Projeto PingPongOS

Autor
Prof. Dr. Carlos Maziero
http://wiki.inf.ufpr.br/maziero/doku.php?id=so:pingpongos

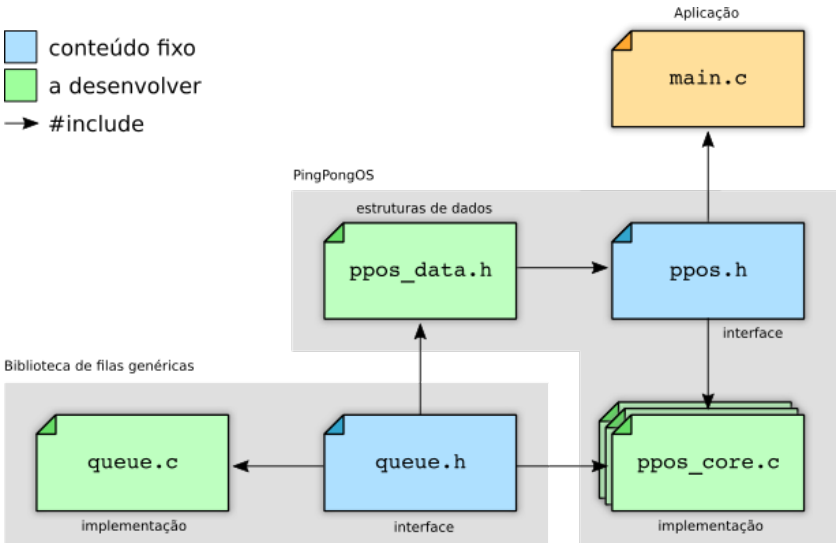
Adaptado por
Prof. Dr. Marco Aurélio Wehrmeister
para o contexto da disciplina CSO30 dos cursos de
Engenharia da Computação e Sistemas de Informação da UTFPR

INTRODUÇÃO

O projeto PingPongOS visa construir, de forma incremental, um pequeno sistema operacional didático. O sistema é construído inicialmente na forma de uma biblioteca de threads cooperativas dentro de um processo do sistema operacional real (Linux, MacOS ou outro Unix).

O desenvolvimento é incremental, adicionando gradativamente funcionalidades como preempção, contabilização, semáforos, filas de mensagens e acesso a um disco virtual. Essa abordagem simplifica o desenvolvimento e depuração do núcleo, além de dispensar o uso de linguagem de máquina.

A estrutura geral do código a ser desenvolvido é apresentada na figura abaixo. Os arquivos em azul são fixos (fornecidos pelo professor), enquanto os arquivos em verde devem ser desenvolvidos pelos alunos.



No contexto das disciplinas "CSO30 – Sistemas Operacionais", turmas S71 (curso de Engenharia de Computação) e S73 (curso de Sistemas de Informação), a implementação de várias funções do PingPongOS será fornecida pelo professor. Os alunos deverão completar essa implementação conforme a descrição dos dois projetos apresentada nas próximas seções.

PROJETO A – Implementação de um Escalonador e Contabilização de Métricas

Objetivos e Requisitos

1. Implementar um escalonador preemptivo baseado em prioridades com envelhecimento
2. Implementar a preempção por tempo (Sistema Multitarefa de Tempo Compartilhado)
3. Fazer a contabilização de métricas sobre a execução das tarefas

1. Implementar um escalonador preemptivo baseado em prioridades com envelhecimento

Este objetivo consiste em implementar um *escalonador* ***preemptivo*** *baseado em* ***prioridades com envelhecimento*** no PingPongOS.

As seguintes operações devem ser implementadas:

1. Uma função *scheduler* que analisa a fila de tarefas prontas, devolvendo um ponteiro para a próxima tarefa a receber o processador.
2. Funções para consulta e ajuste de prioridades:
   a. `void task_setprio (task_t *task, int prio)` - Esta função ajusta a prioridade estática da tarefa *task* para o valor *prio* (que deve estar entre -20 e +20). Caso *task* seja nulo, ajusta a prioridade da tarefa atual.
   b. `int task_getprio (task_t *task)` - Esta função devolve o valor da prioridade estática da tarefa *task* (ou da tarefa corrente, se *task* for nulo).

