DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Trabalhando com o Tempo Real em Aplicações Sobre o Linux

Relatório Técnico

Carlos Alexandre Piccioni
Cássia Yuri Tatibana
Rômulo Silva de Oliveira

Florianópolis, dezembro de 2001.

Sumário

1. Introdução

O sistema operacional Linux [1] é um sistema operacional tipo Unix, com código fonte aberto e disponível através de licença GPL, isto é, de graça mediante certas restrições. Este sistema tornou-se muito popular nos últimos anos, tanto para uso em máquinas *desktop* como para uso em máquinas servidoras. Graças a sua robustez, seu bom desempenho e a disponibilidade de ambientes de desenvolvimento, tem sido cada vez mais usado como base para aplicações de automação industrial.

Muitas aplicações do Linux necessitam manipular medidas de tempo, de alguma forma. Por exemplo, em sistemas de automação é necessário executar tarefas periódicas. Na comunicação de dados é comum a utilização de temporizações para detectar *time-outs*. Para registrar eventos é necessário saber a hora corrente. Para medir desempenho é necessário medir a duração de intervalos de tempo. Freqüentemente os programadores deparam-se com situações onde devem lidar com valores e medições de tempo, além de orientar o próprio fluxo de execução em função da passagem do tempo, como no caso de alarmes e tarefas periódicas.

O sistema operacional Linux provê diversos mecanismos para manipular o tempo em nível de aplicação. Diversas chamadas de sistema estão disponíveis para isto. Entretanto, algumas vezes o uso desses mecanismos exige cuidados na programação. Medidas de tempo em computadores são sempre aproximadas, mas uma programação cuidadosa é capaz de minimizar os erros. Além disso, alguns "truques" de programação não são óbvios a partir da simples leitura da descrição das chamadas de sistema e rotinas da biblioteca.

O objetivo deste texto é orientar o leitor sobre os recursos existentes no sistema operacional Linux para a manipulação do tempo, em nível de aplicação. Não serão descritos métodos que exijam a alteração do kernel do Linux nem que necessitem a preparação de módulos para serem incorporados dinamicamente ao kernel. A informação na qual este texto foi baseado está dispersa em livros, *faqs* e manuais de usuário. O propósito é portanto reunir em um só manual todas as informações relevantes para um programador de aplicação que necessite lidar com o tempo em seu programa. Não é do conhecimento dos autores outro texto que concentre as informações relacionadas com a manipulação do tempo por aplicações executando no sistema operacional Linux.

A maior parte do texto trata de recursos existentes no sistema operacional Linux. Dessa forma, eles estarão disponíveis em qualquer arquitetura para o Linux for portado. Entretanto, também serão descritos alguns recursos que estão disponíveis apenas na arquitetura PC Pentium, pois utilizam registradores específicos desse processador. Como esta é a arquitetura mais utilizada para executar Linux, tais informações serão úteis para muitas pessoas. Embora o texto tenha sido escrito com base na versão 2.4 do Linux, a maioria das chamadas de sistema utilizada é comum e estão disponíveis em todas as outras versões. Muitas delas seguem o padrão POSIX [2] e podem ser encontradas também em outros sistemas operacionais.

A seção 2 deste trabalho descreve como são os relógios de hardware presentes na arquitetura PC baseada no processador Intel Pentium. Os relógios de hardware são a base para a marcação do tempo no computador e o seu conhecimento facilita a compreensão das possibilidades e limitações da marcação do tempo no sistema. A seção 3 mostra como é possível obter a hora, através de chamadas de sistema. A seção 4 descreve duas maneiras diferentes de medir a passagem do tempo: usando a diferença entre horas e usando o registrador *Time Stamp Clock*. Na seção 5 são

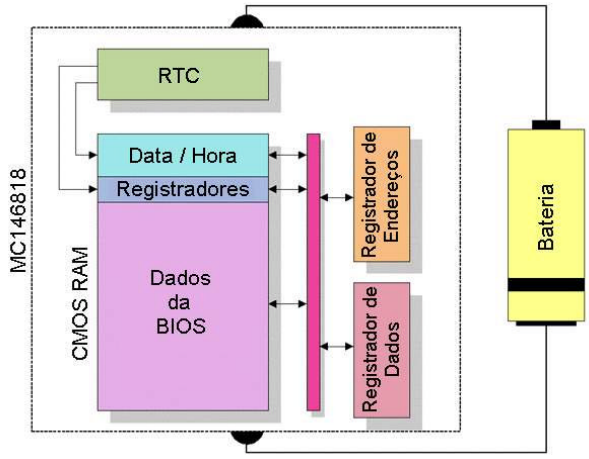
descritas as várias formas que o programador dispõe para esperar o tempo passar. Os mecanismos dos temporizadores e dos sinais são descritos na seção 6. Essa mesma mostra como uma aplicação pode estabelecer alarmes do tipo *time-out*. As diferentes formas possíveis de implementar tarefas periódicas são apresentadas na seção 7. Finalmente, a seção 8 traz algumas conclusões. No anexo estão incluídas informações sobre as chamadas de sistema e rotinas de biblioteca utilizadas ao longo do texto.

2. Relógios de Hardware do PC - Pentium

O kernel do Linux interage com três relógios de hardware diferentes: *Real Time Clock (RTC)*, *Time Stamp Counter (TSC)* e o *Programmable Interval Time (PIT)*. Como veremos a seguir, os dois primeiros auxiliam o kernel a obter a data e hora atual e o ultimo é programado para gerar interrupções fixas em uma freqüência determinada pelo kernel, que serão importantes para execução de tarefas do kernel e de programas do usuário [1].

2.1 Real Time Clock (RTC)

O Real Time Clock, RTC, ou relógio de tempo real, é um relógio de hardware implementado por um circuito integrado, geralmente o *Motorola 146818* e integrado a placa mãe, mais precisamente ao CMOS RAM, presente em todos os PCs atuais. Sua função é registrar a data e a hora atual. Ele é alimentado por uma pequena bateria, a mesma que alimenta a memória com a configuração do hardware do PC, ou seja, esse relógio continua funcionando mesmo com o PC desligado. Com relação a sua precisão, espera-se um erro de 10 segundos a cada mês. Além de registrar a data e a hora, pode funcionar como um timer, gerando interrupções na IRQ 8 quando seu relógio atingir determinado valor. O Motorola 146818 possui a seguinte estrutura interna [3]:



A CMOS RAM possui 64 bytes, onde estão gravados dados da BIOS, como configuração da máquina e a data e hora providos pelo RTC. Os endereços de memória internos da CMOS são:

Endereço:	**Conteúdo**
0x00	Segundos
0x02	Minutos
0x04	Horas
0x06	Dia da semana
0x07	Dia do mês
0x08	Mês
0x09	Ano
0x32	Século

A palavra contida em cada endereço possui 1 byte, sendo atualizada sempre pelo RTC e está no formato BCD. O acesso à essas informações são feitas nos endereços 0x70 e 0x71 das portas de entrada / saída da máquina. Por exemplo, se quisermos ter o valor do minuto corrente, devemos inserir o valor do endereço da CMOS RAM correspondente a minutos no endereço 0x70 do PC, e a leitura será feita no endereço 0x71. Pode ser implementado em C da seguinte forma (leitura dos minutos, por exemplo):

```
unsigned short int minuto;
outb (0x02,0x70);
minutos=inb (0x71);
```

Na primeira linha é declarada a variável *minuto*. Na segunda, escreve-se na porta 0x70 o valor correspondente ao endereço da CMOS RAM a qual se quer ter acesso, que em nosso caso é 0x02 que corresponde ao campo onde o RTC registra o valor dos minutos. Na terceira, guarda-se na variável *minuto* o valor correspondente ao minuto atual, lido na porta 0x71. Não devemos esquecer que este valor está no formato BCD.

O Linux usa o RTC somente para ler a data e a hora atual. Ele faz isso apenas uma vez, quando o kernel é iniciado. Para manter a hora atualizada, o kernel usa um outro artifício. Após ler a hora do RTC, ele não o usa mais para saber a hora e data atual. A hora em vigor no sistema é então a hora lida no RTC atualizada através de timers, como veremos adiante.

O administrador do sistema pode também setar esse clock usando o programa de sistema */sbin/clock* para agir diretamente nessas duas portas, ou pode programá-lo através de */dev/rtc*.

2.2 Time Stamp Counter (TSC)

O Time Stamp Counter, ou TSC, é um registrador que começou a estar presente na família dos microprocessadores 80x86 a partir do Pentium. Ele é um registrador de 64 bits que é incrementado a cada sinal, ou período de clock, ou seja, a cada fração de tempo correspondente ao inverso do clock do processador em Hz. Por exemplo, em um Pentium de 500 MHz, o TSC é incrementado a cada 2 nanosegundos. O TSC é uma ferramenta de alta precisão para a medição de tempo em PCs. O valor desse registrador pode ser acessado pela função implementada em C abaixo:

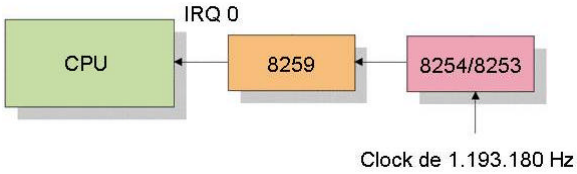
```
static inline long long unsigned lrdtsc (void) {
      long long unsigned time;
      __asm__ __volatile__ (“rdtsc”:”=A”(time));
      return time;
}
```

O Linux se aproveita do TSC pelo fato dele ser muito mais preciso que os derivados do *Programmable Interval Time* (PIT). Um exemplo de aplicação do TSC é a determinação da freqüência real do clock do processador, tarefa que o Linux faz quando é iniciado. Essa freqüência é obtida pela contagem de sinais de clock durante um intervalo de tempo pré-estabelecido (mais precisamente, 50,00077 milisegundos) gerado por um dos canais do PIT. Assim, a freqüência do processador será o número de sinais de clock divido pelo tempo pré-determinado em que ocorreram (freqüência em Hz = sinais do clock / tempo em segundos).

2.3 Programmable Interval Timer (PIT)

O PIT, ou Programmable Interval Timer é a terceira forma de medição de tempo utilizada pelo Linux. O kernel o usa para gerar períodos de tempo com uma freqüência fixa.

O PIT é implementado nos PCs pelo circuito integrado 8253 ou 8254 CMOS, que utiliza os endereços de 0x40 a 0x5F das portas de entrada e saída. Cada PC possui ao menos um PIT, e cada um, três canais. O primeiro canal é usado para atualização da hora, o segundo para controlar a atualização (refresh) da memória RAM e o terceiro para geração de ondas quadradas para o alto-falante do PC. O PIT recebe um sinal de clock no valor de 1.19318 MHz, e é conectado ao C.I. 8259, que é o responsável pela geração de interrupções do micro.



O PIT possui um contador, que é decrementado a cada sinal de clock. Os endereços de memória usados pelo primeiro PIT são [4]:

- 0x40 - Contador do canal 0 (leitura / escrita).
- 0x41 - Contador do canal 1 (leitura / escrita).
- 0x42 - Contador do canal 2 (leitura / escrita).
- 0x43 - Palavra de Controle (somente escrita).

Os dois bits mais significativos da palavra de controle são responsáveis pela escolha do canal a ser utilizado (00b para o canal 0, 01b para o canal 1 e 10b para o ultimo canal). Os próximos dois bits definem o modo de escrita/leitura, sendo:

- 0 (00b) - Contador fechado: O contador passa a operar internamente, mas ao ser lido em seu endereço de memória tem-se a impressão que está parado naquele valor.
- 1 (01b) - Leitura e escrita do byte menos significativo.
- 2 (10b) - Leitura e escrita do byte mais significativo.
- 3 (11b) - Leitura e escrita dos bytes mais e menos significativos.

Os próximos três bits definem o modo em que o PIT irá funcionar, que são:

- Modo 0 (000b) - Interrupção ao final da contagem: Contagem regressiva. A interrupção será ativada após o término da contagem estabelecida no seu endereço de memória. Uma nova contagem será iniciada e a interrupção desligada quando for escrito um novo valor no contador do canal.
- Modo 1 (001) - Semelhante ao primeiro modo, mas nesse caso a contagem pode ser reiniciada sem a necessidade de escrever novamente o valor do contador.
- Modo 2 (010b) - Gerador de freqüência: A cada período de tempo pré-determinado é gerado uma interrupção.
- Modo 3 (011b) - Gerador de onda quadrada: Gera uma onda quadrada de freqüência pré-determinada.
- Modo 4 (100b) - Similar ao modo 0, sendo que a interrupção é iniciada ativada e após o final da contagem é desligada.
- Modo 5 (101b) - Similar ao anterior, mas a contagem aguarda um sinal de hardware para iniciar a contagem.

O bit menos significativo da palavra de controle configura o padrão do número do contador:

- 0b - Binário de 16 bits.
- 1b - BCD de 4 décadas

O Linux programa o primeiro PIT para gerar interrupções na IRQ 0, a freqüência de 100 Hz, ou seja, com um período de 10 milisegundos (o programa no modo 3, gerador de freqüência). A cada interrupção, ou seja, a cada instante em que se passaram mais 10 milisegundos ocorre o que chamamos de *tick*. Esses *ticks* são utilizados para sinalizar todas as atividades do sistema. Por exemplo: atualizar a hora no sistema, atualizar as estatísticas dos recursos usados no kernel, determinar quanto tempo esta ocupando a CPU determinado processo e se esse já estourou o tempo de limite de uso, enfim, várias atividades de responsabilidade do kernel.

O *tick* de 10 milisegundos é o padrão adotado pelo Linux. Pode ser alterado, mas devemos ter em mente certos fatores. *Ticks* curtos produzem melhores repostas do sistema. Isto é devido à resposta do sistema ser largamente dependente de quão rápido um processo rodando é preemptado por um processo de alta prioridade assim que ele se tornou executável. Além disso, o kernel usualmente verifica se o processo rodando pode ser preemptado enquanto for tratada uma interrupção de timer. Ou seja, ticks curtos requerem que a CPU gaste uma larga fração de seu tempo em Modo Kernel, isto é, tem uma menor fração de tempo no Modo Usuário. Como conseqüência disso, programas do usuário rodarão devagar. Portanto, somente máquinas com muito poder de processamento podem adotar ticks curtos e arcar com as conseqüências. Atualmente, somente o Compaq Alpha apresenta um tick em torno de um milisegundo (1024 interrupções por segundo).

A seguir veremos como o Linux programa o PIT. Como vimos, seu contador é decrementado a cada sinal de clock, que é 1.19318 MHz. Assim, a cada segundo, o contador é decrementado de 1193180 unidades. Para que seja zerada uma interrupção a cada 10 ms, um centésimo de segundo, esse valor deverá ser dividido por 100. O Linux o faz da seguinte forma: O kernel possui uma variável interna, *Hz* (asm/param.h), que é setada em 100. Há outra variável, chamada de *Clock_tick_rate* (asm/timex.h), que é setada em 1193180. E a última variável, que será a gravada no endereço de memória do contador do canal 0, é chamada de *Latch* (asm/timex.h), é a razão entre *Clock_tick_rate* e *Hz*, e que será usada para programar o PIT e nos fornecer interrupções a cada 10 milisegundos.

O PIT é então inicializado pela função **init_IRQ()** (arch/i386/kernel/i8259.c), que possui a seguinte programação em C:

```
outb_p (0x34, 0x43);
outb_p (LATCH & 0xFF, 0x40);
outb (LATCH >> 8, 0x40);
```

A função **outb()** em C é equivalente a mesma instrução em assembly: ela copia o primeiro operando para o endereço de memória especificado no segundo operando. Nessa primeira operação, o Linux grava a palavra 0x34 no endereço de memória da palavra de controle do PIT. A palavra em 0x34 corresponde a 00110100 em binário. O bit menos significativo é 0, e como vimos, define que o valor gravado no contador do canal será em binário de 16 bits. Os próximos três, 010b, definem o modo 2, que será um gerador de freqüência. Os próximos dois, 11b, permitem a leitura e escrita dos bytes menos e mais significativos do contador. E os dois bits mais significativos, 00b, informam que será usado o primeiro canal (canal 0) do PIT. A função **outb_p()** é

similar a **outb()**. A primeira linha é um comando para o PIT para assumir a interrupção para uma nova taxa. A segunda e terceira linha fornece uma nova taxa para o 8254. A constante *LATCH* é uma variável de 16 bits, e a porta 0x40 é de apenas 8 bits. Com 2 bytes consecutivos. Como resultado o PIT está programado para gerar interrupções a cada 10 milisegundos. Para se fazer uso da função **outb()** em um programa do usuário acessando endereços de portas I/O é necessário logar-se como super-usuário e usar o a função **ioperm()** para habilitar o acesso a esses endereços, como será descrito na seção 3.4.

