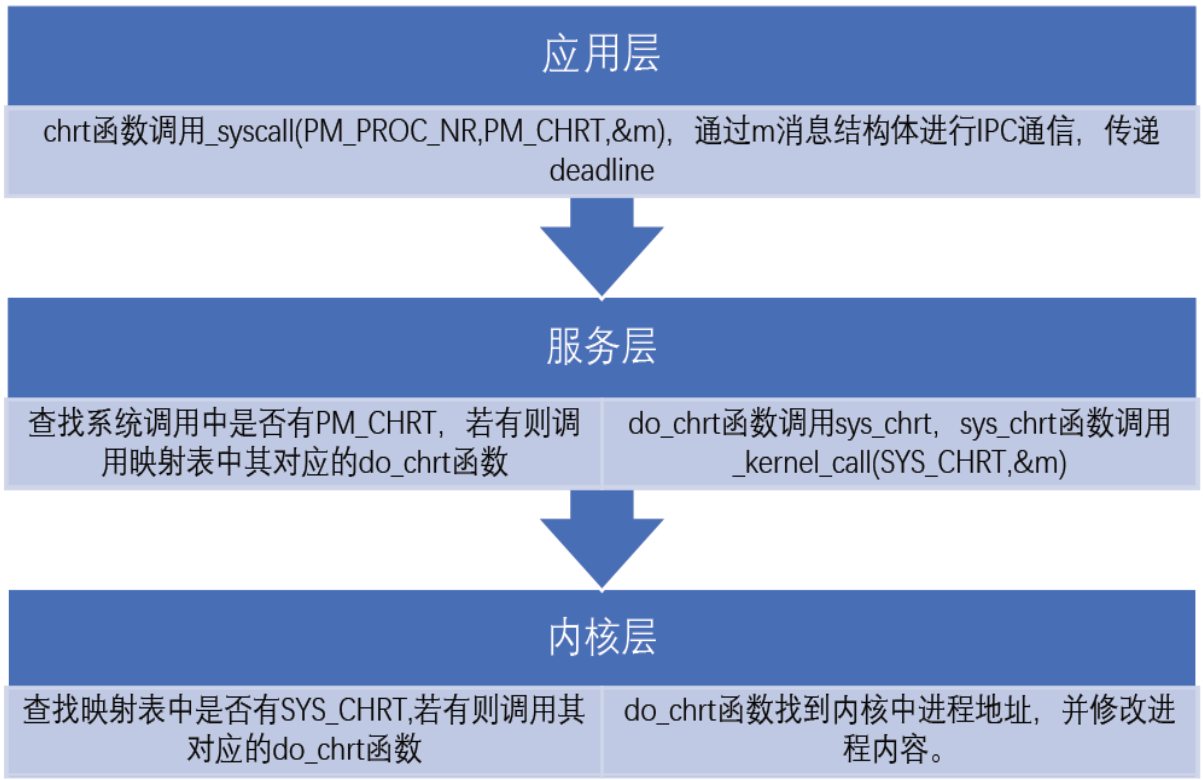
make build #首次编译，或者修改了头文件，Makefile时使用，时间较长

Make build MKUPDATE=yes #增量式编译，适用于少量C源代码修改时使用。

reboot #重启，默认情况下自动选择latest kernel（新生成的kernel），需要原始版本时手工选择。

3、 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。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。



应用层有一个chrt.c文件，调用_syscall,通过消息结构体进行IPC通信，传递deadline到服务层；到了服务层，先查找是否有PM_CHRT,有的话则在映射表中调用它对应的do_chrt函数，do_chrt函数调用sys_chrt，sys_chrt再调用kernel_call将结构体和deadline传到内核层；到了内核层，先查映射表中是否有sys_chrt，如果有，则调用对应的do_chrt函数，do_chrt函数去内核中找到进程的地址，再修改它的deadline。

（1） 应用层（3步）

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

```
#endif /* __CUSERID_DECLARED */
int chrt(long );//添加chrt 函数定义
```

• 在/usr/src/minix/lib/libc/sys/chrt.c中添加chrt函数实现。Chrt的功能主要是将参数deadline添加到message信息中，然后利用_syscall()函数信息结构体进行IPC通信以传递deadine。可用alarm函数实现超时强制终止。可参考该文件夹下的fork.c文件，在实现中通过_syscall(调用号)向系统服务传递。