Observações:

- O escalonador deve usar prioridades no estilo UNIX (valores entre -20 e +20, com escala negativa).
- Para que o escalonador funcione corretamente, ele deve implementar um esquema de envelhecimento de tarefas (*task aging com* $\alpha = -1$). Caso contrário, sempre a mesma tarefa será escalonada para execução. ***O envelhecimento deve ser <u>implementado dentro do escalonador</u>.***
- Ao ser criada, cada tarefa recebe a prioridade default (0).
- Sua implementação deve funcionar com o programa de teste ***pingpong-scheduler.c***. A saída da execução deve ser similar ao *print* que está no arquivo *pingpong-scheduler.txt* (pequenos desvios são aceitáveis). *Para executar este teste, você deve* ***desabilitar a preempção por tempo****.*

- No exemplo acima, observe que a tarefa *pang* executa com mais frequência que *peng*, esta executa com mais frequência que *ping* e assim sucessivamente. Isso mostra claramente a influência das prioridades das tarefas no escalonamento.

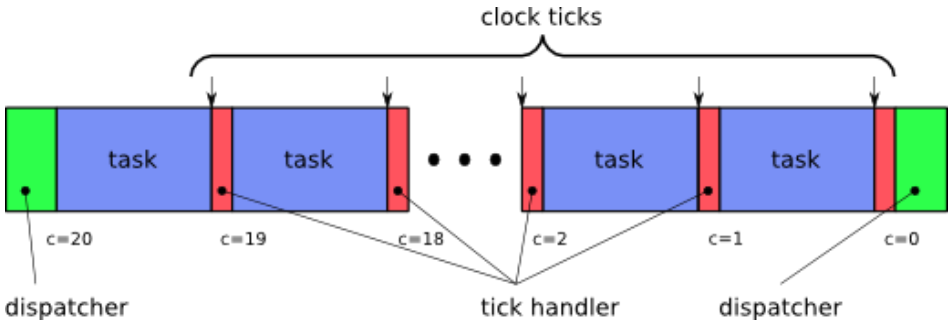
2. Implementar a preempção por tempo (Sistema Multitarefa de Tempo Compartilhado)

Este objetivo consiste em fornecer o suporte necessário para a implementação de tarefas preemptivas, que podem se alternar no uso do processador sem a necessidade de trocas de contexto explícitas através de chamadas da função *task_yield()*.

Em sistemas de tempo compartilhado (*time-sharing*), cada tarefa de usuário recebe uma pequena fatia de tempo de processador, denominada *quantum*. Valores típicos de *quantum* estão entre 1 ms e 100 ms. Ao acabar seu *quantum*, a tarefa em execução retorna à fila de prontas para ceder lugar à próxima tarefa da fila de prontas.

Em um sistema real, a implementação da preempção por tempo tem como base as interrupções geradas pelo temporizador programável do hardware. Esse temporizador é programado para gerar uma interrupção a cada 1 milissegundo, que é tratada por um *interrupt handler* (tratador de interrupção) ou ISR (*Interrupt Service Routine*); essas ativações periódicas do tratador de interrupção são normalmente chamadas de ***ticks do relógio***.

Quando uma tarefa recebe o processador, o *dispatcher* ajusta um contador de *ticks* que essa tarefa pode usar, ou seja, seu quantum definido em número de ticks. A cada *tick*, esse contador deve ser decrementado; quando ele chegar a zero, o processador deve ser devolvido ao *dispatcher* e a tarefa volta à fila de prontas. A figura a seguir ilustra esse conceito:



Importante: *Como um processo UNIX não tem acesso direto aos temporizadores e interrupções do hardware, vamos simular o temporizador de hardware usando um temporizador UNIX, e o mecanismo de interrupção será simulado através de sinais UNIX, que serão explicados a seguir.*

2.1 Sinais UNIX

O mecanismo de sinais do UNIX é similar às interrupções (IRQs) geradas pelo hardware: ao receber um sinal, um processo desvia sua execução para uma função que ele previamente registrou no sistema operacional.

