

Documentação TP02

Sistemas Operacionais

Alunos:

Gabriel Sacoman Teixeira Silva - 2019054560

Lucca Silva Medeiros - 2019054773

Questões Abertas

• Qual a política de escalonamento é utilizada atualmente no XV6?

Segundo a documentação o xv6 troca o processamento de um processo em duas situações:

- Coloca pra dormir processos que estejam esperando IO ou estejam esperando processos filhos acabarem ou esperando no sleep system call.
- Periodicamente o xv6 força a troca de processamento quando um processo está executando uma instrução de usuário.

Logo, podemos concluir que a política de escalonamento utilizada é Round-robin (RR). Uma vez que periodicamente o xv6 força essa troca de processamento, atribuindo fatias de tempo a cada processo.

• Quais processos essa política seleciona para rodar?

Essa política atribui fatias de tempo a cada processo em partes iguais e em ordem circular, manipulando todos os processos sem prioridade. Logo essa política seleciona os processos em ordem de chegada (FIFO).

Especificamente no xv6 o escalonador procura na tabela de processos (ptable.lock) um processo que esteja em estado RUNNABLE. Seguindo a ordem de chegada ele seleciona o próximo processo que será executado e altera seu estado para RUNNING, e o processamento pode ser iniciado.

• O que acontece quando um processo retorna de uma tarefa de I/O?

Enquanto um processo espera uma tarefa de I/O ele está em estado SLEEPING. Ao retornar dessa espera o processo chama a função wakeup(), fazendo com que seu estado mude para RUNNABLE, o deixando elegível novamente para continuar sua execução.

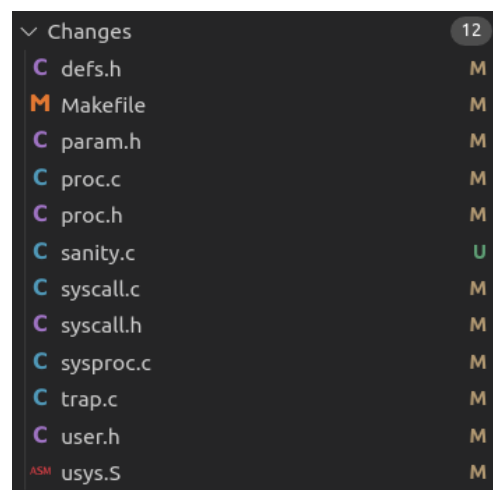
- **O que acontece quando um processo é criado e quando ou quão frequente o escalonamento acontece?**

Quando um processo é criado a função `allocproc()` cria uma estrutura `proc` na tabela de processos (`ptable.lock`), além disso, inicializa as estruturas necessárias do processo para que seu thread do kernel consiga ser executado. Uma vez inicializado, a função `userinit()` marca o processo recém criado como disponível para o escalonamento, setando seu estado (`p->state`) para `RUNNABLE`.

A respeito da frequência, o escalonamento executa um loop e uma interrupção de tempo ocorre a cada 100 vezes por segundo. Isso acontece para que o kernel alterne o uso da CPU entre os processos.

Implementação

Para implementar as modificações exigidas pelo trabalho prático, alteramos os seguintes arquivos do repositório original do xv6:



| Changes | 12 |
|-------------|----|
| C defs.h | M |
| M Makefile | M |
| C param.h | M |
| C proc.c | M |
| C proc.h | M |
| C sanity.c | U |
| C syscall.c | M |
| C syscall.h | M |
| C sysproc.c | M |
| C trap.c | M |
| C user.h | M |
| ASM usys.S | M |

Todas as mudanças podem ser facilmente encontradas ao se pesquisar pelo comentário “MOD”, o qual assinalamos e explicamos tudo o que foi feito para implementar e testar a nova política de escalonamento no xv6.

A primeira ação foi 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. Adicionamos a linha seguinte ao arquivo `param.h` e inicializamos o valor `INTERV` para 5.

