NKE - Um Nanokernel Educacional para Microprocessadores ARM

Celso Maciel da Costa, João Leonardo Fragoso, Lucas Murliky, Leonardo da Luz Silva, Aline Fracalossi, Cássio Brasil, Guilherme Debom
Universidade Estadual do Rio Grande do Sul
Guaíba, Rio Grande do Sul - Brasil
{celsocostars, jl.fragoso, lucasuow, alinee.ff, cassio.infobr, guidebom}@gmail.com, leonardoluz@live.com

Rivalino Matias Jr.
Faculdade de Computação
Universidade Federal de Uberlândia
Uberlândia, Minas Gerais - Brasil
rivalino@fc.ufu.br

***Resumo* — O ensino e o aprendizado de sistemas operacionais são dificultados em face da natureza dos conceitos envolvidos. É necessário não apenas estudos conceituais, mas principalmente a prática em laboratório, o que nem sempre é possível por falta de ambiente e instrumental apropriados. Este artigo apresenta o NKE, um Nanokernel desenvolvido para microprocessadores ARM e voltado para o apoio de atividades práticas em disciplinas de sistemas operacionais e sistemas embarcados. O Nanokernel foi desenvolvido em linguagem C, com um conjunto mínimo de funções em linguagem Assembly. O NKE tem sido usado com sucesso em atividades práticas no curso de Engenharia de Computação da UERGS/Guaíba, proporcionando uma experiência diferenciada aos alunos por meio de um ambiente real de desenvolvimento de sistemas operacionais.**

Palavras-chave—sistemas operacionais; nanokernel; sistemas embarcados

I. Introdução

Os cursos de graduação em computação possuem em seus currículos a disciplina "Sistemas Operacionais", a qual visa qualificar os estudantes no entendimento da estrutura e funcionamento de sistemas operacionais de computador. Os conceitos abordados nessa disciplina apresentam relativa dificuldade, em especial no tocante aos detalhes práticos, tornando o seu ensino e aprendizado uma tarefa não trivial. Observa-se que muitos alunos aprendem os conceitos e algoritmos abordados, contudo, apresentam dificuldade para correlacioná-los e entendê-los de forma integrada [1], [2]. Um elemento facilitador nesse processo é a prática de laboratório. Experimentar esses conceitos de forma prática possibilita ao estudante ter uma compreensão integrada dos diversos assuntos estudados nesta disciplina. Neste caso, considera-se como prática efetiva não o uso de um sistema operacional, mas a programação dos seus subsistemas em nível de núcleo (*kernel space*).

Para isso, é necessário adotar um instrumental adequado como plataforma para as atividades práticas de ensino. Apesar de existentes (ex. [1], [2], [3], [4], [5]), ainda são poucas as iniciativas neste campo. Uma abordagem interessante para construir um sistema operacional didático é utilizar o conceito de Nanokernel [6], [7], [8]. Este tem como principal característica sua simplicidade e flexibilidade de construção, podendo ser baseado na combinação de módulos tais como escalonadores, *drivers* de dispositivos e interfaces.

Neste trabalho é apresentado o NKE, um sistema operacional embarcado baseado no conceito de Nanokernel. O NKE foi desenvolvido objetivando ser usado em atividades de ensino prático de sistemas operacionais e sistemas embarcados em cursos universitários. Sua simplicidade e modularidade possibilitam ao estudante, gradativamente, construir seu próprio sistema operacional, a partir de um código base mínimo já existente. Na medida em que o estudante avança no conteúdo da disciplina, ele pode customizar o NKE incorporando novas funcionalidades ou alterando as já existentes. O NKE é desenvolvido predominantemente em linguagem C, visando facilitar o seu entendimento e modificação por parte dos estudantes. O microprocessador usado atualmente é o ARM [9], principalmente pela sua popularidade no mercado e por seu baixo custo; facilitando a aquisição de *kits* ARM para uso do NKE por parte das instituições e estudantes.

As demais seções deste artigo descrevem o NKE em detalhes, as quais estão organizadas como segue. Na Seção II é apresentada a plataforma de hardware utilizada atualmente nas atividades de ensino com o Nanokernel proposto. Uma visão do Nanokernel é apresentada na Seção III e detalhes de sua implementação na Seção IV. Na Seção V são apresentadas as considerações finais e trabalhos futuros.

