Xv6, um Sistema Operacional Unix

Alesom Zorzi [1]

[1]Ciência da Computação – Universidade Federal da Fronteira Sul (UFFS)
Caixa Postal 181 – Chapecó – SC – Brasil

alesom.zorzi@gmail.com

Abstract. *The goal of this work is give a short explanation about the Operational System* ***xv6****, that is isnpired in the unix V6. Also this work will go to the deep in the scheduling of xv6, with objective of better implementation of a new scheduling of process. This scheduling are of the type* ***stride scheduling****.*

Resumo. *Este trabalho visa explicar de forma sucinta os principais elementos do Sistema Operacional* ***xv6****, que é baseado na versão 6 do Unix. Bem como dissecar o gerenciamento de processos do xv6, tendo em vista uma implementação eficiente de um escalonador de processos do tipo* ***stride scheduling*** *(escalonamento em passos largos).*

1. Introdução

O Sistema Operacional [1] xv6, criado em 2006 pelo curso de Sistemas Operacionais do *MIT* (Massachusetts Institute of Technology), é baseado na versão 6 do Unix, contendo os principais elementos de um sistema operacional, além de ser um sistema de código aberto, didático e multiprocessado, é um sistema operacional perfeito para estudantes.

O xv6, foi projetado tendo apenas um vetor "circular" para realizar o escalonamento dos processos, isso não possibilita, entre outras coisas, atribuir prioridades a certos processos, não importando se um processo é do próprio sistema operacional, ou de um usuário comum. Outro limitante relevante é o fato desse vetor, ser estático com 64 posições, ou seja, são possíveis no máximo, 64 processos rodando paralelamente no xv6.

Assim, a principal proposta do trabalho é projetar um escalonador do tipo **stride scheduling**, de forma a otimizar e dar maior liberdade para o controle das prioridades dos processos, além da possíbilidade de aumentar o número de processos que podem rodar paralelamente no núcleo do processador.

2. Xv6, uma visão geral

Vamos trazer à baila os principais elementos que compõem o sistema xv6, pois, é de suma importância que possamos entender melhor o funcionamento do xv6, ao passo de que não será possível estudar profundamente o xv6. Assim, poderemos desenvolver um projeto de implementação completo, respeitando as limitações do sistema, ao mesmo tempo explorando as melhores qualidades do mesmo.

[1] *Um sistema operacional é um conjunto de processos que tem como objetivo gerenciar os recursos do hardware, dispondo de uma interface para que diversos periféricos possam fazer uso destes recursos sem se preocupar em como as coisas realmente estão acontecendo em um nível mais abstrato.*[Tan09, p. 2]

2.1. Interface

Um dos principais objetivos de um sistema operacional é facilitar a comunicação dos aplicativos com o hardware, essa comunicação é feita através de uma *interface*.

Mas, obviamente não é possível que qualquer processo tenha comunicação com qualquer parte do hardware, um processo criado por um usuário comum não pode interferir em outro processo de um usuário, muito menos em um processo criado pelo sistema operacional. Para realizar esse controle, o sistema operacional impede que um processo comum possa realizar todo e qualquer tipo de instruções sozinho. Então para realizar essas instruções os processos precisam realizar uma *chamada de sistema*.

No xv6 os programas podem ser do tipo *user* (usuário) ou *kernel*(núcleo), quando um processo do tipo user precisa realizar uma ação que deve ser feita por um processo do tipo kernel, é feita uma chamada de sistema e esta chamada solicita que a ação que o programa do tipo user necessita realizar, seja realizada. E para que todos os programas possam conhecer essas chamadas tais chamadas são realizadas através de uma interface. Essa interface nada mais é do que um conjunto de chamadas, são elas:

Chamada	**Descrição**
fork()	Cria um processo
exit()	Termina o processo atual
wait()	Espera por um processo filho para sair
kill(pid)	Finaliza o processo atual
getpid()	Retorna o id do processo atual
sleep(n)	Faz o processo não ser executado por n segundos
exec(filename, *argv)	Carrega um arquivo e executa o mesmo
sbrk(n)	Aumenta a memória do processo em n bytes
open(filename, flag)	Abre um arquivo; flag indica se é para escrita ou leitura
read(fd, buf, n)	Lê n bytes do arquivo já aberto fd e coloca em buf
write(fd, buf, n)	Escreve n bytes no arquivo aberto fd
close(fd)	Libera o arquivo aberto fd
dup(fd)	duplica fd
pipe(p)	Cria um pipe e retorna o fd em p
chdir(dirname)	Muda o diretorio atual
mkdir(dirname)	Cria um novo diretório
mknod(name, major, minor)	Cria um arquivo de dispositivo
fstat(fd)	Retorna informações sobre um arquivo aberto
link(f1, f2)	Cria outro nome (f2) para o arquivo (f1)
unlink(filename)	Remove um arquivo

Assim todo e qualquer processo pode ter acesso tais serviços sem que o sistema operacional perca o controle do hardware, além de facilitar muito a vida dos processos que precisam realizar tais tarefas.

2.2. Memória e Processos

Um processo é fundamentalmente um contêiner que armazena todas as informacões necessárias para executar um programa[Tan09, p. 23].

No xv6 um processo consiste, basicamente, de um *user-space* (espaço de usuário), memória (a memória é dividia em instrução, dados e pilha) e *per-process* (estado privado do kernel).

O primeiro processo criado é o *userinit*, após esse processo ser criado, todos os outros processos são criados a partir de outros processos, quem gerou o novo processo, é denominado *pai*, por óbvio o processo criado é chamado *filho*. Todos os processos tem um identificador único, chamado de *pid*. O sistema operacional usa endereços virtuais, isso faz com que os processos tenham a impressão que tem todo o hardware disponível para seu uso. O sistema operacional também interrompe a execução dos programas, salvando o contexto dos mesmos e retomando a execução exatamente de onde havia parado, assim todos os processos tem a impressão que tem todo o processamento ao seu dispor.

O gerenciamento dos endereços virtuais de cada processo, é feito através da *tabela de páginas* (implementada em hardware). Cada processo tem sua própria tabela de páginas e o endereçamento é feito na mesma. Isso ajuda ainda a manter o isolamento dos processos, deixando fácil verificar caso algum processo queira interferir em um espaço de endereçamento indevido. Cada espaço de endereçamento inicia no endereço virtual zero e consiste em Instruções (código), variáveis globais, pilha e heap (usado para alocação dinâmica), nessa ordem.

2.3. Tabela de Páginas

A tabela de páginas é um mecanismo muito poderoso, a maioria dos sistemas operacionais faz uso com o intuito de melhorar o gerenciamento da memória.

O xv6 faz uso da tabela de página que os processadores x86 implementam em hardware. A tabela de páginas tem 2^{20} entradas, essas entradas são chamadas de PTEs (page table entries). Cada um das PTEs contém 20 bits referentes ao número da página física, chamado de PPN (physical page number) e mais algumas *flags* (bandeiras) para realizar os devidos controles. Para reduzir o tamanho da tabela de páginas criou-se dois níveis: uma tabela (chamada de diretório) direciona para outras tabelas que contém as informações sobre a página que foi requerida. O endereço virtual possui 22 bits, dos quais 10 bits selecionam o diretório, 10 bits a tabela de páginas e 12 bits o *offset* (deslocamento) na página requerida.

2.4. Traps e Interrupções

Outro tema que exige uma cuidadosa atenção, são as interrupções. As interrupções podem ser de dois tipos: De Software ou de Hardware. As interrupções de software são chamadas de traps (no português, armadilhas), já as interrupções de hardware são chamadas apenas de interrupções.

As interrupções servem para avisar algum processo, seja ele o sistema operacional ou não, de que alguma ação ocorreu. Um exemplo de interrupção é quando o tempo que o processo fica no núcleo do processador expira, quando isso acontece, o próprio hardware gera uma interrupção para que aquele processo que estava executando seja retirado do núcleo, (salvando o contexto do processo, obviamente), deixando assim, outro processo, que será escalonado, assumir o núcleo. Não poderíamos deixar de citar alguns conceitos essênciais, que nada mais são do que interrupções ou traps, são eles: *chamada de sistema*, *exceção* e *interrupção*.

- Uma *chamada de sistema*, basicamente, permite a um processo de usuário socilitar que um comando seja executado em modo kernel.
- Uma *exceção*, acontece quando um processo executa uma ação não permitida (e.g.: divisão por zero, tentar acessar uma página que não pertence ao processo, etc).
- Uma *interrupção* é um sinal gerado pelo hardware solicitando atenção do sistema operacional

2.5. Bloqueio

Quando dois, ou mais, processos precisam acessar um recurso que é compartilhado entre os mesmos, ou seja, eles podem realizar atualiações no recurso, geralmente por meio da escrita, alguém, necessariamente precisa controlar quem faz, e, em que tempo faz, os acessos. Quando um processo quer acessar um desses recursos que são compartilhados, ele deve solicitar para o sistema operacional, se conseguir acessar o recurso, esse recurso fica *bloqueado* até que o processo libere tal recurso, só assim, outro processo pode acessar o recurso compartilhado.

Se dois, ou mais, processos tentam acessar um recurso essa condição é chamada de *race condition* (condição de disputa). Os mecanismos que o xv6 implementa, não serão discutidos nesse artigo, tais mecanismos podem ser encontrados em [RC12].

2.6. Sistema de Arquivos

O sistema de arquivo tem uma função bem definida, ele serve para armazenar dados não voláteis de forma mais organizada. O xv6 usa o sistema proveniente do Unix-Like, e para fazer isso várias coisas são necessárias:

- O sistema precisa ter estruturas armazenadas com as árvores dos nomes dos diretórios, para identificar cada parte ocupada e cada parte livre do disco.
- O sistema de arquivo precisa ter suporte a desligamentos inesperados.
- O sistema de arquivo precisa dar suporte para que vários programas usem o mesmo simultaneamente.
- Acessar disco tem um custo de tempo bastante elevado, por isso blocos de dados mais acessados devem ser guardados na memória.

O sistema de arquivo do xv6 é organizado em 6 níveis, do mais baixo para o mais alto, são eles:

- *Buffer Cache*: Responsável por fazer a leitura e escrita nos blocos do disco. Controlando para que somente um processo de kernel, por vez, consiga armazenar dados naquele bloco.
- *Logging*: Responsável pelas transações. Precisa garantir a atomicidade de seus atos.
- *Inode*: É uma estrutura que guarda uma sequência de blocos do disco. Essa estrutura contém informações de quantos blocos e o qual endereço dos mesmos.
- *Directory Inodes*: É a implementação de um diretório de inodes. Cada diretório contém um nome e faz referência a um "arquivo de inode".
- *Recursive Lookup*: Essa camada cria a parte hierarquica dos diretórios, fazendo possivel a navegação por eles.
- *File Descriptors*: E essa camada, faz com que programadores e usuários possam usar arquivos e diretórios de forma mais simples, abstraindo todo o sistema envolvido (dispositivos, arquivos, blocos).

3. Escalonamento

Como já é sabido, em um sistema operacional deve ter diversos processos rodando simultaneamente para conseguir um desempenho aceitável, ainda mais que, com os processadores multinúcleos (dois, ou mais *cores*), isso possíbilita que dois processos rodem ao mesmo tempo. Mas mesmo com apenas um núcleo, temos vários processos disputando uma fatia de tempo do processador, pois assim, o sistema fica mais interativo, i.e. o tempo médio de espera de um processo é reduzido. O xv6 foi desenvolvido para ser multiprocessado, assim necessita ainda mais de um escalonador efiente para selecionar um processo mais rápido possível, para que seus núcleos não fiquem ociosos.

Uma abordagem bastante comum é, dar aos processos a ilusão de que eles não tem concorrência e tem o controle do núcleo do processador em todos os momentos. Assim são cedidos para os processos "processadores virtuais", sendo que na verdade estão todos disputando o mesmo processador físico.

Um processo, num dado momento pode estar em um, dentre três estados possíveis: *pronto*, *bloqueado*, ou *executando*.

- **Pronto**: O processo esta pronto para executar e só espera o sistema operacinar seleciona-lo para executar.
- **Bloqueado**: O processo, por algum motivo se encontra bloqueado e não pode ser selecionado para ganhar o núcleo do processador.
- **Executando**: processo está executando no núcleo do processador.

Tento estes três estados em mãos, o sistema operacional deve fazer uma escolha, dentre os processos que estão em estado de pronto, para conseder ao escolhido o núcleo do processador. Perceba que a escolha deve ser feita o mais rápido possível, pois, isso influencia diretamente no desempenho do sistema operacional, já que, durante esse período, o processador está ocioso.

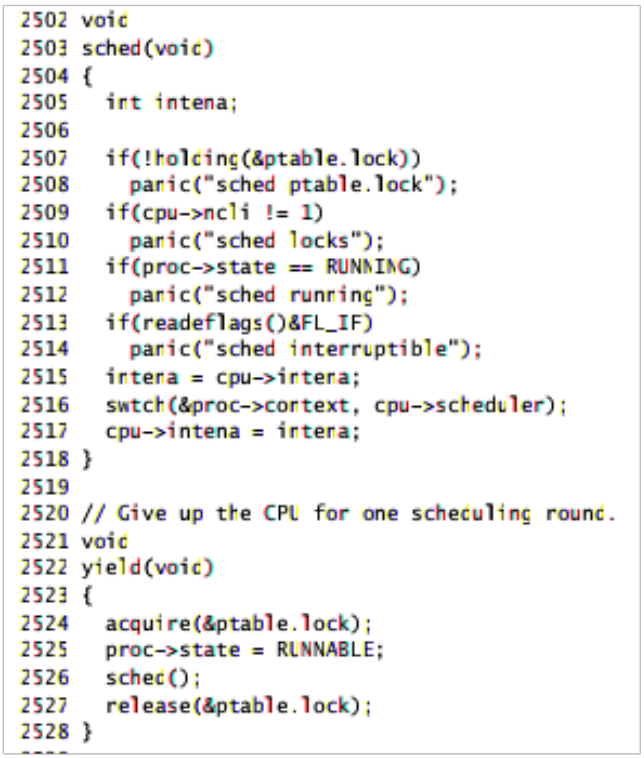
No restante deste capítulos as explicações serão mais profundas, com códigos de apoio para um melhor entendimento do projeto, já que o objetivo do mesmo envolve especialemnte esse capítulo.

3.1. Troca de Contexto

Existem dois contextos possíveis em um processador, ou ele esta executando instruções do kernel, ou está executando instruções de um programa/processo qualquer. Quando a instrução executada é de usuário, geralmente, é necessário, antes da troca de contexto, salvar o contexto do processo que estava executando, e somente após este passo o contexto é trocado. O salvamento do contexto não acontece apenas caso o processo termine sua execução por conta própria. Já se a instrução que esta sendo executada é de kernel, antes de haver a troca de contexto, é necessário recuperar o contexto do processo que será executado, obviamente, caso não seja o início do um novo processo. O código da figura 1 mostra como é feito a troca de contexto através da função *swtch*, que salva o contexto atual no registrador antigo e carrega o novo contexto no novo registrador.

3.2. Escalonador

O escalonador tem a função de escolher, rapidamente, um processo para ceder-lhe o núcleo do processador, como já haviamos falado anteriormente, o escalonador do xv6 é um



```
2502 void
2503 sched(void)
2504 {
2505   int intena;
2506
2507   if(!holding(&ptable.lock))
2508     panic("sched ptable.lock");
2509   if(cpu->ncli != 1)
2510     panic("sched locks");
2511   if(proc->state == RUNNING)
2512     panic("sched running");
2513   if(readeflags()&FL_IF)
2514     panic("sched interruptible");
2515   intena = cpu->intena;
2516   swtch(&proc->context, cpu->scheduler);
2517   cpu->intena = intena;
2518 }
2519
2520 // Give up the CPU for one scheduling round.
2521 void
2522 yield(void)
2523 {
2524   acquire(&ptable.lock);
2525   proc->state = RUNNABLE;
2526   sched();
2527   release(&ptable.lock);
2528 }
```

Figura 1. Quando se faz necessário a troca de contexto é chamada a função *yield*

tanto quanto modesto, utiliza-se de um laço circular para escolher o primeiro processo que encontrar com o estado *RUNNABLE*.

Um processo que quer assumir o núcleo do processador, precisa bloquear a tabela de processos *ptable.lock*, realizar outros bloqueios, atualizar seu estado e assim chamar do *sched*. A função sched faz algumas verificações, e, desde que o bloqueio possa ser feito, as instruões podem ser executadas com interrupções desabilitadas. Depois disso, finalmente, o sched chama o swtch para salvar o contexto atual no *proc->context* e troca o contexto no *cpu->scheduler*. Após isso, retorna para a pilha do escalonador e o escalonador continua seu laço a procura de processos para executar.

Quando o primeiro processo é escalonado ele começa com um forkret (figura 2). O forkret existe apenas pela realização do bloqueio ptable.lock, entretanto o novo processo pode começar com uma trap.

ptable.lock possíbilita, entre outras coisas, a alocação de um identificador de processos e a liberação de espaços na tabela de processos.

```
2530 // A fork child's very first scheduling by scheduler()
2531 // will swtch here.  "Return" to user space.
2532 void
2533 forkret(void)
2534 {
2535   static int first = 1;
2536   // Still holding ptable.lock from scheduler.
2537   release(&ptable.lock);
2538
2539   if (first) {
2540     // Some initialization functions must be run in the context
2541     // of a regular process (e.g., they call sleep), and thus cannot
2542     // be run from main().
2543     first = 0;
2544     initlog();
2545   }
2546
2547   // Return to "caller", actually trapret (see allocproc).
2548 }
```

Figura 2. Quando se faz necessário a troca de contexto é chamada a função *yield*

4. Proposta de Implementação

Esse trabalho visa a implementação de um escalonador *stride scheduling* (pode ser traduzido como escalonamento a passos largos) para o sistema operacional xv6.

4.1. Stride Scheduling

O escalonamento stride scheduling é um mecanismo de atribuição determistica para recursos compartilhados em tempo [CAW95, p. 2]. No escalonamento stride scheduling o direito aos recursos são distribuidos conforme o número de *bilhetes* que um processo possui. A distribuição dos bilhetes é feita da seguinte forma:

1- Defina o número N, que corresponde ao máximo de bilhetes que podem ser distribuidos.
2- Quando um processo é iniciado, distribua um número n_i de bilhetes para cada processo, de forma que $\sum_{i=1}^{n} n_i \leq N$.

Com os bilhetes distribuidos conforme for conveniente ao sistema, podemos definir um *passo*. O passo do processo i será dado por $\frac{N}{n_i}$. O processo que será escalonado é o que tiver menor *distância percorrida*. Inicialmente da distância percorrida é 0, e cada vez que o processo é escalonado, a distância percorrida é incrementada em *passo* unidades. Por conveniência, iremos usar como desempate o identificador do processo (no xv6, pid).

4.2. Implementação

Na implementação foi usada uma *heap binária*, tendo a mesma função de uma fila de prioridades, onde o peso dos nós da árvore será a *distância percorrida* do processo até o momento. As operações de inserção e decremento de chave, são feitas em tempo $\mathcal{O}(\log n)$, já a operaração de escolha do processo que vai ser escalonado é feita em tempo $\mathcal{O}(1)$, já

que o objetivo do heap é que o menor, ou maior, elemento seja a raiz do heap. Maiores informações sobre *heap binário* podem ser encontradas em [CLRS91, p. 151-169].

O *heap binário* será construido (em tempo $\mathcal{O}(n \log n)$) sobre o vetor *ptable.proc*, que guarda todos os processos, os processos que estão bloqueados, terão um peso "infinito" (número arbitrário, por exemplo, 10^{10}), ou seja, não serão escalonados. Quando um processo volta do bloqueio, recebe o *"peso"* que tinha antes de ser bloqueado. Caso nenhum processo esteja pronto para ser executado, o escalonador espera até que algum processo possa ser executado, ou seja, enquanto a raiz do *heap* é igual a infinito, espere.

Seguem algumas imagens do código quando o escalonador usava uma fila de prioridades (heap) para selecionar o processo que ganhará o núcleo do processador:

```
30  /*Begin Heap CODE*/
31  void troque(int i, int j) {
32    int tmp;
33    tmp = H[i];
34    H[i] = H[j]; H[j] = tmp; emH[H[i]] = i; emH[H[j]] = j;
35  }
36
37  void heapify(int i) {
38    int menor = i;
39    if (ESQ(i) <= tamH && dist[H[ESQ(i)]] < dist[H[i]])
40      menor = ESQ(i);
41
42    if (DIR(i) <= tamH && dist[H[DIR(i)]] < dist[H[menor]])
43      menor = DIR(i);
44
45    if (menor == i) return;
46    troque(i, menor);
47    heapify(menor);
48  }
49
50  int extraia_min(void) {
51    troque(1, tamH);
52    emH[H[tamH--]] = 0;
53    heapify(1);
54    return H[tamH + 1];
55  }
56
57  void rebaixe(int i) {
58    for (; i > 1 && dist[H[i]] < dist[H[PAI(i)]]; i = PAI(i))
59      troque(i, PAI(i));
60  }
```

Figura 3. Código usado na implementação do heap

```
349 void scheduler(void) {
350   struct proc *p;
351
352   for(;;){
353     // Enable interrupts on this processor.
354     sti();
355     int u = extraia_min();
356
357     acquire(&ptable.lock);
358
359     while (dist[u] >=INFINITO)
360       u =extraia_min();
361
362     p = getprocessById(u);
363     if(p==0) continue;
364
365     proc = p;
366     switchuvm(p);
367
368     p->state = RUNNING;
369     dist[p->pid] = INFINITO;
370     swtch(&cpu->scheduler, proc->context);
371
372     switchkvm();
373
374     p->curr_pass += p->pass;
375     dist[p->pid] = p->curr_pass;
376     rebaixe(emH[p->pid]); //atualiza a heap
377     proc = 0;
378     release(&ptable.lock);
379   }
380 }
```

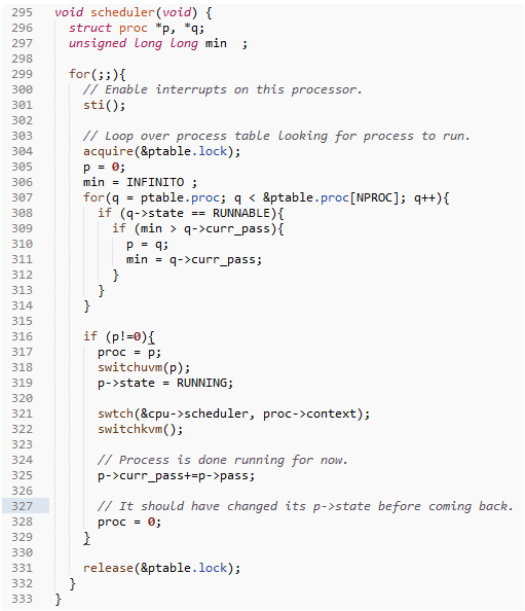
Figura 4. Escalonador quando implementado a escolha do processo por heap

```
331 struct proc * getprocessById(int id){
332   struct proc *p;
333
334   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
335     if (p->pid == id){
336       return p;
337     }
338   }
339   return 0;
340 }
```

Figura 5. A função GetProcessById retorna um o processo pelo pid do processo.

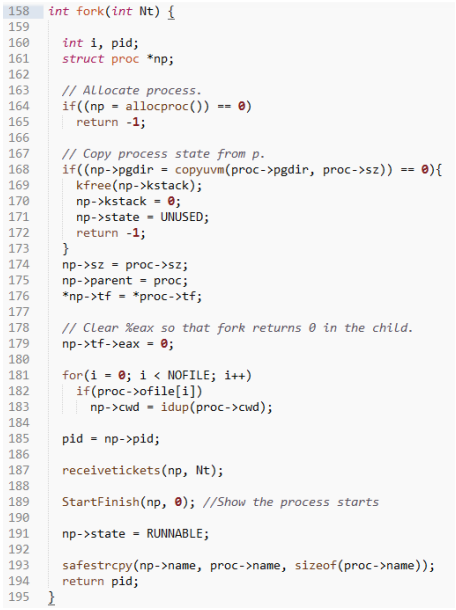
No entanto, infelizmente, por conta de alguns detalhes não previstos no projeto, decidi mudar a implementação do trabalho. O heap foi implementado, devolvendo um inteiro que seria o identificador do processo (*pid*), e eu acreditava que fazendo uma conta rápida, conseguiria saber de qual processo se tratava, ledo engano, de qualquer forma, teria que procurar pelo processo que tem o id que a heap retornou, ou seja, a escolha do processo, ficaria com tempo $\mathcal{O}(n)$, e como o tempo de atualização da heap é $\mathcal{O}(log(n))$, ficaria com uma complexidade superior ao método simples e linear sem o uso de uma heap binária.

Assim, decidi implementar o escalonador do método mais simples, eficaz e menos suscetível a erros. Buscando, no mesmo vetor já usado no escalonador original do xv6, o processo com menor *distância percorrida* (no código curr_pass), e assim dar-lhe o núcleo do processador. Seguem alguns trechos de código do xv6, com o escalonador *stride scheduling*.



```
void scheduler(void) {
  struct proc *p, *q;
  unsigned long long min  ;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    p = 0;
    min = INFINITO ;
    for(q = ptable.proc; q < &ptable.proc[NPROC]; q++){
      if (q->state == RUNNABLE){
        if (min > q->curr_pass){
          p = q;
          min = q->curr_pass;
        }
      }
    }

    if (p!=0){
      proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      p->curr_pass+=p->pass;

      // It should have changed its p->state before coming back.
      proc = 0;
    }

    release(&ptable.lock);
  }
}
```

Figura 6. Escalonador de processos



```
int fork(int Nt) {

  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process state from p.
  if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->cwd = idup(proc->cwd);

  pid = np->pid;

  receivetickets(np, Nt);

  StartFinish(np, 0); //Show the process starts

  np->state = RUNNABLE;

  safestrcpy(np->name, proc->name, sizeof(proc->name));
  return pid;
}
```

Figura 7. Fork, criando um novo processo

4.3. Módulos modificados/acrescentados no xv6

Para a implementação, foram modificados vários módulos, a primeira modificação foi na estrutura *proc*, foram acrescentadas três variáveis, *Ntickts*, *pass* e *curr_pass*, que correspondem, respectivamente, ao número de tickets do processo, passo do processo e a distância percorrida até o momento. A função *scheduler* foi bastante alterada, devido ao novo formato da escolha do processo que ganhará o núcleo do processador, veja na imagem abaixo:

```
void scheduler(void) {
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

```
void scheduler(void) {
  struct proc *p, *q;
  unsigned long long min  ;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    p = 0;
    min = INFINITO ;
    for(q = ptable.proc; q < &ptable.proc[NPROC]; q++){
      if (q->state == RUNNABLE){
        if (min > q->curr_pass){
          p = q;
          min = q->curr_pass;
        }
      }
    }

    if (p!=0){
      proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      p->curr_pass+=p->pass;

      // It should have changed its p->state before coming back.
      proc = 0;
    }

    release(&ptable.lock);
  }
}
```

Figura 8. No lado esquerdo vemos o escalonador original do xv6, já no lado direito temos o novo escalonador do xv6, o *stride scheduling*

Outras funções como *fork*, que antes não recebia nenhum parâmetro, agora passou a receber um inteiro como parâmetro, que corresponde ao número de tickets do processo que será criado pelo *fork*, com isso, todas as funções que chamavam a função *fork*, tiveram que ser alteradas. Outra modificação na função *fork*, foi a inclusão de duas chamadas, uma para fazer a inicialização de algumas variáveis da estrutura *proc*, denominada *receivetickets*, que são usadas para o escalonamento e a outra função é para sinalizar que o processo iniciou, denominada *StartFinish*. Outra modificação foi na função *sys_fork*, que passou a retornar um número aleatório, correspondente ao número de bilhetes que um processo receberá.

4.4. Estratégias para a Validação do Desempenho do Escalonador

Foi criado um novo programa, nomeado *TestProc*, que tem apenas a função de criar N novos processos que sempre ocupam completamente suas fatias de tempo no processador, isto é, não são interrompidos durante a execução. Assim, como cada um destes novos programas recebe uma prioridade aleatória, podemos verificar o desempenho e validar a corretude do escalonador. Para essas validações, foram colocados, em pontos estratégicos, avisos, que sinalizam quando um processo iniciou ou terminou. Com essa finalidade foi usado uma função denominada *StartFinish* e acrescentado a estrutura *proc* uma nova variável, denomidada *slices* que representa a quantidade de vezes que o processo ganhou o núcleo do processador.

5. Implementação e Testes

5.1. Implementação

O primeiro passo para a implementação foi a criação de de três novas variáveis essenciais para o funcionamento do escalonamento *stride scheduling*, são elas, o número de bilhetes do processo, o passo do processo e o passo atual do processo (Ntickts, pass e curr_pass, respectivamente), além de uma constante, que corresponde ao número máximo de tickets.

Após isso, outra mudança foi a possibilidade de passar um inteiro como parâmetro na função *fork*, que representa o número de bilhetes do processo, ainda na função *fork*, foi feita a chamada da função *receivetickets*, que faz a inicialização das 3 variáveis ditas anteriormente. Feito isso, foi alterado, em todos os lugares onde havia a chamada *fork*, e passado um parâmetro.

No escalonador foi usado mesmo vetor de processos, para escolher, dentre os processos no estado *RUNNABLE*, o que tem menor *curr_pass*. Outras alterações foram a atualização do *curr_pass* e do *slices* do processo.

5.2. Testes

Foi criado um programa denominado *TestProc*, que cria *N* novos processos, que tem como código, um único *for*, todos os processos criados, tem o mesmo código, o único intuito deste *for* é ocupar ao máximo a fatia de tempo do processador possível. No código do xv6, foi inserido em dois pontos a chamada da função *StartFinish*, que recebe zero se o processo está iniciando, ou um se o processo esta terminando.

```
/* Function StartFinish warning when a process starts or have finish */

void StartFinish(struct proc *p, int SF){
  if (SF){
    //termina
  //  cprintf("Finish\n");
  //  cprintf("Id: %d Passo: %d Tickts %d\n", p->pid, p->pass, p->Ntickts);
    cprintf("%d, %d\n", p->pid, p->Ntickts);
  }else{
  //inicia
  //  cprintf("Start\n");
  //  cprintf("Id: %d Passo: %d Tickts %d\n", p->pid, p->pass, p->Ntickts);
    cprintf("%d, %d\n", p->pid, p->Ntickts);
  }
}
/* End StartFinish */


/*receivetickets  give n tickets to process and set some thinks */

void receivetickets(struct proc *np, int n){
  np->Ntickts = n;
  np->pass = NumMaxTickts /  np->Ntickts;
  np->curr_pass = 0;
}
//end
```

Figura 9. As funções StartFinish e recievetickts

6. Resultados Obtidos

O novo escalonador, teve resultados dentro do esperado, os processos que recebiam um número maior de bilhetes acabavam sua execução antes dos processos que tinham um número maior de bilhetes, veja as tabelas a baixo:

Ordem de inicio		**Ordem de finalização**	
pid do processo	Número de Tickets	*pid* do processo	Número de Tickets
4	10995	6	27316
5	18073	7	24833
6	27316	9	19468
7	24833	5	18073
8	6231	10	12075
9	19468	13	11079
10	12075	4	10995
11	6590	11	6590
12	1702	8	6231
13	11079	12	1702

Tabela 1. Os processos estão em ordem de início e finalização, com isso é possível notar que os processos que receberam maior número de tickets, finalizaram sua execução antes.

Ordem de inicio		**Ordem de finalização**	
pid do processo	Número de Tickets	*pid* do processo	Número de Tickets
4	10995	20	29076
5	18073	15	28176
6	27316	6	27316
7	24833	7	24833
8	6231	17	24759
9	19468	9	19468
10	12075	5	18073
11	6590	19	18605
12	1702	21	15365
13	11079	23	15253
14	287	10	12075
15	28176	16	11888
16	11888	13	11079
17	24759	4	10995
18	6907	18	6907
19	18605	22	6723
20	29076	11	6590
21	15365	8	6231
22	6723	12	1702
23	15253	14	287

Tabela 2. Os processos estão em ordem de início e finalização, com isso é possível notar que os processos que receberam maior número de tickets, finalizaram sua execução antes.

Foram feitos mais testes com um número maior de processos, os resultados se mantêm os mesmos.

Outro teste que foi realizado e comprova os resultados obtidos, é o quadro a baixo, que mostra, em um período de tempo, a quantidade de vezes que um determinado processo recebeu o núcleo do processador, desde seu início até o dado momento:

pid do processo	Número de Tickets	*slices* do processo
4	10995	71
5	18073	116
6	27316	175
7	24833	159
8	6231	40
9	19468	125
10	12075	78
11	6590	43
12	1702	11
13	11079	71

Tabela 3. Os processos estão em ordem de início. É possível notar que os processos que receberam maior número de tickets, recebem mais vezes o núcleo do processador do que os processos que receberam menos tickets, veja por exemplo os processos com *pid* 6 e 12, o processo com *pid* 6 recebeu 175 vezes o núcleo, enquanto o processo de *pid* 12 recebeu apenas 11 vezes, isso se explica pelo número de bilhetes que cada um recebeu, 27316 e 1702 respectivamente.

7. Conclusão

Com o novo escalonador, os processos poderão ter uma prioridade conforme for conveniente, com isso existe um ganho muito grande em desempenho, pois processos que são mais críticos, podem ser executados com maior prioridade, enquanto os processos que mais são dispensáveis, podem rodar com uma prioridade mais baixa, não afetando tando o bom funcionamento do sistema com um todo.

O desempenho do novo escalonador, em termos de complexidade, permaneceu o mesmo, ambos $\mathcal{O}(n)$, com n representando o número de processos.

8. Referências Bibliográficas

Todo fundamental teórico desse artigo foi construido com duas fortes bases, que são o livro Sistemas Operacionais Modernos [Tan09] de Andrew S. Tanenbaum e o próprio manual do sistema operacional xv6 [RC12] escrito por Russ Cox, Frans Kaashoek e Robert Morris. Foi usado como aprimoramento teórico o artigo *Stride Scheduling: Deterministic Proportional-Share Resource Management* [CAW95] escrito por Carl A. Waldspurger e William E. Weihl, mais especificamente as páginas onde é descrito o funcionamento do escalonamento em questão [2].

O livro Sistemas Operacionais modernos, traz uma descrição detalhada e, ao mesmo tempo didática, sobre sistemas operacionais. Conceitos teóricos sobre sistemas operacionais (e.g.: como o que é um processo, o que é e quais são as funções de um sistema operacional) são detalhadamente explicados no livro, com várias imagens ilustrativas que facilitam e muito a vida de quem ainda não esta habituado com tais termos.

Todas as informações sobre o sistema operacional xv6, bem como o manual do mesmo, história e código fonte, podem ser encontradas em [3]. No manual, são encontrados, com todos os detalhes, as informações essencias para que as modificações que serão implementadas no sistema possam ser executadas com êxito, de forma sucinta e com fragmentos de código, para evitar pequenas dubiedades que possam vir a ocorrer.

[2] *link:* `http://www.vuse.vanderbilt.edu/~dowdy/courses/cs381/waldspurger.weihl.pdf` *Referênciado dia 17/04/2015*

[3] *link:* `http://pdos.csail.mit.edu/6.828/2012/xv6.html` *Referênciado dia 04/04/2015*

Referências

[CAW95] William E. Weihl Carl A. Waldspurger. *Stride Scheduling: Deterministic Proportional-Share Resource Management*. MIT, Reading, Massachusetts, 1th edition, 1995.

[CLRS91] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 3th edition, 1991.

[RC12] Robert Morris Russ Cox, Frans Kaashoek. *xv6 a simple, Unix-like teaching operational system*. MIT, Reading, Massachusetts, 1th edition, 2012.

[Tan09] Andrew S Tanenbaum. *Modern operating systems*. Pearson Education, Reading, Massachusetts, 1th edition, 2009.