[ARQUIVO] param.h

```
// [MOD] // Adicionando constantes

// Define 5 ticks de processo para execução antes de preemptar
#define INTERV 5
```

Essa constante é utilizada na função trap(struct trapframe *tf), responsável por gerir os recursos quando ocorre uma interrupção no sistema:

[ARQUIVO] trap.c

```
if(myproc() && myproc()->state == RUNNING &&
    tf->trapno == T_IRQ0+IRQ_TIMER &&
    // [MOD] // Verifica se INTERV ticks já se passaram desde que o processo foi alocado na CPU.
    myproc()->rtime % INTERV == 0)
    yield();
```

Então, para implementar a política de escalonamento multinível inicia-se a implementação da função: int set_prio(int priority):

importando a nova função corretamente nos arquivos necessários:

1)

[ARQUIVO] defs.h

```
// [MOD] // Adicionando importações
int set_prio(int);
```

2)

[ARQUIVO] user.h

```
// [MOD] // Adicionando importações
int set_prio(int);
```

3)

[ARQUIVO] usys.S

```
SYSCALL(set_prio)
```

Segue a implementação da função responsável pela atribuição de prioridade dos processos. Ela somente aceita as prioridades válidas 1, 2 ou 3. Sendo 3 a prioridade mais alta e 1 a mais baixa.

Definindo o inteiro que armazenará a prioridade de cada processo:

[ARQUIVO] `proc.h`

```
int priority; // [MOD] // Prioridade
```

Implementando a função:

[ARQUIVO] `proc.c`

```
// [MOD] // Função de atribuição de prioridade
int
set_prio(int priority)
{
    if((myproc()->killed) || (priority < 1) || (priority > 3)){
        return -1;
    }

    else{
        myproc()->priority = priority;
        return 0;
    }
}
```

Finalizada essa implementação, criamos a chamada de sistema que aciona de fato a já implementada `set_prio()`, a fim de realizar a alocação de prioridade nos processos a serem escalonados:

Implementando a `sys_set_prio(void)`:

[ARQUIVO] `sysproc.c`

```
// [MOD] // Adicionando funções
int
sys_set_prio(void)
{
    int priority;
    argint(0, &priority);

    if((priority < 1) || (priority > 3)){
        return -1;
    }

    return set_prio(priority);
}
```

Importando a nova função criada:

1)

[ARQUIVO] syscall.c

```
// [MOD] // Adicionando systemcalls
extern int sys_set_prio(void);
```

```
// [MOD] // Adicionando systemcalls
static int (*syscalls[])(void) = {
//...
[SYS_set_prio] sys_set_prio,
//...
};
```

2)

[ARQUIVO] syscall.h

```
// [MOD] // Adicionando constantes
#define SYS_set_prio 22
```

Agora, inicia-se a implementação da nova política de escalonamento de fato. Criamos 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 (futuramente utilizaremos essas constantes para o algoritmo de aging, a fim de evitar inanição dos processos).

[ARQUIVO] param.h

```
// [MOD] // Adicionando constantes

//Promove um processo da fila 1 para fila 2 se tempo de espera maior que 5 ticks
#define T1T02 5

//Promove um processo da fila 2 para fila 3 se tempo de espera maior que 5 ticks
#define T2T03 5
```

Definimos também a prioridade padrão = 2 a todos os processos:

[ARQUIVO] proc.c

```
// [MOD] // Prioridade padrão = 2 para todo processo alocado.
p->priority = 2;
```

Implementando o escalonador...

Essa é a função responsável pelo escalonamento dos processos no xv6. Ela procura por processos que estejam em estado RUNNABLE. Ao encontrar um processo nesse estado o escalonador o deixa pronto para ser processado pela CPU.

