NKE-Arduíno - Um Nanokernel Educacional

Celso Maciel da Costa†
Unidade da Uergs em Guaíba
Universidade Estadual do RS
Guaíba RS Brasil
celso-costa@uergs.edu.br

Ismael Soller Vianna
Unidade da Uergs em Guaíba
Universidade Estadual do RS
Guaíba RS Brasil
ismael-vianna@uergs.edu.br

Daniel Ferraz Tavares Jr.
Unidade da Uergs em Guaíba
Universidade Estadual do RS
Guaíba RS Brasil
daniel-junior@uergs.edu.br

Jordan de Andrade Pereira
Unidade da Uergs em Guaíba
Universidade Estadual do RS
Guaíba RS Brasil
jordan-pereira@uergs.edu.br

Matheus Luiz Debom Karr
Unidade da Uergs em Guaíba
Universidade Estadual do RS
Guaíba RS Brasil
matheus-karr@uergs.edu.br

Lucas Karr Osielski
Unidade da Uergs em Guaíba
Universidade Estadual do RS
Guaíba RS Brasil
lucas-osielski@uergs.edu.br

Maikon Michel de Almeida
Unidade da Uergs em Guaíba
Universidade Estadual do RS
Guaíba RS Brasil
maikon-almeida@uergs.edu.br

Wellington da Silva da Luz
Unidade da Uergs em Guaíba
Universidade Estadual do RS
Guaíba RS Brasil
wellington-luz@uergs.edu.br

Allan Demetrio Pereira de Deus
Unidade da Uergs em Guaíba
Universidade Estadual do RS
Guaíba RS Brasil
allan-deus@uergs.edu.br

ABSTRACT

This paper presents the development and implementation of NKE-Arduino, an educational multiprogrammable operating system specifically designed for the Arduino platform. This project involved porting the existing NKE system, originally developed for ARM-based platforms, to the Arduino environment. NKE-Arduino supports multiple concurrent tasks, scheduling and timing mechanisms provided by the TimerOne library. The system incorporates task create operations, semaphore-based synchronization to manage access to shared resources, preventing race conditions and ensuring data integrity. Additional functionalities include control of LEDs, serial input reading analogous to scanf, and serial output printing similar to printf. Designed to operate on the resource-limited hardware of the Arduino Uno, NKE-Arduino offers a practical and accessible tool for educational purposes. The system provides students with the opportunity to understand and experiment with fundamental operating system concepts. The development of NKE-Arduino not only serves as an instructional tool but also showcases the feasibility of implementing a multitasking operating system on constrained hardware.

. CCS CONCEPTS

Computer Systems Organization. Embedded and cyber-physical systems Embedded Systems. Embedded Software.

KEYWORDS

Sistemas Embarcados, Nanokernel, Sistemas Operacionais Embarcados. Arduíno.

1 INTRODUÇÃO

Este trabalho apresenta a implementação do nanokernel NKE[1], originalmente desenvolvido para a família ARM, na plataforma Arduino. A escolha do Arduino se justifica por seu baixo custo, simplicidade de desenvolvimento, ampla documentação disponível e grande aplicabilidade em prototipagem. Além disso, microcontroladores de 8 bits, como o Arduino, são capazes de executar sistemas operacionais mínimos com desempenho satisfatório [2].

A portabilidade de sistemas operacionais exige a adaptação às características específicas da arquitetura de destino [3,4,5, 6] Sistemas desenvolvidos para uma família de processadores, como ARM, não são diretamente compatíveis com outras, como x86 ou AVR, devido às diferenças no conjunto de instruções, compiladores e ambientes de desenvolvimento. Para portar um sistema operacional é necessário ajustar seu código às particularidades da nova plataforma, respeitando recursos como registradores, timers, mecanismos de interrupção e o ambiente de programação.

O NKE, versão 0.7c, é um nanokernel projetado para fins educacionais, com foco no ensino de sistemas operacionais e programação concorrente. Seu pequeno tamanho (38 KB de código em C e menos de 300 linhas em Assembly) e a predominância do código em C tornam o NKE uma ferramenta ideal para experimentações práticas em disciplinas acadêmicas. Essas características simplificam a compreensão de conceitos fundamentais de sistemas operacionais e permitem a execução de atividades práticas em um curto período de tempo.

O porte do NKE para o Arduino envolveu a adaptação do nanokernel a aspectos específicos da arquitetura AVR, como registradores, timers e mecanismos de interrupção. Questões

críticas, como salvamento e restauração de contexto, tratamento de interrupções do temporizador e troca de contexto entre tarefas, foram tratadas para garantir a funcionalidade multitarefa.

