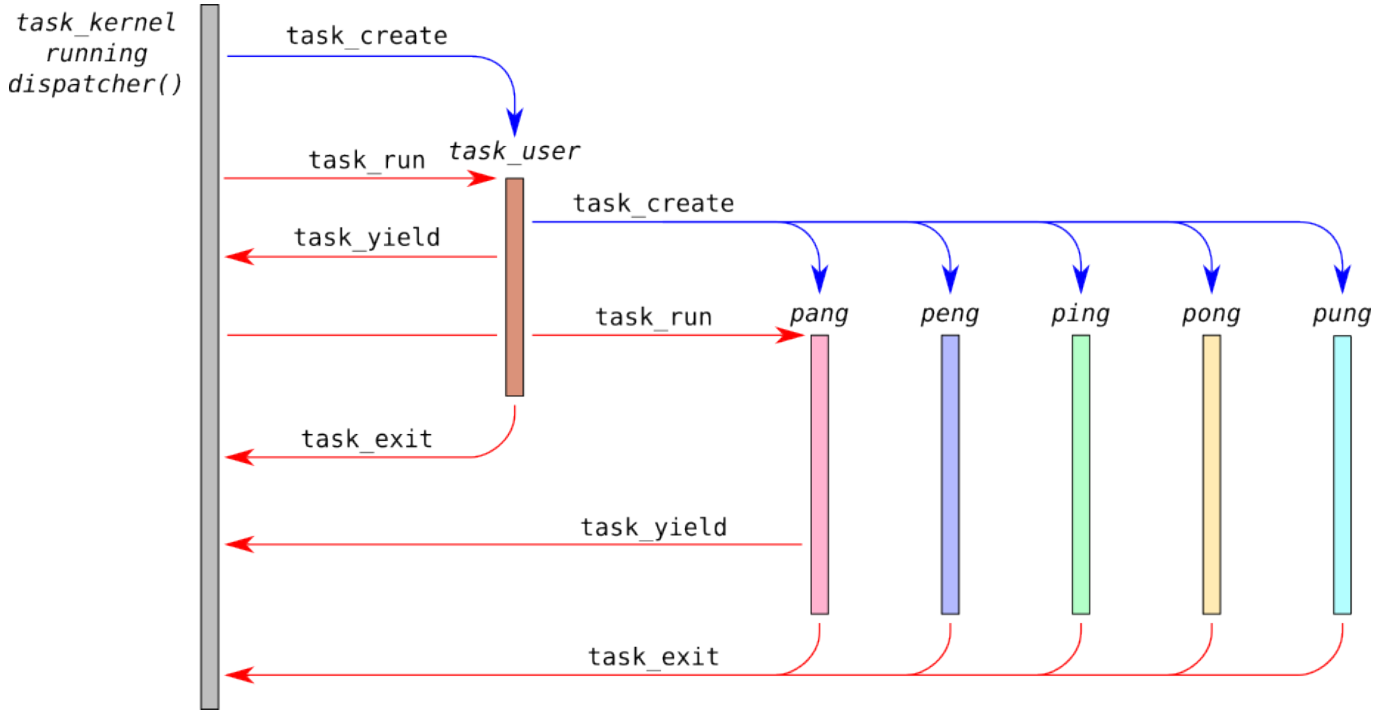
Despachante de tarefas

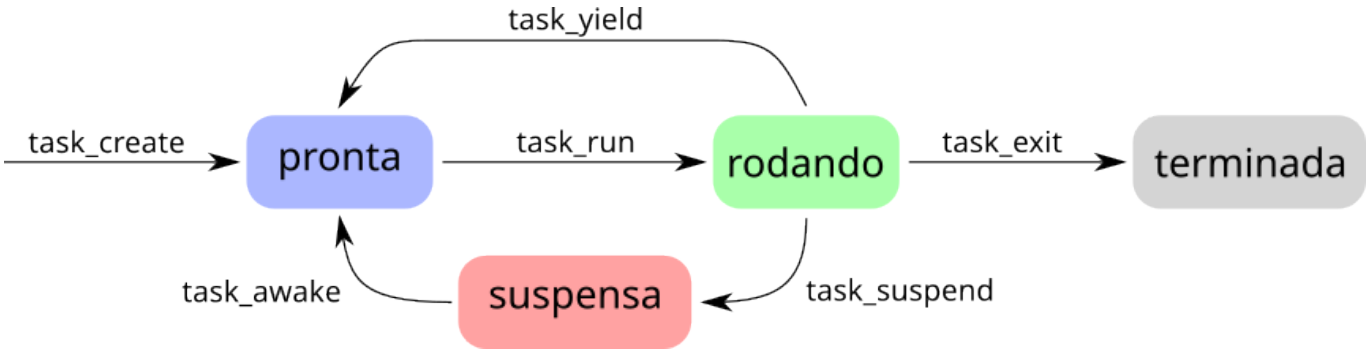
Este projeto faz parte do PPOS v2.