[ARQUIVO] `proc.c`

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;
    for(;;){
        sti();

        acquire(&ptable.lock);
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->state != RUNNABLE)
                continue;

            c->proc = p;
            switchvm(p);
            p->state = RUNNING;

            swtch(&(c->scheduler), p->context);
            switchkvm();

            c->proc = 0;
        }
        release(&ptable.lock);
    }
}
```

Após implementar a nova política de escalonamento proposta, a função scheduler(void) ficou assim:

[ARQUIVO] proc.c

```
void
scheduler(void)
{
    struct proc *p;
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){

        sti();

        acquire(&ptable.lock);

        // [MOD] // Evitar Inanição aumentando a prioridade dos processos com o tempo
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

            if(p->priority == 1 && p->state == RUNNABLE && p->retime % T1T02 == 0) {
                p->priority = 2;
                continue;
            }

            if(p->priority == 2 && p->state == RUNNABLE && p->retime % T2T03 == 0) {
                p->priority = 3;
                continue;
            }
        }

        // [MOD] // Escalonamento multinivel
        int flag = 0;
        // Prioriza os processos com prioridade = 3
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
            if(p->priority == 3 && p->state == RUNNABLE) {
                flag = 1;
                break;
            }
        }
    }
}
```

```

if (!flag) {
    // Após procura processos de prioridade = 2
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->priority == 2 && p->state == RUNNABLE) {
            flag = 1;
            break;
        }
    }
}

if (!flag) {
    // Por último processos de prioridade = 1
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
        if(p->priority == 1 && p->state == RUNNABLE) {
            flag = 1;
            break;
        }
    }
}

if (flag) {
    c->proc = p;
    switchvm(p);
    p->state = RUNNING;
    swtch(&(c->scheduler), p->context);
    switchkvm();
    c->proc = 0;
}
release(&ptable.lock);
}
}

```

Agora o escalonador segue a política de Filas Multinível. Onde se prioriza processos com prioridade = 3, depois = 2, e por último processos com prioridade = 1. Além disso foi implementada também uma lógica de aging, a fim de evitar a inanição de processos:

```

if(p->priority == 1 && p->state == RUNNABLE && p->retime % T1T02 == 0) {
    p->priority = 2;
    continue;
}

if(p->priority == 2 && p->state == RUNNABLE && p->retime % T2T03 == 0) {
    p->priority = 3;
    continue;
}

```


O que esses condicionais fazem é basicamente verificar se para cada processo disponível para processamento (`p->state == RUNNABLE`), se o tempo em que o processo está em `READY(RUNNABLE)` é maior do que a constante que define esse limite. Caso um processo satisfaça essas condições sua prioridade é aumentada a fim de garantir seu processamento.

Vê-se então necessário implementar uma estrutura que seja capaz de medir esse tempo, e estendendo para análises futuras, criar medições para os seguintes parâmetros:

- Tempo quando o processo foi criado
- Tempo `SLEEPING`
- Tempo `READY(RUNNABLE)`
- Tempo executando (`RUNNING`)

Declarando as novas variáveis que serão responsáveis por armazenar o tempo de cada processo:

[ARQUIVO] `proc.h`

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

Agora, dentro da função responsável por alocar todos os processos (`allocproc(void)`) no arquivo `proc.c`, inicializamos todos os tempos em 0 e realizamos a medição de `ctime` (que mede em quanto tempo o processo foi criado):

[ARQUIVO] `proc.c`

```
// [MOD] // Inicialização dos controladores de tempo.
p->retime = 0;
p->rutime = 0;
p->stime = 0;

// [MOD] // Prioridade padrão = 2 para todo processo alocado.
p->priority = 2;
acquire(&tickslock);

// [MOD] // Assinala o valor de ticks na variável de momento de criação do processo.
p->ctime = ticks;
release(&tickslock);

release(&ptable.lock);
```

E para realizar as medições de fato são necessárias três funções auxiliares: `update_times(void)`, `wait2(int* retime, int* rutime, int* stime)` e `user_yield(void)`.

A função `wait2` extrai as informações de cada processo e as apresenta para o usuário. Segue sua implementação:

[ARQUIVO] `proc.c`

