Tarefas cooperativas

Este projeto faz parte do PPOS v2.

O objetivo deste projeto é construir as funções básicas de gestão de tarefas usando as funções de troca de contexto vistas no projeto anterior.

Descritor de tarefa

Cada tarefa existente no sistema deve ter uma estrutura que a representa, ou seja, um descritor de tarefa (*TCB - Task Control Block*). Essa estrutura deve ser definida no arquivo `kernel/tcb.h`:

```
struct task_t
{
   int id;          // identificador da tarefa
   char *name;      // nome da tarefa
   ctx_t context;   // contexto da tarefa
   int status;      // pronta, executando, terminada, ...
   ...              // demais informações, a completar
};
```

Implementação

As seguintes funções devem ser implementadas em `kernel/task.c` ou `kernel/dispatcher.c`, de acordo com os arquivos de cabeçalho correspondentes:

Inicia a gestão de tarefas

```
void task_init()
```

Esta função é chamada por `ppos.c:ppos_init` e inicia as variáveis necessárias à gestão de tarefas. Neste momento, sua principal responsabilidade é definir uma tarefa para o fluxo de execução do núcleo, que pode ser chamada de `task_kernel`, com nome “kernel” e ID 0. Essa tarefa pode ser vista como o equivalente da função `main` no programa `contexts.c` do projeto anterior.

A tarefa do núcleo tem ID 0 e não deve ser criada com `task_create`, pois não usa uma pilha separada nem uma função separada.

Cria uma nova tarefa

```
struct task_t *task_create(char *name,
                           void (*entry)(void *),
                           void *arg)
```

Parâmetros:

- `name`: nome da tarefa (útil nas mensagens na tela), pode ser nulo
- `entry`: função que será executada pela tarefa
- `arg`: parâmetro recebido pela função `entry` ao iniciar a tarefa
- retorno: ponteiro para uma nova tarefa ou NULL, se houver erro

Esta função cria uma nova tarefa. Para isso ela deve:

- alocar e preencher um TCB
- alocar uma pilha
- criar um contexto
- definir um ID
- devolver um ponteiro para o TCB

Ao ser criada, cada tarefa recebe um ID inteiro crescente, iniciando com 1 (o ID 0 (zero) é reservado para a tarefa principal do núcleo).

Destrói uma tarefa

```
int task_destroy(struct task_t *task)
```

Esta função destrói um descritor de tarefa e libera seus recursos (pilha e TCB).

Informa o identificador da tarefa corrente

```
int task_id(struct task_t *task)
```

Esta função informa o identificador numérico (ID) de uma tarefa (ou da tarefa tarefa atual, se `task` for nulo). Não devem existir duas tarefas com o mesmo ID.

Informa o nome da tarefa corrente

```
char *task_name(struct task_t *task)
```

Esta função informa o nome de uma tarefa (ou da tarefa atual, se `task` for nulo).

Transfere o processador para outra tarefa

```
int task_switch(struct task_t *task)
```

Parâmetros:

- `task`: tarefa que irá receber o processador, ou NULL.
- retorno: valor negativo se houver erro, ou zero

Esta é a operação básica de troca de contexto entre tarefas, que usa a função `ctx_switch` da biblioteca de contextos. Ela será chamada sempre que for necessária uma troca de contexto.

Caso `task` seja nulo, a CPU deve ser transferida para a tarefa que criou a tarefa atual, ou seja, a tarefa que estava ativa quando `task_create` criou a tarefa que está executando neste momento.

Observe que a função `task_switch` recebe somente o ponteiro da **próxima** tarefa, então ela deve ter alguma forma de saber quem é a tarefa atualmente em execução, para poder efetuar a troca de contexto.

O despachante

Será necessário implementar um despachante de tarefas básico na função `dispatcher.c:dispatcher`. Ele deve executar estas ações:

1. criar uma tarefa `task_user` para executar o código inicial da aplicação (definido na função `pingpong-task*.c:user_main`)
2. o nome dessa tarefa deve ser `user_main`
3. passar a CPU para essa tarefa, usando a função `task_switch`.
4. quando retornar dessa tarefa, destruí-la e encerrar.

```
dispatcher()
{
    task_user = task_create(...)
    task_switch(task_user)
    task_destroy(task_user)
}
```

A função `dispatcher` é acionada pelo núcleo ao final da inicialização, na função `ppos.c:main`; quando a função `dispatcher` encerrar o núcleo irá desligar o SO.

Observações

A implementação completa deste projeto compreende definir as estruturas de dados necessários para gerenciar as tarefas e implementar as funções acima descritas, comentando detalhadamente o código.

Capriche na implementação, pois esse código será a base de todos os projetos posteriores.

Avisos de compilação (*warnings*) geram **desconto** na nota!

Validação

Seu código deve funcionar adequadamente com os programas de teste, fornecendo as saídas esperadas:

- `test/pingpong-tasks1.c` (saída esperada `test/pingpong-tasks1.txt`)
- `test/pingpong-tasks2.c` (saída esperada `test/pingpong-tasks2.txt`)
- `test/pingpong-tasks3.c` (saída esperada `test/pingpong-tasks3.txt`)

A compilação usando `make` irá gerar os executáveis dos programas de teste acima.

O primeiro programa de teste corresponde ao código `contexts.c` do projeto anterior, reescrito com as funções propostas neste projeto. Comparar os dois códigos pode ajudar a

compreender o que deve ser implementado em cada função.

Arquivos

Os seguintes arquivos são relevantes para este projeto:

- `kernel/tcb.h`: define a estrutura do *task control block*
- `kernel/task.h`: interface da gestão básica de tarefas (**não alterar**)
- `kernel/task.c`: implementação da gestão básica de tarefas
- `kernel/dispatcher.c`: implementação básica do despachante de tarefas
- `test/pingpong-tasks*.c`: programas de teste
- `test/pingpong-tasks*.txt`: saídas esperadas dos programas de teste

Depuração

Todas as funções implementadas devem gerar mensagens de depuração, que permitam acompanhar a execução das tarefas, como no exemplo a seguir:

```
PPOS: system starting
DEBUG: subsystem task initiated
DEBUG: subsystem dispatcher initiated
PPOS: system started (uptime 0 ms)
DEBUG: dispatcher started
DEBUG: task 0 (kernel) create task 1 (user)
DEBUG: task 0 (kernel) switch to task 1 (user)
user: inicio
DEBUG: task 1 (user) create task 2 (ping)
DEBUG: task 1 (user) create task 3 (pong)
DEBUG: task 1 (user) switch to task 2 (ping)
    ping: inicio
    ping: 0
DEBUG: task 2 (ping) switch to task 3 (pong)
        pong: inicio
        pong: 0
DEBUG: task 3 (pong) switch to task 2 (ping)
    ping: 1
DEBUG: task 2 (ping) switch to task 3 (pong)
        pong: 1
DEBUG: task 3 (pong) switch to task 2 (ping)
    ping: 2
DEBUG: task 2 (ping) switch to task 3 (pong)
        pong: 2
DEBUG: task 3 (pong) switch to task 2 (ping)
    ping: 3
DEBUG: task 2 (ping) switch to task 3 (pong)
        pong: 3
DEBUG: task 3 (pong) switch to task 2 (ping)
    ping: fim
DEBUG: task 2 (ping) switch to task 1 (user)
DEBUG: task 1 (user) switch to task 3 (pong)
        pong: fim
DEBUG: task 3 (pong) switch to task 1 (user)
```

```
user: fim
DEBUG: task 1 (user) destroy task 2 (ping)
DEBUG: task 1 (user) destroy task 3 (pong)
DEBUG: task 1 (user) switch to task 0 (kernel)
DEBUG: dispatcher stopping, no more user tasks
DEBUG: task 0 (kernel) destroy task 1 (user)
PPOS: system stopping
PPOS: system stopped (uptime 0 ms)
```

Para facilitar seu trabalho, o arquivo `kernel/macros.h` define uma macro `ppos_debug` para a geração de mensagens de depuração, que funciona igual à função `printf`:

```
ppos_debug("switch task %d (%s) to task %d (%s)\n", ...);
```

Essas mensagens de depuração somente serão geradas se a macro global DEBUG estiver definida no código-fonte (`#define DEBUG`) ou na linha de comando, no momento da compilação (`make debug p1`).

Outras informações

- Duração estimada: 4 horas.
- Dependências:
  - Trocas de contexto