Fork.c代码：

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

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

#ifdef __weak_alias
__weak_alias(fork, _fork)
#endif

pid_t fork(void)
{
  message m;

  memset(&m, 0, sizeof(m));
  return(_syscall(PM_PROC_NR, PM_FORK, &m));
}
```

Chrt.c代码

C: > Users > 86138 > Documents > 操作系统 > project 2 > code > 应用层 > minix > lib > libc

```c
#include <lib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include<sys/time.h>

int chrt(long deadline){
//struct timespec time={0,0};
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));
}
```

注意：

1、因为message中应该包含进程结束时的时间，所以还需要获取到现在系统上的时间加上设置的deadline才可以得到message中应该传输的时间。系统时间可以通过time.tv_sec得到。

2、m.m2_l1存放deadline，deadline是一个long型数据，根据书中P98页的图可以看出数据存放的顺序是先int型数据，其次是long型数据，因此我们选择m2_l2存放deadline。

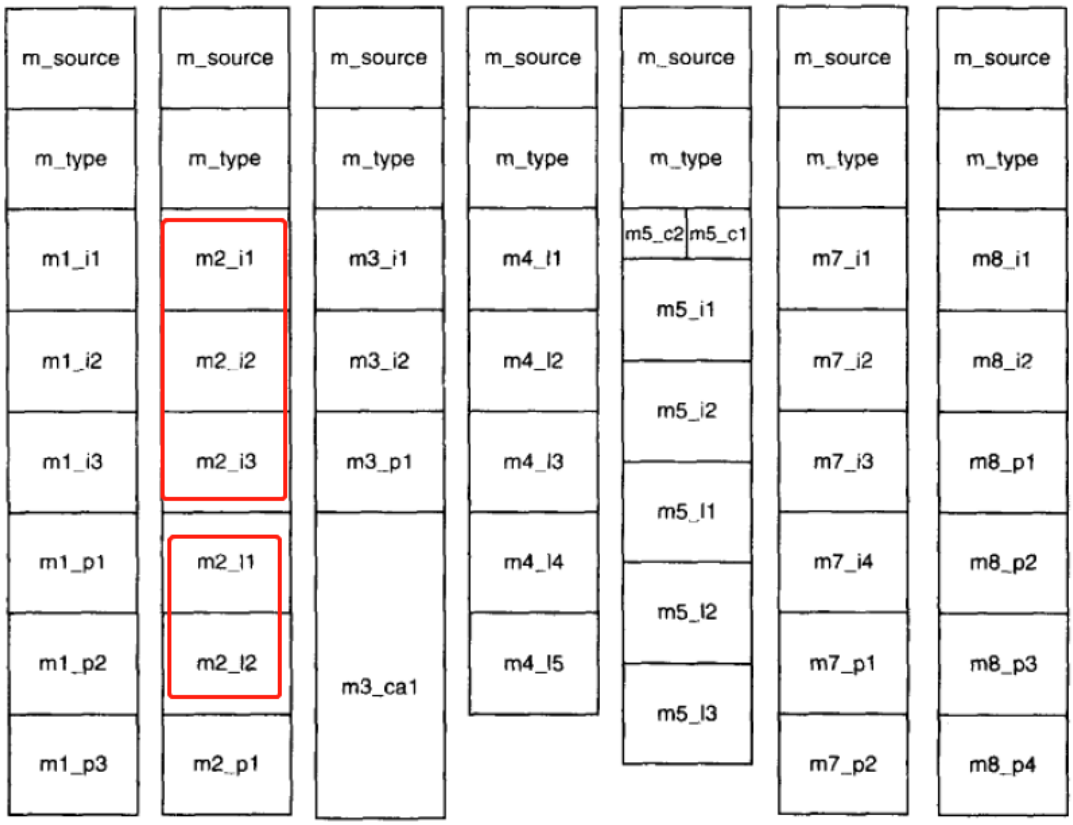


图 2.34　MINIX 3中所用的七种消息。消息大小随机器体系结构的不同会有所不同。本图展示的是类似奔腾系列的拥有 32 位指针的 CPU 的情况

• 在/usr/src/minix/lib/libc/sys中Makefile.inc文件添加chrt.c条目，（因为添加c文件后，需要在同目录下的Makefile/Makefile.inc中添加条目）。

