Universidade do Vale do Itajaí
Curso: Engenharia de Computação
Disciplina: Sistemas em Tempo Real
Professor: Felipe Viel
Avaliação M2 – Linux RT e Servidor Aperiódico

Instruções

- Deverá ser usada as linguagens **C/C++** para o desenvolvimento.
- Não será permitido o uso de bibliotecas ou funções prontas e disponibilizadas em base de dados ou em outros meios, salvo situações expressas no trabalho.
- Os códigos desenvolvidos devem ser postados e estarem funcionais. Caso a aplicação não esteja correta, será atribuída nota proporcional ao trabalho.
- O trabalho poderá ser feito sozinho, em dupla ou trio.
- O(s) aluno(s) deverá(ão) apresentar o trabalho presencialmente. Caso seja requisitado, deverão dar esclarecimentos adicionais. O não comparecimento a explicação, mesmo a adicional, implicará em nota máxima (proporcional ao desenvolvido) 5,0.
- Trabalhos copiados implicará na nota zero para todos os envolvidos.
- Não serão aceitos trabalhos entregues em atraso.
- As aplicações desenvolvidas deverão exibir resultados corretos para qualquer caso de teste possível dentro do especificado pelo enunciado.
- O código fonte deverá estar comentado para auxiliar o entendimento. A postagem deverá ser feita no github. O usuário do professor é: VielF.
- A responsabilidade pela demonstração do código funcionando é do(s) aluno(s).
- Deverá ser entregue (postado) um relatório contendo: a) Identificação do autor e do trabalho; b) enunciado de cada projeto; c) imagens, tabelas e/ou quadros demonstrando os resultados obtidos; d) explicação dos resultados obtidos; e e) descrição das técnicas utilizadas. Poderá ser usado modelo de artigo, usando o template IEEE (versão A4) ou ABNT. O relatório poderá ser entregue até dia 11/12/2025 até as 12h00 (meio-dia). Não será aceito entrega em atraso.
- O trabalho deverá ser postado por um dos alunos no material didático.
- O trabalho deverá ser apresentado e código postado no material didático até as 8h00 do dia 03/12/2025.Não serão aceitos trabalhos entregue em atraso.
    - Apresentação, descrição do problema, utilização correta do português, profundidade de análise a partir das questões elencadas, diagramação (uso de imagens e diagrama para explicar);
  - Codificação totalmente funcional e atendimento aos requisitos.
  - Qualidade da apresentação do trabalho (presencialmente). Compilação do software no momento da apresentação é obrigatório.
  - Pontualidade.
- Distribuição das notas:
  - Desenvolvimento, teste e metodologia: 3,5
  - Defesa explicando desenvolvimento, testes e metodologia: 3,5
  - Relatório: 3,0

Parte 1 – Análise de Linux Kernel RT

Baseado nos conceitos vistos em aula sobre Linux Kernel RT e também demonstração em como carregar um kernel pré-compilado em VirtualBox, você deverá testar 3 aplicações presentes em: Link. Nesse link há uma série de “ferramentas” que poderão ser usadas para testar diferentes características do kernel na sua versão RT. As ferramentas deverão ser explicadas (o que faz, como faz, etc) e também os resultados obtidos. Lembrando que você pode ver o código fonte da ferramenta. Caso opte por variar as configurações da VM (núcleo e memória), poderá testar 2 códigos apenas.

Parte 2 – Evoluir o Trabalho da M2 e a Análise Temporal

Problemática

O conceito de servidores periódicos em sistemas em tempo real é importante para trazer previsibilidade para uma tarefa e sistema. Tarefas periódicas/esporádicas (T1..T4) são escalonadas diretamente pelo processador (ex.: RM, EDF). Tarefas aperiódicas (Tw..Tx) vão para a fila de serviço. Um Servidor (que é ele próprio uma tarefa periódica) acorda todo período Ts, com um "orçamento/slot" de execução Cs. Durante aquele período ele:

- consome a fila de requisições aperiódicas enquanto ainda tem orçamento;
- quando o orçamento acaba, ele volta a "dormir" até o próximo período.

Com isso, a thread Servidor é periódica e protege a escalonabilidade das tarefas periódicas: o uso do processador pelo servidor é limitado a Cs/Ts. Abaixo alguns códigos:

1. Estruturas básicas

Uma fila de requisições aperiódicas e um servidor periódico:

```
#include <pthread.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

////////////////////////  Fila de requisições aperiódicas ////////////////////////////
typedef void (*job_func_t)(void *arg);

typedef struct job {
    job_func_t func;
    void *arg;
    struct job *next;
} job_t;

static job_t *queue_head = NULL;
static job_t *queue_tail = NULL;
static pthread_mutex_t queue_mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t  queue_cond  = PTHREAD_COND_INITIALIZER;

////////////////////////  Enfileira um job (chamado pelas tarefas aperiódicas) ////////////////////////
void enqueue_job(job_func_t f, void *arg) {
    job_t *j = malloc(sizeof(job_t));
    j->func = f;
    j->arg  = arg;
    j->next = NULL;

    pthread_mutex_lock(&queue_mutex);
    if (queue_tail)
        queue_tail->next = j;
    else
        queue_head = j;
    queue_tail = j;
    pthread_cond_signal(&queue_cond);
    pthread_mutex_unlock(&queue_mutex);
}

////////////////////////  Retira um job da fila (usado pelo servidor) ////////////////////////
job_t *dequeue_job(void) {
    job_t *j = queue_head;
    if (!j) return NULL;
    queue_head = j->next;
    if (!queue_head) queue_tail = NULL;
    return j;
}
```