Você irá construir um despachante de tarefas baseado em duas entidades: uma função *dispatcher*, responsável pelo controle dos estados das tarefas, e uma função *scheduler*, responsável por determinar qual a próxima tarefa a executar a cada troca de contexto.

A figura abaixo ilustra o funcionamento geral do despachante:



Essencialmente o despachante funciona como um gerenciador dos estados das tarefas, conforme pode ser observado na figura abaixo. Por isso, a estrutura de controle de cada tarefa deve possuir um campo “estado”, que é mantido atualizado pelas funções implementadas neste projeto.



Implementação

As seguintes funções estão definidas em `dispatcher.h` e devem ser implementadas em `dispatcher.c`:

Inicia o despachante

```
void dispatcher_init()
```

Esta função inicia as estruturas de controle do despachante. Neste projeto, sua ação mais importante é criar a fila de tarefas prontas.

O despachante

```
void dispatcher()
```

Esta função, ativada imediatamente após a inicialização do sistema (em `ppos.c:main`) contém o código principal da função despachante, que executará as tarefas existentes até sua conclusão.

Na versão anterior (1) do PPOS, o despachante era uma tarefa. Nesta versão 2 **o despachante é uma função**, invocada pelo núcleo ao final da inicialização.

O código da função *dispatcher* deve seguir +/- o seguinte modelo:

```
dispatcher()
{
    // cria a tarefa inicial de usuário, que executará user_main()
    task_user = task_create(...)
    // enquanto houver tarefas de usuário
    enquanto ( userTasks > 0 )
    {
        // escolhe a próxima tarefa a executar
        próxima = scheduler()

        // escalonador escolheu uma tarefa?
        se próxima != NULL
        {
            // transfere controle para a próxima tarefa
            task_run(próxima)
            // ao voltar ao dispatcher, trata a tarefa de acordo com seu estado
            caso o estado da tarefa "próxima" seja
            {
                PRONTA    : ...
                TERMINADA : ...
                SUSPENSA  : ...
                (etc)
            }
        }
    }

    // destrói a tarefa inicial do usuário
    task_destroy(task_user)
}
```

Executa uma tarefa

```
void task_run(struct task_t *task)
```

Esta função transfere a CPU para a tarefa `task`, através das seguintes ações:

1. retira a tarefa `task` da fila de prontas;
2. muda o estado da tarefa para “rodando”;
3. transfere a CPU para ela usando `task_switch`.

Libera o processador

```
void task_yield()
```

Esta função interrompe a tarefa atual e retorna a execução ao *dispatcher*, através das seguintes ações:

1. muda o estado da tarefa atual para “pronta”;
2. coloca a tarefa atual no fim da fila de prontas;
3. retorna ao *dispatcher* usando `task_switch`.

Suspende uma tarefa

```
void task_suspend(struct queue_t *queue)
```

Esta função suspende a **tarefa atual** através das seguintes ações:

1. ajusta o status da tarefa atual para “suspensa”;
2. insere a tarefa atual na fila `queue` (se não for nula);
3. retorna ao *dispatcher* usando `task_switch`.