2 ARQUITETURA E FUNCIONAMENTO DO ARDUINO

O Arduino é uma plataforma de prototipagem eletrônica open-source que combina hardware e software de fácil uso. A arquitetura do Arduino é baseada no microcontrolador Atmel AVR, especificamente o ATmega328 no modelo Arduino Uno. Este microcontrolador é responsável por executar o código do usuário, interagir com sensores e atuadores, e gerenciar a comunicação com outros dispositivos.

A arquitetura do AVR possui registradores de uso geral e instruções de baixo nível que possibilitam a manipulação do contexto das tarefas. As funções de manipulação do contexto, escritas em Assembly, possibilitam gerenciar, de forma eficiente, os registradores do microcontrolador para alternar entre diferentes tarefas.

Para tratar as interrupções temporizadas utilizou-se a biblioteca TimerOne, usada para a troca periódica de tarefas, garantindo que cada tarefa receba tempo de CPU adequadamente, essencial em um sistema operacional multitarefas.

O Arduino Uno possui uma memória flash de 32 KB para armazenamento de código, SRAM de 2 KB para dados temporários e EEPROM de 1 KB para dados permanentes. Tem um oscilador de cristal de 16 MHz que fornece um clock que determina a velocidade de execução das instruções no microcontrolador. Os pinos de entrada e saída (GPIO) são configuráveis (digitais ou analógicos) e permitem a interação com sensores e atuadores, enquanto o conversor analógico-digital (ADC) de 10 bits possibilita a leitura de sinais analógicos. O suporte a comunicação serial via interface UART facilita a integração com computadores e outros dispositivos.

A facilidade de uso de sua IDE, permitiu que o trabalho fosse desenvolvido de maneira rápida e eficiente. A IDE do Arduino facilita a escrita, compilação e upload do código para o microcontrolador, permitindo uma interação rápida com o sistema operacional.

3. NKE

O NKE - Nanokernel Educacional para Microprocessadores ARM[4], foi concebido para fins educacionais, projetado com o objetivo de ensinar e explorar conceitos fundamentais de sistemas operacionais. O NKE foi implementado especificamente para microprocessadores ARM, amplamente utilizado em sistemas embarcados. Ele serve como uma plataforma prática para estudantes e desenvolvedores que desejam entender a interação entre o software de baixo nível e o hardware. É um sistema mínimo, com 2600 linhas de código escrito na linguagem C e cerca de 200 linhas em Assembler do microprocessador ARM.

O NKE incorpora funcionalidades básicas de um sistema operacional, como criação e destruição de tarefas, escalonamento de tarefas, gerenciamento de interrupções, temporização e sincronização através de semáforos. O sistema também foi projetado para ser modular, permitindo a expansão e customização conforme necessário, facilitando sua integração em diferentes cenários educacionais.

O sistema suporta diversos algoritmos de escalonamento, incluindo Round Robin (RR), Rate Monothonic (RM) e Earliest Deadline First (EDF)[7], além de permitir a implementação de outros algoritmos.

No NKE, os programas do usuário são executados no nível de privilégio usuário, que possui acesso limitado aos recursos do sistema e ao hardware. Quando um programa necessita de um serviço fornecido pelo kernel, como manipulação de tarefas, temporização ou acesso a dispositivos, ele realiza uma chamada de sistema. Cada serviço disponível no kernel possui uma chamada de sistema correspondente, que serve como a interface entre o nível de usuário e o nível de kernel. Essas chamadas permitem que o programa especifique a operação desejada e os parâmetros necessários.

Em uma chamada de sistema, os parâmetros e a identificação do serviço do kernel são organizados em uma struct, que é passada como referência. A transição do contexto de execução para o nível kernel é realizada por meio da instrução SWI (Software Interrupt), disponível na arquitetura ARM. Essa instrução dispara a execução de uma função no nível kernel, que salva o contexto da task em execução e dispara a rotina correspondente ao serviço solicitado. Ao entrar no nível kernel, ocorre um chaveamento da pilha, onde a pilha de usuário é substituída temporariamente pela pilha do kernel. Isso garante que as operações realizadas no nível kernel não interfiram no estado da pilha do programa do usuário. Após a execução do serviço solicitado, o controle retorna ao programa do usuário, restaurando seu contexto original.

No NKE, o salvamento e restauração de contexto são executados na pilha de cada tarefa. Durante o escalonamento de tarefas, o estado atual de uma tarefa (registradores de uso geral, progam counter e Stack pointer) é salvo para que possa ser restaurado posteriormente, permitindo que a execução continue do ponto onde foi interrompida.

Entre as funcionalidades disponíveis, destacam-se a criação e término de tarefas, o uso de semáforos, mecanismos de temporização, funções de entrada e saída, e o suporte ao salvamento e restauração de contexto.

