Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Sistemas Operacionais – 2023/2

Trabalho Prático #2 – Escalonador de Processos

1) Introdução

Ao longo desse trabalho utilizaremos um sistema operacional de aprendizado chamado XV6. O XV6 é simples o bastante para modificar e entender em poucas semanas e ao mesmo tempo abrange os conceitos mais importantes da estrutura do UNIX. Para rodá-lo, você irá precisar de compilar os arquivos fontes e usar o emulador de processador QEMU.

Dica: O XV6 era e ainda é desenvolvido como parte da disciplina de Sistemas Operacionais do MIT. Você poderá encontrar informações úteis aqui http://pdos.csail.mit.edu/6.828/2014/xv6.html

Dica: O XV6 também tem um manual bastante detalhado. Esse manual irá te ajudar muito ao longo do trabalho. **Leia**! https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf

2) Obtendo o XV6

Primeiramente devemos baixar o XV6 a partir do repositório no github do MIT.

- Abra o *shell* no seu sistema operacional, crie uma pasta para o trabalho, por exemplo, tp2 e acesse essa pasta:

```
$ mkdir ~/tp2
$ cd tp2
```

- Execute o comando abaixo:

```
$ git clone https://github.com/mit-pdos/xv6-public.git
```

  Esse comando irá baixar o XV6 para a pasta xv6-public.

- Execute o comando a seguir para instalar o QEMU:

```
$ sudo apt install qemu
```

- Agora, acesse a pasta xv6-public e compile o XV6 com o make:

```
$ cd xv6-public
$ make
```

- Para rodar o XV6 dentro do QEMU execute o comando abaixo:

```
$ make qemu
```

Esse comando irá compilar o QEMU e vai iniciar automaticamente o XV6 em uma janela separada dentro do QEMU. Para sair, feche a janela ou digite o comando `quit` no emulador.

3) Escalonador

O escalonador é um dos componentes mais básicos e importantes de um sistema operacional. O escalonador deve satisfazer vários objetivos conflitantes: rápido tempo de processamento, boa vazão para tarefas que rodam em segundo plano, evitar a inanição de processos, conciliar as necessidades de processos de alta a baixa prioridades, etc.

O conjunto de regras usados para determinar quando e como selecionar um novo processo para executar é chamado de *política de escalonamento*.

Sua primeira tarefa é entender a política atual de escalonamento do XV6. Para isso, **LEIA** o capítulo 5 do manual do XV6[1]. Em seguida, localize o código do escalonador no código fonte do XV6 e tente entender o seu fluxo de execução. Em seguida, responda as seguintes questões:

- Qual a política de escalonamento é utilizada atualmente no XV6 ?
- Quais processo essa política seleciona para rodar ?
- O que acontece quando um processo retorna de uma tarefa de I/O ?
- O que acontece quando um processo é criado e quando ou quão frequente o escalonamento acontece ?

Sua primeira tarefa será modificar a política atual do escalonador para que o processo de preempção ocorra a cada intervalo `n` de tempo (medidos em *ticks* do clock) ao invés de a cada 1 *tick* do clock. Adicione a linha seguinte ao arquivo `param.h` e inicialize o valor INTERV para 5. Utilize essa constante para estabelecer o intervalo entre preempções.

```
#define INTERV 5
```

4) Escalonamento de Filas Multinível

O escalonamento de filas multinível (*Multi-Level Queue Scheduling*) é uma política de escalonamento preemptiva que inclui 3 filas de prioridades. Inicialmente, cada processo deve ser iniciado com prioridade padrão 2 e essa prioridade deve ser atribuída a partir da chamada `fork`. Nessa política de escalonamento, o escalonador irá selecionar um processo de uma fila de baixa prioridade somente se não houver processo pronto para rodar em uma fila de prioridade mais alta. A prioridade 3 é a mais alta, a prioridade 2 é a intermediária e a prioridade 1 é a mais baixa. A mudança manual de um processo entre filas de prioridades é possível via *system call*. **Para isso, você deverá criar a chamada de sistema change_`prio` que altera a prioridade atual do processo.**

[1] https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf

```
int change_prio(int priority)
```

O parâmetro `priority` será um número de inteiro do conjunto `{1,2,3}` indicando a nova prioridade do processo. Essa função retorna `0` se bem sucedido ou `-1` se houver um erro.

Dica: para aprender como criar uma chamada de sistema consulte o tutorial em http://cse.csusb.edu/tongyu/courses/cs660/labs/lab3.php

Sua tarefa agorá será modificar o atual escalonador do `XV6` para implementar uma política de escalonamento de filas multinível como descrito acima. Posteriormente, você deverá ainda adicionar um mecanismo de *aging* para evitar inanição. Isto é, processos que estejam por muito tempo em uma fila de baixa prioridade podem ser promovidas à fila de prioridade mais alta seguinte. Dessa forma, você deverá criar mais duas constantes de tempo para comparar com a informação do tempo de espera de um processo e verificar se ele deve ser "passado" para uma fila de maior prioridade. Você pode começar com o valor de 200 *ticks* mas deverá também experimentar outros valores diferentes para as constantes abaixo:

```
//Promove um processo da fila 1 para fila 2 se tempo de espera maior que 5 ticks
#define 1TO2 200

//Promove um processo da fila 2 para fila 3 se tempo de espera maior que 100
ticks
#define 2TO3 100
```

5) Testes

Para analisar o efeito da política implementada deveremos extrair algumas estatísticas de cada processo executado**. Para isso você deverá estender a estrutura `proc` no arquivo `proc.h`. Adicione à estrutura `proc` os seguintes campos:**

```
uint ctime; // Tempo quando o processo foi criado
int stime; //Tempo SLEEPING
int retime; //Tempo READY(RUNNABLE) time
int rutime; // Tempo executando (RUNNING)
```

Esses campos representam respectivamente o tempo da criação do processo, o tempo gasto no estado `SLEEPING`, o tempo gasto no estado `READY` e tempo gasto executando (`RUNNING`).

A partir da criação do processo o *kernel* deverá atualizar o campo `ctime`. Os outros campos deverão ser atualizados a cada *tick* do clock. Assuma que um processo está no estado `SLEEPING` somente quando estiver esperando uma tarefa de I/O. Tome cuidado ao marcar o tempo de terminação de um processo: note que um processo pode ficar um tempo arbitrário no estado `ZOMBIE`.

Para extrair essas informações de cada processo e apresentá-la para o usuário você também deverá criar outra chamada de sistema. Essa nova chamada de sistema `wait2` deverá estender a chamada de sistema atual `wait`:

```
int wait2(int* retime, int* rutime, int* stime)
```

Essa chamada irá atribuir ao parâmetro `retime` o tempo em que o processo esteve no estado `READY`, irá atribui ao parâmetro `rutime` o tempo em que o processe esteve no estado `RUNNING` e ao parâmetro `stime` o tempo em que o processo esteve no estado `SLEEPING`. Deverá ainda retornar `0` se for bem sucedido ou `-1` se houver erro.

Uma vez criada essa chamada de sistema **você deverá criar um programa chamado `sanity.c` que recebe como argumento um parâmetro inteiro `n`. Esse programa irá criar `3*n` processo com `fork` e esperar até que cada um deles termine imprimindo as estatísticas da chamada de sistema `wait2` para cada processo terminado.**

Cada um dos `3n` processos será de um dos 3 tipos abaixo:

- Processos com (`pid mod 3 == 0`) serão processos do tipo **`CPU-Bound`**: executam 100 vezes um *loop* vazio de 1000000 iterações.
- Processos com (`pid mod 3 == 1`) serão processos de tarefas curtas **`S-CPU`**: executam 20 vezes um *loop* vazio de 1000000 iterações e a cada passada das 100 chama a função do sistema `yield`.
- Processos com (`pid mod 3 == 2`) serão processos **`IO-Bound`**: para simular chamadas de IO executa 100 vezes a chamada de sistema `sleep(1)`.

Esse programa deverá ainda imprimir para cada processo terminado:

- O identificado do processo (`pid`) e o seu tipo `{CPU-Bound, S-Bound, IO-Bound}`.
- O seu tempo de espera, tempo executando e tempo de IO.

Após todos os `3n` processo serem executados imprima:

- Tempo médio no estado `SLEEPING` para cada tipo de processo.
- Tempo médio `READY` para tipo de processo.
- Tempo médio para completar para completar (*turnaround time*) a tarefa para tipo de processo.

Para verificar se o algoritmo de escalonamento está obedecendo a ordem de prioridades você poderá ainda criar um programa que cria, por exemplo, 20 processos do tipo `CPU-Bound` e atribui a cada um deles prioridades diferentes através da chamada de sistema `set_prio`. Avalie também qual o efeito do *aging* na ordem de execução dos processo à medida em que as constantes do mesmo são modificadas.

6) Entrega

Esse trabalho poderá ser feito em dupla.

A data de entrega é 27/11/2023 até 23:55 via *moodle*.

Seu grupo deverá submeter no *moodle* um único arquivo no formato `zip` contendo o código fonte do `XV6` (pasta `xv6-public`) e um relatório (`pdf`) contendo nomes dos componentes do grupo e explicando os algoritmos implementados e a análise dos resultados dos testes obtidos. A análise de resultados é ponto importante do trabalho. Varie as constantes levantadas acima, por exemplos de ticks, para os algoritmos e aging e análise o impacto para os resultados. Compare o algoritmo multi-filas com o padrão do xv6.

7) Esclarecimentos

1. **Leia o manual do XV6!** Principalmente o capítulo 5. O entendimento desse capítulo irá ajudar bastante você a entender o funcionamento do escalonador do `XV6` e como modificá-lo para esse trabalho.
2. Não deixe para fazer esse trabalho na última hora! Comece a fazê-lo o quanto antes!
3. O trabalho será avaliado em um Linux Ubuntu.
4. Bom trabalho a **todos!**