```
symlink.c \
sync.c syscall.c sysuname.c truncate.c umask.c unlink.c write.c \
utimensat.c utimes.c futimes.c lutimes.c futimens.c \
_exit.c _ucontext.c environ.c __getcwd.c vfork.c sizeup.c init.c \
getrusage.c setrlimit.c setpgid.c chrt.c
```

（2） 服务层（八步）

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

```
int do_srv_fork(void);
int do_chrt(void);//添加chrt 函数定义
int do_exit(void);
```

这里虽然说是chrt函数的定义，但是这里定义的其实是服务层对chrt函数的处理,向内核层发送message，而不是应用层中用户使用的chrt函数。

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

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

Sys_chrt()的参数来自同文件下的glo.h

```
EXTERN message m_in;        /* the incoming message itself is kept here. */
EXTERN int who_p, who_e;    /* caller's proc number, endpoint */
```

其中，m_in是结构体message，who_p是进程的进程号

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

```
#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

```
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

```
CALL(PM_EXIT)       = do_exit,      /* _exit(2) */
CALL(PM_FORK)       = do_fork,      /* fork(2) */
CALL(PM_CHRT)       = do_chrt, //参照fork、exit 定义进行添加。
```

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

```
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。可参考该文件夹下的 sys_fork 文件，在实现中通过 _kernal_call（调用号）向内核传递。

Sys_fork:

```
#include "syslib.h"

int sys_fork(parent, child, child_endpoint, flags, msgaddr)
endpoint_t parent;    /* process doing the fork */
endpoint_t child;   /* which proc has been created by the fork */
endpoint_t *child_endpoint;
u32_t flags;
vir_bytes *msgaddr;
{
/* A process has forked.  Tell the kernel. */

  message m;
  int r;

  m.m_lsys_krn_sys_fork.endpt = parent;
  m.m_lsys_krn_sys_fork.slot = child;
  m.m_lsys_krn_sys_fork.flags = flags;
  r = _kernel_call(SYS_FORK, &m);
  *child_endpoint = m.m_krn_lsys_sys_fork.endpt;
  *msgaddr = m.m_krn_lsys_sys_fork.msgaddr;
  return r;
}
```

Sys_chrt()实现:

```
#include"syslib.h"
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;
}
//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。
```

- 在/usr/src/minix/lib/libc/sys 中 Makefile.inc 文件添加 chrt.c 条目

```
sef_signal.c \
sqrt_approx.c \
srv_fork.c \
srv_kill.c \
stacktrace.c \
sys_abort.c \
sys_clear.c \
sys_chrt.c \
sys_cprof.c \
sys_diagctl.c \
sys_endsig.c \
sys_exec.c \
sys_exit.c \
```

（3） 内核层（六步）

在MINIX内核中实现进程调度功能，此处可以直接修改内核信息，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

```
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
```

- 在/usr/src/minix/kernel/system/do_chrt.c 中添加 do_chrt 函数实现。

```c
#include "kernel/system.h"
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <lib.h>
#include <minix/endpoint.h>
#include <string.h>

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);
}
```

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

```
do_protbuf.c \
do_vmctl.c \
do_mcontext.c \
do_schedule.c \
do_schedctl.c \
do_statectl.c \
do_chrt.c

.if ${MACHINE_ARCH} == "i386"
SRCS+= \
```

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

```
#  define SYS_SCHEDCTL (KERNEL_CALL + 54)   /* sys_schedctl() */
#  define SYS_STATECTL (KERNEL_CALL + 55)   /* sys_statectl() */

#  define SYS_SAFEMEMSET (KERNEL_CALL + 56) /* sys_safememset() */

#  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

```
map(SYS_FORK, do_fork);      /* a process forked a new process */
map(SYS_EXEC, do_exec);    /* update process after execute */
map(SYS_CHRT,do_chrt);    //参考do_fork、do_exec 按照相应格式添加。
```

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

```
844     { "READBIOS",       SYS_READBIOS },
845     { "STIME",      SYS_STIME },
846     { "VMCTL",      SYS_VMCTL },
847     { "MEMSET",     SYS_MEMSET },
848     { "PADCONF",        SYS_PADCONF },
849     { "CHRT", SYS_CHRT },
850     { NULL,     0 }
851 };
```

4、 MINIX3中的进程调度

进程调度模块位于/usr/src/minix/kernel/下的proc.h和proc.c，修改影响进程调度顺序的部分。

- struct proc 维护每个进程的信息，用于调度决策。添加 deadline 成员。修改/usr/src/minix/kernel/下的 proc.h