A criação e término de tarefas é implementada através das funções taskcreate e taskexit. Com taskcreate, é possível instanciar novas tarefas, associando uma função específica que será executada e configurando as prioridades e parâmetros necessários. Já a função taskexit permite que uma tarefa encerre sua execução de forma controlada, liberando os recursos alocados. Os semáforos são utilizados para a sincronização entre tarefas e controle de acesso a recursos compartilhados. No NKE os semáforos são contadores, podendo assumir valores maiores que 1(um). As funções

seminit, semwait e sempost implementam essa funcionalidade. A função seminit inicializa um semáforo, configurando o valor inicial. A função semwait é usada para bloquear uma tarefa até que o recurso esteja disponível, enquanto sempost libera o semáforo, sinalizando que o recurso pode ser utilizado por outras tarefas.

As funções de temporização, como sleep, msleep e usleep, permitem que tarefas sejam suspensas por períodos específicos, expressos em segundos, milissegundos ou microssegundos, respectivamente.

Operações de entrada e saída são executadas com nkprint e nkread. A função nkprint é usada para exibir mensagens e dados em um terminal, enquanto nkread permite que dados sejam lidos do terminal. Essas funções proporcionam uma interface prática para a comunicação entre o sistema operacional e o mundo externo.

4. NKEArduíno

O porte do NKE da plataforma ARM para o Arduino representou um desafio técnico significativo, mas também destacou a flexibilidade e portabilidade do código do kernel, que foi escrito majoritariamente em C. Essa característica permitiu que grande parte das rotinas, incluindo a criação e término de tarefas, semáforos, temporização e funções de entrada e saída, fosse reaproveitada sem modificações substanciais.

No entanto, dois pontos críticos exigiram maior esforço de adaptação: as rotinas de salvamento e restauração de contexto e o mecanismo de troca de contexto. No caso do Arduino, que utiliza o microcontrolador ATmega328, essas rotinas tiveram que ser implementadas diretamente em Assembly, o que exigiu a manipulação dos registradores, incluindo ponteiros de pilha e estados de interrupção, para garantir que o sistema pudesse alternar entre tarefas de maneira correta e segura.

Outro desafio foi a ausência de instruções equivalentes à SWI (Software Interrupt) do ARM, utilizada para chamadas de sistema e troca de contexto. No Arduino, esse mecanismo foi implementado por uma função, disparada pelas chamadas de sistema, que desabilita interrupções e invoca a rotina do kernel que deverá executar o serviço. Se a chamada de sistema é bloqueante (ex. semwait), a task do usuário é bloqueada e é feito um reescalonamento. Caso contrário (ex. getmynumber), as interrupções são habilitadas e a execução retorna a task do usuário.

O uso da IDE Arduino foi fundamental durante o processo de desenvolvimento e porte do NKE para essa plataforma. A simplicidade da IDE facilitou tanto a escrita quanto a carga do sistema no microcontrolador. Além disso, as facilidades de compilação e upload de código, contribuíram para reduzir o tempo necessário para testes e depurações.

Outro recurso indispensável foi o uso das funções Serial.print, essenciais para a implementação de logs de depuração. Essas funções permitiram o registro e a análise de eventos e estados do sistema durante a execução, facilitando a identificação e correção de falhas, especialmente nas partes mais críticas, como o salvamento e restauração de contexto e o escalonamento de tarefas.

O sistema completo, que inclui o kernel NKE e a aplicação, é implementado como um sketch Arduino, resultando em um único executável que pode ser carregado no microcontrolador.

5. Detalhes da Implementação

Uma chamada de sistema no programa de aplicação chama uma User Call. Para cada User Call existe uma System Call correspondente, que é uma rotina do kernel. A troca do modo usuário para o modo kernel é realizada pela rotina callSVC, que recebe os parâmetros da User Call, desabilita interrupções e invoca a rotina kernel, que seleciona a Sys Call que deverá executar o serviço solicitado (Figura 1).

```
// Programa do usuário
Sleep(10);

// User Call
void sleep(int time) {
    Parameters arg;
    arg.CallNumber = SLEEP;
    arg.p0 = (unsigned char *)time;
    callsvc(&arg);
}

// call SVC
void callsvc(Parameters *args) {
    noInterrupts();
    kernelargs = *args;
    kernel();
    interrupts();
}

// kernel
void kernel() {
    switch (kernelargs.CallNumber) {
        case SEM_WAIT:
            sys_semwait((sem_t *)kernelargs.p0);
            break;
        case SLEEP:
           sys_sleep((int)kernelargs.p0);
            break;
        ...
    }
}

// Sys Call
void sys_sleep(unsigned int segundo) {
    Descriptors[TaskRunning].Time = segundo / ClkT;
    if (Descriptors[TaskRunning].Time > 0) {
        Descriptors[TaskRunning].State = BLOCKED;
        switchTask();
}
```

Figura 1: Chamada de sistema sleep

Em uma chamada de sistema, os parâmetros, que podem ser quatro, são armazenados na estrutura de dados *parameters*, juntamente com o número da chamada de sistema (Figura 2).

