# UNIX 的实时改造策略初探

施广宇 （大连理工大学自动化系 116023）

**摘要**：本文根据 AT&T UNIX System V Release 4.0 及后续的一些与其兼容的常见 UNIX 变种的特点，给出了基于 IPC 消息队列的对 UNIX 操作系统进行实时化改造的具体策略。

**关键词**：UNIX 实时 IPC 消息队列

## 1. 问题的提出

计算机技术的飞速发展，对于自动控制学科意味着巨大的机遇和挑战，作为计算机控制的关键，则在于控制软件的研制。目前国内的研究主要集中在建模方法和算法等应用软件的具体实现领域，关于应用软件的基础平台—如操作系统，针对控制工程的需要而进行的研究工作则相对较少，已有的一些报道也主要集中在基于 PC 的 DOS/WINDOWS 系列[2]。

作为开放系统的象征，UNIX 众多的优秀性能是不争的事实。但由于 UNIX 的分时特性，使得工程控制领域（特别在国内）在一些对实时性要求较高的场合一直对它“忍痛割爱”，所以如果能够对 UNIX 进行实时化改造，这将是一种很有益的方案。

## 2. UNIX 实时改造的目标与基础

实时性是指系统能在可以接受的时间限内对外部事件作出恰当的响应。对于大多数大型复杂控制工程（如大型 DCS，电力系统控制中心，机场空中交通管理中心）的上位机管理软件来说，如果能得到百分秒级的进程调度就足以满足实时方面的要求。

在传统的标准 UNIX 中，进程的定时激活 alarm()，进程睡眠 sleep()，系统时间获取 time()这样一些对于进程调度机制而言至关重要的系统调用的时间操作单位均是整秒，所以就不能精确控制进程的状态，以前这一直被认为是利用 UNIX 开发控制软件的障碍。但由于今天一些 UNIX 平台（如 HPUX 7.0 forHP9000－700，SUN OS 4.1/Solaris for SUN，IRIX 5.0 for SGI，At&T UNIX System V Release 4.0 及它们的后续版本）已经提供了一些实时/类实时的功能，利用这些功能就可以对 UNIX 进行实时化改造以适应控制系统的需要。由于 At&T UNIX System V Release 4.0 在 UNIX 发展史上的突出地位以及许多常见 UNIX 后续变种都宣称与其兼容，本文的讨论将主要基于 AT&T UNIX SV R4。

在详细研究 At&T UNIX SV R4 提供的让人眼花缭乱的众多系统调用和函数后，可以发现其中有一些对于实时改造至关重要，正是基于这些新的系统调用和函数以及 IPC（InterProcess Communication）消息通信机构，才有可能编程实现 UNIX 百分秒级的进程调用机制。

## 3. UNIX 的实时改造策略

结合作者在清华大学仿真工程研究所参与研制的我国第二代开放式电网仿真系统（Power System Simulator，即 PSS）的经验，对 UNIX 的实时改造主要应包括以下几个方面：

（1）设定实时进程。从 AT&T UNIX SV R4 开始，系统支持实时（RT）和分时共享（TS）两类进程，可以通过如下的系统调用来动态设置某个或某一组进程的为实时类别以及与其相关的其他调度参数：

priocntl（idtype-t idtype，id-t id，PC-SETPARMS，pcarms-targ）；

其中 idtype 和 id 联用决定设置哪一个（组）进程，arg－＞pc-pid 为实时进程类别标识号，arg－＞pc-clparms 为调度参数，包括实时优先级和分配给该实时类进程的时间片长度。系统对实时类提供了按优先级的抢占式调度方法。如果能使整个系统的进程都工作在实时方式下，那么立刻就万事大吉，但可惜的是由于 UNIX 分时内核的限制，使得许多重要进程（如 init）不能够或不适合工作在实时方式下，另一方面，可运行的实时进程总要先于其他任何进程运行，因此 UNIX 文档指出，不适当的使用实时进程会给系统性能造成极大的负影响。所以在实践中，只对那些特别需要快速和确定的响应时间或对调度优先级有绝对控制要求的进程才将其设为实时进程，对其余的绝大多数进程则采用下述的调度方法来满足实时要求。

(2)微秒或毫秒级系统时间的获取。在标准 UNIX 中,只能利用 time()获得秒级的系统时间,SUN OS 中的 ftime()和 gettimeofday()分别可以得到毫秒级和微秒级的系统时间,IRIX 和 At&T UNIX 中只有 gettimeofday(),当然使用 gettimeofday()可以很方便地构造出 ftime()来,gettimeofday()函数调用方式如下:

```
#include<sys/time.h>
int gettimeofday(struct timeval *tp);
```

函数返回后,tp->tx-sec 中存放自 1970/01/01 以来经过的秒数,tp->tv-usec 中存放微秒数。

(3)微秒或毫秒级别进程睡眠的实现。在 SUN OS 中,usleep()可得到微秒级的进程睡眠,在 SCO UNIX 中的 nap()也有这样的功能,其他大部分的 UNIX(如 IPIX for SGI, HPUX for HP, AT&T UNIX)中尚未见到类似功能,这种情况下要使用 setitimer()或其他类似功能的函数来实现,AT&T UNIX 中为 setitimer(),,setitimer()可以在微秒级的指定时间向进程发出一个 SIGALRM 或 SIGVTALRM 信号,使进程结束睡眠恢复正常运行。下面给出微秒级的进程睡眠函数 usleep()中的关键代码:

```
#include<sys/time.h>
#include<math.h>
int usleep(usec)
int usec;      /*usec:进程要睡眠的微秒数*/
{
  struct timeval tp;
  struct itimerval value;
  ...
  gettimeofday(&ty);/*获取系统微秒级时间*/
  tp.tv-usec+=usec;/*加上偏移量*/
  if(tp.tv-usec>1000000){
    tp.tv-sec+=tp.tv-usec/1000000;
    tp.tv-usec-=mod(tp.tv-usec,1000000);
  }
  value.it-interval=tp;
  setitimer(ITIMER-REAL,&value,NULL);/*设置微秒级定时器*/
  pause();
  ...
}
```

说明:UNIX 系统为每个调用 setitimer()的进程保留三个微秒级别的定时器,用 ITIMER—REAL 作为第一个参数指定使用按系统实际时间缩减的定时器,到时向该调用进程发送 SIGALRM 信号,从而唤醒因调用 pause()而睡眠的进程。

(4)进程的实时调度。UNIX 不提供秒级以下调度进程的直接的系统调用,所以必须自己编程来实现。在第二代 PSS 对 UNIX 的实时改造部分中,采用通过编程实现的 usleep()进程睡眠),schwk()(进程百分秒级的定时激活),canwk()(结束进程)来实现进程的动态实时调度,schwk()是其中的关键。

实现的基础是 UNIX 的 IPC(InterProcess Communication)消息队列,巧妙地利用函数 msgrcv()具有使进程睡眠直到接收特定类型消息的功能。在 schwk()函数中,调用进程(客户)先用 msgsnd()向消息队列中发出一个包含定时激活的内容的消息,其中包括进程号(PID)和微秒或毫秒级的唤醒时间,然后使用该客户进程的进程号(PID)作为确定要接收消息的类型去调用 msgrcv(),从而进入睡眠。当后台服务进程"schwkd"收到该消息后,就产生一个定时子进程,这个子进程用 3.2 中的方法睡眠到指定时间后,使用要激活客户进程的 PID 作为消息类型发出一个消息,这个消息激活因调用 msgrcv()而挂起的进程,当函数调用结束后可以从消息中得到激活者的 PID 等消息。下面给出 schwk()和 schwkd 的关键实现代码:

①头文件 schwk.h,其中定义了对应于客户向服务者发出要求定时激活的消息的键值(key),消息类型和消息格式

```
#define MSGSCHKEY 100
#define MSGSCHTYPE 100
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
struct msgschform{
  long mtype;      /*消息类型*/
  char mtext[256];/*消息正文*/
}
...
```