```
// [MOD] // Função para extrair informações de cada processo e apresentá-la para o usuário
int
wait2(int *retime, int *rutime, int *stime)
{
    struct proc *p;
    struct proc *current_process = myproc();

    int pid;
    int havekids = 0;

    acquire(&ptable.lock);

    for(;;){
        havekids = 0;
        for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

            if(p->parent != current_process){
                continue;
            }

            havekids = 1;
            if(p->state == ZOMBIE){

                *stime = p->stime;
                *retime = p->retime;
                *rutime = p->rutime;

                pid = p->pid;
                kfree(p->kstack);
                p->kstack = 0;
                freevm(p->pgdir);
                p->state = UNUSED;
                p->pid = 0;
                p->parent = 0;
                p->name[0] = 0;
                p->killed = 0;
                p->ctime = 0;
                p->stime = 0;
                p->retime = 0;
                p->rutime = 0;

                release(&ptable.lock);
                return pid;
            }
        }

        if(!havekids || current_process->killed){
            release(&ptable.lock);
            return -1;
        }

        sleep(current_process, &ptable.lock);
    }
}
```

Já a função `update_times` realiza, a cada período de tempo, o devido incremento nas variáveis de controle. Caso o estado do processo seja `RUNNABLE`, ele incrementa `retime`. Caso seja `RUNNING`, ele incrementa `rutime` e se caso for `SLEEPING`, ele incrementa `stime`. Segue a implementação:

[ARQUIVO] proc.c

```
// [MOD] // Função para atualizar variáveis de tempo
void
update_times(void)
{
    struct proc *p;

    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

        if(p->state == RUNNABLE){
            p->retime++;
        }
        else if(p->state == RUNNING){
            p->rutime++;
        }
        else if(p->state == SLEEPING){
            p->stime++;
        }
    }
    release(&ptable.lock);
}
```

Chamamos `update_times()` dentro da já alterada função `trap()`, para que cada vez que ocorra uma interrupção no sistema as variáveis de controle sejam incrementadas.

Já a função `user_yield()`, apenas possibilita ao usuário realizar a chamada de sistema `yield()`.

[ARQUIVO] proc.c

```
// [MOD] // Função para possibilitar a chamada de sistema yield() pelo usuário
int
user_yield(void)
{
    yield();
    return 0;
}
```

Importando as novas funções criadas:

1)

[ARQUIVO] defs.h

```
// [MOD] // Adicionando importações
int          set_prio(int);
void         update_times(void);
int          wait2(int* retime, int* rutime, int* stime);
```

2)

[ARQUIVO] user.h

```
// [MOD] // Adicionando importações
int set_prio(int);
int wait2(int*, int*, int*);
int user_yield(void);
```

Criamos também as chamadas de sistema: sys_wait2 (que aciona de fato a função criada anteriormente) e sys_yield (que permite a chamada de sistema yield).

[ARQUIVO] sysproc.c

```
int
sys_wait2(void)
{
    int *rettime;
    int *ruptime;
    int *stime;

    if ((argptr(0, (char**)&rettime, sizeof(int)) < 0) ||
        (argptr(1, (char**)&ruptime, sizeof(int)) < 0) ||
        (argptr(2, (char**)&stime, sizeof(int)) < 0)) {
        return -1;
    }

    return wait2(rettime, runtime, stime);
}
```

E por último importamos essas novas chamadas de sistema nos respectivos arquivos:

[ARQUIVO] usys.c

```
SYSCALL(wait2)
SYSCALL(user_yield)
```

[ARQUIVO] syscall.h

```
#define SYS_wait2    23
#define SYS_user_yield 24
```

[ARQUIVO] syscall.c

```
// [MOD] // Adicionando systemcalls
extern int sys_set_prio(void);
extern int sys_wait2(void);
extern int sys_yield(void);
```