CallNumber	p0	p1	p2	p3

Figura 2: *E*strutura de dados Parameters.

O sistema possui um conjunto de procedimentos que implementam o gerenciamento de tarefas, sincronização, espera de passagem de tempo e operações de entrada e saída.

5.1 Gerenciamento de tarefas

O gerenciamento de tarefas no sistema é implementado pelas primitivas taskcreate e taskexit. A cada task corresponde um registro descritor no kernel, conforme a figura 3, a seguir.

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

Figura 3: Descritor de tarefas.

O Tid (Task Identifier) é o identificador único de cada tarefa, Esse identificador é fundamental para operações como escalonamento, sincronização e término de tarefas.

Já o campo Prio (Prioridade) define a ordem de execução da tarefa, permitindo que tarefas mais importantes sejam executadas antes das menos prioritárias, dentro do esquema de escalonamento do kernel. O atributo Time indica o tempo que a tarefa deve permanecer em estado de suspensão (sleep), sendo útil para implementar funções de temporização como sleep, msleep e usleep.

O State registra o estado atual da tarefa, que pode ser READY (pronta para executar), RUNNING (em execução) ou BLOCKED (aguardando um recurso ou evento). O campo Stack (um vetor) armazena a pilha da tarefa, utilizada para armazenar variáveis locais e contextos de execução durante as chamadas de função. Por fim, o P (ponteiro para o topo da pilha) aponta para a posição atual na pilha da tarefa, sendo essencial para operações como salvamento e restauração de contexto durante a troca de tarefas.

A função sys_taskcreate, Figura 4, é responsável pela criação de novas tarefas e pela inicialização de seus descritores. Primeiro, o identificador global de tarefas (NumberTaskAdd) é incrementado, garantindo que cada tarefa receba um identificador único. Esse identificador é armazenado no campo Tid do descritor da tarefa correspondente. Além disso, outros campos são inicializados: State é definido como READY, indicando que a tarefa está pronta para ser escalonada; Time é inicializado como 0, representando que não há tempo de espera; e Prio é configurado como 0, indicando a prioridade padrão.

```
void sys_task_create(int *tid, void (*taskFunction)(void)) {
    NumberTaskAdd++;
    *tid = NumberTaskAdd;
    Descriptors[NumberTaskAdd].Tid=*tid;
    Descriptors[NumberTaskAdd].State=READY;
    Descriptors[NumberTaskAdd].Time=0;
    Descriptors[NumberTaskAdd].Prio = 0;
    uint8_t* stack = Descriptors[*tid].Stack+SizeTaskStack-1;
    Descriptors[*tid].P = stack;
    *(stack--) = ((uint16_t)taskFunction) & 0xFF;
    *(stack--) = ((uint16_t)taskFunction >> 8) & 0xFF;
    *(stack--) = 0x00;
    *(stack--) = 0x80;
    for (int i = 1; i < 32; i++) {
         *(stack--) = i;
     }
     Descriptors[*tid].P = stack;
}
```

Figura 4: Função sys_task_create.

A pilha da tarefa é então configurada como sendo o último elemento do vetor Stack. O ponteiro da pilha (P) é ajustado para essa posição e armazenado no descritor da tarefa. Em seguida, o endereço da função que a tarefa executará (EP) é armazenado no topo da pilha, dividido em dois bytes (menos significativo e mais significativo). Um valor fixo 0x00 é empilhado para o registrador R0, e o registrador de status (SREG) é configurado com 0x80 para habilitar interrupções globais. Após isso, os registradores simulados da arquitetura (de R1 a R31) são inicializados com valores fictícios, representando o estado inicial dos registradores. Por fim, o ponteiro da pilha atualizado, após a configuração de todos os valores iniciais, é novamente armazenado no descritor da tarefa.

5.2 Escalonamento de tarefas

O algoritmo de escalonamento utilizado é especificado durante a inicialização do sistema, por meio da chamada de sistema *start(algoritmo),* Atualmente, o único algoritmo suportado é o Round-Robin (RR), mas pretende-se implementar os algoritmos de tempo real RM e EDF.