3. Obtendo a Hora

Para se obter a hora no Linux através de chamadas de sistema, em nível de programação, tem-se disponíveis as seguintes funções: **time()**; **ftime()** e **gettimeofday()**. Podemos também fazer acesso a informações sobre hora através do **acesso direto ao RTC**, como explicado na seção anterior e que será visto novamente com mais algumas considerações. Existem também as funções responsáveis pela formatação das unidades de tempo, como **ctime(), asctime(), gmtime(), localtime(), mktime()** e **strftime()**. A seguir, veremos o uso e característica de cada forma de se obter a hora no PC em nível de programação em C [5].

3.1 Função Time

A função **time()** é a mais simples de todas. Com ela obtemos apenas um valor, que é o número de segundos passados desde a primeira hora do dia primeiro de janeiro de 1970. Esse valor não conta os segundos que foram retirados ou acrescentados ao calendário desde então para correção do próprio. Para ela, cada dia transcorrido desde então possuí exatamente 24 horas. A função é declarada da seguinte forma: **time_t time (time_t *time)**, sendo *time* um ponteiro para a área de memória contendo o tempo em segundos. É necessária a inclusão de *time.h* para o funcionamento da mesma, já que esta contém a estrutura *time_t*. Podemos fazer uso da função **time()** da seguinte forma:

```
time_t time;
time (&time);
```

Assim, na variável *time* teríamos agora o valor de tempo em segundos transcorridos desde 1970, GMT (horário do Meridiano de Greenwich). A função time() retorna zero quando é bem sucedida e -1 quando não. *Time_t* é uma variável definida em *time.h* como *long int*.

A função **time()** demora em média 2200 ciclos de clock de CPU para ser executada, considerando sua execução com prioridade normal e interrupções do kernel do Linux habilitadas, além de outros processos em atividade. Para se chegar em tal valor de ciclos de clock, se faz uso da função **lrdtsc()** demonstrada na seção anterior. Faz-se uma chamada antes e uma depois da função **time()**, e o número de ciclos será a subtração do valor fornecido da segunda pela primeira, além de descontarmos o número de ciclos de clock que a **lrdtsc()** também consome.

Operações que exigem precisão não poderão ser feitas com a função **time()**. Por definição, a melhor resolução que conseguimos com ela é de 1 segundo, ou seja, o erro é em média meio segundo. Assim, ela só poderia ser usada para execução de tarefas de controle por exemplo em processos com uma resposta extremamente lenta.

3.2 Função Ftime

A função **ftime()** é mais elaborada que a **time()**. Possui uma estrutura de dados interna que nos fornecem mais informações sobre a hora atual. Era a função principal na obtenção da hora no Unix *BSD* versão 4.2, sendo então suplantada pela função

gettimeofday() na versão 4.3 desse sistema operacional. A estrutura interna usada é a seguinte:

```
struct timeb {
      time_t time;
      short unsigned millitm;
      short timezone;
      short dstflag;
}
```

Onde, em *time* teremos o número de segundos transcorridos desde primeiro de janeiro de 1970, GMT, igual a ao retornado pela função **time()**; *millitm* nos dá os milisegundos transcorridos desde então; *timezone* o valor em minutos a oeste do meridiano de Greenwich em que nos encontramos; *dstflag* será setado em 1 se estivermos nos horário de verão ou 0 se não.

A biblioteca a ser inclusa para seu uso, onde é declarada essa estrutura *timeb*, é a *sys/timeb.h*. A chamada de **ftime()** é na forma **int ftime (struct timeb *time)**. Abaixo, *time* será a variável que acessará as informações sobre a hora. A função **ftime()** sempre retorna 0. Para fazer uso da função, declaramos a variável *time*:

```
struct timeb time;
```

A chamada a função será então:

```
Ftime (&time);
```

Assim, a partir de *time*, podemos acessar os dados da estrutura *timeb* que conterão as informações sobre a hora no momento em que **ftime()** fora chamada.

A função **ftime()** demora em média 500 ciclos de clock da CPU para ser executada, considerando sua execução com prioridade normal e interrupções do kernel do Linux habilitadas, além de outros processos em atividade.

Ftime(), como vimos na sua variável *millitm*, apresenta um resolução de milisegundos. Sua execução e suas operações internas demoram na ordem de centenas de nanosegundos, tempo cerca de mil vezes menor que sua menor unidade de medição de tempo. Sendo assim, é bastante precisa, e pode ser usada em operações onde se exija uma precisão de milisegundos.

3.3 Função Gettimeofday

A função **gettimeofday()** é a função mais utilizada atualmente para a obtenção da hora atual. Passou a ocupar o lugar da **ftime()** a partir do BSD 4.3. Sua principal diferença em relação a anterior é que possui duas estruturas de dados internas:

```
struct timeval {
      long tv_sec;
      long tv_usec;
}

struct timezone {
      int tz_minuteswest;
      int tz_dsttime;
}
```

A biblioteca que contém essa estrutura é a *sys/time.h*. A chamada de **gettimeofday()** é na forma: **int gettimeofday(struct timeval *tv, struct timezone *tz)**, sendo que *tv* e *tz* serão nossas variáveis que acessarão as informações sobre a hora atual. **Gettimeofday()** retorna 0 se a operação foi executada com sucesso ou -1 se falhou.

Após uma chamada de **gettimeofday()**, a variável *tv_sec* guardará o valor em segundos que se passaram desde a primeira hora de primeiro de janeiro de 1970, assim como faz a função **time()**. A variável *tv_usec* retorna o número de microssegundos transcorridos após o último segundo. Essas duas variáveis pertencem a estrutura *timeval*, como já fora citado. Já as duas variáveis pertencentes a *timezone*, são *tz_minuteswest*, que como a variável *timezone* de **ftime()** apresenta o valor em minutos a oeste do meridiano de Greenwich em que nos encontramos. A variável *tz_dsttime* está em desuso, sendo que nem pode ser utilizada sobre o Linux (seu uso gera erro na execução do programa), e armazena um valor simbólico com informações sobre a área do planeta no qual nos encontramos, para que a partir de um algoritmo presente em algum programa do usuário seja calculado os dias em que se aplicará o horário de verão. Seu desuso se deve ao fato de que esses dias, na atualidade, não podem ser calculados por um simples algorítmo, pois dependem mais de decisões políticas do que de qualquer outro fator. Assim, para fazermos uso de **gettimeofday()**:

```
struct timeval tv;
struct timezone tz;
```

E chamamos a função:

```
int gettimeofday (&tv,&tz);
```

Assim, temos em *tv.tv_sec* e *tv.tv_usec* as informações já expostas anteriormente.

A função **gettimeofday()** obtém o tempo da seguinte forma: a cada interrupção do PIT, na IRQ 0, ocorre um tratamento dessa interrupção pelo kernel. Nessa rotina, o Linux sabe que se acabaram de passar dez milisegundos desde o último *tick*, e também faz a leitura do TSC e guarda seu valor em uma variável. Assim, a cada chamada de **gettimeofday()**, o Linux sabe o tempo atual até o último tick, e o restante ele obtém fazendo uma nova leitura do TSC, subtraindo esse valor por aquele lido na interrupção e convertendo-o em microssegundos.

A função **gettimeofday()** consome em média 3200 ciclos de clock da CPU para ser executada, considerando sua execução com prioridade normal e interrupções do kernel do Linux habilitadas, além de outros processos em atividade. Em um Pentium de 100 MHz, 3000 ciclos de clock demoram 3 microssegundos. Como a resolução dessa função é de 1 microssegundo, podemos notar que a precisão não é igual a resolução, sendo um pouco menor. Porque? Porque a função demorando cerca de 3 microssegundos para ser executada, após terminada o tempo já não será o mesmo do qual ela o leu em microssegundos (provavelmente já se terão passados de 1 a 3 microssegundos). Assim, quanto mais rápido a função **gettimeofday()** for executada, maior será a precisão, ou seja, deveremos levar em conta o clock da CPU.

3.4 Acesso Direto ao RTC

Uma outra forma de obtermos informações sobre a hora é através do acesso direto ao RTC. Mas para tal é necessário ter em mente alguns fatores. O primeiro é

que a hora do sistema não é exatamente a mesma do RTC. O Linux ao ser iniciado lê a hora do RTC e a partir de então gerencia a hora do sistema por conta própria, sem mais consultá-lo. A próxima hora em que fará isso será ao ser desligado, quando atualizará a hora do RTC com a do sistema. Ou seja, como o RTC possui um erro em torno de 10 segundos ao mês e o artifício usado pelo Linux para atualizar a hora depende do PIT e este também não é perfeito, em um sistema ligado a muito tempo pode haver uma grande disparidade entre a hora dada pelo RTC e a do sistema. Outro fator que devemos ter em mente é que para lermos o valor do RTC devemos escrever em endereços de memória protegida pelo Linux, sendo necessária a inclusão da seguinte linha de comando no programa do usuário:

```
ioperm (0x70,2,1);
```

Essa função, bem como a **outb()** e a **inb()** estão na biblioteca *sys/io.h*. Para o usuário rodar esse programa deverá primeiro logar-se como *root*. Essa função tem por objetivo liberar (usando 1 no terceiro argumento) o uso de 2 bytes (segundo argumento) a partir do endereço 0x70. Assim, poderão ser escritos / lidos os bytes 0x70 e 0x71 das portas de entrada e saída da máquina. Como já fora explicado, deveremos guardar o valor no endereço 0x70 respectivo ao que queremos receber no endereço 0x71, conforme a tabela abaixo:

Endereço (CMOS RAM):	**Conteúdo**
0x00	Segundos
0x02	Minutos
0x04	Horas
0x06	Dia da semana
0x07	Dia do mês
0x08	Mês
0x09	Ano
0x32	Século

Assim, para qualquer conteúdo da tabela que desejarmos obter devemos, utilizamos a função **outb (y,0x70)**, onde y é o endereço da CMOS RAM presente na tabela acima correspondente. Para se ler o resultado basta invocarmos a função **inb (0x71)**, que retornará o valor correspondente, em um byte, mas no formato BCD.

Uma chamada **outb()** seguida de um **inb()** nesse contexto demora aproximadamente de 400 a 500 ciclos de clock. A resolução mínima acessível é também na ordem de segundos, ou seja, não satisfaz aplicações críticas.

3.5 Funções de Formatação do Tempo

As funções **ctime(), asctime(), gmtime(), localtime(), mktime()** e **strftime()**, são responsáveis pela manipulação e formatação do tempo, ou seja, transformam uma forma de indicação de tempo em outra correspondente. Para o funcionamento das mesmas, é necessária a inclusão da biblioteca *time.h*.

A função **ctime()**, declarada **char *ctime (const time_t *timep)** retorna o valor de apontado por *timep* (segundos desde primeiro de janeiro de 1970) em uma string, da forma: *"Mon Oct 2 12:30:21 2001"*.

Gmtime() é declarada como **struct tm *gmtime (const time_t *timep)**, sendo *timep* os segundos desde janeiro de 1970. Ela retorna um ponteiro para a

estrutura *tm* com as informações sobre a hora atual (GMT), que tem a seguinte declaração em *time.h*:

```
struct tm {
      int tm_sec;
      int tm_min;
      int tm_hour;
      int tm_mday;
      int tm_mon;
      int tm_year;
      int tm_wday;
      int tm_yday;
      int tm_isdst;
};
```

Sendo *tm_sec* o número de segundos (0 a 60), *tm_min* o tempo em minutos (0-59), *tm_hour* as horas (0 a 23), *tm_mday* o dia do mês (1 a 31), *tm_mon* o mês (0 a 11), *tm_year* o ano (se em 1999, nos retorna 99, se em 2001, 101), *tm_wday* o dia da semana (0 a 6), *tm_yday* o dia do ano (0 a 365) e *tm_isdst* a informação a respeito do horário de verão, sendo -1 para uma hora de atraso, 1 para uma hora adiantada e 0 para desativado.

Já **localtime()** é igual a **gmtime()**, mas retorna o tempo regional, enquanto a anterior, o valor GMT, além de gerar variáveis externas, como *tzname*, com informações sobre a região onde nos encontramos, *timezone* com o número de segundos a oeste está essa região do Meridiano de Greenwich, e *daylight* indicando se são aplicadas ou não políticas sobre o horário de verão.

Asctime() tem a seguinte declaração: **char *asctime (const struct tm *timeptr)**. Retorna a data na forma de uma *string*, como **ctime()**, a partir de um ponteiro para uma estrutura *tm* com informações sobre a hora atual.

A função **mktime()** é declarada **time_t mktime (struct tm *timeptr)** e faz o inverso de **gmtime()** ou **localtime()**, retornando a partir de um ponteiro para um estrutura *tm* o tempo em segundos transcorridos desde primeiro de janeiro de 1970.

E por fim, **strftime()**, que é declarada: **size_t strftime (char *s, size_t max, const char *format, const struct tm *tm)**. A partir de um ponteiro para a estrutura *tm*, retorna uma *string s* de tamanho *max* no formato definido por *format*, em um formato semelhante a **printf()**. Por exemplo: **strftime ("%r %A %e %B %Y", localtime (time())** retorna a string: "00:30:21 AM Monday 2 October 2001". Os parâmetros passados em *format* podem ser vistos através do *man strftime* no terminal do Linux:

Parâmetro	**Conteúdo**
%n	Nova Linha
%t	Tabulação
%H	Horas (00 a 23)
%I	Horas (01 a 12)
%k	Horas (0 a 23)
%l	Horas (1 a 12)
%M	Minutos (00-59)
%p	AM ou PM
%r	Tempo, 12 horas (hh:mm:ss [AP]m)
%R	Tempo, 24 horas (hh:mm)
%s	Segundos desde 01/01/1970
%S	Segundos (0 a 59)
%T	Tempo, 24 horas (hh:mm:ss)

%a	Abreviação do dia da semana
%A	Dia da semana (nome completo)
%b ou %h	Abreviação do nome do mês
%B	Nome do mês completo
%C	Século (00 a 99)
%d	Dia do mês (01 a 31)
%e	Dia do mês (1 a 31)
%D	Data GMT (mm/dd/aa)
%j	Dia do ano
%m	Mês (01-12)
%U	Número da semana (00 a 53)
%w	Dia da semana (0 a 6)
%x	Data local (mm/dd/aa)
%y	Últimos dois dígitos do ano (00 a 99)
%Y	Ano (19xx ou 20xx)

4. Medindo a Passagem do Tempo

Medir a passagem do tempo é uma tarefa freqüentemente exigida de aplicações de áreas diversas, seja para sincronização dos processos que a compõe, ou para utilização por um tipo específico de aplicação, como programas de avaliação de desempenho. Aplicações de tempo real, entretanto, lidam com o tempo de maneira explícita e portanto, tem a medição da passagem de tempo como uma tarefa de intrínseca e de extrema importância.

Como ferramentas de suporte no estudo de sistemas computacionais de tempo real, programas com o propósito específico de capturar intervalos de tempo são freqüentemente construídos. Iterações contínuas de um laço de medição de tempo revelam informações importantes sobre o comportamento do sistema em tempo de execução. Embora existam diversos outros meios de obter a mesma informação, muito deles interferem demasiadamente nos valores capturados, o que eventualmente compromete a qualidade dos estudos sobre os mesmos. Este é o caso por exemplo, de simuladores e depuradores.

A construção de programas que possam garantir determinada precisão na medição de tempo e conseqüentemente fornecer base confiável para o estudo desenvolvido, deve apresentar cuidados especiais quanto a escolha dos valores a serem adotados como parâmetros. O parâmetro utilizado é um valor sobre o qual todos os demais cálculos do programa são efetuados. Ele é lido a partir de alguma fonte de medição de tempo externa ao programa através de consultas, realizadas por chamadas do sistema. Contadores de hardware caracterizam uma fonte confiável e relativamente precisa de medição de tempo. Como visto anteriormente, os contadores não são afetados por flutuações de desempenho do sistema, falhas de software ou sobrecarga. Embora o processo de leitura dos mesmos o seja, eles são freqüentemente utilizados como fonte de parâmetros para os programas em questão.

No Linux, como citado anteriormente, existem três mecanismos voltados para o registro de passagem de tempo: o *Real Time Clock, (RTC),* o *Time Stamp Counter* (TSC), e o *Programable Interrupt Timer (PIT).* Portanto, a medição de passagem de tempo nesta plataforma é feita direta ou indiretamente a partir de uma dessas fontes.