```
// [MOD] // Adicionando systemcalls
static int (*syscalls[])(void) = {
    [SYS_wait2]    sys_wait2,
    [SYS_user_yield] user_yield,
};
```

Análise

Para analisar o comportamento do novo escalonador implementado criamos o programa sanity.c. Ele recebe como argumento um parâmetro inteiro n, e cria 3*n processos com fork (). Depois disso, espera até que cada um deles termine e imprime 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) são processos do tipo CPU-Bound : executam 100 vezes um loop vazio de 1000000 iterações.
- Processos com (pid mod 3 == 1) são processos de tarefas curtas S-CPU : executam 100 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) são processos IO-Bound : para simular chamadas de IO executa 100 vezes a chamada de sistema sleep(1).

Rodamos esse programa com n = 2, n = 6, n = 10, n = 14 e n = 20 para apurar dados e analisar o comportamento. Para cada n, realizamos 3 execuções e esses foram os resultados médios apurados:

| n = 2 | | | | | |
|-----------------------|-----------|-----------------------|---------|-----------------------|----------|
| Execução 1 | CPU BOUND | Execução 1 | S BOUND | Execução 1 | IO BOUND |
| Tempo READY(RUNNABLE) | 7 | Tempo READY(RUNNABLE) | 6 | Tempo READY(RUNNABLE) | 15 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 19 | Tempo TURNAROUND | 17 | Tempo TURNAROUND | 115 |
| Execução 2 | CPU BOUND | Execução 2 | S BOUND | Execução 2 | IO BOUND |
| Tempo READY(RUNNABLE) | 4 | Tempo READY(RUNNABLE) | 4 | Tempo READY(RUNNABLE) | 4 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 11 | Tempo TURNAROUND | 11 | Tempo TURNAROUND | 104 |
| Execução 3 | CPU BOUND | Execução 3 | S BOUND | Execução 3 | IO BOUND |
| Tempo READY(RUNNABLE) | 2 | Tempo READY(RUNNABLE) | 8 | Tempo READY(RUNNABLE) | 8 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 9 | Tempo TURNAROUND | 15 | Tempo TURNAROUND | 108 |
| n = 6 | | | | | |
| Execução 1 | CPU BOUND | Execução 1 | S BOUND | Execução 1 | IO BOUND |