O escalonador é implementado pelas funções switchTask, sortReadyList,e insertReadyList. A função switchTask é responsável pela alternância entre tarefas, gerenciando o salvamento do contexto da tarefa atual e a restauração do contexto da próxima tarefa a ser executada. Já a função insertReadyList organiza as tarefas na fila de prontas (ready list) com base na prioridade atribuída a cada uma. Ao ser chamada, essa função insere a tarefa na no final da fila e chama a rotina sortReadyList garantindo que tarefas mais prioritárias sejam escalonadas antes das de menor prioridade. A switchTask,Figura 5, funciona da seguinte maneira: Se a task atual não é Idle (TaskRunning != 0), a task é removida da Ready List. Caso não esteja bloqueada, ela é reinserida no final da Ready List. A remoção é feita com o deslocamento para a esquerda Chama a função sortReadyList() Atualiza TaskRunning com a primeira task da Ready List (TaskRunning =

ready_queue.queue[0]). Se a Ready List estiver vazia, TaskRunning será 0 (Idle)

```
void switchTask( ) {
  saveContext(&Descriptors[TaskRunning]);
  if (TaskRunning != 0){
    for (int i = 0; i < ready_queue.head - 1; i++) {
      ready_queue.queue[i] = ready_queue.queue[i + 1];
    }
    ready_queue.head--;
    if (Descriptors[TaskRunning].State != BLOCKED)  {
      InsertReadyList(TaskRunning);
    }
  }
  sortReadyList( );
  if (ready_queue.head > 0){
    TaskRunning = ready_queue.queue[0];
  } else {
    TaskRunning = 0;
  }
  Descriptors[TaskRunning].State = RUNNING;
  restoreContext(&Descriptors[TaskRunning]);
}
```

Figura 5: Implementação switchTask.

Algoritmo Bubble Sort (Figura 6) é utilizado para a reordenação da Ready List. O critério de ordenação é a prioridade (Prio) definida para a task Menor valor numérico indica maior prioridade.

```
void sortReadyList( ) {
    for (int i = 0; i < ready_queue.head - 1; i++) {
        for (int j = 0; j < ready_queue.head - i - 1; j++) {
            if (Descriptors[ready_queue.queue[j]].Prio >
            Descriptors[ready_queue.queue[j + 1]].Prio) {
                int temp = ready_queue.queue[j];
                ready_queue.queue[j] = ready_queue.queue[j + 1];
                ready_queue.queue[j + 1] = temp;
            }
        }
    }
}
```

Figura 6: Implementação sortReadyList.

A função insertReadyList, Figura 7, é chamada quando a espera por passagem de tempo (funções sleep, mslepp e usleep) termina, quando a task deve ser acordada de uma função bloqueante (semwait, nkread) e quando termina a fatia de tempo da task running, que deve ser inserida novamente na ready list.

```
void InsertReadyList(int id) {
   ready_queue.queue[ready_queue.head] = id;
   ready_queue.head++;
}
```

Figura 7: Implementação InsertReadyList.

5.3 Salvamento e Restauração de Contexto

O salvamento e a restauração de contexto são realizados na pilha da task, Figuras 8 e 9, a seguir. São as únicas rotinas escritas em Assembly do microprocessador.

```
void saveContext(TaskDescriptor* task) {
    asm volatile (
        "push r0 \n\t"
        "in r0, __SREG__ \n\t"
        "cli \n\t"
        "push r0 \n\t"
        "push r1 \n\t"
        "clr r1 \n\t"
        "push r2 \n\t"
        "push r3 \n\t"
        "push r4 \n\t"
        "push r5 \n\t"
        "push r6 \n\t"
        "push r7 \n\t"
        "push r8 \n\t"
        "push r9 \n\t"
        "push r10 \n\t"
        "push r11 \n\t"
        "push r12 \n\t"
        "push r13 \n\t"
        "push r14 \n\t"
        "push r15 \n\t"
        "push r16 \n\t"
        "push r17 \n\t"
        "push r18 \n\t"
        "push r19 \n\t"
        "push r20 \n\t"
        "push r21 \n\t"
        "push r22 \n\t"
        "push r23 \n\t"
        "push r24 \n\t"
        "push r25 \n\t"
        "push r26 \n\t
        "push r27 \n\t"
        "push r28 \n\t"
        "push r29 \n\t"
        "push r30 \n\t"
        "push r31 \n\t"
        "in %A0, __SP_L__ \n\t"
        "in %B0, __SP_H__ \n\t"
        : "=r" (task->P)
    );
}
```