4.1 Diferença na Hora

Em sua grande maioria, os estudos relacionados a tempo real sobre plataformas existentes exigem em dado momento o conhecimento do comportamento temporal desta plataforma. Seja para realização de testes que comprovem uma determinada teoria, ou testes de caráter ilustrativo ou coleta de dados. Esse conhecimento pode ser adquirido pelo estudo detalhado do código fonte do sistema e posterior aproximação de valores de tempo para sua execução, ou pela medição do intervalo de tempo transcorrido durante a execução de um conjunto de instruções conhecidas, cujo tempo de execução se deseja medir. No segundo caso, a medição revela apenas uma aproximação do tempo consumido pela instrução ou instruções executadas. Entretanto, a precisão desta medição pode ser melhorada caso executadas dentro de um laço. Obtendo um vetor de valores para execução de uma mesma instrução ou seqüência de instruções que permite a verificação do tempo médio de execução das mesmas dentro do *loop*. O trecho de código de programa a seguir ilustra tais mecanismos.

```
int j=1;
int media=0;
while (j<N) {
      i1=gettimeofday (&tv1, &tz1);
      t1[j]=tv1.tv_usec;
      nanosleep (&ts, &tsr);        /*intervalo a ser medido */
      i1=gettimeofday (&tv2, &tz2);
      t2[j]=tv2.tv_usec;
      j++;
}
for(i=0;i<N;i++) {
      media+=t2[i]-t1[i];
}
printf("Média dos Intervalos de tempo coletados:%llu\n",
media/N );
```

A seqüência de instruções contidas no laço *while* é executada *N* vezes, a coleta de valores é feita pelas linhas das chamadas **gettimeofday()** e das instruções de armazenamento de valores no vetor *t1* e *t2*. Ambos os vetores armazenam valores de tempo em nanosegundos, fornecidos como valores de retorno da chamada **gettimeofday()** na estrutura tv1. Tv1 é uma *struct timeval*, como já fora explicado anteriormente. A instrução **nanosleep()** simplesmente preenche o laço afim de fornecer um tempo suficiente para medição de tempo transcorrido no intervalo. Os valores de parâmetros fornecidos para esta função são passadas através das estruturas *ts* e *tsr*, ambas possuem um campo para valores de tempo em segundos e um campo para valores de tempo em nanosegundos. **Nanosleep()** provoca uma pausa na execução do programa por pelo menos o tempo especificado em seu primeiro parâmetros. Entretanto, o tempo máximo que esta chamada pode durar não pode ser determinado com precisão.

4.2 Utilizando o TSC

O TSC, pode ser facilmente acessado por qualquer aplicação, não é exigido da mesma que execute em modo supervisor. A função abaixo ilustra como pode ser feita a chamada de consulta ao TSC, como já fora citado:

```
long long unsigned lrdtsc (void) {
      long long unsigned time;
      __asm__ __volatile__("rdtsc": "=A"(time));
      return time;
}
```

A função **lrdtsc()** é bastante simples, ao ser invocada executa a instrução assembly rdtsc que é responsável pela leitura do registrador Time Stamp Counter (TSC). Tendo em vista que este é um registrador de 64 bits incrementado desde a inicialização do processador, seu valor exige uma variável *long long* em que possa ser armazenado. Por isso, o tipo da função e da variável de retorno é definido como *long long unsigned*. A utilização da função **lrdtsc()** é ilustrada pelo trecho de programa abaixo:

```
while(j<N) {
      t1[j]=lrdtsc();
      nanosleep (&ts, &tsr);  /* Intervalo a ser medido */
      t2[j]=lrdtsc();
      j++;
```

```
}
for(i=0 ;i<N;i++) {
      media=+t2[i]-t1[i];
}
printf("Média dos Intervalos de ciclos de clock
            coletados:%llu\n", media/N );
```

Por se tratar de valores altos, os valores de retorno da chamada **lrdtsc()** são armazenados em variáveis de 64 bits. Entretanto, durante a manipulação de tais valores, existe a possibilidade da ocorrência de overflow em algumas variáveis (principalmente as variáveis do tipo *long long int* ou *long unsigned*) e conseqüente perda do valor. A fim de diminuir as chances de tal ocorrência durante os cálculos do programa, um macro pode ser utilizado. A macro simplesmente subtrai do valor devolvido por **lrdtsc()** o valor capturado no inicio da execução do programa. Tornando assim, todos os valores de tempo capturados dentro da aplicação, relativos a um instante inicial.

Embora a utilização do contador de ciclos de clock seja bastante simples, e forneça um parâmetro bastante preciso para a medição da passagem de tempo, em geral a informação desejada deve estar sob a forma de unidades de tempo. O conhecimento prévio da freqüência de clock do processador permite que o valor em número de ciclos de clock fornecido por **lrdtsc()** seja facilmente convertido em unidades de tempo. Embora este valor possa ser encontrado em alguma variável do código do kernel, tendo em vista o cálculo desse valor na inicialização do sistema, ele só pode ser acessado através do uso de módulos do kernel, executando com permissões do modo supervisor.

Afim de que esta informação possa ser conhecida por qualquer usuário, um valor aproximado do valor da freqüência de CPU é fornecido em */proc/cpuinfo*. Entretanto, a falta de precisão do valor fornecido combinado aos altos valores retornados pelo contador
de tempo real provocam um erro no valor calculado. O erro provocado, embora seja inicialmente pequeno, é amplificado no decorrer dos cálculos realizados sobre o valor fornecido, resultando em um erro cumulativo de proporções consideráveis, comprometendo a validade da resposta da aplicação.

Ainda assim, o cálculo do valor de freqüência, por parte de uma aplicação de usuário é possível, como ilustrado pelo código do programa *freq.c*. O programa *freq.c* é também um exemplo simples, um programa que captura intervalos de tempo e correspondentes valores de contador de ciclos de clock para o cálculo da freqüência do processador da máquina em uso. Este programa permite a captura deste valor com precisão de algumas dezenas de Hz e pode ser executada por qualquer usuário como uma aplicação comum, não exigindo permissões do modo supervisor.

A utilização do contador TSC e do macro utilizado para leitura do contador de ciclos de clock acima mencionada é ilustrada pelo programa *freq.c* abaixo (calculo da freqüência do processador):

```
#include <stdio.h>
#include "pthread.h"
#include <sys/time.h>
#include <stdlib.h>

long long unsigned  c0, *c1,  *c2 ;
int g1, g2;
struct timeval *t1, *t2;
int nciclos = 12;
```

```
#define agora() ( lrdtsc() - c0 )

long long unsigned lrdtsc (void) {
      long long unsigned time;
      __asm__ __volatile__("rdtsc": "=A"(time));
      return time;
}

void * thread(void * arg) {
      int i, j, ds;
      long long unsigned dus, media;
      int cont;
      struct timespec ts;
      struct timespec tsr;
      struct timeval tv;
      struct timezone tz;
      j=1;
      fprintf (stderr, "Iniciando processo! %s\n", (char *)
arg);
      c0=lrdtsc();
      while(j < nciclos) {
            ts.tv_sec=10;
            ts.tv_nsec=0;
            tsr.tv_sec=0;
            tsr.tv_nsec=0;
            c1[j]=agora();
            if((g1=gettimeofday (&tv, &tz))<0)
                  exit(0);
            t1[j]=tv;
            nanosleep (&ts, &tsr);  //espera    pelo    próximo
      período
                  c2[j]=agora();    /*c2marca o fim da execução
                                    da tarefa da thread*/
            if((g2=gettimeofday(&tv, &tz))<0)
                  exit(0);
            t2[j]=tv;
            j++;
      }
      cont=2;
      while(cont<nciclos){
            printf("t2[cont]:   %lu   %lu\n",   t2[cont].tv_sec,
      t2[cont].tv_usec);
            printf("t1[cont]:   %lu   %lu\n",   t1[cont].tv_sec,
      t1[cont].tv_usec);
            ds=t2[cont].tv_sec-t1[cont].tv_sec;
            dus=ds*1000000L+t2[cont].tv_usec-t1[cont].tv_usec;
            printf("dus: %llu\n", dus);
            printf("c2[cont]: %llu\n", c2[cont]);
            printf("c1[cont]: %llu\n", c1[cont]);
            printf("c2[cont]-c1[cont]:    %llu\n",    c2[cont]-
      c1[cont]);
            printf(" Freqüência em Hz: %llu \n", ((c2[cont]-
      c1[cont])*1000000L/dus)) ;
            cont++;
      }
      media=0;
      cont=2;
      while(cont<nciclos){
      ds=t2[cont].tv_sec-t1[cont].tv_sec;
      dus=ds*1000000L+t2[cont].tv_usec-t1[cont].tv_usec;
```

```
      printf("Freqüência   em   Hz:   %llu   \n",   ((c2[cont]-
c1[cont])*1000000L/dus));
      media+=((c2[cont]-c1[cont])*1000000L/dus);
      cont++;
      }
printf("media Hz; %llu\n",  media/10);
return NULL;
}

int main (void) {
      int retcode;
      pthread_t th_a;
      void *retval;
      struct sched_param mysched;
      mysched.sched_priority                                    =
sched_get_priority_max(SCHED_FIFO) -1;
      if (sched_setscheduler(0,SCHED_FIFO,&mysched)==-1){
            printf ("Erro na definição de escalonador!\n");
            perror("errno");
            exit(0);
      }
      t1=calloc (nciclos ,sizeof(struct timeval));
      t2=calloc (nciclos, sizeof(struct timeval));
      c1=calloc (nciclos, sizeof(long long unsigned));
      c2=calloc (nciclos, sizeof(long long unsigned));
      retcode=pthread_create(&th_a,  NULL,  thread,  (void  *)
"1");
      if (retcode!=0) fprintf (stderr, "create a failed %d\n",
retcode);
            retcode=pthread_join (th_a, &retval);
      if  (retcode!=0)  fprintf(stderr,  "join  a  failed  %d\n",
retcode);
            return 0;
}
```

A rotina do programa, descrita no corpo da *thread*, tem o único propósito de capturar intervalos de tempo. A captura de valores de ciclos de clock, realizada pela chamada da função **lrdtsc()** é sempre imediatamente seguida pela chamada a sub-rotina **gettimeofday()**.
Desta forma, supondo a execução imediata da seqüência dessas duas linhas de código, os valores de ciclos de clock e tempo permitirão o cálculo da freqüência do processador com precisão satisfatória para a maioria das aplicações. Valores do contador de ciclos de clock e valores de tempo são armazenados duas vezes dentro de cada laço de execução. Entre as duplas de função e sub-rotina de captura destes valores, uma chamada a sub-rotina **nanosleep()** realiza uma pausa na execução pelo tempo especificado na variável do primeiro parâmetro da mesma. O cálculo da freqüência, ou qualquer outro valor que possa ser adquirido a partir da execução do laço e dos valores capturados é realizado após o término das iterações, não comprometendo desta maneira, os valores em si.

5 Esperando o Tempo Passar

No Linux há três chamadas de sistemas que nos permitem criar um período de espera, inativo, em nossos processos. São elas: **sleep()**, **usleep()** e **nanosleep()**. Abaixo, a descrição de cada uma, e por final, uma avaliação da precisão e formas de utilização desse tipo de chamada de sistema [5].

5.1 Função Sleep

A função **sleep()** é declarada da seguinte forma: **unsigned int sleep (unsigned int *seconds*)**. Ela deixará o processo paralisado, dormente, por tantos segundos quantos forem declarados na variável *seconds*. Essa função pode ser interrompida por algum sinal que o processo esteja esperando, e caso isso ocorra, ela retornará o número de segundos que faltavam para o seu término. Caso ela 'durma' todo o tempo pré-estabelecido, retornará zero. A biblioteca a ser inclusa é a *unistd.h*. A função **sleep()** usa a variável interna *SIGALRM*. Alterações nessa quando a função em execução, ou o uso da função **longjmp()** em tratadores de sinal, pode gerar um comportamento inesperado. O uso conjunto da função **alarm()** também não é recomendável, pois faz uso da mesma variável.

5.2 Função Usleep

A função **usleep()** (lida como micro sleep), é declarada como **void usleep (unsigned long *usec*)**, onde *usec* será a variável em que informaremos quanto, em microssegundos, o processo deve 'dormir'. *Usec* pode ser maior que 999.999, ou seja, a função pode dormir mais que um segundo. A biblioteca a ser inclusa é a mesma da **sleep()**: *unistd.h*.

5.3 Função Nanosleep

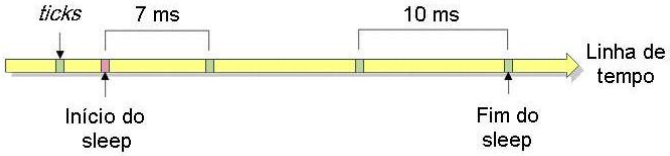
A função **nanosleep()** é declarada **int nanosleep (const struct timespec *req, struct timespec *rem)**, onde *req* e *rem* são duas estruturas de tipo *timespec* declaradas em *time.h*:

```
struct timespec {
      time_t tv_sec;
      long tv_sec;
}
```

Onde *time_t* é do tipo *long int*, O tempo de dormência deve ser passado em *req*, em segundos na variável *tv_sec* e em nanosegundos na variável *tv_usec*. Se a função for bem sucedida, ou seja, não for interrompida, retornará zero. Caso contrário, retornará -1 e em *rem* ficará registrado o tempo que faltava para o final da execução. A valor setado em *rem->tv_usec* pode ir de 0 a 999.999.999 nanosegundos. Uma das vantagens da função **nanosleep()** é que não interfere com sinais.

5.4 Precisão das Funções Sleep

As três funções possuem funcionamento interno semelhante. Sem prioridade alta, o processo que as roda não ocupa o processador após a chamada da função, e o libera para outros processos. As funções **usleep()** e **nanosleep()** deveriam ter teoricamente uma precisão na ordem de microssegundos e nanosegundos, respectivamente. Mas tal feito não ocorre na realidade em um processo rodando com prioridade normal, sem privilégios. Sua precisão fica dependente da política de gerenciamento de tempo do Linux, através dos *ticks*. Seu funcionamento é o seguinte: Suponha que queiramos que o processo durma 15 milisegundos, por exemplo. O Linux entende que esse processo deve ficar dormente no mínimo 15 ms, e não exatamente. Assim, utiliza a ocorrência de *ticks* para determinar o tempo mínimo que o processo deverá permanecer parado. Como cada tick ocorre a cada 10 ms, o Linux acordará esse processo após dois *ticks*, o que garantirá que os 15 ms tenham passado. Outro fator que aumentará o tempo de dormência é que a função não iniciará exatamente na ocorrência do primeiro *tick*, e sim antes. Nesse exemplo, pode ocorrer que a função seja chamada 7 milisegundos antes do próximo tick. A chamada pede uma dormência de 15 milisegundos, o que o Linux tratará como a passagem de dois *ticks*. Assim, o processo só acordará 2 *ticks* mais 7 milisegundos depois, o que totaliza 27 milisegundos. Abaixo, uma linha do tempo ilustrando esse exemplo (uma chamada de alguma função do tipo **sleep()** para dormir 15 milisegundos):



Pode ocorrer ainda que atividades internas do kernel façam com que o processo durma um *tick* a mais. Nesse caso, serão mais 10 milisegundos perdidos. Para calcular o maior erro da chamada **sleep()** e suas derivadas, devemos pensar no pior caso. Como explicamos acima, o Linux arredonda o tempo de dormência para cima do valor desejado, e em múltiplos de 10 milisegundos. Se for requisitado para o processo dormir 30 milisegundos, o Linux o programará para esperar 3 *ticks*, o que dá realmente 30 ms, sem contar o tempo antes do primeiro *tick* ser disparado. Mas se quisermos uma latência de 30,001 ms, por exemplo (30001 microssegundos), o Linux arredondará para quatro *ticks*. Assim, já possuímos um erro de 10 ms, que deve ser somado a hipótese da chamada ocorrer logo após um *tick*, ou seja, a contagem de *ticks* só iniciará a partir do próximo, o que demorará quase 10 ms. Assim, temos um erro máximo de 20 milisegundos (contando que o kernel não perca nenhum *tick*).

