Sistemas Operacionais
Escalonador de Processos

Felipe Cadar Chamone

October 2020

1 Introdução

A tarefa geral desse trabalho é ver na prática como funciona a implementação de um escalonador de processos e avaliar o resultado das mudanças aplicadas.

2 Tarefa 1

A primeira tarefa consiste em implementar a variável INTERV para definir um intervalo de preempção em ticks do processador. Pela documentação do XV6 sabemos que a parte de traps cuida da preempção de processos. No arquivo *trap.c* temos um if que verifica se a interrupção foi causada pelo timer, então apenas adicionamos a condição de que o número de ticks módulo INTERV deve ser zero.

3 Tarefa 2

A segunda tarefa envolve o escalonamento de múltiplas filas de prioridade. A parde de escalonamento atualmente é feita dentro do proc.c na função scheduler(). Primeiro adicionamos a propriedade "priority" ao processo, depois iniciamos essa variável com "2" na função allocproc e em seguida, ao procurar pelo próximo processo pronto, a função scheduler primeiro procura pelos processos com prioridade 2, depois com prioridade 1 e depois com prioridade 0. Felizmente conseguimos fazer isso com apenas um loop.

Para adicionar a função *set_prio*() como chamada de sistema foi preciso consultar um tutorial [2]. Na implementação da chamada de sistema chamamos um outra função que implementei dentro do proc.c para mudar a prioridade do processo.

4 Tarefa 3

A tarefa 3 envolve em implementar o sistema de envelhecimento de processos, para evitar inanição. Para isso adicionamos também as variáveis de contabilização de tempo em cada estado do processo, essas variáveis são atualizadas pela trap acionada para atualizar o tick do sistema. Sempre que o tick é atualizado também verificamos se algum processo precisa passar pelo envelhecimento.

5 tarefa 4

Agora precisamos fazer o programa sanity.c. Para ele funcionar foi preciso inserir outra chamada de sistema, a yield().

Um output do programa sanity é:

```
$ sanity 2
[SANITY] - Running 10 procs
PID: 4 S-CPU     - RET 0 RUT 2 ST 476
PID: 5 IO-Bound  - RET 0 RUT 0 ST 100
PID: 6 CPU-Bound - RET 0 RUT 1 ST 0
PID: 7 S-CPU     - RET 0 RUT 1 ST 494
PID: 8 IO-Bound  - RET 0 RUT 0 ST 100
PID: 9 CPU-Bound - RET 0 RUT 1 ST 0
PID: 10 S-CPU     - RET 0 RUT 1 ST 515
PID: 11 IO-Bound  - RET 0 RUT 1 ST 100
PID: 12 CPU-Bound - RET 0 RUT 1 ST 0
PID: 13 S-CPU     - RET 0 RUT 1 ST 500
Results:
CPU-Boud: Sleep 0 Ready 0 Turnaround 1
S-CPU:    Sleep 496 Ready 0 Turnaround 497
IO-Bound: Sleep 100 Ready 0 Turnaround 100
```

O loop básico do programa é:

```
for i in n_procs:
    f = fork()
    if f == 0:
        process()
        exit()
    else:
        wait()
```

Isso significa que sempre teremos no máximo 2 processos, um processo pai (o main) e um filho (o fork), ao mesmo tempo no processador. Então é de se esperar que o tempo de *ready* dos processos seja zero, uma vez que sempre que um fork entra no processador ele já está pronto e livre para ser executado. No caso de **CPU-Bound** a execução é extremamente rápida e eficiente, entao na maioria das vezes ela ocorre dentro de um mesmo tick. No caso **S-CPU** percebemos uma quantidade elevada de tempo em sleep, como era de se esperar pela quantidade de *yield()* usada. E no tipo **IO-Bound** usamos apenas o *sleep(n)*, que faz o programa dormir por uma quantidade n de ticks (no caso 1) e mudar o contexto para o scheduler().

Para adicionar esse programa ao sistema operacional como um comando de terminal foi usado um tutorial no internet [1]

References

[1] How to add a user program to xv6 – ampleux. `https://ampleux.wordpress.com/2018/02/22/how-to-add-a-user-program-to-xv6/`. (Accessed on 10/01/2020).

[2] Xv6 system call. `https://arjunkrishnababu96.gitlab.io/post/xv6-system-call/`. (Accessed on 10/01/2020).