Figura: 8: Implementação saveContext.

```
void restoreContext(TaskDescriptor* task) {
    asm volatile (
        "out __SP_L__, %A0 \n\t"
        "out __SP_H__, %B0 \n\t"
        "pop r31 \n\t"
        "pop r30 \n\t"
        "pop r29 \n\t"
        "pop r28 \n\t"
        "pop r27 \n\t"
        "pop r26 \n\t"
        "pop r25 \n\t"
        "pop r24 \n\t"
        "pop r23 \n\t"
        "pop r22 \n\t"
        "pop r21 \n\t"
        "pop r20 \n\t"
        "pop r19 \n\t"
        "pop r18 \n\t"
        "pop r17 \n\t"
        "pop r16 \n\t"
        "pop r15 \n\t"
        "pop r14 \n\t"
        "pop r13 \n\t"
        "pop r12 \n\t"
        "pop r11 \n\t"
        "pop r10 \n\t"
        "pop r9 \n\t"
        "pop r8 \n\t"
        "pop r7 \n\t"
        "pop r6 \n\t"
        "pop r5 \n\t"
        "pop r4 \n\t"
        "pop r3 \n\t"
        "pop r2 \n\t"
        "pop r1 \n\t"
        "pop r0 \n\t"
        "out__SREG__, switchTask r0 \n\t"
        "pop r0 \n\t"
        : : "r" (task->P)
    );
}
```

Figura 9: Implementação restoreContext.

5.4 Interrupções

O NKEArduíno trata somente as interrupções do timer, para fazer escalonamento e gestão de tempo nas chamadas de sistema sleep, msleep e usleep. Para tal, foi utilizada a biblioteca TimerOne, do Arduíno, que abstrai a complexidade do uso de timers em baixo nível. Em vez de manipular diretamente os registradores, se utilizam as funções *Timer1.initialize()* e *Timer1.attachInterrupt()* [6], o que torna o desenvolvimento mais rápido e menos propenso a erros. Na Figura 10, o timer é configurado para gerar interrupções a cada segundo e, sempre que ocorrer uma interrupção (término da fatia de tempo), será disparada a função systemContext (Figura 11), que decrementa o tempo de espera das tasks bloqueadas a espera de passagem de tempo (rotina wakeup), chama SeriaEvent, que testa a serial para ver se há dados. Em caso positivo, os dados são lidos e associados a task que efetuou a operação nkread, que será desbloqueada. A seguir, será disparada a rotina switchTask, que vai realizar um escalonamento.

```
#include <TimerOne.h>
void setup() {
  Timer1.initialize(1000000); // interrompe a cada segundo
  Timer1.attachInterrupt(systemContext); //Função executada
}
```

Figura 10: Uso da biblioteca TimerOne.h.

```
void systemContext() { //Chamada pela interrupcao do Timer
  wakeUP();
  serialEvent();
  switchTask();
}
```

Figura 11: Implementação systemContext.

5.5 Semáforos

Os semáforos são utilizados para implementar mecanismos de exclusão mútua e sincronização, que garantem que processos ou threads possam acessar recursos compartilhados de maneira segura e coordenada. As funções são semwait, sempost e seminit permitem a criação, inicialização, espera pela liberação de recursos compartilhados. A primitiva semwait é apresentada na Figura 12.

```
void semwait(sem_t *semaforo){
    Parameters arg;
    arg.CallNumber=SEM_WAIT;
    arg.p0=(unsigned char  *) semaforo;
    CallSWI(0,&arg);
}
```

Figura 12: Implementação semwait.

sem_t *semaforo: É um ponteiro para o semáforo que será usado na operação. arg.CallNumber = SEM_WAIT: Define o número da chamada de sistema ou ação a ser executada. arg.p0 = (unsigned char *)semáforo é o identificador do semáforo. Ao parâmetros são passados para a rotina callSVC(&arg) (Figura 13), que desabilita interrupções e chama o kernel. A função kernel dispara a sys_semwait() (Figura 14) que efetivamente executa a chamada de sistema. Ao final da execução da função sys_semwait as interrupções são novamente habilitadas.

```
void callsvc(Parameters *args){
  noInterrupts();
  kernelargs = *args ;
  kernel() ;
  interrupts();
}
```

Figura 13: Implementação callsvc.

```
void sys_semwait(sem_t *semaforo){
    semaforo->count--;
    if(semaforo->count < 0) {
      semaforo->sem_queue[semaforo->tail] = TaskRunning;
      Descriptors[TaskRunning].State = BLOCKED ;
      semaforo->tail++;
      if(semaforo->tail == MaxNumberTask-1) semaforo->tail = 0;
        switchTask();
    }
}
```

Figura 14: Implementação sys_semwait.