Abaixo, uma tabela com os resultados de chamadas da função **nanosleep()**, que é equivalente a função **usleep()** com relação ao desempenho:

Tempo de dormência em microssegundos	200000	200001	201000	205000	209000	210000
Resultado de 10 amostras, em microssegundos	208546	217782	213108	213208	212873	212307
	207297	211766	217309	210830	216833	217204
	209646	210273	215632	211206	218644	217117
	205151	218846	217284	218367	210670	216361
	207156	213399	219110	213606	218889	219933
	202444	212059	214145	210846	217215	211203
	200861	219938	216183	217839	219300	215805
	202471	217321	211216	218474	218935	212470
	200654	216847	219570	218914	217179	211897
	209940	214108	214727	212669	216745	213202

No entanto, existem certas artimanhas que nos permitem melhorar a precisão dos sleeps. Primeiro, é possível fugir das políticas dos *ticks* para as funções **nanosleep()** e **usleep()**. Para tal, precisamos configurar o processo que as chama como um processo de alta prioridade. Assim, para valores de dormência abaixo de 2 milisegundos [5], essas duas funções deixarão de depender dos *ticks* do Linux e executarão o que chamamos de *busy awaiting*, que faz com que o processador fique executando um loop durante a faixa de tempo definida na função. Assim, elas funcionam com uma alta precisão, na ordem dos microssegundos, para esses valores de dormência abaixo dos 2 milisegundos. Porém, durante a execução dessas funções, nesse modo, o processador fica ocupado, não permitindo a execução de outro processo. Devemos lembrar que para alterar a prioridade do processo este deve ser executado em modo *super-usuário*.

Para setar o processo como um processo de alta prioridade, usamos a função **sched_setscheduler()**, que é declarada como **int sched_setscheduler (pid_t pid, int policy, const struct sched_param *p)**, onde *pid* é o número de processo a ser modificado, que no caso do processo atual é definido como 0, *policy* é o tipo de política de prioridade, e **p* um ponteiro para uma estrutura do tipo *sched_param* que conterá as informações a respeito da prioridade desejada. A biblioteca que contém a estrutura *sched_param* é a s*ched.h.* Com as seguintes linhas, configuramos um processo para rodar em alta prioridade:

```
struct sched_param realtime;
realtime.sched_priority=sched_get_priority_max (SCHED_RR)-1;
sched_setscheduler(0,SCHED_RR,&realtime);
```

As políticas de prioridade *SCHED_FIFO* e *SCHED_RR* são destinadas a aplicações de tempo real. A política *SCHED_OTHER* é padrão para processos normais do Linux. Em nosso caso, optamos por usar a *SCHED_FIFO*, que só libera o processamento quando o processo terminar, for bloqueado ou interrompido por um processo de prioridade maior. A função **sched_get_priority_max()** destina-se a obter o valor máximo de prioridade da atual política disponível que fará com que o processo seja de prioridade máxima. Corresponde a um inteiro de 0 a 99. Assim, a função **sched_setsheduler()** configura o processo com esse valor e fará com que o Linux entenda que esse possui a máxima prioridade no momento. Abaixo, uma tabela com amostras da chamada **nanosleep()** com prioridade máxima (o resultado pode variar de PC para PC, pois depende também do poder de processamento da máquina. Esses testes foram realizados em um Pentium de 120 MHz):

Tempo de dormência em microssegundos	10	100	500	1000	2000
Resultado de 5 amostras, em microssegundos	36	126	524	1023	2018
	37	127	524	1022	2017
	36	126	524	1023	2019
	37	126	525	1022	2018
	37	127	524	1023	2017

Podemos notar que existe um pequeno erro, quase constante, para cada valor medido. Tendo conhecimento da forma como o erro aparece, podemos realizar correções na chamada da função. Apesar do erro, conseguimos uma boa precisão com prioridade alta para valores de dormência abaixo dos dois milisegundos. Caso o tempo de dormência for 2001 microssegundos, o Linux passará a utilizar a política dos *ticks* e os resultados serão semelhantes aos apresentados na primeira tabela. No entanto, diversas aplicações de tempo real necessitam uma precisão de microssegundos para tempos de inatividade superiores aos dois milisegundos. Para tanto, podemos criar uma função que consiste em um laço de “sleeps” que nos permite aumentar essa faixa de tempo. Abaixo, um exemplo de função que pode realizar tal feito (*sleeptime* é o valor a dormir, e pode ser de 0 a 999999 microssegundos. Valores maiores poderiam ser obtidos com pequenas modificações):

```
#include <time.h>
#include <sched.h>

void NSleep (long unsigned sleeptime) {
      long i;
      struct timespec req,rem;
      struct sched_param realtime;
      realtime.sched_priority=sched_get_priority_max
(SCHED_FIFO)-1;
      sched_setscheduler (0,SCHED_FIFO,&realtime);
      req.tv_sec=0;
      req.tv_nsec=(sleeptime % 1000)*1000;
      nanosleep (&req,&rem);
      req.tv_nsec=1000000;
      for (i=0;i<sleeptime/1000;i++) {
            nanosleep(&req,&rem);
      }
}
```

O parâmetro passado à função **NSleep()** é o tempo de dormência em microssegundos. Por exemplo, se passarmos para a função um período de 5500 microssegundos, o primeiro **nanosleep()** se encarregará de dormir as frações de milisegundos, que nesse caso é 500 (*req.tv_nsec=(sleeptime % 1000)*1000*) e o loop será executado 5 vezes, cada um durando teoricamente 1 milisegundo. Assim, teremos um tempo total de 5,5 ms ou 5500 microssegundos. Devemos ter em mente também que o processo será rodado o tempo todo com alta prioridade. Assim, outros processos estarão impedidos de ocupar o processador. Para evitar isso poderíamos modificar o loop, setando uma prioridade normal após o **nanosleep()** e uma prioridade máxima antes, ficando o loop dessa forma:

```
      For (i=0;i<sleeptime/1000;i++) {
            sched_setscheduler (0,SCHED_FIFO,&realtime);
            nanosleep (&req,&rem);
            sched_setscheduler (0,SCHED_OTHER,&normal);
      }
```

Onde *normal* é declarado como *struct sched_param normal*, e definido como *normal.sched_priority=0* (o valor 0 corresponde à prioridade normal). Essa alteração diminui a precisão, e é necessária apenas se houver a necessidade de executar outro processo simultaneamente. Abaixo, uma tabela com amostras de diferentes chamadas da função **NSleep()** original, sem as alterações citadas acima:

Tempo de dormência em microssegundos	**5000**	**10000**	**50000**	**200000**	**900000**
Resultado de 5 amostras, em microssegundos	5025	10010	49890	199447	897300
	5026	10009	49889	199428	897281
	5026	10014	49888	199428	897285
	5026	10010	49888	199443	897304
	5025	10010	49886	199430	897302

Ao analisarmos esses resultados, podemos tirar várias conclusões sobre os erros da função: existem dois tipos de erros: um erro acumulativo, que aumenta ao longo do tempo, e um erro de *offset*, que permanece constante ao longo do tempo. Mas esses erros podem ser corrigidos. Como o resultado desses testes depende da máquina em que se está executando, não podemos definir uma correção universal. Podemos sim criar um programa que calcule a partir de cada máquina qual será a correção.

O erro de *offset* pode ser medido ao mandarmos a função **NSleep()** dormir 0 microssegundos. Em nossos testes, ela dormiu na realidade 42 microssegundos (esse e os outros testes podem ser realizados com auxilio da função **gettimeofday()** ou da **lrdtsc()**, já mostradas anteriormente. No caso da **lrdtsc()** deve-se realizar a conversão para tempo, (já que ela nos retorna o número de sinais de clock passados até então). Esse erro é um erro constante, independente de qual valor de tempo for passado para a função, e se deve ao processamento das instruções da mesma. Assim, para corrigi-lo, basta calcular quanto tempo à função **NSleep()** demora a ser executada com parâmetro nulo, e subtrair esse valor do total que será passado como tempo de dormência em sua próxima chamada.

O outro erro é o erro acumulativo. Esse erro não é um erro simplesmente linear. A melhor forma de corrigi-lo seria amostrando vários resultados de diversas chamadas, obtendo-se um gráfico do valor pedido pelo valor retornado, e a partir desse, através de uma regressão linear, obter uma equação que descreva o comportamento desse erro. Assim, ao ser feita uma chamada da função **NSleep()**, corrigiríamos o valor a ser passado com base nesse erro.

Uma forma mais simples é pela aproximação linear. Após a correção de *offset*, calculamos quanto ela retorna ao pedirmos uma dormência de 200000 (200000 – erro de *offset*) microssegundos por exemplo. Assim, possuímos uma relação entre o que se pediu e o que foi retornado, e a cada próximo valor requerido haverá uma correção através de uma simples regra de três. Abaixo, um pequeno programa que realizaria essas correções e que depois chama a função **NSLeep()**. Note que os valores do erro de *offset* e o valor da chamada de 200000 microssegundos foram previamente calculados, já que esses valores dependem de máquina para máquina.

```
void XSleep (long long unsigned sleeptime) {
      sleeptime-=41;
      sleeptime=200000*sleeptime/(199429-41);
      NSLeep (sleeptime);
      return;
}
```

Onde 41 é o erro de *offset* em microssegundos em nossos testes e 199429 o valor médio em microssegundos da chamada de 200000 microssegundos da função **NSleep**. A terceira linha do código corresponde a regra de três. Abaixo, os valores retornados para diversas chamadas:

Tempo de dormência em microssegundos	**5000**	**10000**	**50000**	**200000**	**900000**
Resultado de 5 amostras, em microssegundos	5000	10001	50005	200000	899998
	5000	10001	50009	200006	899989
	5001	10004	50004	200004	900045
	5000	10004	50004	200023	900009
	5001	10004	50004	200046	900067

Podemos notar que esses são os melhores resultados obtidos até agora. Podem sofrer ainda um ajuste fino, alterando os parâmetros de correção, para se conseguir uma pequena melhora.

Nessa última forma de criar uma dormência no processo, conseguimos valores precisos de tempo, mas notamos que o processo roda todo em prioridade alta, e a função **nanosleep()** em alta prioridade cria o *busy awaiting*, ou seja, o processador fica ocupado durante todo o processo. Uma solução é criar um função mista, que utilize o **nanosleep()** sem prioridade alta para liberar o processador e a função XSleep para promover uma correção precisa do tempo restante de dormência. Por exemplo, ao mandarmos dormir 55000 microssegundos, fazemos um chamada normal de **nanosleep()** com 40 milisegundos. Sabemos que ele dormirá entre 40 e 50 milisegundos. Se medirmos precisamente esse tempo de dormência, saberemos quanto falta, que será um valor entre 40 e 55 milisegundos. Então usamos a função **XSleep()** para esse restante. O programa abaixo realiza esse feito:

```
      #include <unistd.h>
      #include <time.h>
      #define FreqCPU 119755

      static inline long long unsigned lrdtsc (void) {
            long long unsigned time;
            __asm__ __volatile__ ("rdtsc":"=A"(time));
            return time;
      }

      void ZSleep (long unsigned timex) {
            struct timespec req,rem;
            long long unsigned clock_1, clock_2, clock_delay,
clock_total;
            long unsigned time_req, time_rem;
            clock_1=lrdtsc();
            time_req=(timex/10000)*10000-10000;
            req.tv_sec=0;
            req.tv_nsec=time_req*1000;
            nanosleep (&req,&rem);
            clock_2=lrdtsc();
            time_rem=timex-(clock_2-clock_1)*1000/FreqCPU;
            XSleep (time_rem);
            return;
      }
```

Em *FreqCPU* é definida a freqüência do processador. Em nosso caso, foi usado um Pentium 120 MHz, mas sua freqüência real é de 119,755 MHz. Esse valor é calculado

na inicialização do Linux e pode ser obtido no Linuxconf ou em */proc/cpuinfo*. Podem ser necessários ajustes nas correções para que os resultados mantenham a precisão da função Zsleep, ou até melhorem. Os resultados obtidos seguem abaixo:

Tempo de dormência em microssegundos	**5000**	**10000**	**50000**	**200000**	**900000**
Resultado de 5 amostras, em microssegundos	4998	10001	50001	200005	900025
	4999	9998	50004	199999	900001
	5004	10000	50000	200004	900004
	4996	9998	50002	200002	900004
	4998	10002	49996	200001	900005

A precisão é semelhante à da função **XSleep()**, com a diferença de saturar o processador por 20 milisegundos no máximo.

6 Sinais, Temporizadores e Alarmes

6.1 Os Sinais

Os sinais são interrupções de software que são enviadas a processos pelo sistema operacional de forma assíncrona para informá-los de eventos no sistema, como erros de entrada e saída ou violação de memória e / ou para forçar um processo executar determinada rotina. Os sinais são tratados de três maneiras diferentes por um processo (com exceção do sinal *SIGKILL* e *SIGSTOP*): podem ser simplesmente ignorados; podem ser interceptados, fazendo com que o processo execute um procedimento específico para depois retornar a sua execução normal, ou podem ainda finalizar a execução do processo ou executar outra ação *default* de cada sinal [6]. Ao contrário das interrupções normais e exceções, os sinais são visíveis em processos do usuário [1].

Cada sinal é identificado por um número inteiro, associado a um mnemônico. A lista de sinais acessíveis está em *signal.h*. Os sinais para Linux em PCs (i386) são:

Nº	Sinal	Ação	Significado
1	SIGHUP	A	Corte, é emitido aos processos associados a um terminal quando este se “desconecta”'.
2	SIGINT	A	Interrupção, é emitido aos processos do terminal quando as teclas de interrupção (ctrl+c) são acionadas.
3	SIGQUIT	C	Abandono, é emitido aos processos do terminal quando as teclas de abandono (ctrl+d) são acionadas.
4	SIGILL	C	Instrução ilegal, é emitido quando uma instrução ilegal é detectada.
5	SIGTRAP	C	Sinal emitido em breakpoints de debugs.
6	SIGABRT	C	Abortar, emitido pela chamada abort().
7	SIGBUS	A	Erro de barramento.
8	SIGFPE	C	Erro de cálculo em ponto flutuante, assim como no caso de um número em ponto flutuante em formato ilegal. Indica sempre um erro de programação.
9	SIGKILL	AEF	Destruição para um processo. Esse sinal não pode ser ignorado nem interceptado.
10	SIGUSR1	A	Primeiro sinal disponível ao usuário, utilizado para a comunicação entre processos.
11	SIGSEGV	C	Violação da segmentação, ou seja, tentativa de acesso a um dado fora do domínio de endereçamento do processo.
12	SIGUSR2	A	Segundo sinal disponível ao usuário, é utilizado para a comunicação entre processos.
13	SIGPIPE	A	Notificação de escrita sobre um pipe não aberto em leitura.
14	SIGALRM	A	Sinal emitido quando o temporizador de um processo para (o temporizador é colocado em funcionamento através da primitiva alarm())
15	SIGTERM	A	Finalização por software, é emitido quando o processo termina de maneira normal. Também pode ser usado quando o sistema deseja finalizar a execução de todos os processos ativos (o que o Linux faz quando é desligado).
16	SIGSTKFLT	A	Indicativo de erro da pilha do co-processador.
17	SIGCHLD	B	Sinal enviado a um processo pai quando o filho termina.
18	SIGCONT	G	Continua uma execução de um processo, se o mesmo estava interrompido.

19	SIGSTOP	DEF	Interrupção de um processo.
20	SIGTSTP	D	Interrupção da digitação no terminal do usuário.
21	SIGTTIN	D	Entrada do terminal para o processo em segundo plano.
22	SIGTTOU	D	Saída do terminal para o processo em segundo plano.
23	SIGURG	B	Condição urgente em um socket.
24	SIGXCPU	A	Tempo de processador esgotado.
25	SIGXFSZ	A	Tamanho de arquivo excedido.
26	SIGVTALRM	A	Sinal do timer quando opera no modo *ITIMER_VIRTUAL*.
27	SIGPROF	A	Sinal do timer quando opera no modo *ITIMER_PROF*.
28	SIGWINCH	B	Alteração no tamanho de uma janela.
29	SIGIO	A	Entrada / Saída (I/O) não possível.
30	SIGPWR	A	Problema no suprimento de energia elétrica.
31	SIGUNUSED	A	Sinal não usado.

Sendo que as ações são:

- A: A ação padrão é terminar o processo.
- B: A ação padrão é ignorar o sinal.
- C: A ação padrão é terminar o processo e mostrar o arquivo core gerado.
- D: A ação padrão é parar o processo.
- E: O sinal não pode ser capturado.
- F: O sinal não pode ser ignorado.
- G: A ação padrão é continuar o processo, se o mesmo estava parado.