```
struct proc {
  struct stackframe_s p_reg;    /* process' registers saved in stack frame */
  struct segframe p_seg;    /* segment descriptors */
  proc_nr_t p_nr;       /* number of this process (for fast access) */
  struct priv *p_priv;      /* system privileges structure */
  volatile u32_t p_rts_flags;   /* process is runnable only if zero */
  volatile u32_t p_misc_flags;  /* flags that do not suspend the process */

  char p_priority;      /* current process priority */
  u64_t p_cpu_time_left;    /* time left to use the cpu */
  unsigned p_quantum_size_ms;   /* assigned time quantum in ms
                   FIXME remove this */
  struct proc *p_scheduler; /* who should get out of quantum msg */
  unsigned p_cpu;       /* what CPU is the process running on */
#ifdef CONFIG_SMP
  bitchunk_t p_cpu_mask[BITMAP_CHUNKS(CONFIG_MAX_CPUS)]; /* what CPUs is the
                                process allowed to
                                run on */
  bitchunk_t p_stale_tlb[BITMAP_CHUNKS(CONFIG_MAX_CPUS)]; /* On which cpu are
                possibly stale entries from this process and has
                to be fresed the next kernel touches this
                processes memory
                 */
#endif

  /* Accounting statistics that get passed to the process' scheduler */
  struct {
    u64_t enter_queue;  /* time when enqueued (cycles) */
    u64_t time_in_queue;    /* time spent in queue */
    unsigned long dequeues;
    unsigned long ipc_sync;
    unsigned long ipc_async;
    unsigned long preempted;
  } p_accounting;

  clock_t p_user_time;      /* user time in ticks */
  clock_t p_sys_time;       /* sys time in ticks */

  clock_t p_virt_left;      /* number of ticks left on virtual timer */
  clock_t p_prof_left;      /* number of ticks left on profile timer */
  long long p_deadline;     //设置deadline
  u64_t p_cycles;       /* how many cycles did the process use */
```

- switch_to_user() 选择进程进行切换

```
/*===========================================================================*
 *                switch_to_user                                             *
 *===========================================================================*/
void switch_to_user(void)
{
    /* This function is called an instant before proc_ptr is
     * to be scheduled again.
     */
    struct proc * p;
#ifdef CONFIG_SMP
    int tlb_must_refresh = 0;
#endif
```

- enqueue_head() 按优先级将进程加入列队首。实验中需要将实时进程的优先级设置成合适的优先级

```
/*===========================================================================*
 *              enqueue_head                                                 *
 *===========================================================================*/
/*
 * put a process at the front of its run queue. It comes handy when a process is
 * preempted and removed from run queue to not to have a currently not-runnable
 * process on a run queue. We have to put this process back at the fron to be
 * fair
 */
static void enqueue_head(struct proc *rp)
{
  const int q = rp->p_priority;         /* scheduling queue to use */

  struct proc **rdy_head, **rdy_tail;

  assert(proc_ptr_ok(rp));
  assert(proc_is_runnable(rp));

  /*
   * the process was runnable without its quantum expired when dequeued. A
   * process with no time left should have been handled else and differently
   */
  assert(rp->p_cpu_time_left);

  assert(q >= 0);


  rdy_head = get_cpu_var(rp->p_cpu, run_q_head);
  rdy_tail = get_cpu_var(rp->p_cpu, run_q_tail);

  /* Now add the process to the queue. */
  if (!rdy_head[q]) {       /* add to empty queue */
    rdy_head[q] = rdy_tail[q] = rp;     /* create a new queue */
    rp->p_nextready = NULL;         /* mark new end */
  } else {                  /* add to head of queue */
    rp->p_nextready = rdy_head[q];      /* chain head of queue */
    rdy_head[q] = rp;           /* set new queue head */
  }

  /* Make note of when this process was added to queue */
  read_tsc_64(&(get_cpulocal_var(proc_ptr->p_accounting.enter_queue)));


  /* Process accounting for scheduling */
  rp->p_accounting.dequeues--;
  rp->p_accounting.preempted++;

#if DEBUG_SANITYCHECKS
  assert(runqueues_ok_local());
#endif
}
```

- enqueue()按优先级将进程加入列队尾