| Tempo READY(RUNNABLE) | 24 | Tempo READY(RUNNABLE) | 22 | Tempo READY(RUNNABLE) | 29 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 32 | Tempo TURNAROUND | 31 | Tempo TURNAROUND | 129 |
| Execução 2 | CPU BOUND | Execução 2 | S BOUND | Execução 2 | IO BOUND |
| Tempo READY(RUNNABLE) | 20 | Tempo READY(RUNNABLE) | 32 | Tempo READY(RUNNABLE) | 31 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 29 | Tempo TURNAROUND | 40 | Tempo TURNAROUND | 131 |
| Execução 3 | CPU BOUND | Execução 3 | S BOUND | Execução 3 | IO BOUND |
| Tempo READY(RUNNABLE) | 39 | Tempo READY(RUNNABLE) | 42 | Tempo READY(RUNNABLE) | 58 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 55 | Tempo TURNAROUND | 57 | Tempo TURNAROUND | 158 |

| n = 10 | | | | | |
|-----------------------|-----------|-----------------------|---------|-----------------------|----------|
| Execução 1 | CPU BOUND | Execução 1 | S BOUND | Execução 1 | IO BOUND |
| Tempo READY(RUNNABLE) | 70 | Tempo READY(RUNNABLE) | 66 | Tempo READY(RUNNABLE) | 80 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 83 | Tempo TURNAROUND | 81 | Tempo TURNAROUND | 180 |
| Execução 2 | CPU BOUND | Execução 2 | S BOUND | Execução 2 | IO BOUND |
| Tempo READY(RUNNABLE) | 76 | Tempo READY(RUNNABLE) | 81 | Tempo READY(RUNNABLE) | 81 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 91 | Tempo TURNAROUND | 95 | Tempo TURNAROUND | 181 |
| Execução 3 | CPU BOUND | Execução 3 | S BOUND | Execução 3 | IO BOUND |
| Tempo READY(RUNNABLE) | 66 | Tempo READY(RUNNABLE) | 74 | Tempo READY(RUNNABLE) | 94 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 82 | Tempo TURNAROUND | 87 | Tempo TURNAROUND | 194 |
| n = 14 | | | | | |
| Execução 1 | CPU BOUND | Execução 1 | S BOUND | Execução 1 | IO BOUND |
| Tempo READY(RUNNABLE) | 62 | Tempo READY(RUNNABLE) | 58 | Tempo READY(RUNNABLE) | 66 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 71 | Tempo TURNAROUND | 67 | Tempo TURNAROUND | 166 |