Quando um sinal é enviado mas ainda não foi recebido ele é chamado de sinal pendente. Ao mesmo tempo, apenas um sinal pendente de cada tipo pode existir para um processo. Caso um outro sinal do mesmo tipo do sinal pendente for enviado, ele será simplesmente descartado. Um sinal pode ficar pendente por tempo indeterminado. Isso porquê um sinal só é recebido por um processo quando esse estiver sendo executado. Um sinal também pode estar selecionado como bloqueado por um processo, ou seja, ele só será recebido quando o mesmo for retirado da lista de sinais bloqueados. Quando um processo executa uma rotina de manipulação de um sinal que fora recebido, ele automaticamente bloqueia esse sinal até que o manipulador acabe. A diferença entre um sinal bloqueado e um sinal ignorado é que o sinal bloqueado nunca é recebido enquanto o mesmo estiver na lista dos sinais bloqueados de um processo. O sinal ignorado simplesmente é recebido sem executar nenhuma ação.

Em Linux, existem várias chamadas de sistema para o tratamento e envio de sinais. São as mais conhecidas: **signal()**, **sigaction()**, **kill()** e **alarm()**.

6.2 Função Signal

A função mais simples para a recepção de sinais é a chamada **signal()**. Ela é declarada como **void (*signal (int signum, void (*handler) (int))) (int)**, e necessita da biblioteca *signal.h*. Apesar dessa chamada parecer complicada, ela necessita de apenas dois parâmetros: o primeiro, *signum*, é o número do sinal a ser recebido, e o segundo, é um ponteiro, *handler*, para uma função que será executada quando o sinal for interceptado e para a qual será passado como parâmetro um inteiro. Caso *handler* for definido como *SIG_IGN*, ao invés de apontar para uma função, o sinal será ignorado. Caso seja *SIG_DFL*, a ação executada será a default. O valor de retorno de **signal()** é o ponteiro para uma função com um inteiro de parâmetro e que não

retornará nada (void). Abaixo, um exemplo que ilustra o uso dessa chamada de sistema [8]:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

int ctrl_c_count = 0;
void (*old_handler)(int);

void ctrl_c (int signum) {
        (void) signal (SIGINT, ctrl_c);
        ++ctrl_c_count;
}

main() {
        int c;
        old_handler=signal(SIGINT,ctrl_c);
        while ((c=getchar())!='\n');
        printf ("ctrl-c count = %d\n", ctrl_c_count);
        (void) signal (SIGINT, old_handler);
        for (;;);
}
```

Esse programa recebe caracteres digitados no terminal até o pressionamento da tecla *ENTER*. Na linha *old_handler=signal (SIGINT ,ctrl_c)*, é armazenado em *old_handler* o endereço da antiga função de tratamento do sinal *SIGINT*, e essa passa a ter um novo tratamento, que será executado pela função *ctrl_c()*. Assim, cada vez que o usuário pressionar as teclas *ctrl+c*, ele passará a executar a função *ctrl_c* que nada mais é que um contador. Após ser pressionada a tecla *ENTER*, o processo sai desse loop, e redefine o tratamento para o sinal *SIGINT*, em *signal (SIGINT, old_handler)*, fazendo com que volte para a sua ação anterior pontada por *old_handler*. Devemos notar que na chamada de *ctrl_c*, a chamada *signal* é realizada novamente. Isso porque a cada chamada de *signal*, o tratamento do sinal é reiniciado e volta para a sua ação *default* (no caso de *SIGINT*, finalizar o processo).

6.3 Função Sigaction

A próxima função para a recepção de sinais é a **sigaction()**. Pelo POSIX, cada processo tem uma mascara de sinais que contém um conjunto de sinais que estão no momento bloqueados. Caso esse processo receba um sinal que está definido nessa máscara, ele será adicionado ao conjunto de sinais pendentes e será recebido quando o bloqueio for removido.

Quando um sinal é enviado ele é automaticamente adicionado à máscara de sinais do processo que o esta recebendo para mais adiante quando este estiver sendo tratado e chegarem outros sinais do mesmo tipo, eles estarão bloqueados. Quando um manipulador de sinal retorna sem erros, a máscara de sinais é restaurada com seu valor anterior a execução desse.

A função **sigaction()** é declarada como **int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact)**, onde *signum* é o número do sinal a ser tratado, *act* um ponteiro para uma estrutura do tipo *sigaction* que contém as informações sobre a ação que será executada na interrupção do sinal e *oldact* uma estrutura do mesmo tipo onde serão gravadas informações sobre a ação prévia de

tratamento desse sinal (como vimos, a chamada **signal()** cria um ponteiro para essa antiga função). Essa estrutura, declarada em *signal.h* tem a seguinte forma:

```
struct sigaction {
        void (*sa_handler)(int);
        sigset_t sa_mask;
        unsigned long sa_flags;
        void (*sa_restorer)(void);
}
```

Sa_handler é o ponteiro para a função de manipulação do sinal, ou pode ser definido como *SIG_DFL* para executar a ação default do sinal ou *SIG_IGN* para ignora-lo. *Sa_mask* é uma estrutura do tipo *sigset_t* que contém as informações sobre a máscara de sinais. Ela é declarada como:

```
struct sigset_t {
        unsigned long sig[2];
}
```

A variável *sig* é composta por dois elementos. O primeiro liga flags para indicar quais dos 31 sinais normais estão bloqueados (é uma variável long, ou seja, de 32 bits). O segundo faz o mesmo para os sinais de tempo real, que serão citados adiante. *Sa_flags* é uma combinação de bits que atribui propriedades ao tratamento do sinal. Pode ter os seguintes mnemônicos, além de outros:

Nome da flag	Função
SA_NOCLDSTOP	Não envia o sinal SIGCHLD quando o processo está parado.
SA_NODEFER, SA_NOMASK	Não mascara o sinal quando está sendo executada a função de tratamento para a mesma.
SA_RESETHAND, SA_ONESHOT	Reseta para o tratamento default de um sinal após o recebimento.
SA_ONSTACK	Usa uma pilha alternativa para o tratamento do sinal.
SA_RESTART	As chamadas de sistemas interrompidas são automaticamente reiniciadas.
SA_SIGINFO	Passa informações adicionais para o tratador de sinal.

Por exemplo, *SA_ONESHOT* realizará o mesmo comportamento da chamada **signal()**, que será de retornar a ação *default* após o recebimento de um sinal, ou *SA_NOMASK*, que ignorará a presença de máscaras de sinais.

Sa_restorer está obsoleto e não é mais utilizado. Existem ainda as funções de manipulação da máscara de sinais bloqueados. A primeira ;é a **int sigemptyset (sigset_t *mask)**, que cria o ponteiro *mask* para uma máscara de sinais do tipo *sigset_t* vazia, onde nenhum sinal estaria bloqueado. A função **int sigfillset (sigset_t *mask)** cria uma máscara onde todos os sinais estariam bloqueados. As funções **int sigaddset (sigset_t *mask, int signum)** e **int sigdelset (sigset_t *mask)** adicionam e retiram respectivamente o sinal de número *signum* dessa máscara de sinais. Já a função **int sigismember (sigset_t *mask, int signum)** verifica se o sinal *signum* está setado como bloqueado em *mask*.

A função **sigprocmask**, declarada como **int sigprocmask (int fcn, const sigset_t *mask, sigset_t *oldmask)**, é usada para alterar o valor da máscara de sinais bloqueados do processo de acordo com o valor declarado em *fcn* e com os sinais indicados em *mask*. Caso *fcn* for *SIG_BLOCK*, a máscara atual será composta pela união dos sinais bloqueados atuais e os definidos em *mask*. Caso seja

SIG_UNBLOCK, os sinais em *mask* serão removidos do atual conjunto de sinais bloqueados e caso seja *SIG_SETMASK*, os sinais bloqueados passarão a ser os definidos em *mask*. **Sigprocmask** realmente altera a lista dos sinais que estão realmente bloqueados ou não pelo processo. As anteriores, como **sigfillset()**, **sigemptyset()**, **sigaddset()** e **sigdelset()** apenas alteram a estrutura *mask* que será carregada por **sigprocmask()**.

Há também a função **int sigpending (sigset_t *mask)**, onde *mask* apontará para a lista de sinais pendentes, e a função **int sigsuspend (const sigset_t *mask)**, que substitui a máscara de sinais bloqueados atual pela definida em *mask* e suspende o processo até que o sinal seja recebido.

Outra função é a **pause()**. Declarada **int pause (void)**, em *unistd.h*, gera uma pausa simples. Não faz nada nem espera nada em particular, apenas libera o processo quando algum sinal chega. Geralmente esse sinal esperado é o sinal de alarme gerado por **alarm()**. Essa primitiva sempre retorna o valor –1.

6.4 Função Kill

Para se emitir um sinal, faz-se o uso da primitiva **kill()** declarada em *signal.h* como **int kill (pid,sig)**, onde *pid* é o número do processo a se enviar o sinal e *sig* o número do sinal a ser enviado. A função **kill()** retorna 0 se o sinal foi enviado e –1 se não. Caso *sig* for igual a zero, nenhum sinal será enviado, e o valor de retorno informará se existe um processo com o *pid* declarado ou não. Se *pid* for zero, o sinal será enviado a todos os processos do mesmo grupo do emissor. Caso o *pid* for igual a 1, podem ocorrer duas situações: o sinal será enviado a todos os processos, exceto aos processos do sistema e ao processo que envia o sinal se o processo que o chamou pertencer ao super-usuário, ou senão o sinal será enviado a todos os processos com ID do usuário real igual ao ID do usuário efetivo do processo que envia o sinal. Essa última é uma forma de matar todos os processos dos quais se é proprietário, independente do grupo de processos a que eles pertençam. Caso *pid* for menor que 0, o sinal será enviado a todos os processos para os quais o ID do grupo de processos (*pgid*) for igual ao valor absoluto de *pid*.

6.5 Função Alarm

Há também a primitiva **alarm()**, declarada em *unistd.h* como **unsigned int alarm (unsigned int secs)**, para o envio do sinal específico *SIGALARM* após um intervalo de tempo em segundos passado no argumento *secs*, funcionando como um alarme. O tratamento desse sinal deve estar previsto no programa, senão o processo será finalizado ao recebê-lo. Caso se faça a chamada com *secs* igual a zero, ela retornará o valor de tempo restante para o envio do sinal. Abaixo, um exemplo de implementação da função **sleep()** usando essas duas chamadas:

```
void nullfcn() { }
void sleep (int secs) {
        signal (SIGALRM,nullfcn);
        alarm (secs);
        pause();
}
```

Nesse exemplo, cria-se uma função **nullfcn()** que não faz absolutamente nada, mas é necessária para o uso de **signal()**. O alarme é armado, e a função fica bloqueada por **pause()**. Quando o alarme disparar, o sinal será recebido e a função terminada. Existem ainda em Linux outras funções para a manipulação de sinais, como a **raise()** que funcionamento semelhante a **kill()**, mas que manda sinais apenas para o processo que a chamou.

6.6 Sinais de Tempo Real

O Linux também oferece suporte ao chamados sinais de tempo real. A diferença entre estes e os sinais convencionais apresentados é que sinais de tempo real, quando são enviados e já há um sinal do mesmo tipo pendente, entram para um fila até serem recebidos, ao invés de serem descartados como são os sinais convencionais nessa situação. Como vimos, os sinais convencionais são descritos por números entre 1 e 31. Os sinais de tempo real ocupam as posições de 32 a 63. As chamadas de sistema para sinais de tempo real são: **rt_sigaction()**, **rt_sigpending()**, **rt_sigprocmask()**, **rt_sigqueueinfo()** e **rt_sigtimedwait()**. Com exceção das duas últimas, elas são equivalentes às chamadas já apresentadas (**sigaction()**, **sigpending()**, **sigprocmask()** e **sigqueueinfo()** respectivamente). A chamada **rt_sigqueueinfo()** obtém informações sobre a fila de sinais pendentes e a chamada **rt_sigtimedwait()** é semelhando à **sigsuspend()**, mas faz com que o processo continue suspenso apenas por um determinado período de tempo.

6.7 Implementação de Temporizadores e Alarmes

Na implementação de temporizadores ou alarmes, utilizamos duas funções, além das já descritas na manipulação de sinais. A primeira é a **setitimer**, declarada como **int setitimer (int which, const struct itimerval *value, struct itimerval *ovalue)** em *sys/time.h*, que gera um período de temporização, descrito em *value*, no qual um contador será decrementado e ao final deste será emitido um sinal. De acordo com o valor de *which*, pode trabalhar de três formas: *ITIMER_REAL*: decremento em tempo real, e envio do sinal *SIGALRM* após a finalização; *ITIMER_VIRTUAL*: decremento apenas quando o processo esta sendo executado, e o sinal enviado é o *SIGVTALRM*; *ITIMER_PROF*: decremento quando o processo está sendo executado e também quando o sistema esta executando em nome do processo. Assim como *ITIMER_VIRTUAL*, ele é utilizado para medir o tempo gasto por uma aplicação do usuário no sistema. O sinal enviado é *SIGPROF*. *Itimerval* tem a seguinte estrutura:

```
struct itimerval {
        struct timeval it_interval;
        struct timeval it_value;
}
```

Onde *it_interval* e *it_value* são estruturas do tipo *timeval*, já vista:

```
struct timeval {
        long tv_sec;
        long tv_usec;
}
```

It_interval é valor de tempo periódico no qual o timer fará seus disparos periódicos. *It_value* é o valor inicial do contador onde a partir do qual será feito o primeiro disparo. Caso *ovalue seja* diferente de zero, o valor do antigo timer será gravado nele. A outra função utilizada para temporizadores é a **getitimer()**, declarada como **int getitimer(int which, struct itimerval *value)** e preenche a estrutura *value* com o valor do timer atual.

Sobre o comportamento temporal dessas funções, devemos ressaltar que elas adotam a mesma política de *ticks* do Linux, ou seja, possuem resolução de 10 milisegundos. As funções do tipo sleep apresentadas nas seções anteriores possuem o funcionamento interno semelhante ao desses tipos de timers, fazendo o uso de sinais. Abaixo, um exemplo do uso de **setitimer()** com a recepção do sinal enviado por esta [10]:

```
#include <sys/types.h>
#include <sys/time.h>
#include <stdio.h>

struct timeval tv;
struct timezone tz;
long otime_usec, otime_sec, ntime_usec, ntime_sec, dif_sec,
dif_usec;

void nullhandler(){
      gettimeofday(&tv,&tz);
      ntime_sec=tv.tv_sec;
      ntime_usec=tv.tv_usec;
      dif_sec=ntime_sec-otime_sec;
      dif_usec=ntime_usec-otime_usec;
      if (dif_usec<0) {
            dif_usec+=1000000;
            dif_sec-=1;
      }
      printf ("Diferença de tempo: %d s %d us\n", dif_sec,
      dif_usec);
      otime_sec=ntime_sec;
      otime_usec=ntime_usec;
}

int main(){
      sigset_t block_these, pause_mask;
      struct sigaction s;
      struct itimerval interval;
      long secs, usecs;

      gettimeofday(&tv,&tz);
      otime_sec=tv.tv_sec;
      otime_usec=tv.tv_usec;

      sigemptyset (&block_these);
      sigaddset (&block_these, SIGALRM);
      sigprocmask (SIG_BLOCK, &block_these, &pause_mask);

      sigemptyset (&s.sa_mask);
      s.sa_flags=0;
      s.sa_handler=nullhandler;
      sigaction (SIGALRM, &s, NULL);
```