```
/*===========================================================================*
 *				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->p_deadline >0)
  {
   rp->p_priority = 6;
  }
  int q = rp->p_priority;		   /* scheduling queue to use */
  struct proc **rdy_head, **rdy_tail;

  assert(proc_is_runnable(rp));

  assert(q >= 0);

  rdy_head = get_cpu_var(rp->p_cpu, run_q_head);
  rdy_tail = get_cpu_var(rp->p_cpu, run_q_tail);

  /* Now add the process to the queue. */
  if (!rdy_head[q]) {		/* add to empty queue */
      rdy_head[q] = rdy_tail[q] = rp;		/* create a new queue */
      rp->p_nextready = NULL;		/* mark new end */
  }
  else {					/* add to tail of queue */
      rdy_tail[q]->p_nextready = rp;		/* chain tail of queue */
      rdy_tail[q] = rp;			/* set new queue tail */
      rp->p_nextready = NULL;		/* mark new end */
  }

  if (cpuid == rp->p_cpu) {
      /*
       * enqueueing a process with a higher priority than the current one,
       * it gets preempted. The current process must be preemptible. Testing
       * the priority also makes sure that a process does not preempt itself
       */
      struct proc * p;
      p = get_cpulocal_var(proc_ptr);
      assert(p);
      if((p->p_priority > rp->p_priority) &&
              (priv(p)->s_flags & PREEMPTIBLE))
          RTS_SET(p, RTS_PREEMPTED); /* calls dequeue() */
  }
#ifdef CONFIG_SMP
  /*
   * if the process was enqueued on a different cpu and the cpu is idle, i.e.
   * the time is off, we need to wake up that cpu and let it schedule this new
   * process
   */
  else if (get_cpu_var(rp->p_cpu, cpu_is_idle)) {
      smp_schedule(rp->p_cpu);
  }
#endif

  /* Make note of when this process was added to queue */
  read_tsc_64(&(get_cpulocal_var(proc_ptr)->p_accounting.enter_queue));


#if DEBUG_SANITYCHECKS
  assert(runqueues_ok_local());
#endif
}
```

将优先级设置为6，在MINIX3中，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。这里经过尝试，设为5和6都可以。具体原因参照课本P124页的图：

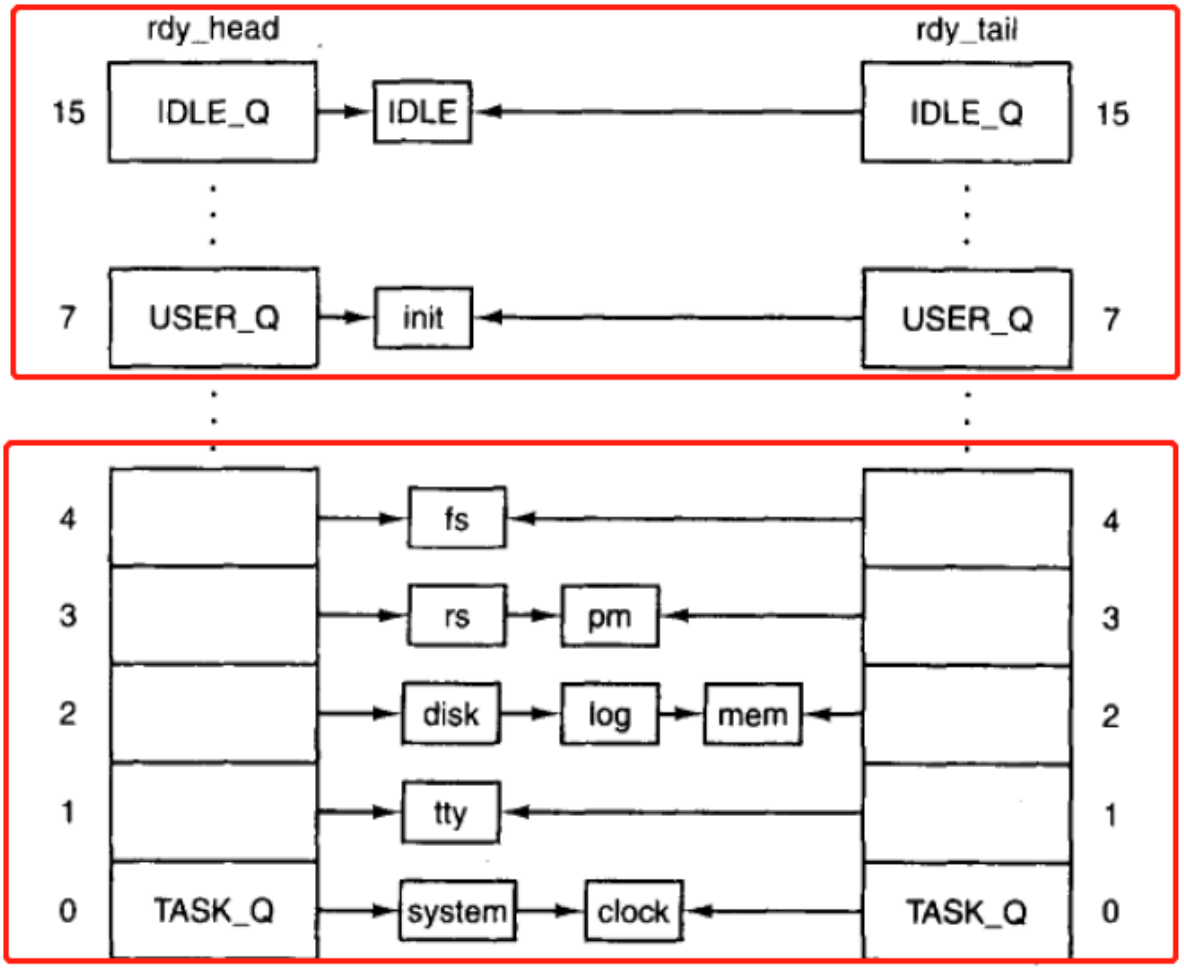


图 2.43　调度器维护16个队列，每个队列具有一个优先级，图中示出了 MINIX 3 启动后的初始进程队列情况

可以看到优先级1-4是系统的进程，7-15是系统的进程，所以我们实验中设置的优先级只能是5或6。

- 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