II. Plataforma de Hardware

O hardware utilizado neste trabalho é um *kit* baseado na placa ARM LPC2378, um processador ARM7-TDMI, 64 Kbytes de memória RAM e 256 Kbytes de memória *flash*. O processador ARM utilizado possui sete modos de operação: *user mode* é o modo de execução normal (nível de usuário); *supervisor* é o modo de execução do sistema operacional; *interrupt* é o modo de operação para o tratamento de interrupções de propósito geral; *fast interrupt* é um modo de operação especial para o tratamento de interrupções prioritárias; *undefined instruction* é um modo executado quando uma instrução inválida é encontrada na posição de memória apontada pelo registrador PC; *system* é usado para tratamento de interrupções e chamadas de sistema aninhadas;

o modo *abort* ocorre quando o processador busca uma instrução ou dado na memória mas o barramento gera um erro. Atualmente, ao NKE utiliza somente os dois primeiros modos.

Independente do modo de operação, o processador ARM possui 16 registradores visíveis ao programador (r0 a r15) e duas instruções importantes para a implementação de sistemas operacionais: STMFD para salvar todos os registradores na pilha e LDMFD para mover os valores salvos na pilha para os registradores; ambas as instruções são usadas pelo NKE para salvamento e restauração de contexto, respectivamente. Outra instrução importante para o NKE é a SWI, a qual troca o modo de operação do processador de *user* para *supervisor*, sendo usada na implementação de chamadas de sistema.

III. Visão Geral do NKE

Um Nanokernel é um sistema operacional que executa no modo privilegiado do processador (*supervisor*) e implementa um conjunto reduzido de funcionalidades [8]. Sua organização tem como principal propósito a modularização dos seus subsistemas para fins de customização; dependendo da aplicação novas funcionalidades são facilmente incorporadas [10], [11], [12]. Do ponto de vista didático, este tipo de organização é muito conveniente para o ensino prático de sistemas operacionais. Primeiro, porque expõe o estudante a um conjunto mínimo de código base que forma o núcleo do sistema operacional, facilitando o seu domínio em menor tempo, diferente de outras abordagens (ex. núcleos monolíticos) onde a quantidade de código que deve ser compreendida para dar início aos trabalhos práticos se torna um forte obstáculo.

O NKE possui dois níveis lógicos: *kernel* e *usuário*. Os serviços oferecidos pelo NKE são executados no nível *kernel* e acessados pelos programas (do nível *usuário*) por meio de chamadas de sistemas. Atualmente, o NKE disponibiliza um conjunto mínimo de funcionalidades que permite ao estudante construir seus próprios subsistemas e rotinas no espaço do *kernel* e usá-los do espaço do *usuário*. Exemplos destas funcionalidades são: salvar e restaurar contexto; tratar interrupções e exceções de hardware; criar, terminar e sincronizar tarefas; escalonamento de tarefas; gerenciamento de tempo; serviços básicos de entrada e saída.

No NKE, um programa de aplicação é escrito em linguagem C e, depois de compilado, deve ser ligado ao NKE antes de ser carregado para execução. A seguir serão apresentados maiores detalhes sobre as principais funcionalidades disponíveis atualmente no NKE.

A. *Chamadas de Sistema*

As seguintes chamadas de sistema estão disponíveis.

1) Criação de tarefas: É realizada no início da execução do sistema. Uma tarefa é implementada como uma função em linguagem C. A criação de tarefas é feita com a chamada de sistema *TaskCreate*(*int &tid, void * func*), onde o parâmetro *tid* armazena a identificação da tarefa criada e *func* é o ponteiro para a função contendo o código da tarefa. Toda tarefa deve terminar executando a chamada de sisztema *TaskExit*().

2) Inicialização de tarefas: As tarefas criadas pela *TaskCreate* são inseridas na *ready queue* pela chamada de sistema *Start*(SCHEDULER). A *ready queue* é organizada de acordo com o algoritmo de escalonamento definido no parâmetro SCHEDULER (ver Seção IV.E).