```
        interval.it_value.tv_sec=5;
        interval.it_value.tv_usec=0;
        interval.it_interval.tv_sec=1;
        interval.it_interval.tv_usec=0;

        setitimer (ITIMER_REAL, &interval, NULL);

        while (1){
              sigsuspend (&pause_mask);
        }
}
```

Nesse programa, primeiramente cria-se a função *nullhandler*, que imprime na tela o tempo transcorrido entre a sua última chamada e a atual. Ela é chamada a cada recebimento do sinal *SIGALARM*, que é enviado por temporizador a cada 1 segundo, após cinco segundos iniciais. Devemos notar que no tratamento do sinal foi criada uma máscara para bloquear o sinal *SIGALARM*, que poderá ser recebido apenas quando a função entrar em *loop* infinito e for feita a chamada a **sigsuspend()**.

7. Tarefas Periódicas

Uma grande variedade de processos e sistemas de controle atuais necessita, em nível de computação, da realização de tarefas periódicas. Devido a isso, esforços significativos têm sido feitos na formalização de modelos para sistemas periódicos de tempo real.

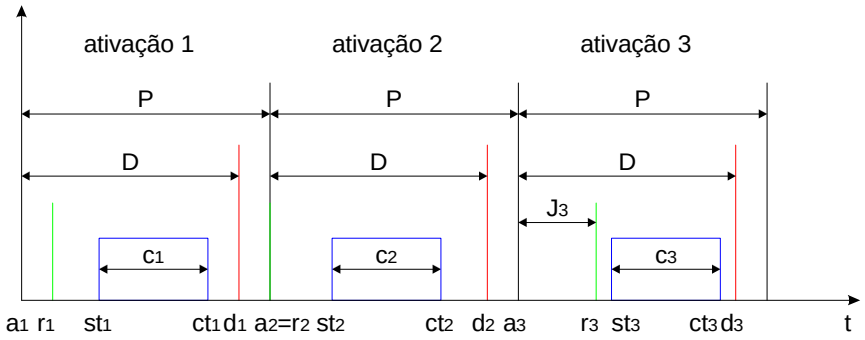
Uma tarefa, ou processo, pode ser conceituado como uma unidade seqüencial de processamento, que concorre com outras tarefas sobre um ou mais recursos computacionais de um sistema. Assim, para a realização de eventos de tempo real, esse comportamento deve ser o mais previsível e conhecido possível.

De acordo com a freqüência em que ocorrem, as tarefas podem ser classificadas de duas formas: tarefas periódicas e tarefas aperiódicas. As tarefas aperiódicas são acionadas de forma aleatória, em resposta a eventos internos ou externos do sistema. As tarefas periódicas são acionadas em intervalos constantes e pré-definidos. Esse intervalo entre cada chamada da tarefa é chamado de período (será referenciado como P). Podemos atribuir a uma tarefa de tempo real várias características [10]:

- Computation Time (C): É o tempo de computação (uso real do processador e outros recursos do sistema) para a execução completa da tarefa.
- Start Time (st): É o instante em que a tarefa "ganha" o processador e inicia o processamento.
- Completion Time (ct): É o instante de tempo em que a tarefa é concluída. Caso a tarefa, ao ganhar o processador, é processada sem interrupções, ou seja, sem perder os recursos do sistema para outra tarefa, o tempo de computação (C) corresponderá ao intervalo entre o tempo de inicio *st* e o tempo de conclusão *ct*.
- Arrival Time (a): É o tempo em que o escalonador do kernel (ferramenta responsável pela decisão de qual tarefa ganhará o processador e quando) toma conhecimento da ativação de uma tarefa.
- Release Time (r): Ou Tempo de Liberação, é o instante em que a tarefa é incluída na fila de tarefas prontas para executar pelo escalonador do sistema. Geralmente, assume-se que *a* e *r* coincidem. Ou seja, tão logo a tarefa seja acionada, ela é imediatamente liberada e inserida na fila de tarefas aptas para serem executadas (*Ready Queue*). Entretanto, isso nem sempre ocorre, como no caso de tarefas ativadas por mensagens. Por exemplo, o bloqueio na recepção de uma mensagem pode retardar a liberação da tarefa. Outro problema é quando o escalonador é ativado a cada período de tempo, como no Linux. Nesse sistema, a cada 10 milisegundos ocorre o chamado *tick* de clock, que é na realidade uma interrupção de hardware, que faz com que o Linux ative o escalonador. Essa incapacidade dele ser executado a qualquer hora, só ocorrendo a cada período de 10 ms, é chamada de granularidade do sistema. Assim fica caracterizado o próximo intervalo de tempo.
- Release Jitter (J): Intervalo de tempo correspondente entre a ativação, *a*, de uma tarefa e a sua liberação pelo escalonador do sistema.
- Deadline (D): É o prazo para a tarefa ser concluída. Ela não deve acabar depois de seu *deadline*.

Outro fator que devemos ter em mente é a diferença entre o instante de ativação, *st*, e o tempo de liberação, *r*. Uma tarefa ao ser liberada e entrar na fila das tarefas aptas pode não ser executada imediatamente, pois pode haver alguma outra tarefa com

prioridade maior sendo executada ou esperando na mesma fila. Assim, *r* e *s* podem não coincidir. Abaixo, um gráfico com uma linha do tempo exemplificando uma tarefa periódica:



ai:	instante da ativação de i	ri:	instante da liberação de i
sti:	instante do início de i	Di:	deadline de i
Ci:	tempo de computação de i	Ji:	release jitter de i
cti:	instante de término de i	P:	período da tarefa

Neste gráfico observamos a ocorrência de uma tarefa periódica, ativada três vezes. Podemos notar na segunda ativação que *a* e *r* coincidiram. Isso se deve ao fato da ativação ocorrer no mesmo instante do *tick* do Linux, onde o escalonador está sendo executado e pôde liberar a tarefa para a fila das tarefas aptas a executar. Já na primeira e terceira e ativações isso não ocorre, pois a essa não coincide com a execução do escalonador.

A implementação de uma tarefa periódica de tempo real geralmente não é um processo simples. As exigências temporais que ela necessita não são discriminadas explicitamente no código de programação. Para criarmos uma tarefa periódica, devemos ter um conhecimento consideravelmente preciso a respeito do tempo, de por exemplo, que instante é agora, qual instante a tarefa deverá ser acionada novamente e de como poderemos implementar um controle sobre esses tempos. Os relógios de hardware são um exemplo de como se pode acessar a hora atual. Entretanto, a escolha do relógio adotado na aplicação e o modo com que é feito o uso destas leituras estão atrelados à precisão desejada na aplicação (seção 2). Outro fator importante a se considerar é o de o Linux ser um sistema operacional desenvolvido para propósitos gerais e portanto não possuí uma preocupação grande em relação ao determinismo do tempo de resposta de suas funções. Assim, apesar da existência desses relógios hardware, toda operação realizada com relação ao tempo está sujeita as conseqüências da granularidade dos *ticks* do Linux. Variações do Linux, como o RTAI[11] ou o RTLinux[12] possuem chamadas específicas para a implementação de tarefas periódicas, visando a execução de tarefas periódicas de tempo real críticas.

No Linux convencional, são disponibilizados mecanismos que tentam melhorar o desempenho do sistema para tarefas de tempo real, porém são soluções que apenas melhoram o desempenho em relação ao funcionamento convencional. Uma delas é a capacidade do programador definir a política de escalonamento de sua tarefa e sua prioridade. O Linux apresenta alguns algoritmos de escalonamento para isso, como o SCHED_FIFO e o SCHED_RR, definidos pela extensão POSIX para tempo real (POSIX.1b). O algoritmo SCHED_FIFO implementa a política *first-in first-out*, ou seja, a tarefa de maior prioridade, ao chegar, será executada até o seu término. Já o algoritmo SCHED_RR implementa a política *round robin*, onde cada tarefa executa por um determinado período de tempo até ceder o processador para

outro processo de mesma prioridade. Outros processos podem utilizar a política padrão do Linux, que partilha o tempo de uso do processador entre os processos, a SCHED_OTHER. Processos utilizando os dois primeiros algoritmos possuem sempre prioridade mais alta que os processos comuns. E entre eles ainda é possível selecionador o valor da prioridade. O Linux estabelece valores máximos e mínimos de prioridade de acordo com a política de escalonamento pré-selecionada.

Como citado anteriormente, o Linux não apresenta rotinas ou chamadas que tornem uma tarefa periódica. Para tanto é necessária a criação de um laço de instruções. Mas apenas isso não é suficiente para que a tarefa ocorra em períodos pré-determinados. Necessita-se então a inclusão da chamada de sistema *sleep* (ver seção 5), que cria um período de tempo onde a tarefa fica inativa. Assim, uma tarefa periódica poderia ser programada da seguinte forma:

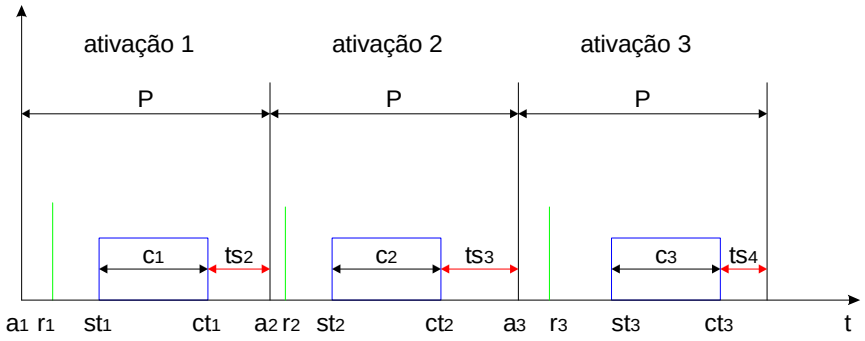
```
while (TRUE){
      sleep (ts);
      instruções;
}
```

O loop *while* garante o número infinito de iterações da tarefa, e a chamada à função **sleep(ts)** faz com que a mesma deixe o processador pelo tempo estipulado pelo parâmetro *ts*. Assim, suponha que a tarefa deve ser executada a cada 50 ms Ao final da execução passaram-se 10 ms. Assim, o valor de dormência *ts* deverá ser de 40 ms. Como o tempo de execução de cada instância da tarefa não é constante, varia conforme a carga do sistema e outros fatores, o tempo de dormência deve variar de acordo, e garantir que a freqüência de ativação da tarefa seja respeitada apesar das variações do estado do ambiente em que a tarefa executa.

Para que a tarefa seja bloqueada pelo tempo determinado por suas especificações temporais, o valor de intervalo do *sleep* é calculado a cada iteração. Baseando o cálculo no valor do período e no instante de tempo de término da instância da tarefa pode-se variar o valor do intervalo de sleep e obter a freqüência de execução desejada respeitando as demais especificações temporais. Assim, o intervalo de sleep (*ts*) calculado em cada iteração é dado por:

$$ts_i=((i-1)*P)-ct_{i-1}$$

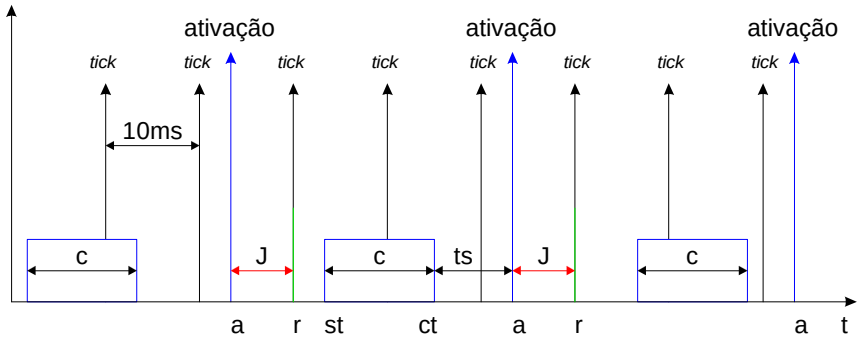
Onde i é cada repetição da tarefa. Por exemplo, no gráfico abaixo:



ai:	instante da ativação de i	ri:	instante da liberação de i
sti:	instante do início de i	ts:	tempo de dormência
Ci:	tempo de computação de i	Ji:	release jitter de i
cti:	instante de término de i	P:	período da tarefa

Podemos observar, como já fora citado, que o tempo de dormência é variável a cada iteração. O valor de ts_3 por exemplo, é calculado como sendo o instante da ativação 3 (instante em que deve ocorrer a_3, dado por (3-1)*P) menos o instante do término da ativação anterior c_2, ct_2.

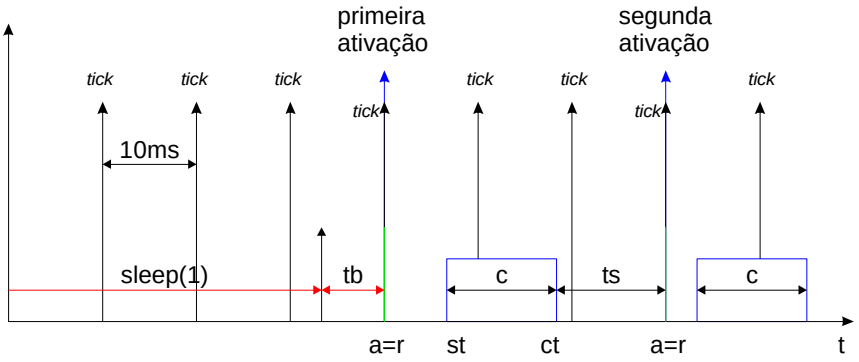
A execução exaustiva de tarefas de tempo real que coletavam valores de intervalos de tempo permitiu a observação da influência da granularidade do relógio do Linux sobre estas. Devemos lembrar que a freqüência da ocorrência dos *ticks* do Linux forma uma grade de tempo, onde o escalonador será acionado a cada 10 ms. Assim, a cada chamada do *sleep*, a tarefa é bloqueada e o processador liberado para outro processo, e aquele só poderá voltar a ser processado em uma ocorrência do *tick* do Linux, onde o escalonador o colocará na fila de tarefas prontas a serem executadas. Assim, essa grade de tempo criada pela ocorrência dos *tick* exerce forte influência sobre o controle temporal de tarefas nesse sistema operacional. Por exemplo, imagine uma tarefa periódica que deva ser executada a cada 30 ms. Na primeira execução do código, o processo tomará conhecimento da hora atual, e suponha que isso ocorra entre dois *ticks* do Linux. Assim, ele executará as instruções que devem ser periódicas e ordenará que o processo durma o tempo restante, para que 30 ms após o início as instruções sejam executadas novamente. Mas na realidade, o processo dormirá o solicitado e mais o tempo necessário até a próxima execução do *tick* do Linux. E isso ocorrerá indefinidamente, pois as duas grades de tempo, a dos *ticks* e da periodicidade da tarefa não estão sincronizadas:



a:	instante da ativação	r:	instante da liberação
ts:	tempo de dormência	C:	tempo de computação
st:	instante do início do processamento		
ct:	instante do término	J:	release jitter

Podemos observar que, devido as duas grades não estarem sincronizadas, sempre haverá o *Release Jitter* (J), que é o tempo entre o instante onde deve ser realizado a ativação e o instante onde o escalonador toma conhecimento dela e a libera.

Para tentar sincronizar essas duas grades, utiliza-se a própria chamada *sleep*, como por exemplo **sleep(1)** antes do inicio do laço de instruções. Dessa forma, a tarefa periódica só começará a ser executada após o término do *sleep*, que ocorrerá um segundo depois mais o tempo até o próximo *tick*, onde o escalonador saberá que a tarefa não deve mais ficar dormente. Assim, a primeira tomada de tempo e a construção da grade de tempo da tarefa ocorrerão logo após um *tick*, e as grades estarão praticamente sincronizadas. Outro fator importante a se considerar é que para se aproveitar esse sincronismo de grades o período de uma tarefa deverá ser um tempo múltiplo de 10 milisegundos:



a: instante da ativação
r: instante da liberação
ts: tempo de dormência
C: tempo de computação
st: instante do início do processamento
ct: instante do término
tb: restante do tempo em que o processo fica bloqueado.

O tempo *tb* é tempo restante em que o processo fica bloqueado, até o escalonador tomar conhecimento do fim do sleep e colocar novamente o processo em execução. Espera-se então que nesse instante sejam executadas as tomadas de tempo para se saber o instante atual e construir a grade de tempo ((i-1)*P) para determinar onde ocorrerá a próxima ativação, que devido a essa artimanha, será extremamente próxima a ocorrência do *tick*, o que possibilitará que a tarefa seja liberada pelo escalonador no instante da sua ativação.

Embora o Linux disponibilize também uma chamada de sistema **nanosleep()**, supostamente destinada a executar a mesma função que a sub-rotina **sleep()**, porém com precisão de nanosegundos, a chamada também funciona sob influência do relógio do Linux. O parâmetro fornecido à chamada é dado em nanosegundos, porém, o intervalo de tempo em que o sistema realmente fica suspenso é de cerca de 18 ms a mais que o tempo solicitado, conforme descrito na seção 5. Assim, subtraindo este valor do cálculo de intervalo de tempo usado como parâmetro da chamada *nanosleep*, obtém-se valores de tempo bastante próximos aos valores desejados. Com isso, garantimos que o *sleep* não durma mais do que o necessário, o que faria a tarefa perder um *tick*, acordando apenas no próximo, perdendo assim 10 milisegundos. O que acontece também, devido a esse desconto de 18 milisegundos, é que o instante de início de processamento, *st*, poderá ocorrer antes do instante teórico da ativação, *a*.

O código de uma tarefa de tempo real genérica, seria semelhante ao código apresentado abaixo:

```
(1)   #include<stdio.h>
(2)   #include<pthread.h>
(3)   #include<sys/time.h>
(4)   #include<sched.h>