```
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;
   register struct proc * temp;     /* 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++) {
    //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;
        }
    }
    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;
}
```

六、 实验结果

修改 test_code 代码

```
43      case 3: //子进程3, 普通进程
44        chrt(0);
45        printf("proc3 set success\n");
46        sleep(1);
47        break;
48      }
49      for (loop = 1; loop < 40; loop++)
```

编译运行

```
# ls
.exrc         hello         shell.c       test_code.c   text.txt
.profile      hello.c       shell.o       test_code.o   your
# clang test_code.c -o test_code.o
# ./test_code.o
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
#
```

结果解释：

在 test_code 代码中

```
case 1: //子进程1，设置deadline=25
    chrt(25);
    printf("proc1 set success\n");
    sleep(1);
    break;
case 2: //子进程2，设置deadline=15
    chrt(15);
    printf("proc2 set success\n");
    sleep(1);
    break;
case 3: //子进程3，普通进程
    chrt(0);
    printf("proc3 set success\n");
    sleep(1);
    break;
}
```

我们设置的子进程 1 deadline 为 25，子进程 2 deadline 为 15，子进程 3 为普通进程，所以刚开始时，优先级为 P2>P1>P3。后来我们将 P1 的 deadline 设置为 5，所以 P1 的优先级提高，全局优先级排序变为 P1>P2>P3，最后进程 1 消失是因为它的 deadline 已经到了；之后我们将 P3 的 deadline 设置为 3，它的优先级提前，全局优先级排序变为 P3>P2，最后进程 3 消失是因为它的 deadline 已经到了。

七、 实验错误原因

实验编译的结果总是出错，但是在 VWware WROKstation 虚拟机里总是看不到报错信息，所以连接 ssh 查看报错信息，发现原因是在所有定义编号的行为中，每次定义完，下一行的编号也要更新。

```
#define PM_GETSYSINFO       (PM_BASE + 47)
#define PM_CHRT (PM_BASE + 48)

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

/*===========================================================================*
```

```
#  define SYS_SCHEDCTL (KERNEL_CALL + 54)   /* sys_schedctl() */
#  define SYS_STATECTL (KERNEL_CALL + 55)   /* sys_statectl() */

#  define SYS_SAFEMEMSET (KERNEL_CALL + 56) /* sys_safememset() */

#  define SYS_PADCONF (KERNEL_CALL + 57)    /* sys_padconf() */

# define SYS_CHRT (KERNEL_CALL + 58)

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

八、 实验总结

通过本次实验，我了解了 MINIX 操作系统进程管理的相关知识，熟悉其系统调用在三个层结构上的实现顺序和调度器的实现，实现自己的近似实时调度算法，它其实类似于书上最短剩余时间优先算法。