②客户 schwk(),其中调用自定义函数 SetMessageValue(),功能为将消息类型 MSGSCHTYPE,进程号 pid 和定时激活时间 usec 等信息填入结构 msg 中相应的域中,返回 msg 的实际长度。

```
#include"schwk.h"
```

```
    ...
    schwk(usec)
    int usec;/ * 微秒级定时激活的时间 * /
    {
      struct msgschform msg;
      int msgsize;
...
      msgid = msgget(MSGSCHKEY,0777);
      pid = getpid();
        msgsize = SetMessageValue (&msg, MSGSCHTYPE, pid, usec);
      msgsnd(msgid, &msg, msgsize, 0);/ * 向服务者发送定时激活请求消息 * /
      msgrcv(msgid, &msg, 256, pid, 0);/ * 用进号作为要接收信号的类型,进入睡眠 * /
      ...
    }
```

③后台服务进程 schwkd，其中调用自定义函数 GetMessageValue()，功能为从结构 msg 中相应的域中析出客户进程号 pid 和定时激活时间 usec 等信息。

```
    #include"schwk.h"
    main()
    {
      struct msgschform msg, msg1;
      int msgsize, msgid, cpid, usec, spid, cpid1, usec1;
    ...
       msgid = msgget (MSGSCHKEY, 0777 | IPCCREDT);
      for(;;){
        msgrcv(msgid, &msg, 256, MSGSCHTYPE, 0);
        GetMessageValue(&msg, &cpid, &usec);
        ...
        spid = fork();
        if(spid = =0){ / * 定时子进程 * /
        cpid1 = cpid;
        usec1 = usec;
          usleep(usec1);
        / * 用客户进程号作为要接收信号的类型 * /
        msgsize = SetMessageValue(&msg1, cpid1, spid, 0);
          msgsnd(msgid, &msg, msgsize, 0);/ * 向客户发送消息,激活客户进程 * /
        ...
        }
        ...
      }
    }
```

另外值得指出的是，XtToolKit 库中的 XtAppAddTime()函数可以实现对调用进程的毫秒级别的实时调度，常用于基于 X/Motif 的 MMI 中。但在另一方面，X/Motif 占用系统资源较多，基于 X 的进程较多时，系统性能将有很大下降。

(5)进程的产生。由于 UNIX 只能使用 fork()产生进程，而 fork()机制是将父进程完整地复制一份从而产生子进程，故当父进程很大而子进程只需做很小一件工作时就会产生内存浪费，所以在程序设计时应把子进程必须实现的功能放在一个单独的函数执行文件中，由该子进程再产生子进程浪费的内存空间要少，用 exec 覆盖子进程空间，也可以节省内存。另外的途径是把子进程要作的各种工作汇集成一个后台的功能代理来统一完成，这也是(4)中为什么采用后台服务进程"schwkd"的原因。

(6)进程的结束。UNIX 进程自然结束(调用 exit())时，就进入称为"Zombie"的状态，这时它虽然已经释放了内存，但这个进程本身还在内存中(占据进程表页中的一项)，这种死进程多了必然造成系统性能下降进而影响它的实时性，避免这种情况的唯一途径是在创建子进程前，在父进程中捕捉子进程终止信号 SIGCLD，如果有此信号，则执行一个 wait()即可。

上述 UNIX 的实时改造方案，已经在 SUN SPARC，SGI，DEC Alpha 等 RISC 工作站上实现，并实际使用于清华仿真所开发的第二代 PSS 中，现运行于广东和河南两个电网的中调。

## 参考文献


[1] GuoSheng Sun、XinFeng Wang, Implementation of an open Power System Simulator, Int Conf. on Advances in Power System Control and Management, 1995 Singapore

[2] 王凡、席裕庚等"实时工业控制软件平台选取及界面生成的一些讨论",《电气自动化》95/4

[3] 胡希明等《UNIX 结构分析》，浙江大学出版社，92/9


(来稿时间：1997 年 9 月)