(5)   #define DEZNA6 1000000

(6)   pthread_t th;
(7)   const nciclos=1000;
(8)   long t1[1000],sleept[1000],t2[1000],tp[1000];

(9)   long agora(long t){
(10)        struct timeval tvA;
(11)        struct timezone tzA;
(12)        long  now;
(13)        gettimeofday(&tvA,&tzA);
```

```
(14)        now=((tvA.tv_sec*DEZNA6)+tvA.tv_usec);
(15)        return now-t;
(16)  }

(17)  void * thread(void * arg){
(18)        int i, j, g;
(19)        long t0;
(20)        struct timespec ts;
(21)        struct timespec tsr;
(22)        struct timeval tv;
(23)        struct timezone tz;
(24)        double sum;
(25)        float avg;
(26)        sum=0.0;
(27)        j=1;
(28)        sleep(1);
(29)        if((g=gettimeofday(&tv,&tz))<0) exit(0);
(30)        t0=(tv.tv_sec*DEZNA6)+tv.tv_usec;
(31)        while(j<nciclos){
(32)              tp[j]=j*200000;
(33)              ts.tv_sec=0;
(34)              sleept[j]=(j*200000)-agora(t0)-18000;
(35)              ts.tv_nsec=sleept[j]*1000;
(36)              tsr.tv_sec=0;
(37)              tsr.tv_nsec=0;
(38)              nanosleep(&ts, &tsr);
(39)              t1[j]=agora(t0);
(40)              for(i=0;i<1000;i++){                //Algoritmo
(41)                    sum=((t1[i]/131.7)/131.45);   //da
(42)                    avg=(sum/t2[i])-sum;          //aplicação
(43)              }
(44)              t2[j]=agora(t0);
(45)              j++;
(46)        }
(47)        return NULL;
(48)  }

(49)  int main(void){
(50)        int retcode,cont;
(51)        struct sched_param mysched;
(52)        void * retval;
(53)        pthread_attr_t atributo;
(54)        pthread_attr_init(&atributo);
(55)        pthread_attr_setinheritsched(&atributo,
            PTHREAD_EXPLICIT_SCHED);
(56)        if(pthread_attr_setschedpolicy(&atributo,SCHED_FIFO)==-
1){
(57)              printf("Erro na definição da política!\n");
(58)              exit(0);
(59)        };
(60)
                  mysched.sched_priority=sched_get_priority_max(SCHED
            _FIFO)-3;
(61)        pthread_attr_setschedparam(&atributo,&mysched);
(62)        printf("Main:          Lançando          thread          0
            %d\n",mysched.sched_priority);
(63)        retcode=pthread_create(&th,&atributo,(void *)thread,"0");
(64)        if(retcode!=0)    fprintf(stderr,    "create    a    failed
            %d\n",retcode);
(65)        retcode=pthread_join(th,&retval);
```

```
(66)        if(retcode!=0)     fprintf(stderr,"join     a     failed
            %d\n",retcode);
(67)        for(cont=1;cont<nciclos;cont++){
(68)              printf("T %2d sleept: %6ld  tp: %7ld  t1: %7ld  t2:
                  %7ld  t1-tp: %7ld  t2-tp: %6ld t2-t1: %3ld t1[%d]-
                  t2[%d]:  %ld  \n",cont,  sleept[cont],  tp[cont],
                  t1[cont],   t2[cont],   t1[cont]-tp[cont],t2[cont]-
                  tp[cont],    t2[cont]-t1[cont],    cont,    cont-1,
                  t1[cont]- t2[cont-1]);
(69)        }
(70)        return 0;
(71)  }
```

Este programa cria uma tarefa periódica, com um período de 200 milisegundos, e realiza 1000 iterações (linha 7). O instante 0, onde será criada a grade de tempo é lido na linha 29 e 30. Podemos notar que, um linha antes (28), há a chamada **sleep(1)**, com o objetivo de "casar" a grade de tempo das interrupções de hardware do Linux (*ticks*) com a da tarefa. As linhas 9 a 16 declaram uma função que retorna a diferença entre o tempo fornecido como parâmetro, que será o instante 0 (*t0*) lido após o **sleep(1)** e o instante atual, em microssegundos. A partir da linha 17, até a 48 é declarada a *thread* que será encarregada da realização dessa tarefa periódica. O *loop* encontra-se entre as linhas 31 e 46. Em *tp* é definido o próximo instante de ativação (*a*), que será a cada 200 ms. Note que no primeiro período, a tarefa apenas dormirá até os 200 ms, e não será realizada nenhuma instrução que desejamos que seja periódica. Ao chegar em 200 ms, a thread retomará o processador, executará as instruções descritas entre as linhas 40 e 43, que seria a tarefa periódica em questão, e após o tempo de computação (C), tomará o instante *ct* na linha 44. O instante *st* é tomado na linha 39, apenas para fins ilustrativos. O contador é então incrementado, e o *loop* volta para seu início, onde é definido o próximo instante de ativação (assim é construída a grade de tempo da tarefa!) que será em 400 ms após o instante inicial, definido na linha 32. É calculado então o quanto a tarefa dormirá na linha 34, até acordar em 400 ms e realizar novamente a seqüência de instruções. Note a presença da subtração dos 18 ms, como descrito anteriormente. Na função *main* estão todas as chamadas e sub-rotinas para inicialização e finalização da *thread* de tempo real.

Abaixo, uma tabela comparativa, mostrando a diferença de tempo entre a ativação, *a*, e o início do processamento da tarefa, *st*, de uma execução com 20 repetições da tarefa, com o alinhamento das grades através do **sleep** e sem. Observe que, devido ao ajuste da chamada **nanosleep**, onde subtraímos 18 ms do valor de dormência passado a ela, o instante *st* ocorre antes do instante *a* (instante da ativação). Essa subtração do *sleep* pode ser modificada para encontrarmos uma diferença positiva entre *st* e *a*, mas em nossos testes, o valor 18 ms foi o que resultou em uma menor diferença, em módulo, entre esses dois instantes:

Com as grades de tempo ajustadas (tempos em microssegundos):

Instante de ativação (a)	*Instante de início (st)*	*Diferença entre a e st*
200000	199935	-65
400000	399929	-71
600000	599931	-69
800000	799931	-69
1000000	1000001	1
1200000	1199927	-73
1400000	1399925	-75
1600000	1599926	-74
1800000	1799927	-73
2000000	1999924	-76
2200000	2199925	-75
2400000	2399925	-75
2600000	2599927	-73
2800000	2799943	-57
3000000	2999936	-64
3200000	3199927	-73
3400000	3399924	-76
3600000	3599927	-73
3800000	3799927	-73
4000000	3999925	-75

Sem o ajuste das grades de tempo (tempos em microssegundos):

Instante de ativação (a)	*Instante de início (st)*	*Diferença entre a e st*
200000	194727	-5273
400000	394722	-5278
600000	594720	-5280
800000	794720	-5280
1000000	994720	-5280
1200000	1194722	-5278
1400000	1394722	-5278
1600000	1594720	-5280
1800000	1794725	-5275
2000000	1994724	-5276
2200000	2194726	-5274
2400000	2394722	-5278
2600000	2594730	-5270
2800000	2794728	-5272
3000000	2994721	-5279
3200000	3194721	-5279
3400000	3394722	-5278
3600000	3594719	-5281
3800000	3794720	-5280
4000000	3994720	-5280

Podemos notar que, apesar do alinhamento da grade não transformar o Linux em um sistema de tempo real, ele melhora consideravelmente o comportamento temporal da tarefa.

8. Conclusão

Este relatório teve por objetivo apresentar ao programador Linux as diferentes formas de se trabalhar com tempo real nesse sistema operacional. O Linux não foi projetado para essa finalidade. Apesar disso, é possível conseguir bons resultados em aplicações de automação e controle com requisitos brandos de tempo real.

Nos capítulos iniciais foi apresentado o funcionamento dos relógios de hardware de um PC, como o Real Time Clock (RTC), o Time Stamp Counter (TSC), que surgiu na arquitetura IBM com o processador Pentium da Intel, e o Programable Interval Timer (PIT). O kernel do Linux os usa em diversas tarefas internas, e estes interferem diretamente no comportamento de um processo criado por um usuário. Mas eles também podem ser acessados em nível de programação, como foi mostrado, o que possibilita boas medidas de tempo. Através de diversas chamadas de sistema, é possível o acesso a eles, e através da informação obtida, é possível medir a passagem do tempo.

Outra importante tarefa quando tratamos de aplicações de tempo real são as tarefas periódicas e a geração de intervalos de tempo pré-determinados. Isso é possível com o uso das funções da família sleep. Com elas, podemos criar intervalos de tempo onde o processo fica dormente, esperando o tempo passar. Apesar dessas chamadas serem criadas para atender expectativas de tarefas que não são de tempo real, podemos, através de algumas artimanhas apresentadas no texto, obter uma ótima precisão para tarefas que a exijam.

Também foi tratado neste relatório o uso de sinais, pois esses são a peça fundamental para a criação de temporizadores e alarmes. Assim, é possível criar processos que notifiquem ou sejam notificados sobre certos eventos após a passagem de certo período de tempo, e assim criar intervalos de tempo pré-determinados.

Neste relatório técnico, um programador pode obter informações a respeito de como funciona o tempo no Linux e como trabalhar com ele, medindo passagens de tempo, criando tarefas periódicas e outras atividades relacionadas com a grandeza "tempo". E também é possível conseguir bons resultados com relação à precisão, através da minimização de erro de medida quando se tem o conhecimento a respeito do funcionamento das diversas ferramentas para esse fim.

ALARM

Section: Linux Programmer's Manual (2)
Updated: 21 July 1993

NAME

alarm - set an alarm clock for delivery of a signal

SYNOPSIS

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

DESCRIPTION

alarm arranges for a **SIGALRM** signal to be delivered to the process in *seconds* seconds.

If *seconds* is zero, no new **alarm** is scheduled.

In any event any previously set **alarm** is cancelled.

RETURN VALUE

alarm returns the number of seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm.

NOTES

alarm and **setitimer** share the same timer; calls to one will interfere with use of the other.

Scheduling delays can, as ever, cause the execution of the process to be delayed by an arbitrary amount of time.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

CTIME

Section: Linux Programmer's Manual (3)
Updated: April 26, 1996

NAME

asctime, ctime, gmtime, localtime, mktime - transform binary date and time to ASCII

SYNOPSIS

```
#include <time.h>

char *asctime(const struct tm *timeptr);

char *ctime(const time_t *timep);

struct tm *gmtime(const time_t *timep);

struct tm *localtime(const time_t *timep);

time_t mktime(struct tm *timeptr);

extern char *tzname[2];
long int timezone;
extern int daylight;
```

DESCRIPTION

The **ctime()**, **gmtime()** and **localtime()** functions all take an argument of data type *time_t* which represents calendar time. When interpreted as an absolute time value, it represents the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time (UTC).

The **asctime()** and **mktime()** functions both take an argument representing broken-down time which is a binary representation separated into year, month, day, etc. Broken-down time is stored in the structure *tm* which is defined in <*time.h*> as follows:

```
struct tm {
        int     tm_sec;         /* seconds */
        int     tm_min;         /* minutes */
        int     tm_hour;        /* hours */
        int     tm_mday;        /* day of the month */
        int     tm_mon;         /* month */
        int     tm_year;        /* year */
        int     tm_wday;        /* day of the week */
        int     tm_yday;        /* day in the year */
        int     tm_isdst;       /* daylight saving time */
};
```

MALLOC

Section: Linux Programmer's Manual (3)
Updated: April 4, 1993

NAME

calloc, malloc, free, realloc - Allocate and free dynamic memory

SYNOPSIS

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

DESCRIPTION

calloc() allocates memory for an array of *nmemb* elements of *size* bytes each and returns a pointer to the allocated memory. The memory is set to zero.

malloc() allocates *size* bytes and returns a pointer to the allocated memory. The memory is not cleared.

free() frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc()**, **calloc()** or **realloc()**. Otherwise, or if **free(***ptr***)** has already been called before, undefined behaviour occurs. If *ptr* is **NULL**, no operation is performed.

realloc() changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If *ptr* is **NULL**, the call is equivalent to **malloc(size)**; if size is equal to zero, the call is equivalent to **free(***ptr***)**. Unless *ptr* is **NULL**, it must have been returned by an earlier call to **malloc()**, **calloc()** or **realloc()**.

RETURN VALUES

For **calloc()** and **malloc()**, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or **NULL** if the request fails.

free() returns no value.

realloc() returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from *ptr*, or **NULL** if the request fails or if size was equal to 0. If **realloc()** fails the original block is left untouched - it is not freed or moved.

FTIME

Section: Linux Programmer's Manual (3)
Updated: 24 July 1993

NAME

ftime - return date and time

SYNOPSIS

```
#include <sys/timeb.h>

int ftime(struct timeb *tp);
```

DESCRIPTION

Return current date and time in *tp*, which is declared as following:

```
struct timeb {
        time_t          time;
        unsigned short  millitm;
        short           timezone;
        short           dstflag;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

RETURN VALUE

This function always returns 0.

BUGS

Under libc4 and libc5 the millitm field is meaningful. But glibc2 is buggy and returns 0 there; glibc 2.1.1 is correct again.

HISTORY

The *ftime* function appeared in 4.2BSD.

GETITIMER

Section: Linux Programmer's Manual (2)
Updated: 5 August 1993

NAME

getitimer, setitimer - get or set value of an interval timer

SYNOPSIS

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
int setitimer(int which, const struct itimerval *value, struct
itimerval *ovalue);
```

DESCRIPTION

The system provides each process with three interval timers, each decrementing in a distinct time domain. When any timer expires, a signal is sent to the process, and the timer (potentially) restarts.

ITIMER_REAL
decrements in real time, and delivers **SIGALRM** upon expiration.

ITIMER_VIRTUAL
decrements only when the process is executing, and delivers **SIGVTALRM** upon expiration.

ITIMER_PROF
decrements both when the process executes and when the system is executing on behalf of the process. Coupled with **ITIMER_VIRTUAL**, this timer is usually used to profile the time spent by the application in user and kernel space. **SIGPROF** is delivered upon expiration.

Timer values are defined by the following structures:

```
struct itimerval {
        struct timeval it_interval;    /* next value */
        struct timeval it_value;       /* current value */
};
struct timeval {
        long tv_sec;           /* seconds */
        long tv_usec;          /* microseconds */
};
```

fills the structure indicated by *value* with the current setting for the timer indicated by *which* (one of **ITIMER_REAL**, **ITIMER_VIRTUAL**, or **ITIMER_PROF**). The element **it_value** is set to the amount of time remaining on the timer, or zero if the timer is disabled. Similarly, **it_interval** is set to the reset value. sets the indicated timer to the value in *value*. If *ovalue* is nonzero, the old value of the timer is stored there.

GETTIMEOFDAY

Section: Linux Programmer's Manual (2)
Updated: 10 December 1997

NAME

gettimeofday, settimeofday - get / set time

SYNOPSIS

```
#include <sys/time.h>
#include <unistd.h>

int gettimeofday(struct timeval *tv, struct timezone *tz);
int settimeofday(const struct timeval *tv , const struct timezone
*tz);
```

DESCRIPTION

gettimeofday and **settimeofday** can set the time as well as a timezone. *tv* is a **timeval** struct, as specified in /usr/include/sys/time.h:

```
struct timeval {
        long    tv_sec;         /* seconds */
        long    tv_usec;        /* microseconds */
};
```

and *tz* is a **timezone** :

```
struct timezone {
        int     tz_minuteswest; /* minutes W of Greenwich */
        int     tz_dsttime;     /* type of dst correction */
};
```

The use of the timezone struct is obsolete; the *tz_dsttime* field has never been used under Linux - it has not been and will not be supported by libc or glibc. Each and every occurrence of this field in the kernel source (other than the declaration) is a bug. Thus, the following is purely of historic interest.

The field *tz_dsttime* contains a symbolic constant (values are given below) that indicates in which part of the year Daylight Saving Time is in force. (Note: its value is constant throughout the year - it does not indicate that DST is in force, it just selects an algorithm.) The daylight saving time algorithms defined are as follows:

DST_NONE /* not on dst */

DST_USA /* USA style dst */

IOPERM

Section: Linux Programmer's Manual (2)
Updated: January 21, 1993

NAME

ioperm - set port input/output permissions

SYNOPSIS