A página de manual *signal* (seção 7, ver signal.txt) relaciona os principais sinais disponíveis em um sistema UNIX e as ações que cada sinal pode desencadear no processo que o recebe. Através da chamada de sistema *sigaction* é possível registrar uma função de tratamento para um determinado sinal (*signal handler function*).

Um exemplo do uso de sinais está no arquivo *signal.c*. Nele, uma função é registrada para tratar o sinal *SIGINT*, que corresponde ao *Ctrl-C* do teclado. Analise atentamente seu código, execute-o e observe seu comportamento.

2.2 Temporizadores UNIX

Para simular as interrupções de relógio do hardware, faremos uso do mecanismo de sinais (para implementar a preempção) e de temporizadores UNIX (para gerar os *ticks* de relógio). O UNIX permite definir temporizadores através das chamadas de sistema *getitimer* e *setitimer*. Ao disparar, um temporizador gera um sinal para o processo, que pode ser capturado por uma função tratadora previamente registrada por ele. O arquivo *timer.c* apresenta um exemplo de uso do temporizador.

2.3 Implementação

O mecanismo a ser implementado pode ser resumido nos seguintes passos:

1. Durante a inicialização do sistema, um temporizador deve ser programado para disparar a cada 1 milissegundo;
2. Os disparos do temporizador devem ser tratados por uma rotina de tratamento de *ticks*;
3. ao ganhar o processador, cada tarefa recebe um *quantum de 20 ticks de relógio* (experimente com diferentes tamanhos de quantum para ver seu efeito);
4. ao ser acionada, a rotina de tratamento de *ticks* de relógio deve decrementar o contador de quantum da tarefa corrente, se for uma tarefa de usuário;
5. se o contador de quantum chegar a zero, a tarefa em execução deve voltar à fila de prontas e o controle do processador deve ser devolvido ao *dispatcher*.

Sua implementação deve funcionar com o programa de teste ***pingpong-preempcao.c*** e deve gerar um resultado similar ao da saída disponível no arquivo ***pingpong-preempcao.txt***. Um teste de

stress, para verificar se seu sistema se comporta bem com muitas tarefas, também está disponível no programa teste ***pingpong-preempcao-stress.c***.

> ***Importante:*** *A rotina de tratamento de ticks de relógio é crítica, pois vai ser executada com muita frequência (1000x por segundo). Essa rotina deve ser pequena e rápida, para não prejudicar o desempenho do sistema e garantir sua estabilidade.*

Observações:

É importante evitar preempções dentro do *dispatcher* ou de funções do sistema, pois estas podem ter resultados imprevisíveis, como condições de disputa e instabilidade. Pode-se controlar a ocorrência de preempções de várias formas. Uma forma conveniente de implementar esse controle usa o conceito de tarefa de sistema:

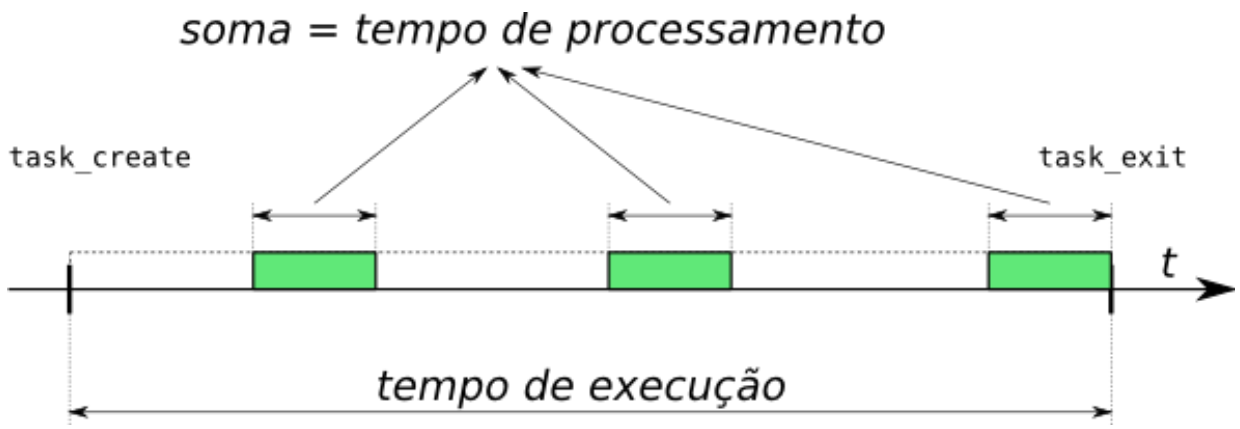
- Tarefas críticas como o *dispatcher* são consideradas tarefas de sistema, pois sua execução correta é fundamental para o bom funcionamento do sistema; ***sua preempção deve ser evitada***.
- As demais tarefas (*main, pang, ..., pung*) são consideradas tarefas de usuário, pois executam código definido pelo usuário do sistema, e podem ser "preemptadas" quando necessário.
- O tratador do temporizador deve sempre verificar se a tarefa corrente é de usuário ou de sistema, antes de preemptá-la devido ao fim de um *quantum*. Pode ser adicionado um flag na estrutura de controle de cada tarefa para indicar se é uma tarefa de sistema ou de usuário.

3. Fazer a contabilização de métricas sobre a execução das tarefas

Este objetivo consiste em adicionar mecanismos para contabilizar o uso do processador pelas tarefas em execução. Sua implementação deve produzir uma mensagem de saída com o seguinte formato, para cada tarefa que finaliza (incluindo o próprio *dispatcher*):

```
Task 17 exit: execution time 4955 ms, processor time 925 ms, 171 activations
```

A figura abaixo ilustra a execução de uma determinada tarefa, de sua criação (*task_create*) ao seu encerramento (*task_exit*). As áreas em verde indicam o uso do processador. É fácil perceber como os valores de contabilização podem ser calculados:



Para a contabilização você precisará de uma referência de tempo (***um relógio***). Para isso, pode ser definida uma variável global para contar *ticks* de relógio, incrementada a cada interrupção do temporizador (1 ms). Dessa forma, essa variável indicará o número de *ticks* decorridos desde a inicialização do sistema na função *ppos_init()*, ou seja, funcionará como um relógio baseado em milissegundos.

Você deverá implementar uma função para informar às tarefas o valor corrente do relógio:

```
unsigned int systime ();
```

Sua implementação deve funcionar com o programa de teste ***pingpong-contab-prio.c*** e deve gerar um resultado similar ao da saída disponível no arquivo ***pingpong-contab-prio.txt***. Neste exemplo, as tarefas devem concluir em instantes bem distintos, mas com consumo de processador e número de ativações similares (pois a carga computacional delas é a mesma).

PROJETO B – Implementação de um Gerenciador de Discos

Objetivos e Requisitos

1. Implementar um gerenciador de disco (virtual)
2. Implementar um escalonador de requisições de acesso ao disco (virtual)

1. Implementar um gerenciador de disco (virtual)

O objetivo é implementar operações de entrada/saída (leitura e escrita) de blocos de dados sobre um disco rígido virtual. A execução dessas operações estará a cargo de um gerente de disco, que cumpre a função de driver de acesso ao disco.

Este gerente deve ser implementado dentro do PingPongOS usando as funções de gerenciamento, comunicação e coordenação de tarefas disponibilizadas no arquivo *ppos.h* cuja implementação é fornecida através dos arquivos objeto.

1.1 Interface de acesso ao disco

A tarefa principal (*main*) inicializa o gerente/driver de disco através da chamada `int disk_mgr_init (&num_blocks, &block_size);`

Ao retornar da chamada, a variável *num_blocks* contém o número de blocos do disco inicializado, enquanto a variável *block_size* contém o tamanho de cada bloco do disco, em bytes. Essa chamada retorna 0 em caso de sucesso ou -1 em caso de erro.

As tarefas podem ler e escrever blocos de dados no disco virtual através das seguintes chamadas (ambas bloqueantes):

```
int disk_block_read  (int block, void* buffer);
int disk_block_write (int block, void* buffer);
```

onde:

- *block*: posição (número do bloco) a ler ou escrever no disco (deve estar entre 0 e numblocks-1);
- *buffer*: endereço dos dados a escrever no disco, ou onde devem ser colocados os dados lidos do disco; esse buffer deve ter capacidade para block_size bytes.
- *retorno*: 0 em caso de sucesso ou -1 em caso de erro.

Cada tarefa que solicita uma operação de leitura/escrita no disco deve ser suspensa até que a operação solicitada seja completada. As tarefas suspensas devem ficar em uma fila de espera

associada ao disco. As solicitações de leitura/escrita presentes nessa fila devem ser atendidas na ordem em que foram geradas, de acordo com a política de escalonamento de disco FCFS (*First Come, First Served*).

1.2 O disco virtual

O disco virtual simula o comportamento lógico e temporal de um disco rígido real, com as seguintes características:

- O conteúdo do disco virtual é mapeado em um arquivo UNIX no diretório corrente da máquina real, com nome default *disk.dat*. O conteúdo do disco virtual é mantido de uma execução para outra.
- O disco contém N blocos de mesmo tamanho. O número de blocos do disco dependerá do tamanho do arquivo subjacente no sistema real.
- Como em um disco real, as operações de leitura/escrita são feitas sempre com um bloco de cada vez. Não é possível ler ou escrever bytes isolados, parte de um bloco, ou vários blocos ao mesmo tempo.
- Os pedidos de leitura/escrita feitos ao disco são assíncronos, ou seja, apenas registram a operação solicitada, sem bloquear a chamada. A finalização de cada operação de leitura/escrita é indicada mais tarde pelo disco através de um sinal UNIX SIGUSR1, que deve ser capturado e tratado.
- O disco só trata uma leitura/escrita por vez. Enquanto o disco está tratando uma solicitação, ele fica em um estado ocupado (*busy*); tentativas de acesso a um disco ocupado retornam um código de erro.
- O tempo de resposta do disco é proporcional à distância entre a posição atual da cabeça de leitura do disco e a posição da operação solicitada. Inicialmente a cabeça de leitura está posicionada sobre o bloco inicial (zero).

O código que simula o disco está em ***disk.c*** e sua interface de acesso está definida em ***disk.h***; estes arquivos não devem ser modificados. ***O arquivo disk.c depende da biblioteca POSIX de tempo real (-lrt)***. Um exemplo de comando para compilação é: `cc -Wall *.o ppos_disk.c disk.c pingpong-disco.c -lrt`

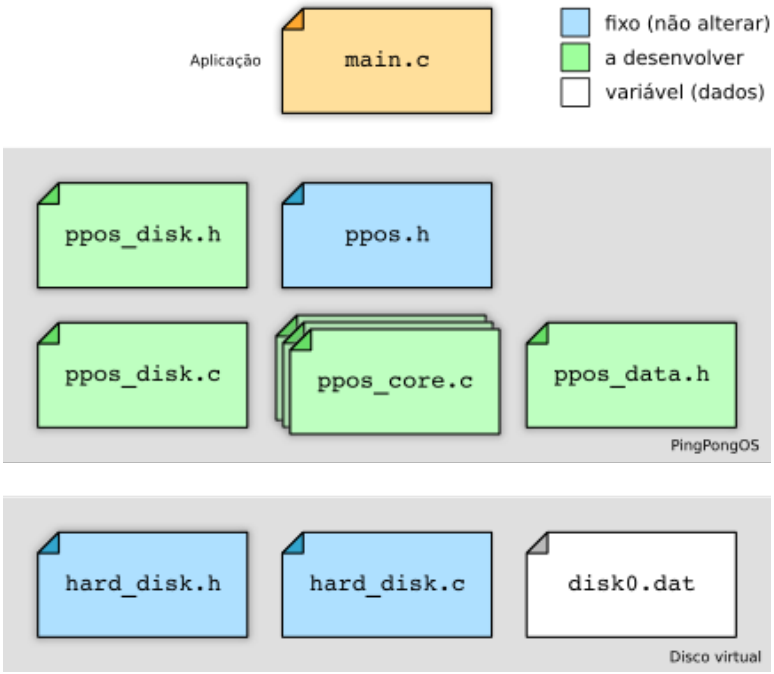
> ***Importante:*** *O acesso ao disco deve feito somente através das definições presentes em disk.h. As* ***definições presentes em disk.c*** *implementam (simulam) o comportamento interno do disco e por isso* ***não devem ser usadas em seu código****.*

1.3 Requisitos para a implementação

A gerência das operações de entrada/saída em disco consiste em implementar:

- Uma tarefa gerenciadora do disco;
- Uma função para tratar os sinais SIGUSR1 gerados pelo disco, que acorda a tarefa gerenciadora de disco quando necessário;
- Uma fila de pedidos de acesso ao disco; cada pedido indica a tarefa solicitante, o tipo de pedido (leitura ou escrita), o bloco desejado e o endereço do buffer de dados;
- As funções de acesso ao disco oferecidas às tarefas através do arquivo *ppos_disk.h* (*disk_mgr_init*, *disk_block_read* e *disk_block_write*).

A implementação do gerenciamento de disco deve ficar no arquivo *ppos_disk.c*, enquanto sua interface fica no arquivo *ppos_disk.h* (fornecido, a completar). A figura a seguir mostra a estrutura geral do código:



Sua implementação deverá funcionar com estes arquivos de teste:

- ***disk.dat***: conteúdo inicial do disco virtual, que tem 256 blocos de 64 bytes cada (b_0, b_1, …, b_{255}), totalizando 16.384 bytes. Para facilitar a visualização, o conteúdo inicial de cada bloco é o número do bloco e alguns caracteres de enchimento para completar o bloco.
- ***pingpong-disco1.c***: uma tarefa única, que lê os blocos do disco em sequência e imprime seu conteúdo na tela. Em seguida, ela escreve nos blocos em sequência, com caracteres aleatórios. A saída deve ser similar ao exemplo contio no arquivo *pingpong-disco1.txt*.

- ***pingpong-disco2.c***: várias tarefas leem e escrevem no disco simultaneamente, com o objetivo de inverter a ordem dos blocos do mesmo (b_{255}, b_{254}, …, b_0). O conteúdo final do disco deve ser igual ao do arquivo disk-final.dat e a saída deve ser similar ao exemplo do arquivo *pingpong-disco2.txt*.

2. Implementar um escalonador de requisições de acesso ao disco (virtual)

O objetivo é implementar diferentes algoritmos de escalonamento de acessos a um disco simulado. A política de escalonamento dos acessos a discos rígidos tem um impacto importante no *throughput* de um sistema (número de bytes lidos ou escritos no disco por segundo). Algumas políticas bem conhecidas são:

- *First Come, First Served* (FCFS): os pedidos são atendidos na ordem em que são gerados pelas tarefas; sua implementação é simples, mas não oferece um bom desempenho;
- *Shortest Seek-Time First* (SSTF): os acessos a disco são ordenados conforme sua distância relativa: primeiro são atendidos os pedidos mais próximos à posição atual da cabeça de leitura do disco.
- *Circular Scan* (CSCAN): os pedidos são atendidos sempre em ordem crescente de suas posições no disco; após tratar o pedido com a maior posição, a cabeça do disco retorna ao próximo pedido com a menor posição no disco.

2.1 Requisitos para a Implementação

Utilizando o gerente de disco desenvolvido anteriormente, o escalonador de requisições de acesso deve:

- Implementar as políticas de escalonamento de disco FCFS, SSTF e CSCAN;
- Para cada política, implementar uma função separada que a implementa;
- Medir o número de blocos percorridos pela cabeça do disco em cada uma dessas três políticas;
- Medir o tempo total de execução para cada uma das três políticas.
- Use o código de teste e os conteúdos de disco do projeto de gerente de disco para efetuar seus testes e medições.