3) Sincronização de tarefas: É realizada por semáforos, os quais podem assumir valores maiores que 1 (semáforos contadores). Uma variável semáforo tem o tipo *sem_t*. As chamadas de sistema relacionadas com semáforos são:

a) SemInit(*sem_t &sem, unsigned int value*): Usada para inicializar o valor do semáforo *sem*. A variável *value* pode assumir valores maiores ou igual a zero.

b) SemWait(*sem_t &sem*): Decrementa o valor do semáforo e caso o resultado seja maior ou igual a zero, então a tarefa não é bloqueada. Caso contrário, a tarefa é bloqueada e entra na fila do semáforo (*sem_queue*).

c) SemPost(*sem_t &sem*): Se a fila do semáforo estiver vazia, ou seja, nenhuma tarefa bloqueada no semáforo, então incrementa o semáforo. Caso contrário, acorda a primeira tarefa aguardando nesta fila.

4) Gerenciamento de tempo: É implementado por meio de duas chamadas de sistema:

a) Sleep(*unsigned int time*): Coloca uma tarefa em estado de espera por *time* segundos.

b) mSleep(*unsigned int time*): O mesmo que a chamada anterior, porém em milissegundos.

B. *Exemplo de Uso*

A Fig. 1 ilustra a criação de duas tarefas no NKE.

```
#include "kernel.h"

unsigned int buffer;
int  t1, t2, buffer=0;
sem_t s;
void  task1 (){
  int i;
  for(i=0;i<100000;i++){
    SemWait(&s);
       buffer = buffer + 1;
    SemPost(&s);
  }
  TaskExit() ;
}
void  task2 (){
  int i;
  for(i=0;i<100000;i++){
    SemWait(&s);
       buffer = buffer + 2;
    SemPost(&s);
  }
  TaskExit();
}
int main(){
  SemInit(&s, 1);
  TaskCreate(&t1, task1);
  TaskCreate(&t2, task2);
  Start(RR)
}
```

Fig. 1. Exemplo de programa no NKE.

Ambas as tarefas utilizam um semáforo para exclusão mútua no acesso a uma variável compartilhada (*buffer*). Após a criação das tarefas, *task1* e *task2*, a chamada *Start*(RR) é executada para colocar as tarefas criadas na fila de tarefas prontas para executar (*ready queue*). Neste caso, o escalonador de tarefas utiliza o algoritmo *Round Robin* [13], o qual irá disparar a primeira tarefa da fila. Como pode ser observada, a programação no NKE é realizada de forma convencional, facilitando seu uso por estudantes já familiarizados com programação em C para outros sistemas operacionais. A seguir serão apresentados detalhes da implementação do NKE.

IV. DETALHES DE IMPLEMENTAÇÃO

A. *Tratamento de Interrupções de Hardware*

O processador ARM possui dois tipos de interrupções: o sinal de interrupção IRQ e o sinal de interrupção rápida FIQ. Atualmente, o NKE trata somente o tipo IRQ, ficando as interrupções rápidas (FIQ) desabilitadas permanentemente. Quando uma interrupção ocorre, as seguintes ações são executadas pelo processador:

1. O endereço da próxima instrução é copiado para o registrador R14 _irq (salva o PC no registrador r14 do modo *interrupt*);
2. O modo de operação corrente, registrador CPSR, é salvo no registrador SPSR_irq;
3. Os bits do registrador CPSR referentes ao modo *interrupt* são ativados;
4. As interrupções do tipo IRQ são desabilitadas;
5. PC recebe o valor da posição 0x00000018 e passa a executar no endereço selecionado pelo controlador de interrupções, sendo este o endereço da rotina de tratamento da interrupção ativada (ISR);
6. A ISR salva os valores dos registradores gerais e de estado da tarefa atual e executa o código de tratamento da interrupção ativada.

No caso da IRQ ativada ser a interrupção do *timer*, a ISR é denominada *TimerInterrupt.* Atualmente, a cada ocorrência dessa interrupção tem-se o término da fatia de tempo de uso do processador por parte da tarefa corrente (*time slice*). Neste caso, após o salvamento de contexto dessa tarefa na sua pilha, a rotina *TimerInterrupt* chama a rotina *Select*, a qual é responsável por realizar a atualização dos tempos das tarefas em espera, *mSleep* ou *Sleep*, bem como selecionar para execução a próxima tarefa da *ready queue*. Por fim, a rotina *Dispatcher* é chamada para disparar a execução da tarefa selecionada, retornando o processador para o modo *user*.