Função auxiliar: dormir até o próximo período. Use tempo absoluto com clock_nanosleep (evita drift):

```
////////////////////  soma nanossegundos a um timespec (periodo) ////////////////////
static void timespec_add_ns(struct timespec *t, long ns) {
    t->tv_nsec += ns;
    while (t->tv_nsec >= 1000000000L) {
        t->tv_nsec -= 1000000000L;
        t->tv_sec  += 1;
    }
}
```

Thread Servidor periódico: Aqui o servidor tem período Ts e orçamento Cs. Para simplificar, assumo que Cs é apenas um limite de tempo em ns; você mede quanto tempo gastou e, quando passar de Cs, para de atender jobs naquele período.

```
typedef struct {
    long period_ns;   // Ts  (por ex. 10ms = 10*10^6 ns)
    long budget_ns;   // Cs  (pode ser <= Ts*Umax)
} server_params_t;

void *server_thread(void *arg) {
    server_params_t *params = (server_params_t *)arg;
    long Ts = params->period_ns;
    long Cs = params->budget_ns;

    struct timespec next_release;
    clock_gettime(CLOCK_MONOTONIC, &next_release);

    while (1) {
        ////////////////////  define momento da próxima ativação
        timespec_add_ns(&next_release, Ts);

        ////////////////////  início do período
        struct timespec start_period;
        clock_gettime(CLOCK_MONOTONIC, &start_period);

        long consumed_ns = 0;

        while (consumed_ns < Cs) {
            ////////////////////  pega um job, se existir
            pthread_mutex_lock(&queue_mutex);
            while (!queue_head) {
                ////////////////////  se não há jobs, pode sair do loop de serviço
                pthread_mutex_unlock(&queue_mutex);
                goto end_of_service;
            }
            job_t *j = dequeue_job();
            pthread_mutex_unlock(&queue_mutex);

            ////////////////////  mede tempo antes/depois de executar o job
            struct timespec t_before, t_after;
            clock_gettime(CLOCK_MONOTONIC, &t_before);
            j->func(j->arg);            // executa requisição aperiódica
            clock_gettime(CLOCK_MONOTONIC, &t_after);

            ////////////////////  atualiza orçamento consumido
            long dt = (t_after.tv_sec - t_before.tv_sec)*1000000000L +
                      (t_after.tv_nsec - t_before.tv_nsec);
            consumed_ns += dt;

            free(j);

            if (consumed_ns >= Cs) {
                // orçamento esgotado neste período
                break;
            }
        }

    end_of_service:
        ////////////////////  Dorme até o instante absoluto da próxima liberação
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                        &next_release, NULL);
    }

    return NULL;
}
```

Criando o servidor com prioridade de tempo real

```
void start_server_thread() {
    pthread_t th;
    pthread_attr_t attr;
    struct sched_param sp;

    pthread_attr_init(&attr);
    pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
    pthread_attr_setschedpolicy(&attr, SCHED_FIFO);  // ou SCHED_RR

    sp.sched_priority = 60; // prioridade RT (ajuste conforme sistema)
    pthread_attr_setschedparam(&attr, &sp);

    static server_params_t params = {
        .period_ns = 10L * 1000000L,   // Ts = 10 ms
        .budget_ns = 3L  * 1000000L    // Cs = 3 ms por período
    };

    pthread_create(&th, &attr, server_thread, &params);
    pthread_attr_destroy(&attr);
}
```

“Tarefas aperiódicas” (Tw, Tz, …): Em vez de criar uma thread nova para cada pedido (ou deixá-la rodar livremente)

```
void meu_job(void *arg) {
    //////////////////////  código que representa a tarefa aperiódica
    printf("Processando requisição aperiódica %d\n", *(int*)arg);
    //////////////////////  ...
}

void requisicao_aperiodica(int id) {
    int *p = malloc(sizeof(int));
    *p = id;
    enqueue_job(meu_job, p);
}
```

Onde entra a “periodicidade da thread”?

A thread servidor se torna periódica porque:

- Calcula next_release += Ts a cada iteração;
- Usa clock_nanosleep(..., TIMER_ABSTIME, &next_release, ...) para acordar exatamente a cada Ts.

Isso implementa exatamente o “Servidor” da figura:

- Fila de tarefas aperiódicas vai para queue_head;
- Servidor periódico vai para server_thread;
- Processador vai para escalonador do Linux com SCHED_FIFO.

Com isso, você deverá escolher uma das funções do trabalho da M2 que é aperiódica e reproduzir em pthread usando o servidor periódico de exemplo nesse link: Código

É interessante compilar com comando **sudo** e executar com comando **sudo** também.