```
#include <unistd.h> /* for libc5 */
#include <sys/io.h> /* for glibc */

int ioperm(unsigned long from, unsigned long num, int turn_on);
```

DESCRIPTION

Ioperm sets the port access permission bits for the process for *num* bytes starting from port address **from** to the value **turn_on**. The use of **ioperm** requires root privileges.

Only the first 0x3ff I/O ports can be specified in this manner. For more ports, the **iopl** function must be used. Permissions are not inherited on fork, but on exec they are. This is useful for giving port access permissions to non-privileged tasks.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

CONFORMING TO

ioperm is Linux specific and should not be used in programs intended to be portable.

NOTES

Libc5 treats it as a system call and has a prototype in <*unistd.h*>. Glibc1 does not have a prototype. Glibc2 has a prototype both in <*sys/io.h*> and in <*sys/perm.h*>. Avoid the latter, it is available on i386 only.

KILL

Section: Linux Programmer's Manual (2)
Updated: 14 September 1997

NAME

kill - send signal to a process

SYNOPSIS

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

DESCRIPTION

The **kill** system call can be used to send any signal to any process group or process.

If *pid* is positive, then signal *sig* is sent to *pid*.

If *pid* equals 0, then *sig* is sent to every process in the process group of the current process.

If *pid* equals -1, then *sig* is sent to every process except for the first one, from higher numbers in the process table to lower.

If *pid* is less than -1, then *sig* is sent to every process in the process group *-pid*.

If *sig* is 0, then no signal is sent, but error checking is still performed.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

ERRORS

EINVAL
An invalid signal was specified.

ESRCH
The pid or process group does not exist. Note that an existing process might be a zombie, a process which already committed termination, but has not yet been **wait()**ed for.

EPERM
The process does not have permission to send the signal to any of the receiving processes. For a process to have permission to send a signal to process *pid* it must

NANOSLEEP

Section: Linux Programmer's Manual (2)
Updated: 1996-04-10

NAME

nanosleep - pause execution for a specified time

SYNOPSIS

```
#include <time.h>

int nanosleep(const struct timespec *req, struct timespec *rem);
```

DESCRIPTION

nanosleep delays the execution of the program for at least the time specified in **req*. The function can return earlier if a signal has been delivered to the process. In this case, it returns -1, sets *errno* to **EINTR**, and writes the remaining time into the structure pointed to by *rem* unless *rem* is **NULL**. The value of **rem* can then be used to call **nanosleep** again and complete the specified pause.

The structure *timespec* is used to specify intervals of time with nanosecond precision. It is specified in <*time.h*> and has the form

```
struct timespec {
        time_t  tv_sec;         /* seconds */
        long    tv_nsec;        /* nanoseconds */
};
```

The value of the nanoseconds field must be in the range 0 to 999 999 999.

Compared to and **nanosleep** has the advantage of not affecting any signals, it is standardized by POSIX, it provides higher timing resolution, and it allows to continue a sleep that has been interrupted by a signal more easily.

ERRORS

In case of an error or exception, the **nanosleep** system call returns -1 instead of 0 and sets *errno* to one of the following values:

EINTR

The pause has been interrupted by a non-blocked signal that was delivered to the process. The remaining sleep time has been written into **rem* so that the process can easily call **nanosleep** again and continue with the pause.

OUTB

Section: Linux Programmer's Manual (2)
Updated: November 29, 1995

NAME

outb, outw, outl, outsb, outsw, outsl - port output
inb, inw, inl, insb, insw, insl - port input
outb_p, outw_p, outl_p, inb_p, inw_p, inl_p - paused I/O

DESCRIPTION

This family of functions is used to do low level port input and output. They are primarily designed for internal kernel use, but can be used from user space, given the following information *in addition* to that given in

You compile with **-O** or **-O2** or similar. The functions are defined as inline macros, and will not be substituted in without optimization enabled, causing unresolved references at link time.

You use or alternatively to tell the kernel to allow the user space application to access the I/O ports in question. Failure to do this will cause the application to receive a segmentation fault.

CONFORMING TO

outb and friends are hardware specific. The *port* and *value* arguments are in the opposite order to most DOS implementations.

PAUSE

Section: Linux Programmer's Manual (2)
Updated: August 31, 1995

NAME

pause - wait for signal

SYNOPSIS

```
#include <unistd.h>

int pause(void);
```

DESCRIPTION

The **pause** system call causes the invoking process to sleep until a signal is received.

RETURN VALUE

pause always returns -1, and *errno* is set to **ERESTARTNOHAND**.

ERRORS

EINTR
signal was received.

CONFORMING TO

SVr4, SVID, POSIX, X/OPEN, BSD 4.3

PRINTF

Section: Linux Programmer's Manual (3)
Updated: 28 January 1996

NAME

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf - formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list
ap);
```

DESCRIPTION

The **printf** family of functions produces output according to a *format* as described below. The functions **printf** and **vprintf** write output to *stdout*, the standard output stream; **fprintf** and **vfprintf** write output to the given output *stream*; **sprintf**, **snprintf**, **vsprintf** and **vsnprintf** write to the character string *str*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of are converted for output.

These functions return the number of characters printed (not including the trailing `\0' used to end output to strings). **snprintf** and **vsnprintf** do not write more than *size* bytes (including the trailing '\0'), and return -1 if the output was truncated due to this limit. (Thus until glibc 2.0.6. Since glibc 2.1 the function **vsnprintf** returns the number of characters (excluding the trailing '\0') which would have been written to the final string if enough space had been available.)

The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**. The arguments must correspond properly (after type promotion) with the conversion specifier. After the **%**, the following appear in sequence:

GET_PRIORITY_MAX

Section: Linux Programmer's Manual (2)
Updated: 1996-04-10

NAME

sched_get_priority_max, sched_get_priority_min - get static priority range

SYNOPSIS

```
#include <sched.h>

int sched_get_priority_max(int policy);

int sched_get_priority_min(int policy);
```

DESCRIPTION

sched_get_priority_max returns the maximum priority value that can be used with the scheduling algorithm identified by *policy*. **sched_get_priority_min** returns the minimum priority value that can be used with the scheduling algorithm identified by *policy*. Supported *policy* values are *SCHED_FIFO*, *SCHED_RR*, and *SCHED_OTHER*.

Processes with numerically higher priority values are scheduled before processes with numerically lower priority values. Thus, the value returned by **sched_get_priority_max** will be greater than the value returned by **sched_get_priority_min**.

Linux allows the static priority value range 1 to 99 for *SCHED_FIFO* and *SCHED_RR* and the priority 0 for *SCHED_OTHER*. Scheduling priority ranges for the various policies are not alterable.

The range of scheduling priorities may vary on other POSIX systems, thus it is a good idea for portable applications to use a virtual priority range and map it to the interval given by **sched_get_priority_max** and **sched_get_priority_min**. POSIX.1b requires a spread of at least 32 between the maximum and the minimum values for *SCHED_FIFO* and *SCHED_RR*.

POSIX systems on which **sched_get_priority_max** and **sched_get_priority_min** are available define *_POSIX_PRIORITY_SCHEDULING* in <unistd.h>.

RETURN VALUE

On success, **sched_get_priority_max** and **sched_get_priority_min** return the maximum/minimum priority value for the named scheduling policy. On error, -1 is returned, *errno* is set appropriately.

SETSCHEDULER

Section: Linux Programmer's Manual (2)
Updated: 1996-04-10

NAME

sched_setscheduler, sched_getscheduler - set and get scheduling algorithm/parameters

SYNOPSIS

```
#include <sched.h>

int sched_setscheduler(pid_t pid, int policy, const struct
sched_param *p);

int sched_getscheduler(pid_t pid);

struct sched_param {
       int sched_priority;
}
```

DESCRIPTION

sched_setscheduler sets both the scheduling policy and the associated parameters for the process identified by *pid*. If *pid* equals zero, the scheduler of the calling process will be set. The interpretation of the parameter *p* depends on the selected policy. Currently, the following three scheduling policies are supported under Linux: *SCHED_FIFO*, *SCHED_RR*, and *SCHED_OTHER*; their respective semantics is described below.

sched_getscheduler queries the scheduling policy currently applied to the process identified by *pid*. If *pid* equals zero, the policy of the calling process will be retrieved.

Scheduling Policies

The scheduler is the kernel part that decides which runnable process will be executed by the CPU next. The Linux scheduler offers three different scheduling policies, one for normal processes and two for real-time applications. A static priority value *sched_priority* is assigned to each process and this value can be changed only via system calls. Conceptually, the scheduler maintains a list of runnable processes for each possible *sched_priority* value, and *sched_priority* can have a value in the range 0 to 99. In order to determine the process that runs next, the Linux scheduler looks for the non-empty list with the highest static priority and takes the process at the head of this list. The scheduling policy determines for each process, where it will be inserted into the list of processes with equal static priority and how it will move inside this list.

SIGACTION

Section: Linux Programmer's Manual (2)
Updated: 8 May 1999

NAME

sigaction, sigprocmask, sigpending, sigsuspend - POSIX signal handling functions.

SYNOPSIS

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);

int sigpending(sigset_t *set);

int sigsuspend(const sigset_t *mask);
```

DESCRIPTION

The **sigaction** system call is used to change the action taken by a process on receipt of a specific signal.

signum specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP**.

If *act* is non-null, the new action for signal *signum* is installed from *act*. If *oldact* is non-null, the previous action is saved in *oldact*.

The **sigaction** structure is defined as

```
struct sigaction {
        void (*sa_handler)(int);
        void (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_restorer)(void);
}
```

The *sa_restorer* element is obsolete and should not be used.

sa_handler specifies the action to be associated with *signum* and may be **SIG_DFL** for the default action, **SIG_IGN** to ignore this signal, or a pointer to a signal handling function.

SIGSETOPS

Section: Linux Programmer's Manual (3)
Updated: 24 September 1994

NAME

sigemptyset, sigfillset, sigaddset, sigdelset, sigismember - POSIX signal set operations.

SYNOPSIS

```
#include <signal.h>

int sigemptyset(sigset_t *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int signum);

int sigdelset(sigset_t *set, int signum);

int sigismember(const sigset_t *set, int signum);
```

DESCRIPTION

The functions allow the manipulation of POSIX signal sets.

sigemptyset initializes the signal set given by *set* to empty, with all signals excluded from the set.

sigfillset initializes *set* to full, including all signals.

sigaddset and **sigdelset** add and delete respectively signal *signum* from *set*.

sigismember tests whether *signum* is a member of *set.*

RETURN VALUES

sigemptyset, **sigfullset**, **sigaddset** and **sigdelset** return 0 on success and -1 on error.

sigismember returns 1 if *signum* is a member of *set*, 0 if *signum* is not a member, and -1 on error.

ERRORS

EINVAL
sig is not a valid signal.

SIGNAL

Section: Linux Programmer's Manual (2)
Updated: 21 July 1996

NAME

signal - ANSI C signal handling

SYNOPSIS

```
#include <signal.h>

void (*signal(int signum, void (*handler)(int)))(int);
```

DESCRIPTION

The **signal** system call installs a new signal handler for the signal with number *signum.* The signal handler is set to *handler* which may be a user specified function, or one of the following:

SIG_IGN
Ignore the signal.
SIG_DFL
Reset the signal to its default behavior.

The integer argument that is handed over to the signal handler routine is the signal number. This makes it possible to use one signal handler for several signals.

RETURN VALUE

signal returns the previous value of the signal handler, or **SIG_ERR** on error.

NOTES

Signal handlers cannot be set for **SIGKILL** or **SIGSTOP**.

Unlike on BSD systems, signals under Linux are reset to their default behavior when raised. However, if you include **<bsd/signal.h>** instead of **<signal.h>** then **signal** is redefined as **__bsd_signal** and signal has the BSD semantics. Both versions of **signal** are library routines built on top of

If you're confused by the prototype at the top of this manpage, it may help to see it separated out thus:

typedef void (*sighandler_t)(int);

sighandler_t signal(int *signum*, sighandler_t *handler*);

SLEEP

Section: Linux Programmer's Manual (3)
Updated: April 7, 1993

NAME

sleep - Sleep for the specified number of seconds

SYNOPSIS

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

DESCRIPTION

sleep() makes the current process sleep until *seconds* seconds have elapsed or a signal arrives which is not ignored.

RETURN VALUE

Zero if the requested time has elapsed, or the number of seconds left to sleep.

CONFORMING TO

POSIX.1

BUGS

sleep() may be implemented using **SIGALRM**; mixing calls to **alarm()** and **sleep()** is a bad idea.

Using **longjmp()** from a signal handler or modifying the handling of **SIGALRM** while sleeping will cause undefined results.

STRFTIME

Section: Linux Programmer's Manual (3)
Updated: 29 March 1999

NAME

strftime - format date and time

SYNOPSIS

```
#include <time.h>

size_t strftime(char *s, size_t max, const char *format,
                                const struct tm *tm);
```

DESCRIPTION

The **strftime()** function formats the broken-down time *tm* according to the format specification *format* and places the result in the character array *s* of size *max*.

Ordinary characters placed in the format string are copied to *s* without conversion. Conversion specifiers are introduced by a `%' character, and are replaced in *s* as follows:

%a
The abbreviated weekday name according to the current locale.

%A
The full weekday name according to the current locale.

%b
The abbreviated month name according to the current locale.

%B
The full month name according to the current locale.

%c
The preferred date and time representation for the current locale.

%C
The century number (year/100) as a 2-digit integer. (SU)

%d
The day of the month as a decimal number (range 01 to 31).

%D
Equivalent to %m/%d/%y. (Yecch - for Americans only. Americans should note that in other countries %d/%m/%y is rather common. This means that in international context this format is ambiguous and should not be used.) (SU)

%e
Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space. (SU)

%E
Modifier: use alternative format, see below. (SU)

TIME

Section: Linux Programmer's Manual (2)
Updated: 9 September 1997

NAME

time - get time in seconds

SYNOPSIS

```
#include <time.h>

time_t time(time_t *t);
```

DESCRIPTION

time returns the time since the Epoch (00:00:00 UTC, January 1, 1970), measured in seconds.

If *t* is non-NULL, the return value is also stored in the memory pointed to by *t*.

RETURN VALUE

On success, the value of time in seconds since the Epoch is returned. On error, ((time_t)-1) is returned, and *errno* is set appropriately.

ERRORS

EFAULT
t points outside your accessible address space.

NOTES

POSIX.1 defines *seconds since the Epoch* as a value to be interpreted as the number of seconds between a specified time and the Epoch, according to a formula for conversion from UTC equivalent to conversion on the naï ve basis that leap seconds are ignored and all years divisible by 4 are leap years. This value is not the same as the actual number of seconds between the time and the Epoch, because of leap seconds and because clocks are not required to be synchronised to a standard reference. The intention is that the interpretation of seconds since the Epoch values be consistent; see POSIX.1 Annex B 2.2.2 for further rationale.

USLEEP

Section: Linux Programmer's Manual (3)
Updated: July 4, 1993

NAME

usleep - suspend execution for interval of microseconds

SYNOPSIS

```
#include <unistd.h>

void usleep(unsigned long usec);
```

DESCRIPTION

The **usleep()** function suspends execution of the calling process for *usec* microseconds. The sleep may be lengthened slightly by any system activity or by the time spent processing the call.

CONFORMING TO

BSD 4.3

Referências Bibliográficas

[1] [BOV 01] BOVET, D.P.; CESATI, M. Understanding the Linux Kernel. O'Reilly & Associates, 2001. 684p.

[2] [Gal95] B. O. Gallmeister. POSIX.4 Programming for the Real World. O'Reilly & Associates, ISBN 1-56592-074-0, 1995.

[3] *What is Real Time Clock?*
http://www.ust.hk/itsc/y2k/Y2k_pc/RTC_detail.html.

[4] *Programming the Intel 8253 Programmable Interval Timer*
http://www.totse.com/en/computers/rare_computers/pit.html.

[5] Manuais *online* do Linux (*manpages*).

[6] Programação em Tempo Real - Módulo II - Comunicação InterProcessos -
http://www.leca.ufrn.br/~adelardo/cursos/ELE409/all.html

[8] HawkLord Web Page
http://www.hawklord.ukLinux.net/system/signals/index.html

[9] Práticas de Sistemas de Tempo Real
http://www.ei.uvigo.es/~cernadas/str01/practica7.htm

[10] Sistemas de Tempo Real .Jean-Marie Farines, Joni da Silva Fraga e Rômulo Silva de Oliveira. 12ª Escola de Computação, IME-USP, São Paulo-SP, 24 a 28 de julho de 2000.

[11] RTAI - Real Time Aplication Interface
www.aero.polimi.it/projects/rtai/

[12] RTLinux - Real Time Linux
www.rtlinux.org