B. *Tratamento de Interrupção de Software*

Uma interrupção de software é gerada quando ocorre a execução da instrução SWI, a qual é usada para implementar chamadas de sistema. A SWI troca o modo de operação do processador de *user* para *supervisor*, causando o chaveamento da pilha da tarefa para a pilha do sistema. O processador passa a executar o código do NKE na pilha do sistema com as interrupções de hardware desabilitadas, inclusive a de *timer*. Quando ocorre uma interrupção de software os seguintes passos são executados:

1. O endereço da próxima instrução é copiado para o registrador R14_svc;
2. O modo de operação corrente é salvo, ou seja, SPSR_svc terá o antigo valor de CPSR;
3. Ativa os bits do modo CPSR referente ao modo *supervisor*;
4. As interrupções do tipo IRQ são desabilitadas;
5. PC recebe o conteúdo da posição 0x00000008 do vetor de interrupções, o qual aponta para a rotina *SystemContext*;
6. A *SystemContext* salva os valores dos registradores gerais e de estado da tarefa atual e chama a rotina *DoSystemCall*.

Após o passo #6, a execução continua na rotina *DoSystemCall* que identifica o número da chamada de sistema a ser executada. Se a chamada não for bloqueante, o contexto da tarefa corrente é restaurado após a conclusão da chamada de sistema e a sua execução continua na instrução seguinte à chamada que causou a interrupção. No caso de chamadas bloqueantes o escalonador é acionado (ver Seção IV.E).

C. *Implementação de Chamadas de Sistema*

Como descrito anteriormente, no NKE as chamadas de sistema são realizadas com o apoio da instrução SWI. Quando ocorre uma chamada de sistema, primeiramente os parâmetros da chamada e o seu identificador (*CallNumber*) são armazenados na estrutura de dados *Parameters* (ver Fig. 2). Atualmente o NKE suporta chamadas de sistema com até quatro parâmetros.

CallNumber	*p0*	*p1*	*p2*	*p3*

Fig. 2. Estrutura de dados do tipo *Parameters*.

Cada chamada de sistema tem uma função de biblioteca correspondente (*system call stub*), ligada ao programa do usuário durante a geração do código executável do programa. Estas *stubs* encapsulam o código que efetivamente realiza a chamada de sistema. Este código salva os parâmetros da chamada na estrutura *Parameters* e passa o controle para o NKE por meio da função *CallSWI*(). Esta por sua vez simplesmente executa a instrução SWI para entrar em modo *supervisor*.

Como descrito na Seção IV.B, após uma chamada SWI a execução do NKE inicia-se na rotina *SystemContext* que em seguida chama a *DoSystemCall.* Esta última identifica a chamada de sistema a ser executada consultando o *CallNumber* na estrutura *Parameters*. A Fig. 3 apresenta um exemplo de programa de usuário que faz uso da chamada de sistema *Sleep*, bem como mostra a sua função de biblioteca (*stub*) correspondente. Na Fig. 4, tem-se parte do código da rotina *DoSystemCall* que chama a implementação da *Sleep*, a qual é efetivamente realizada pela rotina *sys_Sleep*.

```
#include "kernel.h"

int main(){
  ...
  Seep(5);
  ...
}
```

```
...
void Sleep(int time)
{
  Parameters.CallNumber=SLEEP;
  Parameters.p0=(unsigned char *)time;
  CallSWI();
}
```

Fig. 3. Programa e *stub* da chamada de sistema *Sleep*.

```
void DoSystemCall(){
 switch(Parameters.CallNumber){
   ...
   case SLEEP:
      sys_Sleep((int)Parameters.p0);
      break;
   ...
 }
}
```

Fig. 4. Fragmento de código da rotina *DoSystemCall*.

D. Gerência de Tarefas

Cada tarefa no NKE é representada por um descritor de tarefas como apresentado na Fig. 5.

Tid	*Prio*	*State*	*Time*	*EP*	*SP*	*Stack*

Fig. 5. Descritor de tarefas do NKE.