| Execução 2 | CPU BOUND | Execução 2 | S BOUND | Execução 2 | IO BOUND |
| Tempo READY(RUNNABLE) | 103 | Tempo READY(RUNNABLE) | 116 | Tempo READY(RUNNABLE) | 115 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 118 | Tempo TURNAROUND | 132 | Tempo TURNAROUND | 215 |
| Execução 3 | CPU BOUND | Execução 3 | S BOUND | Execução 3 | IO BOUND |
| Tempo READY(RUNNABLE) | 56 | Tempo READY(RUNNABLE) | 61 | Tempo READY(RUNNABLE) | 70 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 65 | Tempo TURNAROUND | 69 | Tempo TURNAROUND | 170 |
| n = 20 | | | | | |
| Execução 1 | CPU BOUND | Execução 1 | S BOUND | Execução 1 | IO BOUND |
| Tempo READY(RUNNABLE) | 88 | Tempo READY(RUNNABLE) | 85 | Tempo READY(RUNNABLE) | 94 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 96 | Tempo TURNAROUND | 93 | Tempo TURNAROUND | 195 |
| Execução 2 | CPU BOUND | Execução 2 | S BOUND | Execução 2 | IO BOUND |
| Tempo READY(RUNNABLE) | 86 | Tempo READY(RUNNABLE) | 91 | Tempo READY(RUNNABLE) | 93 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 93 | Tempo TURNAROUND | 100 | Tempo TURNAROUND | 193 |
| Execução 3 | CPU BOUND | Execução 3 | S BOUND | Execução 3 | IO BOUND |
| Tempo READY(RUNNABLE) | 87 | Tempo READY(RUNNABLE) | 92 | Tempo READY(RUNNABLE) | 103 |
| Tempo SLEEPING | 0 | Tempo SLEEPING | 0 | Tempo SLEEPING | 100 |
| Tempo TURNAROUND | 96 | Tempo TURNAROUND | 100 | Tempo TURNAROUND | 204 |

Visando garantir uma boa análise fizemos uma média das 3 execuções para aí sim partir para a plotagem dos gráficos e análise do programa:

Segue os dados apurados:

| n = 2 | | | |
|-----------------------|-----------|---------|----------|
| Média | CPU BOUND | S BOUND | IO BOUND |
| Tempo READY(RUNNABLE) | 4,33 | 6 | 9 |
| Tempo SLEEPING | 0 | 0 | 100 |
| Tempo TURNAROUND | 13 | 14,33 | 109 |

| n = 6 | | | |
|-----------------------|-----------|---------|----------|
| Média | CPU BOUND | S BOUND | IO BOUND |