Acorda uma tarefa

```
void task_awake(struct task_t *task)
```

Esta função acorda/ativa a tarefa `task` através das seguintes ações:

1. se a tarefa `task` estiver suspensa em alguma fila, retira-a dessa fila;
2. ajusta o status de `task` tarefa para “pronta”;
3. insere `task` na fila de tarefas prontas;
4. continua a tarefa atual (**não retorna** ao *dispatcher*)

Encerra uma tarefa

```
void task_exit(int exit_code)
```

Esta função encerra a execução da tarefa atual, através das seguintes ações:

1. muda o estado da tarefa para “terminada”;
2. retorna ao *dispatcher* usando `task_switch`.

O parâmetro `exit_code` é um valor devolvido pela tarefa atual; ele deve ser ignorado por enquanto, pois

somente será usado mais tarde.

Outras funções

As funções `task_wait` e `task_sleep` estão definidas no arquivo `kernel/dispatcher.h` mas não precisam ser implementadas neste momento. Elas serão implementadas em um projeto posterior.

Observações

- O *dispatcher* deve ser implementado na função `dispatcher.c:dispatcher` , que é chamada durante a inicialização do sistema.
- A função *dispatcher* só encerra quando não existirem mais tarefas de usuário a executar.
- Será necessário criar uma fila de tarefas prontas em `dispatcher.c:dispatcher_init`, usando a biblioteca de fila genérica desenvolvida anteriormente.
- A função `task_create` deve inserir cada tarefa criada na fila de tarefas prontas.
- A **política de escalonamento** é definida pela função `scheduler.c:scheduler`, chamada pelo *dispatcher* para escolher a próxima tarefa a ativar. Neste projeto, deve ser implementada uma política FCFS (ou seja, apenas informar qual é a primeira tarefa da fila).

Sua implementação deve funcionar com o código de teste em `pingpong-dispatcher.c`. A saída da execução deve ser igual ao conteúdo de `pingpong-dispatcher.txt`.

Arquivos

Os seguintes arquivos são relevantes para este projeto:

- `kernel/dispatcher.h`: interface do dispatcher (**não alterar**)
- `kernel/dispatcher.c`: implementação do diA função dispatcher é acionada pspatcher
- `kernel/scheduler.h`: interface do escalonador (**não alterar**)
- `kernel/scheduler.c`: implementação do escalonador
- `kernel/tcb.c`: define a estrutura do task control block
- `kernel/task.c`: implementação da gestão básica de tarefas
- `test/pingpong-dispatcher.c`: código de teste do dispatcher (**não alterar**)
- `test/pingpong-dispatcher.txt`: saída esperada do teste (**não alterar**)

Uso do Valgrind

O Valgrind é uma ferramenta poderosa para a depuração de problemas de memória, que pode ser muito útil neste projeto. Entretanto, o uso de trocas de contexto em modo usuário confunde esse depurador e gera uma imensa quantidade de avisos errôneos.

Para poder usar o Valgrind corretamente neste projeto, deve-se informar ao Valgrind que estão sendo usados vários contextos com pilhas separadas. Isso é feito através de algumas alterações no código-fonte.

Primeiro, deve-se incluir o arquivo de cabeçalhos do Valgrind:

```
#include <valgrind/valgrind.h>
```

No descritor de cada tarefa, deve-se incluir um campo para uso do Valgrind:

```
struct task_t
{
  ...
  int vg_id;      // ID da pilha da tarefa no Valgrind
  ...
}
```

Após a alocação da pilha de cada tarefa, deve-se **registrar** essa pilha junto ao Valgrind:

```
// aloca a pilha da tarefa
task->stack = malloc(STACKSIZE);
if (!task->stack)
  return (-1);

// registra a pilha da tarefa no Valgrind
task->vg_id = VALGRIND_STACK_REGISTER(task->stack, task->stack + STACKSIZE);
```

Finalmente, quando a pilha da tarefa for liberada, deve-se **desfazer o registro** dessa pilha no Valgrind:

```
// libera a pilha da tarefa
free(task->stack);

// dezfaz o registro da pilha no Valgrind
VALGRIND_STACK_DEREGISTER(task->vg_id);
```

Mais informações sobre essas macros e o uso avançado do Valgrind podem ser obtidas nesta página.

Outras informações

- Duração estimada: 5 horas.
- Dependências:
  - Fila genérica
  - Tarefas cooperativas