O campo *Tid* armazena um valor inteiro que identifica unicamente uma tarefa, cuja prioridade é armazenada em *Prio* e o estado corrente no campo *State*. A Fig. 6 ilustra o diagrama de transição de estados de uma tarefa no NKE. Se o estado da tarefa é *Blocked* e o valor do campo *Time* é diferente de zero, então a tarefa está esperando pela passagem do tempo especificado nesse campo. O campo *EP* armazena o endereço inicial (*entry point*) de execução da tarefa. *SP* armazena o endereço do topo da pilha da tarefa e o campo *Stack* é um vetor de inteiros de 100 elementos, o qual serve como pilha para a execução da tarefa. Cada elemento do vetor possui quatro bytes.

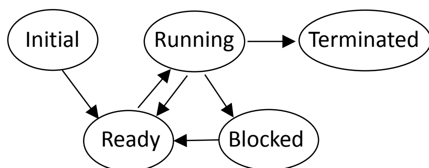

Fig. 6. Diagrama de transição de dados de uma tarefa no NKE.

Após a tarefa ser criada, seu estado inicial é *Initial*. Neste estado a tarefa não é colocada na *ready queue*. A tarefa transita para o estado *Ready* após a execução da chamada de sistema *Start*, quando então ela é movida para a *ready queue*, estando disponível para execução. Do estado *Ready* a tarefa transita para o estado *Running* quando ganha o processador. Deste estado, a tarefa vai para o estado *Blocked* quando precisa aguardar por um evento (ex. liberação de um semáforo) ou para o estado *Ready* caso termine sua fatia de tempo; preempção por tempo no algoritmo de escalonamento *Round Robin*. A transição para o estado *Terminated* ocorre pela execução da chamada *TaskExit*. Nesse estado a tarefa não concorre ao uso do processador e tem todos os seus recursos (ex. descritor de tarefa, semáforos) liberados antes de finalizar.

Como descrito anteriormente, a criação de tarefas no NKE é efetuada por meio da chamada de sistema *TaskCreate*. A rotina de *kernel* que implementa esta chamada é a *sys_TaskCreate* (ver Fig. 7), a qual aloca e inicializa o descritor da tarefa sendo criada. A tabela global de descritores é implementada pelo vetor *Descriptors*, onde cada *NumberTaskAdd* indica a posição alocada no vetor de descritores para o descritor da nova tarefa; *TaskIdentifier* é o identificador da tarefa e *func* é o endereço (*entry point*) do código da nova tarefa.

```
void sys_TaskCreate(int *tid, void (*func)){
  *tid=++TaskIdentifier;
  i = ++NumberTaskAdd;
  Descriptors[i].Tid=TaskIdentifier;
  Descriptors[i].EP=func;
  Descriptors[i].State=Initial;
  Descriptors[i].SP=&Descriptors[i].\
                Stack[SizeTaskStack-1];
}
```

Fig. 7. Código da rotina *sys_TaskCreate*.

A *TaskExit* é implementada pela rotina de *kernel* *sys_TaskExit*, a qual assinala no descritor da tarefa o estado *Terminated* e posteriormente chama o escalonador para selecionar a próxima tarefa para execução. Existem duas tarefas especiais no sistema: *idle* e *main*. A *idle* executa quando a *ready queue* está vazia (sem tarefas prontas para executar) e a *main* é a *thread* principal da aplicação do usuário, resultado da ligação do programa com o NKE. Após a criação das tarefas e da chamada *Start*, a *main* não disputa mais o processador.

E. Escalonamento de Tarefas

O NKE foi projetado para suportar mais de um algoritmo de escalonamento. Atualmente estão implementados os algoritmos: RR (*Round Robin*), RM (*Rate Monotonic*) e EDF (*Earlist Deadline First*) [14]. A seleção do algoritmo ocorre na chamada de sistema *Start*(SCHEDULER). A variável global SCHEDULER armazena o tipo de escalonamento em vigor. Atualmente esse parâmetro pode assumir três valores: RR, RM ou EDF. No caso dos algoritmos RM e EDF, a *ready queue* é organizada levando-se em conta a prioridade de cada tarefa, armazenada no campo *Prio* do descritor da tarefa (ver Fig. 5). No RR, as tarefas não possuem prioridade e a fatia de tempo utilizada é de 100 milissegundos. No NKE, uma vez definido o algoritmo de escalonamento todas as tarefas são executadas com esse algoritmo.

Duas rotinas implementam o escalonamento de tarefas: *Select* (Fig. 8) e *Dispatcher* (Fig. 9). A primeira seleciona a rotina de escalonamento e a tarefa a ser executada. Ela é chamada pela *Start*, na inicialização das tarefas, e também pela rotina *TimerInterrupt* (Seção IV.A), na ocorrência de uma interrupção de *timer*. A segunda é chamada pela *Select* e pelas chamadas de sistema bloqueantes (ex. *sys_SemWait*). Essa rotina dispara a execução da primeira tarefa da fila de tarefas prontas (*ready_queue*). Para isso, recupera o valor do campo

SP no descritor da tarefa e chama a *RestoreContext* com este valor. A *RestoreContext* restaura os valores dos registradores armazenados na pilha da tarefa selecionada a fim de começar ou recomeçar sua execução a partir da instrução apontada pelo registrador PC, recuperado da sua pilha.

```
void Select(){
  switch (SCHEDULER){
    case RR:  GetNextRR();
              UpdateRR();
              break;
    case RM:  GetNextRM();
              UpdateRM();
              break;
    case EDF: GetNextEDF();
              UpdateEDF();
              break;
  }
  Dispatcher();
}
```

Fig. 8. Código da rotina *Select.*

```
void Dispatcher(){
  TaskRunning=ready_queue.queue\
                         [ready_queue.head];
  if(ready_queue.head != ready_queue.tail){
    if(ready_queue.head<MaxNumberTask-1)
       ready_queue.head++;
    else
       ready_queue.head=0;
  }
  Descriptors[TaskRunning].State=Running;
  RestoreContext(Descriptors[TaskRunning]\
                                     .SP);
}
```

Fig. 9. Código da rotina *Dispatcher.*

F. Semáforos

A estrutura de dados usada para representar um semáforo é *sem_t* (ver Fig. 10). Nesta estrutura, *count* representa o valor do semáforo, *head* é o índice do vetor que contém o *tid* da primeira tarefa bloqueada no semáforo, *tail* o índice do vetor que contém o *tid* da última tarefa bloqueada no semáforo, e *sem_queue* é um vetor que armazena o *tid* das tarefas bloqueadas no semáforo. A versão atual do NKE suporta semáforos contadores [9].

Count	*head*	*tail*	*sem_queue*

Fig. 10. Estrutura de dados *sem_t.*

As chamadas de sistema relacionadas a semáforos são *SemInit*, *SemWait* e *SemPost* (ver Fig. 11). Na rotina *sys_SemWait* o valor do semáforo é decrementado. Se esse valor se tornar menor que zero, o estado da tarefa que efetuou a chamada é alterado de *Running* para *Blocked* e a tarefa é inserida no final da fila de espera do semáforo (*sem_queue*), causando o disparo da próxima tarefa da *ready queue* por meio da *Select*. A rotina *sys_SemPost* incrementa o valor do semáforo; se o valor se torna maior ou igual a zero, o estado da primeira tarefa da fila é alterado para *Ready* e a mesma é inserida na *ready queue*.

```
void sys_SemInit(sem_t *s, int value){
  s->count = value;
  s->head = 0;
  s->tail = 0;
}
void sys_SemWait(sem_t *s){
  s->count--;
  if(s->count < 0){
    s->sem_queue[s->tail]=TaskRunning;
    Descriptors[TaskRunning].State=Blocked;
    s->tail++;
    if(s->tail==MaxNumberTask-1)
       s->tail = 0;
    Dispatcher();
  }
}
void sys_SemPost(sem_t* s){
 s->count++;
 if(s->count <= 0){
  Descriptors[s->sem_queue[s->header]].State\
                                     =Ready;
  InsertReadyQueue(s->sem_queue[s->head]);
  s->head++;
  if(s->head==MaxNumberTask-1)
     s->head = 0;
 }
}
```

Fig. 11. Rotinas de *kernel* para semáforos.

G. Gerenciamento de Tempo

As chamadas de sistema de gerenciamento de tempo no NKE, *Sleep* e *mSleep*, são implementadas por *sys_Sleep*(*unsigned int seconds*), *sys_mSleep*(*unsigned int miliseconds*). O valor do parâmetro recebido no *kernel* é transformado em número de interrupções do timer. Ao serem chamadas, o valor da variável *time* é armazenado no campo *Time* do descritor da tarefa, o estado da tarefa é alterado para *Blocked* e tem-se a seleção de uma nova tarefa para execução. Cada vez que ocorre uma interrupção de *timer*, o valor *Time* das tarefas em estado *Blocked* é decrementado. Se esse valor se tornar zero então a tarefa é inserida no final da *ready_queue* (a tarefa foi acordada). O código da rotina *sys_Sleep* é apresentado na Fig. 12. A constante *ClkT* define a frequência das interrupções (em 100 milissegundos) e a variável *ticks* representa o número de interrupções de *timer* que a tarefa ficará bloqueada.

```
void  sys_Sleep(unsigned int seconds){
  int ticks ;
  ticks = (second*1000)/ClkT;
  Descriptors[TaskRunning].Time =ticks;
  Descriptors[TaskRunning].State= Blocked;
  Dispatcher();
}
```

Fig. 12. Rotina de *kernel sys_Sleep.*

V. TRABALHOS RELACIONADOS

Duas alternativas para o ensino prático de sistemas operacionais são o uso de um sistema real ou de um sistema construído especialmente para o ensino. O trabalho apresentado em [15] se enquadra na primeira abordagem e relata o uso do Linux no curso de sistemas operacionais. São propostos cinco projetos relativos ao desenvolvimento de

novas chamadas de sistema, primitivas de sincronização, políticas de escalonamento, mecanismo de visualização do mapeamento de páginas por processos e extensões a um sistema de arquivos. Segundo os autores, os resultados obtidos comprovam a validade desse modelo, com experiências práticas de modificações de código de um sistema operacional real. O uso do Windows para práticas de laboratório é descrito em [16], onde os autores usam o *Windows Research Kernel* cujo código fonte é cedido pela Microsoft. São descritos três projetos, um depurador, uma chamada de sistema e outro referente à gerência de memória. Os autores aferiram como os estudantes avaliaram o uso do Windows para aprendizado de SO, bem como suas principais dificuldades. As respostas demonstraram que os estudantes se sentiram satisfeitos em usar o Windows, mas um percentual expressivo gostaria de usar outros sistemas (ex. Linux). Em [17] o uso do Android no ensino de sistemas operacionais também é proposto, onde cinco projetos são discutidos, os quais exigiram dos estudantes ler, entender e modificar o *kernel* do Android. Os resultados indicaram que o uso do Android despertou o interesse e serviu como motivador para os estudantes, o que mostrou a adequação desse sistema no ensino de sistemas operacionais. Uma experiência de SO construído para o ensino de sistemas operacionais é o *Pintos* [18], projetado para a arquitetura 80x86, tem suporte à carga e execução de programas com *threads* e um sistema de arquivos. *Pintos* executa em hardware real e em um ambiente de simulação e tem sido usado em várias instituições de ensino. De acordo com os autores, o uso desse sistema motiva os alunos ao estudo de SO. Já o NKE, apresentado neste artigo, possui como principal diferencial ser um sistema de fácil compreensão e modificação, para uso embarcado. Seus mecanismos básicos podem ser facilmente entendidos e modificados, viabilizando seu uso concomitante ao ensino teórico de SO em disciplinas típicas de nível de graduação. Se comparado aos demais trabalhos relacionados, o NKE se destaca pela adoção da arquitetura Nanokernel, o que reduz a curva de aprendizagem dos estudantes devido à menor complexidade do seu código base, agilizando o desenvolvimento das experiências práticas de desenvolvimento de um SO em sala de aula.

VI. CONSIDERAÇÕES FINAIS

Atualmente, o NKE se encontra funcional e tem sido usado para o ensino de graduação e projetos de iniciação científica. Em nível de ensino, observa-se que a prática da programação/modificação dos seus subsistemas, bem como a visualização destas mudanças em funcionamento em um sistema real, é um forte motivador para o engajamento dos estudantes no aprendizado dos conceitos teóricos relacionados. Desenvolver diferentes algoritmos de escalonamento e comparar o desempenho das diferentes abordagens é a experiência mais recente disso. Também, conceitos pouco abordados na literatura, mas essenciais para se criar um sistema operacional, tais como carga de código executável e *stubs* de chamadas de sistema, entre outros, têm tido sua compreensão facilitada com a experiência prática do NKE. Trabalhos em andamento incluem estudos para transformar o NKE em um sistema de tempo real, suporte ao multiprocessamento e torná-lo compatível com o padrão POSIX [19].

REFERÊNCIAS

[1] C. Maziero, "Reflexões sobre o ensino prático de Sistemas Operacionais," Anais do XII Congresso da Sociedade Brasileira de Computação, Workshop de Ensino em Computação, 2002.

[2] F. B. Machado, L. P. Maia, "Um Framework Construtivista no Aprendizado de Sistemas Operacionais - Uma Proposta Pedagógica com o uso do Simulador SOSim," Anais do XII Workshop de Educação em Computação, Congresso da Sociedade Brasileira de Computação, 2004.

[3] H.W.M.S. Gonçalves, F. Bortoluzzi, C.A. Zeferino, "Desenvolvimento de um Sistema Operacional de Tempo Real para um Microcontrolador Básico," Anais do III Simpósio Brasileiro de Engenharia de Sistemas Computacionais, 2013.

[4] C. C. de Souza, T. R. Medeiros, R. N. S. Gadelha, T. D. N. de Sousa, E. L. Silva, R. R. de Azevedo, "Um Ambiente Integrado de Simulação para Auxiliar o Processo de Ensino/Aprendizagem da Disciplina de Sistemas Operacionais," Anais do XXII SBIE - XVII WIE, 2011.

[5] Nachos, "General Nachos Documentation" , http://www.cs.washington.edu/homes/tom/nachos/

[6] F. Bruns, S. Traboulsi, D. Szczesny, E. Gonzalez, X. Yang, A. Bilgic, "An Evaluation of NanoKernel-Based Virtualization for Embedded Real-Time Systems," Proceedings of the 22nd Euromicro Conference on Real-Time Systems, 2010, pp. 57-65.

[7] N. Herder, Towards a True NanoKernel Operating System. Vrije Universiteit Amsterdam, Master of Science thesis in Computer Science, 2005.

[8] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, J. S. Shapiro, "The KeyKOS Nanokernel Architecture," Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, 1992, pp. 95-112.

[9] S. B. Furber, ARM System-On-Chip Architecture, Addison-Wesley, 2000.

[10] M. Decky, "Component-based General-purpose Operating System," Proceeding of the 16th Annual Conference of Doctoral Students, 2007, pp. 58-63.

[11] J. Polakovic, J. B. Stefani, "Architecting reconfigurable component-based operating systems," Journal of Systems Architecture: the EUROMICRO Journal, vol. 54:6, pp. 562-575, 2008.

[12] F. Loiret, J. Navas, J. P. Babau and O. Lobry, "Component-Based Real-Time Operating System for Embedded Applications," Proceedings of the 12th International Symposium on Component-Based Software Engineering, 2009, pp. 209-226.

[13] A. S. Tanenbaum, Operating Systems Design and Implementation. Prentice-Hall, 1987.

[14] C. L. Liu, J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," Journal of the ACM, vol. 20:1, pp.46-61, 1973.

[15] O. Laadan, J. Nieh, N. Viennot. "Structured linux kernel projects for teaching operating systems concepts". In Proceedings of the 42nd ACM technical symposium on Computer science education (SIGCSE '11). ACM, New York, NY, USA, 2011, pp. 287-292.

[16] A. Schmidt, A. Polze, D. Probert. "Teaching operating systems: windows kernel projects". In Proceedings of the 41st ACM technical symposium on Computer science education (SIGCSE '10). ACM, New York, NY, USA, 2010, pp. 490-494.

[17] J. Andrus, J. Nieh. "Teaching operating systems using android". In Proceedings of the 43rd ACM technical symposium on Computer Science Education (SIGCSE '12). ACM, New York, NY, USA,2012, pp. 613-618.

[18] B. Pfaff, A. Romano, G. Back. "The pintos instructional operating system kernel". In Proceedings of the 40th ACM technical symposium on Computer science education (SIGCSE '09). ACM, New York, NY, USA, 2009, pp. 453-457.

[19] POSIX Threads. The standard, POSIX.1c, Threads extensions. IEEE Std 1003.1c, 1995.