5.6 Entrada e Saída

Operações de entrada e saída na serial são realizadas por nkread e nkprint. Como qualquer outra chamada de sistema, possuem uma user call e uma sys call relacionada. A função "nkread" faz leitura assíncrona da entrada serial, semelhante ao "scanf" em C. Quando uma task chama "nkread", ela é bloqueada até que a entrada desejada seja recebida, mas outras tasks continuam a executar normalmente. A task que chama "nkread" é adicionada em uma fila, juntamente com o formato esperado da entrada e um ponteiro para onde os dados lidos serão armazenados. A função "serialEvent", chamada em toda interrupção do timer, verifica se há dados disponíveis na Serial. Em caso positivo, acessa o buffer de entrada e chama uma função para converter a string de entrada para o formato adequado (inteiro, float ou string). Após a conversão, a task associada à solicitação de leitura é desbloqueada, reinserida na ready list e pode ser selecionada sua execução.

O nkprint tem interação direta com o Arduino, utilizando a chamada “Serial.print()”. Possibilita a exibição de dados inteiros, float e string.

É importante salientar que nkprint foi de extrema importância no desenvolvimento do NKEArduíno, especialmente no contexto da depuração, permitindo que informações fossem facilmente visualizadas no monitor serial.

5.7 Gerenciamento de Tempo

O gerenciamento de tempo é feito pelas funções Sleep(), mSleep() e usleep, que são implementadas pelas rotinas do kernel sys_sleep(unsigned int seconds) e sys_msleep(unsigned int milliseconds) e sys_usleep(unsigned int croseconds),respectivamente. Os valores dos parâmetros recebidos nas rotinas do kernel são transformados em um número de interrupções do temporizador. Quando essas chamadas de sistema são invocadas, o valor do parâmetro time é armazenado no campo Time do descritor da tarefa chamadora, e o estado da tarefa é alterado para Blocked (Bloqueado), causando a seleção de uma nova tarefa para ser executada no processador.

A cada ocorrência de uma interrupção do temporizador, o campo Time de todas as tarefas no estado Blocked, cujo valor seja diferente de zero, é decrementado. Se esse valor chegar a zero, a tarefa é inserida no final da ready_queue (fila de prontas); o que significa que a tarefa foi despertada.

O código da rotina do kernel sys_sleep() é apresentado na Figura 15.

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

Figura 15: Implementação sys_sleep.

A constante ClkT define a frequência das interrupções do temporizador (em 100 milissegundos), e ticks representa o número de interrupções do temporizador durante as quais a tarefa chamadora permanecerá bloqueada em estado de espera (sleep).

6. Exemplo de programa de aplicação

A Figura 16 apresenta o código de um programa multitask que roda no NKEArduíno.

```
void task1() {
  while (1) {
    nkprint("Task 1 is running");
  }
}
void task2() {
  while (1) {
    nkprint("Task 2 is running");
  }
}
void setup() {
  Serial.begin(9600);
  int tid1, tid2, tid3;
  taskcreate(&tid1, idle1);
  taskcreate(&tid2, task1);
  taskcreate(&tid3, task2);
  Timer1.initialize(1000000);
  Timer1.attachInterrupt(switchTask);
  sei();
  restoreContext(&Descriptors[tid0]);
}
```

Figura 16: Exemplo programa multitask.

O programa possui duas funções, task1 e task2, criadas pela chamada de sistema taskcreate A idle, cujo código faz parte do sistema e consiste somente de um loop eterno, também precisa ser criada. A idle é escalonada quando não tem task da aplicação pronta para rodar. A seguir, configura o timer para gerar interrupções periódicas, a cada segundo, e associa a rotina switchTask a interrupção. Após, habilita interrupções globais (sei) e inicia a execução da tarefa idle (descritor zero) usando restoreContext. Quando ocorrer a primeira interrupção do timer, a switchTask fará

um escalonamento e vai colocar para rodar a primeira task da fila de prontos, que vai executar até que ocorra a próxima interrupção (final da fatia de tempo).

7 DISCUSSÃO

O NKE-Arduíno apresenta uma abordagem didática para o ensino de sistemas operacionais e programação concorrente. Comparado a alguns trabalhos da literatura, sua simplicidade e propósito educacional são seus principais diferenciais. O estudo de Smith e Johnson [8] explora a implementação de sistemas operacionais de tempo real no Arduino, destacando algoritmos de escalonamento e priorização para atender requisitos temporais rigorosos, enquanto o NKE-Arduíno não é projetado para aplicações de tempo real, mas voltado ao ensino dos fundamentos dos sistemas operacionais. Já Brown e Davis [9] analisam a portabilidade de sistemas embarcados para microcontroladores, enfatizando ajustes para lidar com restrições de hardware. O NKE-Arduíno se alinha a esse tema ao mostrar uma implementação prática para o Arduino. O trabalho de Wilson e Martinez [10] propõe melhorias em nanokernels educacionais, com ênfase na acessibilidade ao ensino, uma proposta similar à do NKE-Arduíno, que se concentra em fornecer uma visão clara de multitarefa e sincronização para os estudantes. Por fim, Kaur [11] examina a implementação de sistemas operacionais de tempo real no Arduino, discutindo soluções para limitações de memória e processamento. Embora o NKE-Arduíno compartilhe algumas técnicas básicas, tais como, como gerenciamento de tarefas e tratamento de interrupções, com os sistemas estudados, seu grande diferencial é que prioriza a simplicidade e a utilidade educacional,