| Tempo READY(RUNNABLE) | 27,67 | 32 | 39,33 |
| Tempo SLEEPING | 0 | 0 | 100 |
| Tempo TURNAROUND | 38,67 | 42,67 | 139,33 |

| n = 10 | | | |
|-----------------------|-----------|---------|----------|
| Média | CPU BOUND | S BOUND | IO BOUND |
| Tempo READY(RUNNABLE) | 70,67 | 73,67 | 85 |
| Tempo SLEEPING | 0 | 0 | 100 |
| Tempo TURNAROUND | 85,33 | 87,67 | 185 |

| n = 14 | | | |
|-----------------------|-----------|---------|----------|
| Média | CPU BOUND | S BOUND | IO BOUND |
| Tempo READY(RUNNABLE) | 73,67 | 78,33 | 83,67 |
| Tempo SLEEPING | 0 | 0 | 100 |
| Tempo TURNAROUND | 84,67 | 89,33 | 183,67 |

| n = 20 | | | |
|-----------------------|-----------|---------|----------|
| Média | CPU BOUND | S BOUND | IO BOUND |
| Tempo READY(RUNNABLE) | 87 | 89,33 | 96,67 |
| Tempo SLEEPING | 0 | 0 | 100 |
| Tempo TURNAROUND | 95 | 97,67 | 197,33 |

Gráfico Tempo READY(RUNNABLE) x n :

Tempo READY(RUNNABLE)

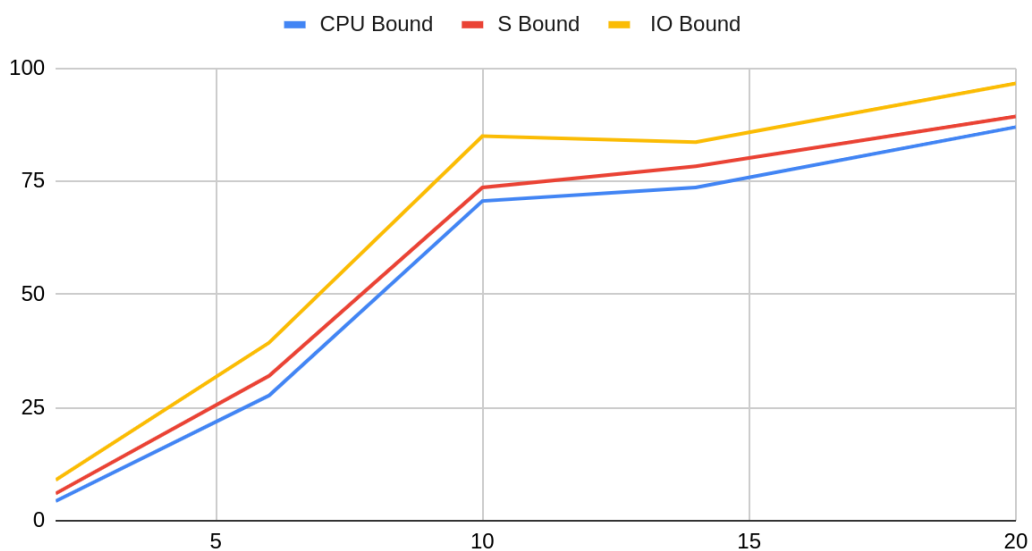


Gráfico Tempo Sleeping x n :

Sleeping

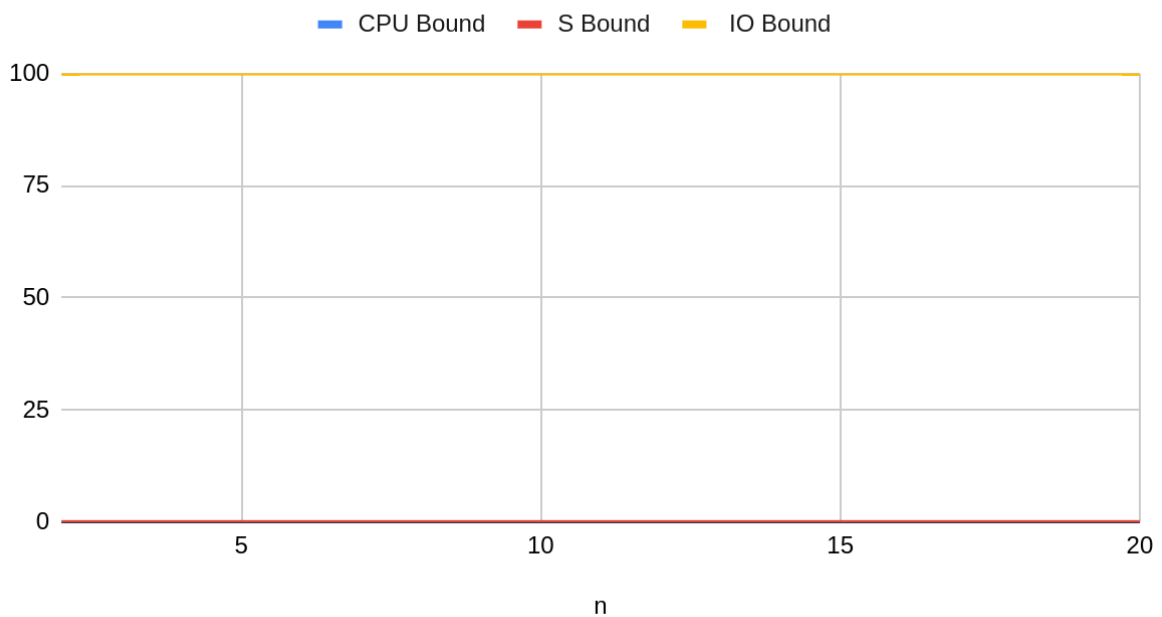
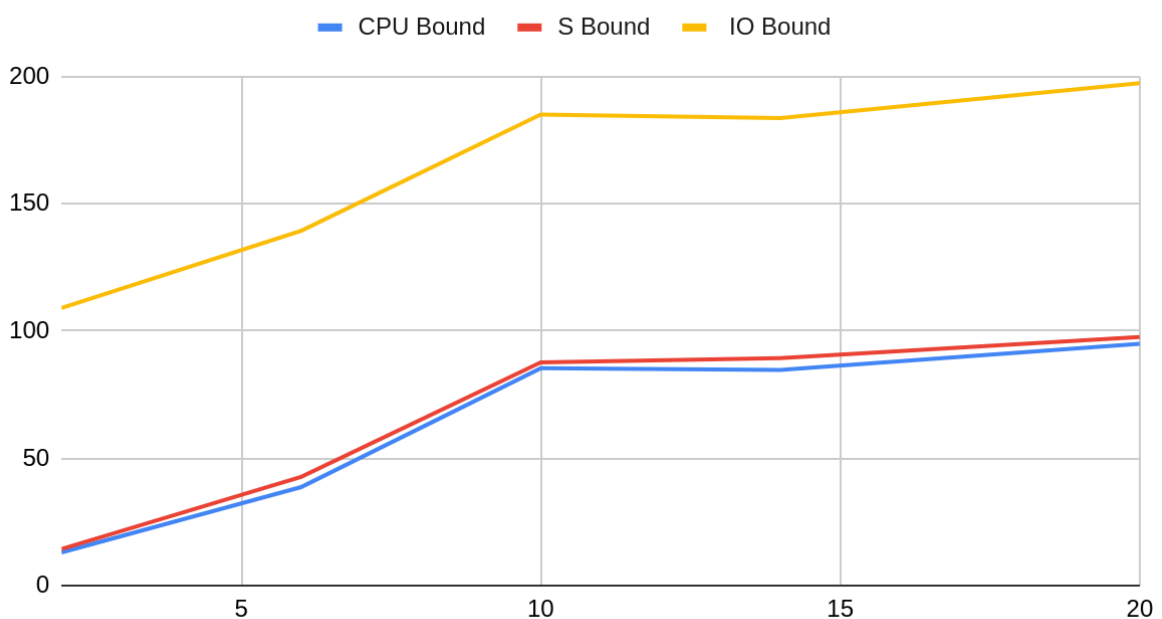


Gráfico Tempo Turnaround x n :

Turnaround



Analisando os gráficos construídos, o que podemos observar foi que o IO Bound possui um Turnaround médio elevado em relação aos outros testes. O que pode causar esse aumento é o fato de que é o único teste que apresentou um tempo SLEEPING diferente de 0, mesmo com a variação de n. Além disso, foi também o teste que apresentou o maior tempo de READY(RUNNABLE). Os demais testes apresentaram tempos similares e com crescimento quase que linear com o aumento de n.

Referências bibliográficas

Fundamentos de Sistemas Operacionais, Oitava edição, Silberschatz, Galvin e Gagne, Ed. LTC.(Livro principal).

<https://www.techtarget.com/whatis/definition/round-robin>

<http://pdos.csail.mit.edu/6.828/2014/xv6.html>

<https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf>

<https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf>