

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

Disciplina de Sistemas Operacionais I-N

Professores Alexandre Silva Carissimi e Nicolas Maillard

Relatório do primeiro trabalho prático – escalonador SJF para a fila de processos de usuário para o sistema operacional Minix 2.0.4

Francieli Zanon Boito – 141949

Márcio Rocha Zacarias – 134685

Porto Alegre, 3 de maio de 2007.

1. Introdução

Sistemas Operacionais são fundamentais para o ideal uso de recursos de Hardware de um computador. Um dos principais fatores de um bom uso de Hardware é o bom uso da CPU pelos processos. Portanto, o gerenciador de processos, também chamado de escalonador, do Sistema Operacional deve ser muito bem projetado e desenvolvido para otimizar o uso da CPU, sendo este justo, independente das cargas de trabalho executadas.

Neste primeiro trabalho da cadeira de Sistemas Operacionais I-N, no semestre 2007/1, temos o importante objetivo de alterar e otimizar o escalonador para fila de processos de usuário do sistema operacional Minix 2.0.4. O atual algoritmo de escalonamento é um FIFO pouco custoso porém com fraco desempenho. Nossa missão foi implementar algoritmo SJF (“Shortest Job First”) para realizar o escalonamento da fila de processos de usuário, esperando um desempenho melhor do que o FIFO anterior, tanto no nível de tempo de resposta quanto em justiça para a atribuição de CPU para os processos.

O algoritmo SJF funciona baseado na equação abaixo:

$$T_{n+1} = a * t_n + (1 - a) T_n$$

Sendo que t_n é a duração do $n^{\text{ésimo}}$ (ou mais recente) burst de CPU, T_n é uma média ponderada dos bursts de CPU desde que o processo foi admitido no sistema, a é um fator que considera o peso do burst de CPU mais recente em relação ao seu histórico ($0 \leq a \leq 1$) e T_{n+1} é a duração prevista para o próximo *burst* de CPU.

Este relatório especifica as alterações no código-fonte do Sistema Operacional Minix 2.0.4 necessárias para o desenvolvimento deste novo gerenciador de processos. Além disso, ainda disserta sobre o desenvolvimento de programas de teste que obtivessem resultados sérios e é discutido o impacto do uso deste novo escalonador para diferentes tipos de carga de trabalho (CPU e I/O-Bound), assim como cargas de trabalho mistas. Por fim, os resultados serão apresentados em forma de gráficos com suas devidas explicações.

2. Implementação

Os arquivos modificados para a implementação do algoritmo SJF no escalonamento da fila de processos de usuário foram:

- **/usr/src/kernel/proc.h:** Primeiramente, foi feita a adição da constante `PROC_A`, que dá a ponderação entre o tempo do último acesso à CPU e o histórico entre os tempos anteriores à este. Os valores variam de zero a um, sendo que zero faz com que o cálculo seja proporcional apenas ao histórico, e um faz com que o cálculo seja proporcional apenas ao último uso da CPU.

```
/*constante PROC_A, controla a ponderacao entre o ultimo tempo e o historico
 * de tempos anteriores do processo para o calculo do tempo estimado para a
 * proxima execucao*/
#define PROC_A 0.50
```

Figura 1. Constante `PROC_A`

Além disso, à *struct proc*, nesse mesmo arquivo, foram acrescentados os campos `clock_t user_time_prev`, `double historico`, `double user_time_expected`. O primeiro se explica pelo fato de que o campo `clock_t user_time`, já existente na estrutura, contém o tempo total que o processo já passou na CPU, sendo sempre atualizado quando este é preemptado ou se bloqueia. Assim, antes da escolha do processo para ir para a CPU, `user_time_prev` recebe `user_time`. Dessa forma, quando o processo sair da CPU, a diferença entre os campos será o tempo que este passou em processamento. Este tempo, juntamente com o campo `historico`, é usado no cálculo do `user_time_expected`, que é a previsão para o próximo tempo de processamento, usado na escolha do escalonador.

```
clock_t user_time_prev;      /* user time of last execution */
double historico;           /* hystory of last executions */
double user_time_expected;  /* time expected based in historico */
```

Figura 2. Campos adicionados à estrutura do processo

- **/usr/src/kernel/proc.c:**
 - **Função sched:** A função `sched` é chamada quando o processo já passou na CPU todo o tempo que ele poderia passar (ou seja, excedeu o *quantum*), e deve ser recolocado na fila de processos usuários aptos. Esse processo era simplesmente recolocado ao fim da fila. Ao contrário disso, a versão modificada atualiza os cálculos dos campos `user_time_expected`, `historico` e `user_time_prev` e então insere o processo no seu lugar devido na fila. Portanto, a fila de usuário é agora uma lista ordenada em ordem crescente de tempo

esperado para o próximo processamento. Isso é uma abordagem intuitiva, considerando que o processo escolhido, ou seja, o primeiro processo da “fila”, deve ser o que, espera-se, vá passar menos tempo na CPU.

```

/* Atualiza os campos do processo e, tendo o novo user_time_expected,
 * o insere na posicao devida na lista de processos aptos da prioridade
 * de usuario. */
rp=rdy_head[USER_Q]; /*o processo que estava executando estara na cabeca*/

/* formula do sjf */
rp->user_time_expected= ((PROC_A*(rp->user_time - rp->user_time_prev)) +
                        ((1 - PROC_A)*(rp->historico)));
/* atualiza o historico */
if(rp->historico)
    rp->historico= ((rp->historico + (rp->user_time - rp->user_time_prev))
                  /2);
else rp->historico= (double) rp->user_time;
/*guarda o user_time para a proxima vez */
rp->user_time_prev=rp->user_time;

```

Figura 3. Atualização dos campos da estrutura do processo na função sched

```

/*remove o processo da fila*/
rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
if(rdy_tail[USER_Q] == rp) rdy_tail[USER_Q]=NIL_PROC;
/*procura a posicao correta para reinseri-lo*/
aux=rdy_head[USER_Q];
while((aux != NIL_PROC) && (aux)
      && (aux->user_time_expected <= rp->user_time_expected))
{
    aux_prev=aux;
    aux=aux->p_nextready;
}
if((aux ==NIL_PROC) || (!aux))
{
    /* se aux eh nula, ou a fila estava vazia, ou o processo deve ser
     * inserido na ultima posicao dela*/
    if(rdy_tail[USER_Q]) (rdy_tail[USER_Q])->p_nextready=rp;
    rp->p_nextready=NIL_PROC;
    rdy_tail[USER_Q]=rp;
    if(!rdy_head[USER_Q]) rdy_head[USER_Q]=rp;
}
else
{
    /*se aux nao eh nula, o processo deve ser inserido entre ela e
     * aux_ant. Mas se aux_prev for nula, entao o processo serah o
     * primeiro da fila */
    if(aux_prev) aux_prev->p_nextready=rp;
    rp->p_nextready=aux;
    if(!aux_prev) rdy_head[USER_Q]=rp;
}
}
pick_proc();

```

Figura 4. Reinserção do processo na fila de acordo com o tempo estimado

- **Função ready:** A função *ready* coloca o processo na fila de aptos, quando ele é criado, ou quando ele volta do estado de bloqueado. Assim, ela apenas deve cuidar da inserção no local correto, pois os campos referentes ao tempo devem estar atualizados nesse momento.

```

/*ponteiros auxiliares nav e nav_aux definidos no inicio da funcao*/
nav=rdy_head[USER_Q];
nav_aux=NULL; /* =D */
/*procura a posicao correta para inserir o processo rp*/
while((nav != NIL_PROC) && (nav)
      && (nav->user_time_expected <= rp->user_time_expected))
{
    nav_aux = nav;
    nav=nav->p_nextready;
}
if((nav == NIL_PROC) || (nav == NULL))
{
    /*nav eh vazio, entao o processo serah o ultimo elemento da fila*/
    /* alem disso, se nav_aux for vazio tambem, entao rp serah o unico
     * elemento da fila*/
    if(nav_aux) nav_aux->p_nextready = rp;
    rdy_tail[USER_Q] = rp;
    rp->p_nextready = NIL_PROC;
    if((!rdy_head[USER_Q]) || (rdy_head[USER_Q] == NIL_PROC))
        rdy_head[USER_Q] = rp;
}
else
{
    /*nav nao eh vazio, entao o processo serah inserido entre os dois*/
    /* se nav_aux for nulo, entao o elemento serah o primeiro da fila */
    if(nav_aux) nav_aux->p_nextready = rp;
    else rdy_head[USER_Q] = rp;
    rp->p_nextready=nav;
}
}

```

Figura 5. Inserção do processo na fila de aptos na função *ready*

- **Função *unready*:** A função *unready* é chamada quando o processo passa para o estado de bloqueado. Então ela deve simplesmente recalculer os seus campos referentes ao tempo de processamento, pois o processo será removido da fila. Posteriormente, o processo será recolocado pela função *ready* usando os campos calculados aqui.

```

/*atualiza os campos de tempo para o sjf*/
rp->user_time_expected=
    ((PROC_A*(rp->user_time - rp->user_time_prev))
     + ((1 - PROC_A)*(rp->historico)));
if(rp->historico)
    rp->historico=((rp->historico
                   + (rp->user_time - rp->user_time_prev))
                 /2);
else rp->historico= (double) rp->historico;
rp->user_time_prev=rp->user_time;

```

Figura 6. Atualização dos campos da estrutura do processo pela função *unready*

3. Testes

Foram utilizados cinco programas de teste, o *Primes*, programas que já veio com a imagem original do Minix 2.0.4, o *Readfile*, que foi modificado a partir da versão que veio com a imagem original e mais três programas criados pelos autores deste trabalho, que são o *dd-modificado*, o *calculos* e o *misto*. Faremos uma breve análise de cada um destes programas de teste:

- **Primes:**

O *Primes* é um programa *CPU-Bound* que calcula todos os primos existentes até o número dado como margem superior de cálculo. Ele faz isso dividindo o valor limite por todos os outros valores inteiros anteriores a ele. Portanto, quanto maior o valor limite, maior o tempo de cálculo.

- **Readfile:**

O *Readfile* original é um programa *I/O-Bound* que realiza a abertura, leitura e fechamento de um arquivo. A modificação realizada neste programa para este trabalho foi incluir um novo parâmetro, *times*, que indica quantas vezes o *readfile* deve ser realizado. Ou seja, para um *times* igual a 1000, o arquivo será aberto, lido e fechado 1000 vezes.

- **Dd-modificado:**

O *dd* é um comando que copia um arquivo, possivelmente realizando conversões sobre o seu conteúdo. O *dd-modificado* copia um arquivo, sem realizar conversão alguma, fazendo isso um número de vezes especificado como argumento para o programa. Portanto, este programa é um *I/O-bound*.

- **Calculos:**

O *calculos* é um programa que realiza repetidamente (número de repetições é o argumento do programa) 9 vezes o comando $\text{sqrt}(\text{exp}(88))$; . As funções *sqrt* e *exp* estão definidas na biblioteca *math.h*. O valor 88 foi escolhido porque o programa gerava erros fatais executando no Minix quando o comando recebia um parâmetro maior. O *calculos* é um programa *CPU-Bound*.

- **Misto:**

O *misto* é uma combinação entre o *calculos* e o *dd-modificado*: uma cópia do arquivo é realizada, seguida de 2000 realizações do *calculos*, seguida de uma nova cópia. Essa seqüência de ações é realizada *times/2*, sendo *times* um dos parâmetros do programa. A intenção ao criar esse programa foi fazer com que o mecanismo SJF “se perdesse”, pensando que o programa fosse *I/O-Bound*, e então ele se tornasse *CPU-Bound*, e depois o contrário.

Além dos programas para o teste, foi desenvolvido o *cria_arquivo*, que recebe como parâmetro um valor em bytes e cria um arquivo com esse tamanho, contendo qualquer lixo existente na memória. Este programa é um auxiliar para criar arquivos parâmetros para o *dd-modificado*.

O comando *time*, do Minix, foi utilizado para obter os tempos real, de usuário e de sistema das execuções. Todas as execuções foram realizadas em uma máquina com processador Pentium 4 de 2.66GHz, com 2Gb de memória RAM. Todos os valores foram obtidos através da média aritmética entre os resultados de 4 execuções consecutivas.

❖ Teste 1 – O impacto sobre os CPU-Bounds

O primeiro teste consistiu na execução de uma única instância do programa *primes*, com o argumento 100000. De forma semelhante, foram obtidos os resultados para o *calculos*, também com argumento 100000.

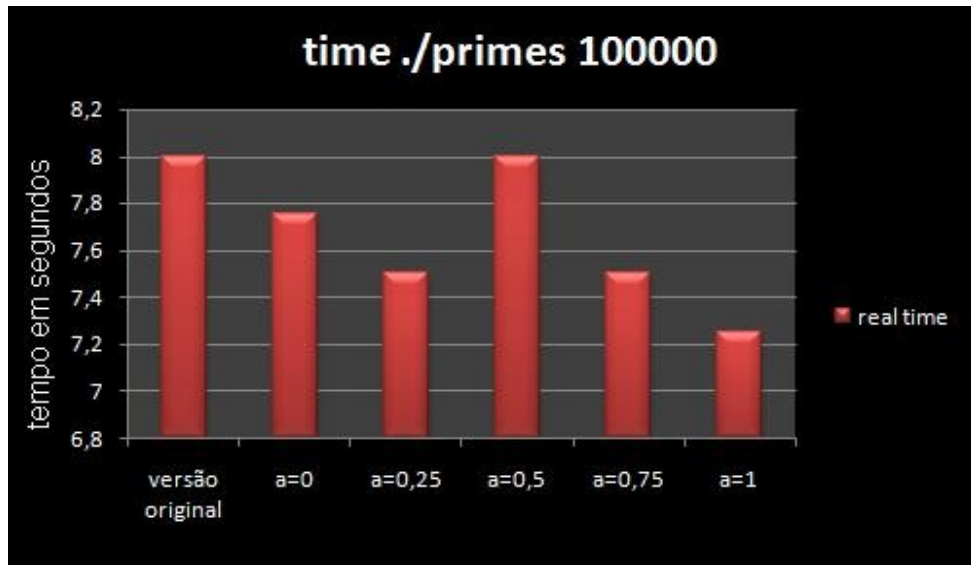


Figura 7. Resultados para a execução do *primes* isoladamente

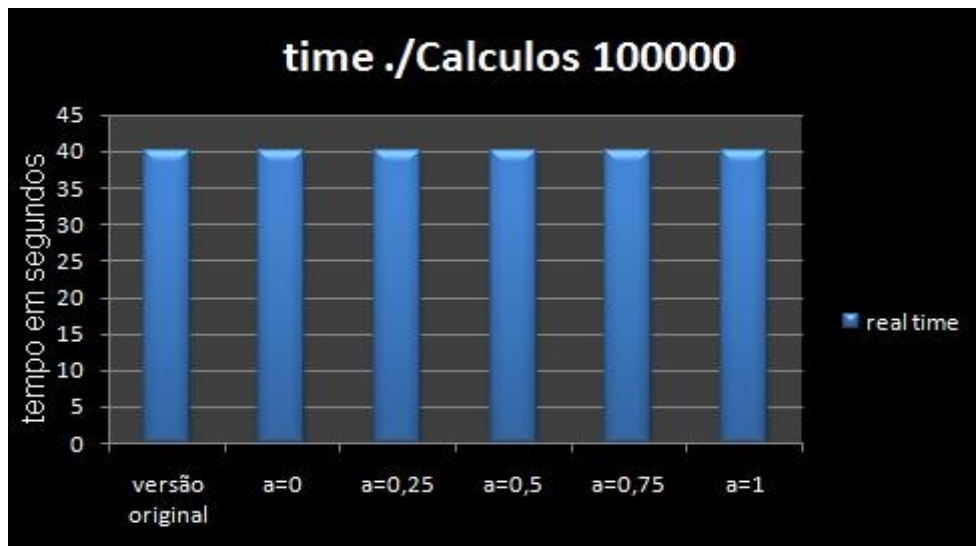


Figura 8. Resultados para a execução do *calculos* isoladamente

O primeiro gráfico nos dá a “ilusão” de uma melhora. No entanto, esta não chega a 0,8s, ou seja, 10%. Este valor pode ser ignorado, considerando-se que os testes não foram realizados em ambiente controlado. Já no segundo gráfico, essa melhora não aparece, mas também não há piora. Assim, conforme esperado, não há diferença no desempenho no caso de um único programa em execução, pois não é necessário competir pela CPU com mais processos.

Continuando as medições, foram disparadas 4 instâncias do *primes* concorrentemente, com os argumentos 25000, 50000, 75000 e 100000. Novamente, o mesmo esquema foi utilizado para medir o *calculos*.

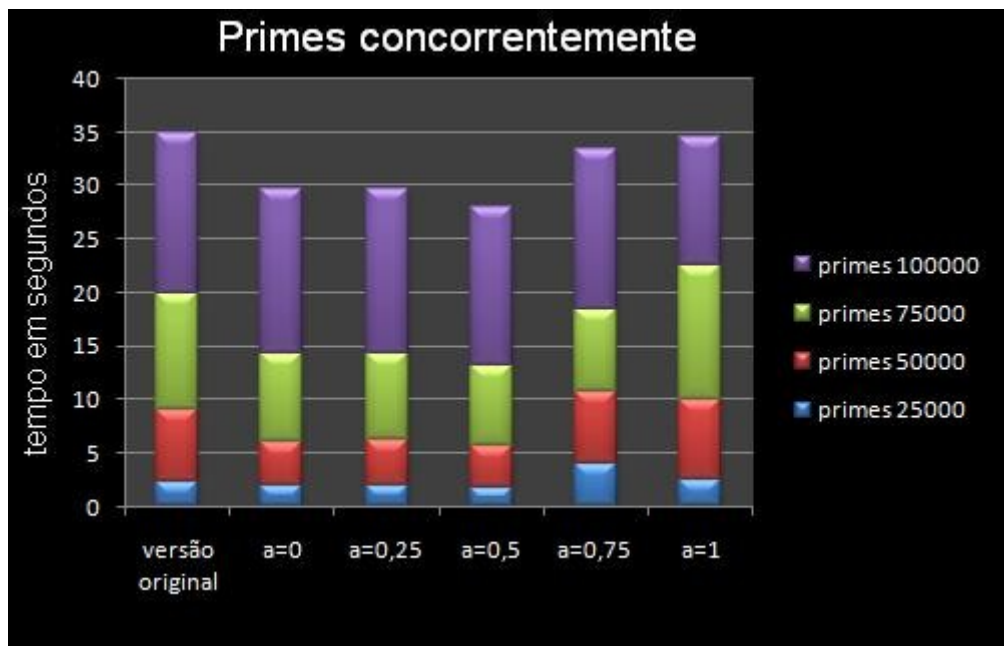


Figura 9. Resultados para a execução do *primes* com 4 instâncias concorrentes

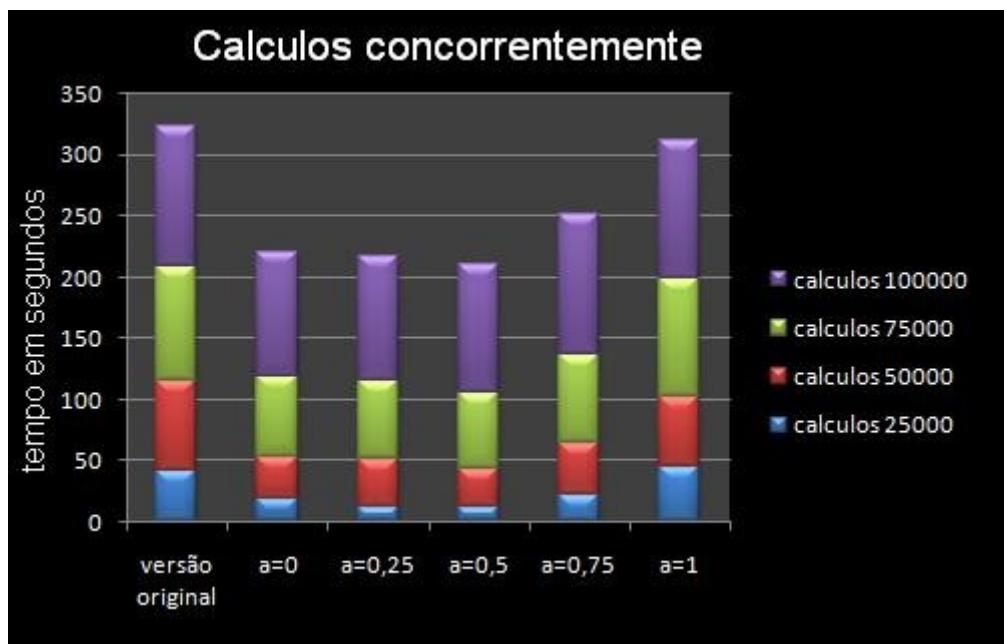


Figura 10. Resultados para a execução do *calculos* com 4 instâncias concorrentes

Apesar de as execuções serem concorrentes, e não seqüenciais, os gráficos mostram os tempos isoladamente para cada fase. Essa abordagem foi escolhida para a melhor visualização dos resultados.

Agora, concorrentemente, podemos ver uma diferença não desprezível entre os resultados. Isso acontece pois temos processos com cargas diferentes: apesar de

todos “representarem” o mesmo programa, a diferença nos parâmetros faz com que uns necessitem mais processamento que outros. Assim, a vantagem do escalonador SJF aparece.

Um comportamento que pode ser observado nos dois gráficos é que, em geral, quanto maior o valor da constante a , ou seja, quanto maior a importância dada à mais recente execução em relação às anteriores, pior fica o desempenho.

Já o contrário não causa tanto impacto. Conseqüentemente, $a=0,5$ parece ser o melhor valor para a nesse caso. Porém, como veremos adiante, tudo muda quando as cargas de trabalho são *I/O-Bound* ou quando são executadas diferentes cargas de trabalho concorrentemente.

❖ Teste 2 – O impacto sobre os I/O-Bounds

De forma análoga aos testes realizados sobre os programas *CPU-Bounds*, começamos pela execução isolada do *readfile* e do *dd-modificado*. Para o *readfile*, foi utilizado como arquivo de entrada o “arquivos.txt”, de 877 bytes, que foi disponibilizado para os testes; e como argumento “*times*” 10000. Já para o *dd-modificado*, foi criado (com o utilitário *cria_arquivo*) um arquivo chamado “arquivo.grande”, de 100000 bytes. O argumento “*times*” para este último é 360.

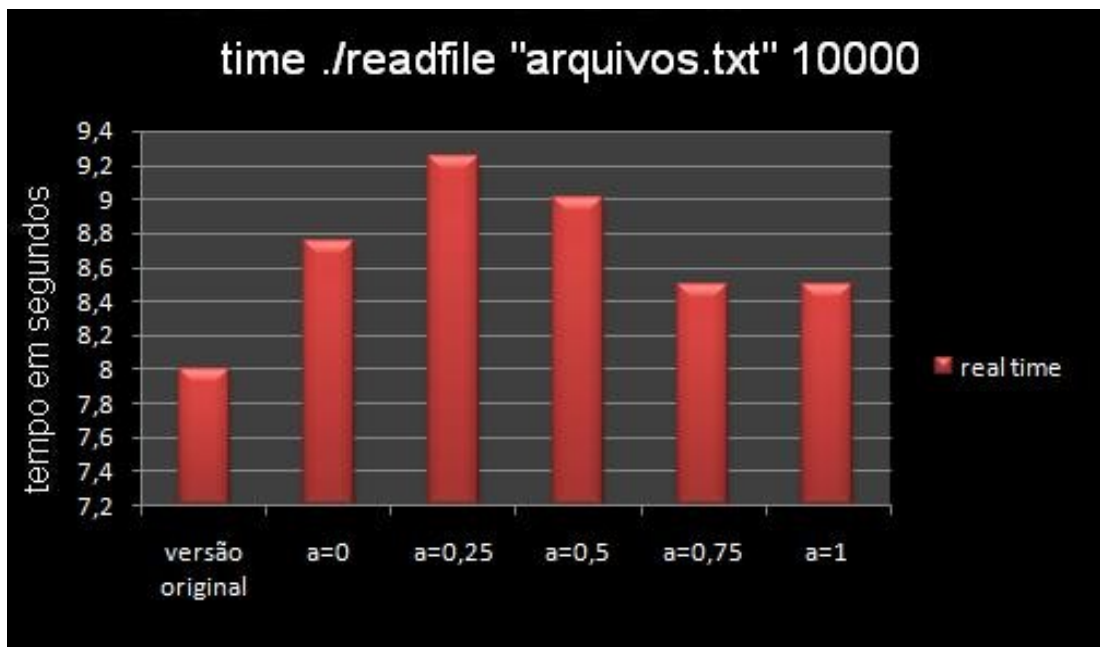


Figura 11. Resultados para a execução do *readfile* isoladamente

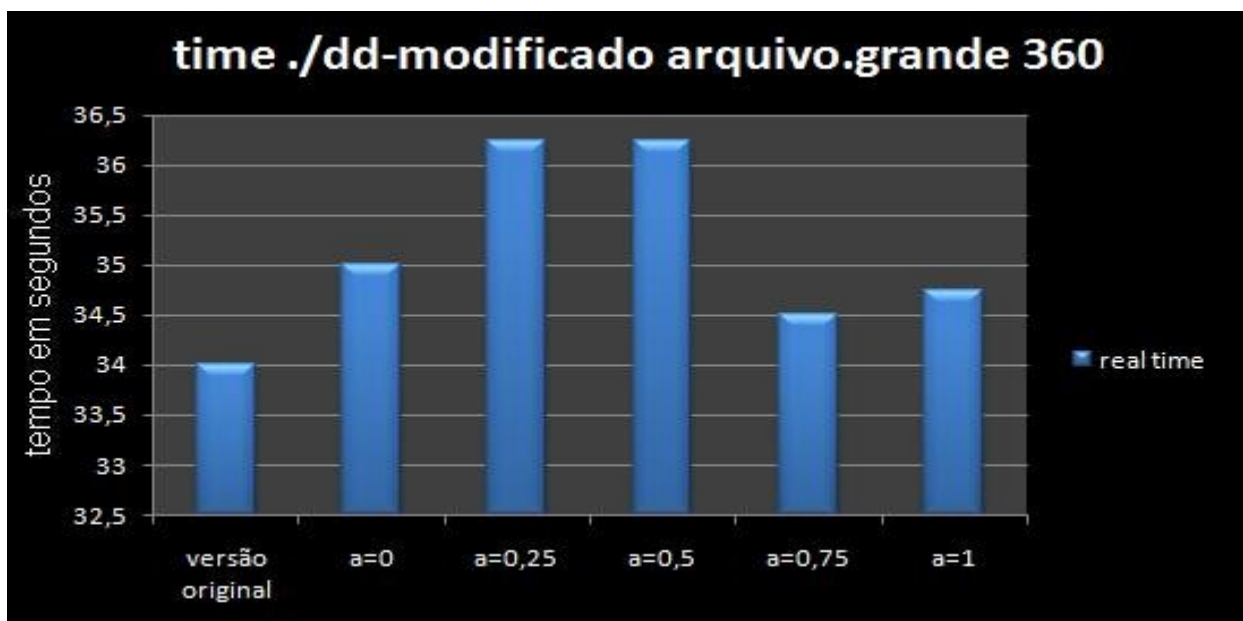


Figura 12. Resultados para a execução do *dd-modificado* isoladamente

Ao contrário do que aconteceu nos gráficos dos *CPU-Bounds*, aqui temos a impressão de uma piora no desempenho. Isso acontece porque os processos *I/O-Bounds* passam muito pouco tempo na CPU, e bastante tempo bloqueados, esperando entrada e saída.

Em outras palavras, cada operação de entrada e saída implica a passagem do processo da CPU para o estado de bloqueado, e depois para o estado de apto novamente. Observando um nível menor, a cada uma dessas operações, as funções *unready* e *ready* são executadas.

Com um grande número de operações, é esperado que o código que acrescentamos a essas funções acabe pesando no desempenho final.

Dessa vez, foram disparadas 4 instâncias do *readfile*, com o arquivo “arquivos.txt” e os argumentos 2500, 5000, 7500 e 10000. Da mesma forma, 4 instâncias do *dd-modificado*, com o arquivo “arquivo.grande” e os argumentos 90, 180, 270 e 360.

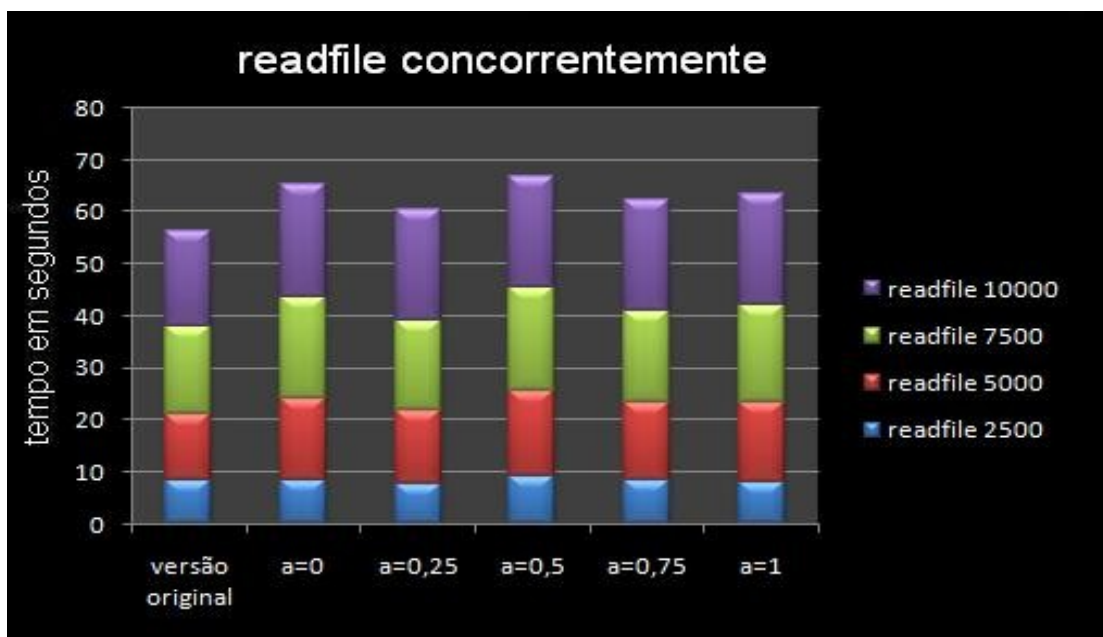


Figura 13. Resultados para a execução do *readfile* com 4 instâncias concorrentes

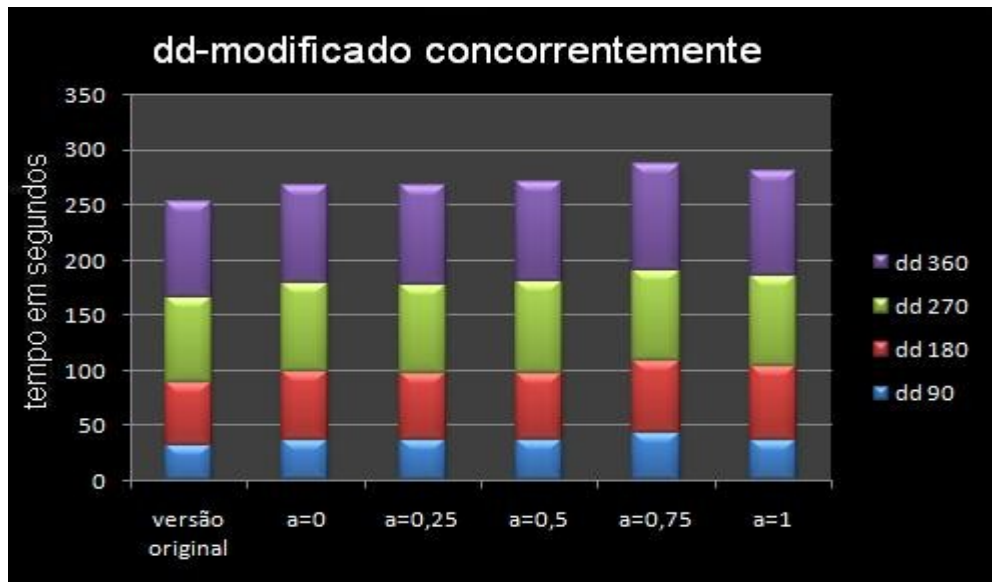


Figura 14. Resultados da execução do *dd-modificado* com 4 instâncias concorrentes

Vemos novamente o impacto causado pelas operações adicionadas ao escalonamento para a implementação do SJF.

Não há diferença significativa entre os valores para a , de forma que por muito pouco $a=0,25$ parece o melhor valor. Porém essa diferença pode existir apenas por não estarmos trabalhando em um ambiente controlado, assim como ocorreu uma diferença desprezível no teste 1.

❖ Teste 3 – O impacto sobre CPU-Bounds e I/O-Bounds concorrentemente

Os testes anteriores são úteis para verificar o peso das operações realizadas durante o escalonamento no desempenho final. No entanto, desejamos saber se o objetivo inicial de justiça no escalonamento em termos de espera do processo foi alcançado, ou seja, se conseguimos fazer com que os *I/O-Bounds* sejam escalonados primeiro, impedindo que eles sejam prejudicados pelo processamento pesado dos *CPU-Bounds*.

O próximo teste foi, então, o disparo de 4 instâncias do *primes* (argumentos 25000, 50000, 75000, 100000) e 4 instâncias do *readfile* (“arquivos.txt” e argumentos 2500, 5000, 7500, 10000), todas concorrentemente. Da mesma forma, 2 instâncias do *calculos* (25000, 50000, 75000 e 100000) com 4 instâncias do *dd-modificado* (“arquivo.grande” e argumentos 90, 180, 270 e 360). Os valores para os argumentos dos programas que foram executados concorrentemente foram escolhidos de forma a levarem, isoladamente, um tempo parecido em cada um deles, ou seja, para balancear a carga entre *CPU-Bounds* e *I/O-Bounds*.

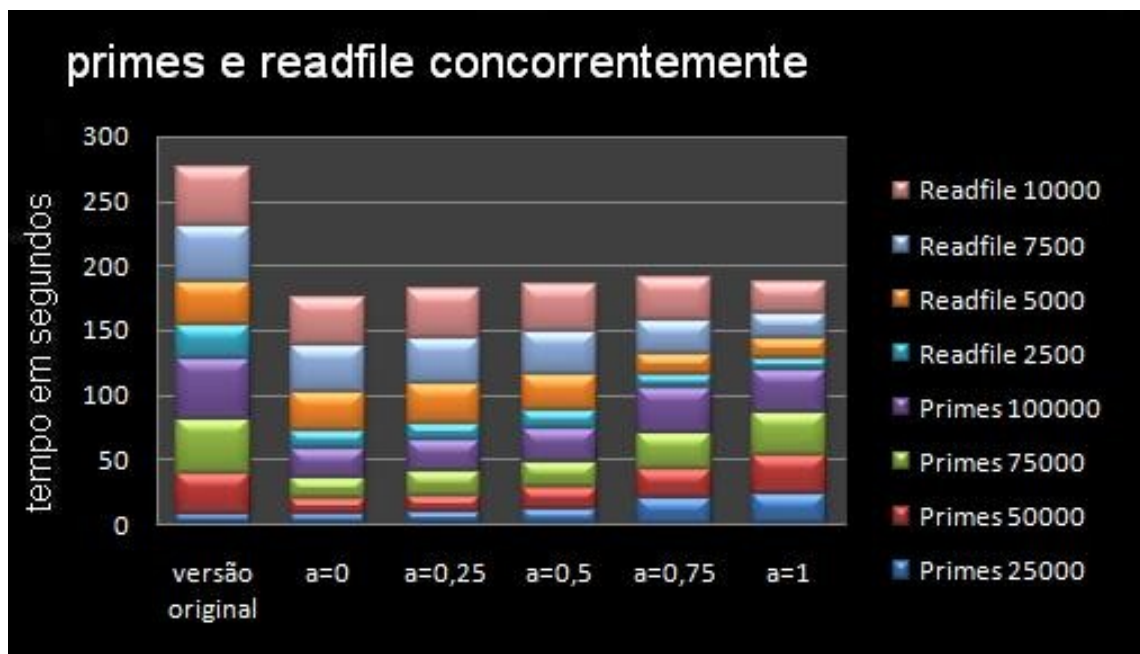


Figura 15. Resultados para 4 instâncias do *primes* e 4 instâncias do *readfile* executando concorrentemente

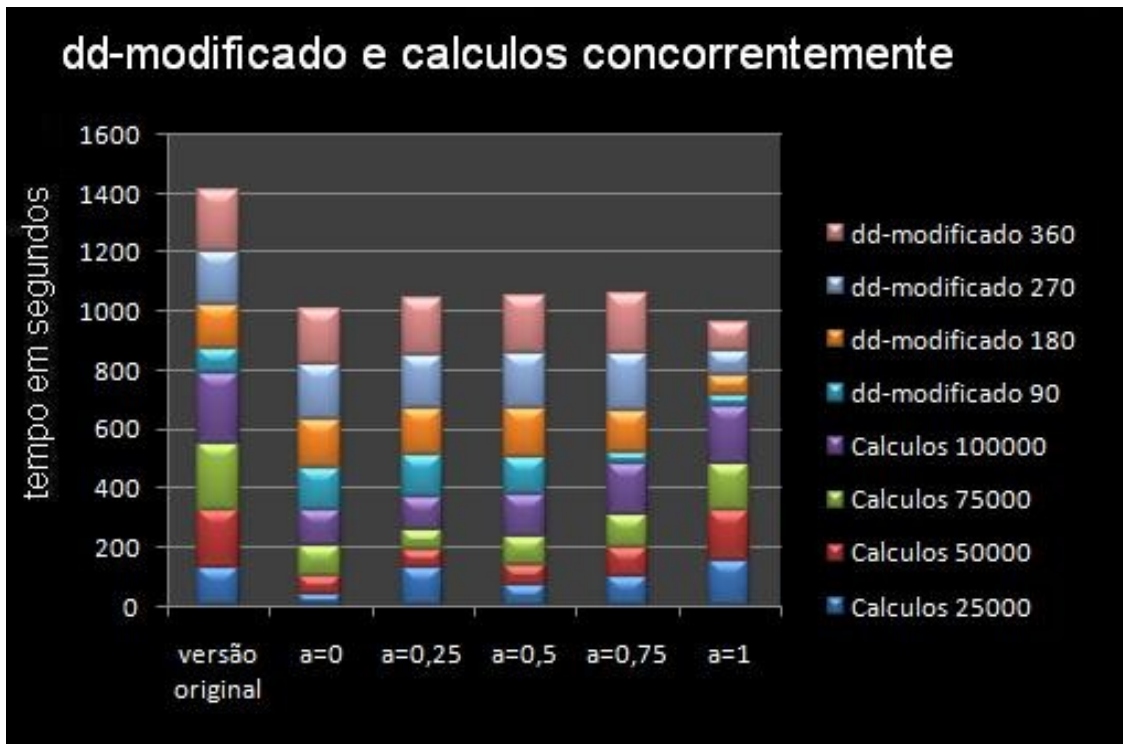


Figura 16. Resultados para 4 instâncias do *dd-modificado* e 4 instâncias do *calculos* executando concorrentemente

Vemos que, conforme o esperado, obteve-se um sensível ganho de desempenho para esse caso, pois agora os processos que passam mais tempo em requisições de entrada e saída não têm que esperar que os que gastarão todo o seu *quantum* para poder usar a CPU. Assim, enquanto os processos *I/O-Bounds* esperam as suas operações de I/O ficarem prontas, os processos *CPU-Bounds* ocupam a CPU para realizarem seus processamentos.

Um fenômeno interessante observado nos gráficos das execuções concorrentes pode ser visto nas figuras 17 a 20: para os processos *CPU-Bounds* (figuras 17 e 18), há ganho de desempenho. Isso ocorre porque os diferentes parâmetros formam cargas diferentes, que passam a ser bem equilibradas. Além do ganho de desempenho, nota-se que, quanto maior for o valor de *a*, pior será o desempenho na execução dos processos porque, a diferença entre as cargas de trabalho não pode ser sentida nas últimas execuções, ou seja, em pequenos blocos do programa, que são iguais para todas. A diferença só é percebida olhando-se todo o histórico e, quanto menor for o *a*, mais importância se dá ao histórico. Já nos processos *I/O-Bounds* (figuras 19 e 20), há até perda de desempenho para o *dd-modificado*, devida ao *overhead* causado pelas operações necessárias ao algoritmo de escalonamento, conforme comentado no teste 2.

Quanto menor o valor de a , melhor o desempenho na execução destes processos. Isso é explicado pelo fato de que, ao contrário dos *CPU-Bounds*, estas cargas têm por característica que as operações de entrada e saída são consecutivas, reunidas em uma parte do programa, ou seja, olhando-se para a última vez que o processo usou a CPU, tem-se o reflexo perfeito do que será a próxima vez. E quanto maior for o a , maior a importância dada ao tempo mais recente.

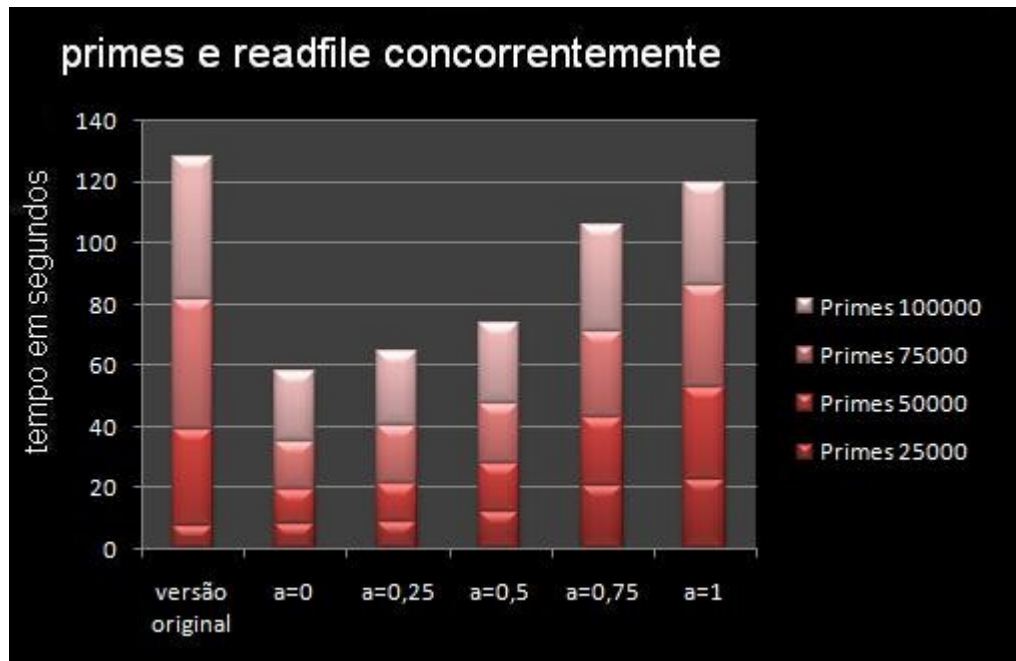


Figura 17. Resultados para as 4 instâncias do *primes* em execução com 4 instâncias do *readfile* concorrentemente

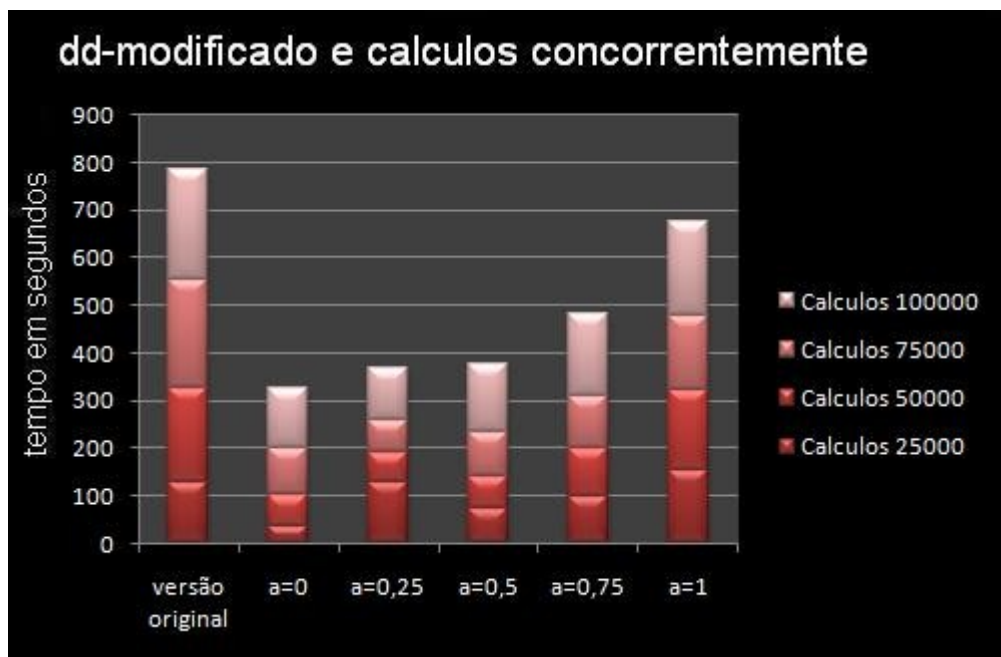


Figura 18. Resultados para as 4 instâncias do *calculos* em execução com 4 instâncias do *dd-modificado* concorrentemente

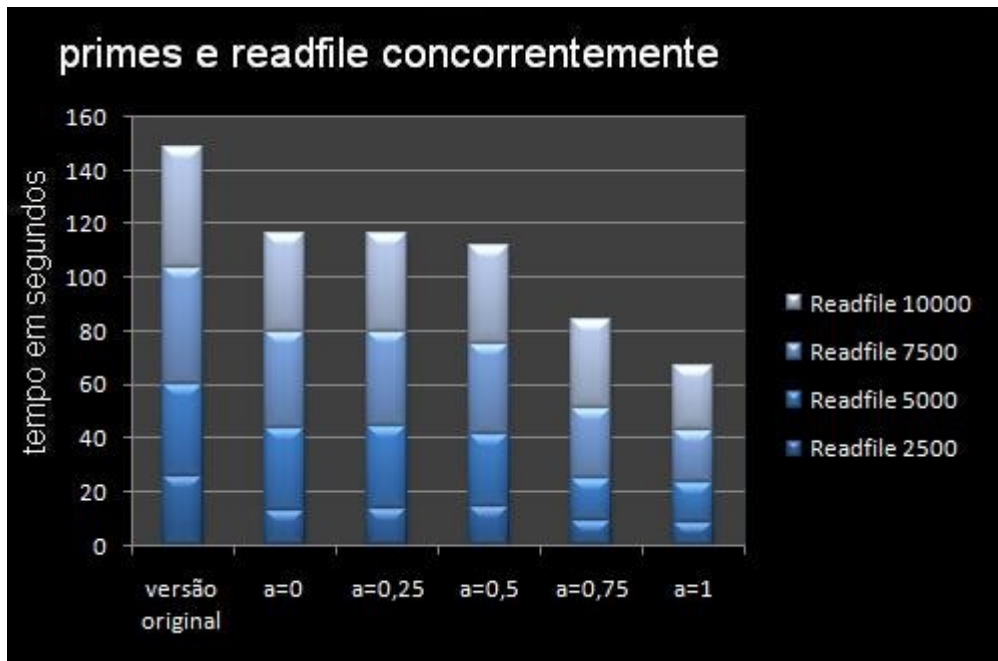


Figura 19. Resultados para as 4 instâncias do *readfile* em execução com 4 instâncias do *primes* concorrentemente

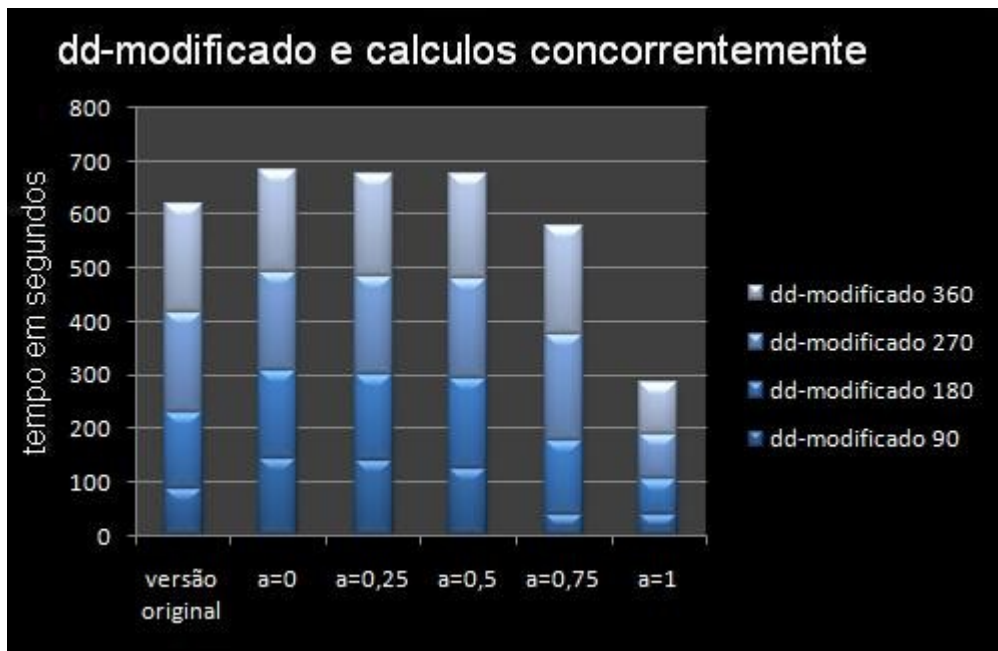


Figura 20. Resultados para as 4 instâncias do *dd-modificado* em execução com 4 instâncias do *calculos* concorrentemente

No entanto, nos gráficos das figuras 15 e 16 vemos diferenças entre os resultados para $a=0$ e $a=1$, o que seria inesperado, considerando-se o que foi dito sobre quem se favorece com que valor de a . Isso ocorreu porque, provavelmente, a quantidade de “trabalho” *CPU-Bound* e *I/O-Bound* não estava tão balanceada quanto esperávamos escolhendo os argumentos para os programas. Teoricamente, os

resultados para $a=0$ e $a=1$ seriam iguais para uma carga devidamente balanceada. Portanto, a escolha do a depende da aplicação destino.

Não é possível de se visualizar nos gráficos anteriores, porém foi notado que o nível de aleatoriedade de término de execução de processos, que é a ordem em que os processos terminam, é dependente de a .

Quanto menor o a , os processos *CPU-Bound* possuem uma aleatoriedade quase nula, terminando em ordem na maioria dos testes, o que é justamente o contrário para os processos *I/O-Bound* com valores pequenos de a , principalmente $a=0$.

Porém quando o a vai aumentando, pode-se notar que esta aleatoriedade vista nos processos *I/O-Bound* vai passando gradativamente para o lado dos *CPU-Bound*, que com $a=1$, possuem aleatoriedade de término de execução análoga à dos processos *I/O-Bound* para $a=1$.

❖ Teste 4 – Um programa que se alterna entre CPU-Bound e I/O-Bound

O último teste consiste na execução concorrente de 4 instâncias do programa *misto*, tendo como arquivo de entrada o “arquivo.grande” e como parâmetros 25, 50, 75 e 100.

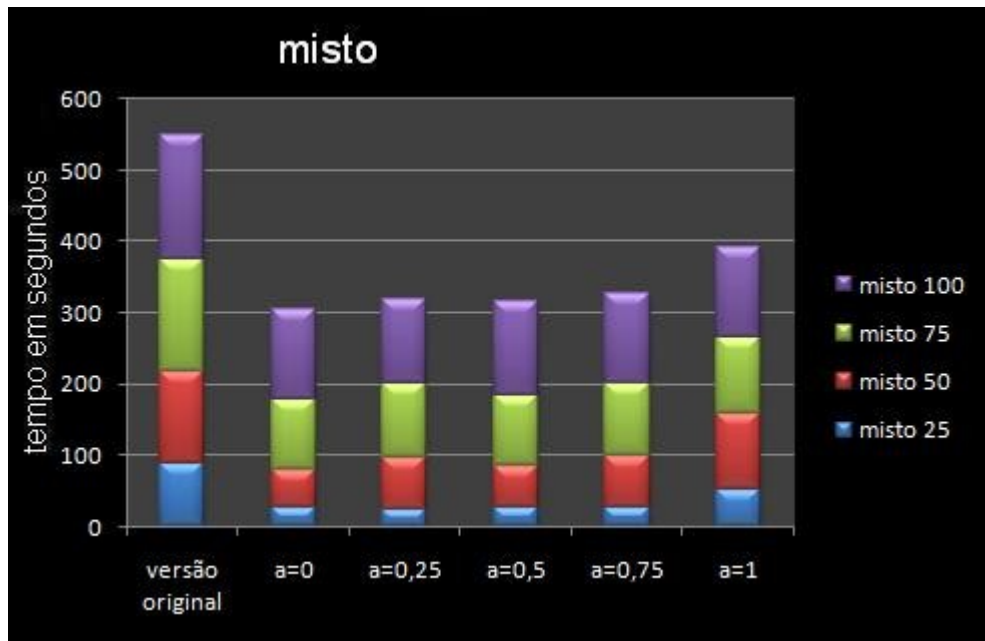


Figura 21. Resultados para 4 execuções concorrentes do *misto*

Ao contrário do esperado, houve ganho de desempenho nessa execução. Isso aconteceu porque, apesar de o escalonador ser “enganado” se olhar para a última execução, o histórico mostra a verdade. E, mesmo que o histórico não seja considerado, como em $a=1$ (com um pouco menos de desempenho, como se observa), o escalonador só será enganado no início de cada fase, e acertará o resto, salvando, assim, da condenação, o algoritmo SJF.

4. Conclusões

Após o desenvolvimento deste gerenciador de processos SJF, podemos constatar que, embora ele gaste certo tempo de processamento para atualizar dados do processo e inserir o mesmo na posição ideal na fila de processos aptos, os resultados gerais dos quatro testes efetuados foram bastante motivadores, chegando a surpreender em alguns casos, como no teste 4.

Isso demonstra o impacto positivo ao se escolher algoritmos mais robustos para soluções em termos de sistema operacional. Por mais que o algoritmo FIFO usado na versão original do Sistema Operacional Minix 2.0.4 seja extremamente simples de implementar e custe pouco processamento, os gastos referentes a implementação do SJF e o aumento do processamento nas operações de atualização da fila são, praticamente, mascarados pelos bons resultados obtidos.