8. CONCLUSÃO

O porte do sistema operacional NKE da plataforma ARM para o Arduino Uno foi concluído com sucesso, mantendo todas as suas funcionalidades. O trabalho seguiu uma metodologia estruturada, com encontros semanais e divisão de tarefas entre os membros da equipe. Essa organização permitiu uma primeira versão operacional com três meses de trabalho. Além disso, o Arduino Uno possui características que foram determinantes para o desenvolvimento do projeto, tais como, Linguagem de programação baseada em C/C++ e Timers do ATmega328, cruciais para temporização e escalonamento de tarefas. A IDE do Arduíno, utilizada para desenvolvimento e carga do sistema, foi outro fator que contribuiu enormemente para que se chagasse ao resultado esperado.

O NKE-Arduíno foi utilizado no ano de 2024 em disciplinas de Sistemas Operacionais, Sistemas de Tempo Real e de Projeto de Sistemas Embarcados, do Curso de Engenharia de Computação da Uergs. A disponibilidade do código em um único programa com tamanho reduzido (1100 linhas), com os protótipos das funções seguindo a definição das variáveis, com as funções a nível de usuário e a nível kernel bem identificadas, com a existência de somente duas funções escritas em Assembly, permitiu aos estudantes uma rápida compreensão do funcionamento do sistema. Possibilitou a realização de trabalhos práticos de implementação de algoritmos de escalonamento, de implementação de chamadas de sistema e trabalhos com o emprego de sensores. Viabilizou aliar experiência prática em um sistema multitarefa aos conceitos vistos em salas de aula.

Como próximo passo, planeja-se portar o NKE-Arduino para a plataforma ESP32, ampliando ainda mais sua aplicabilidade em diferentes ambientes de hardware. Aqui se coloca um enorme desafio, que é transformar o NKE-Arduíno em um nanokernel multicore para sistemas embarcados.

REFERENCIAS

[1] Celso M. da Costa, Cassio Brasil, Leonardo L. Silva, Lucas Murliky, João Leonardo Fragoso, Guilherme Debom, Rivalino Matias, Aline Fracalossi, and Margrit Reni Krug. 2016. NKE: an embedded nanokernel for educational purpose. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16). Association for Computing Machinery, New York, NY, USA, 1900–1902. https://doi.org/10.1145/2851613.2851947

[2] LIMA, Charles B de; VILLAÇA, Marco V. M. **AVR e Arduino - Técnicas de Projeto**. Ed. 2. Florianópolis: Edição dos autores, 2012.

[3] J., & Oliveira, M. (2021). Implementação de um Sistema Operacional de Tempo Real no Arduino. Anais do Congresso Brasileiro de Sistemas Embarcados.

[4] Pereira, C., & Souza, A. (2020). Portabilidade de Sistemas Operacionais para Plataformas de Baixo Custo: Um Estudo de Caso com o Arduino. Simpósio Internacional de Sistemas Embarcados.

[5] Lima, M., & Santos, P. (2022). Adaptação de um Nanokernel Educacional para o Arduino. Workshop de Educação em Sistemas Embarcados.

[6] Ferreira, L., & Almeida, B. (2021). Desafios no Porte de Sistemas Operacionais para Microcontroladores: Experiências com o Arduino. Encontro Nacional de Engenharia de Computação.

[7] Liu, C. L., & Layland, J. W. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. Journal of the ACM (JACM), 20(1), 46–61.: 10.1145/321738.321743

[8] Smith, J., & Johnson, E. (2021). *Design and Implementation of a Real-Time Operating System on Arduino Platform*. In Proceedings of the International Conference on Embedded Systems (pp. 45–53).

[9] Brown, M., & Davis, S. (2022). Porting Embedded Operating Systems to Low-Cost Microcontrollers: A Case Study with Arduino. Journal of Embedded Computing, 15(3), 123-134.

[10] Wilson, L., & Martinez, R. (2023). Enhancing Educational Nanokernels for Arduino-Based Learning Environments. In Proceedings of the IEEE International Conference on Teaching, Assessment, and Learning for Engineering (pp. 88-95).

[11] Kaur, A., & Kaur, R. (2021). "Implementing Real-Time Operating Systems on Arduino Platforms: Challenges and Solutions." International Journal of Embedded Systems, 14(1), 25-33.