



**PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS**  
**ENGENHARIA DE COMPUTAÇÃO**  
**SISTEMAS OPERACIONAIS A**

Carlos Henrique Vieira Marques – 18720367

Erick Matheus Lopes Pacheco – 18711630

Hiago Silva Fernandes – 18726455

Leonardo Sanavio – 18054395

Marcos Antônio Junior Vasconcellos – 18720920

**EXPERIMENTO #1**

**CRIAÇÃO DE PROCESSOS E CONCEITO DE TEMPO DENTRO DO S.O.**

CAMPINAS

Fevereiro 2020

## SUMÁRIO

1.	INTRODUÇÃO .....	3
2.	AMBIENTE DOS TESTES .....	4
3.	PERGUNTAS .....	5
3.1.	REFERENTES AOS SISTEMAS UNIX.....	5
3.2.	REFERENTES AO PROGRAMA EXEMPLO.....	6
4.	O PROGRAMA EXEMPLO .....	8
4.1.	CONCEITOS .....	8
4.2.	OS ERROS.....	9
5.	RESULTADOS .....	11
5.1.	REFERENTES AO PROGRAMA EXEMPLO .....	12
5.2.	REFERENTES AO PROGRAMA MODIFICADO .....	15
6.	ANÁLISE DOS RESULTADOS .....	17
6.1.	PROGRAMA EXEMPLO .....	18
6.2.	PROGRAMA MODIFICADO .....	20
7.	CONCLUSÃO.....	21
8.	BIBLIOGRAFIA.....	22

## 1. INTRODUÇÃO

O presente experimento visa a identificação de características específicas dos sistemas operacionais baseados em Unix na criação e manipulação de processos, bem como a relação entre os processos pais e filhos e informações relevantes a respeito destes durante o tempo de execução, tal como o tempo que cada processo leva para ser executado em um sistema ocioso e também sobrecarregado.

Para realizarmos tal tarefa, utilizar-se-á um programa exemplo fornecido pelo professor cujo código se dá por uma rotina de criação de processos filhos através de chamadas de sistemas em laços de repetição; o filho chamado, por sua vez, salvará o instante (tempo) inicial e fará uma quantidade pré-determinada de iterações, entrando em um estado de dormência utilizando **sleep** nesta mesma quantidade, salvando em seguida o instante (tempo) final e, por fim, fazendo o cálculo de desvio de tempo ocasionado pelas chamadas para o S.O. bem como do uso da própria função sleep.

No programa modificado, por outro lado, aumentamos o número de processos filhos e o tempo de dormência de cada um deles para múltiplos de 200, começando em 400 microssegundos. Outro diferencial é a implementação da chamada do filho por meio da função “execvp()”, especificando o caminho do arquivo do filho no primeiro parâmetro e passando um ponteiro de vetor de chars no segundo para os argumentos (neste, especificando o tempo de dormência). Nisso, o objetivo consistirá em perceber o tempo de medição e também detectar as mudanças em relação ao primeiro.

## 2. AMBIENTE DOS TESTES

Antes de partirmos para os resultados dos testes, é válido que os façamos sob um ambiente de trabalho com a maior equivalência de hardware possível e livre de interferências (de softwares externos, por exemplo), para que os testes feitos nos forneçam um resultado coerente.

Foram utilizadas três máquinas:

- Dell Optiplex 9020 i7 4790 **3.6GHz** & 8GB de RAM DDR3 1600MHz;
- Notebook i7 7500U **2.7GHz** & 8GB de RAM DDR4 2400MHz;
- Desktop i7 4790 **4.0GHz** & 16GB de RAM DDR3 1600MHz.

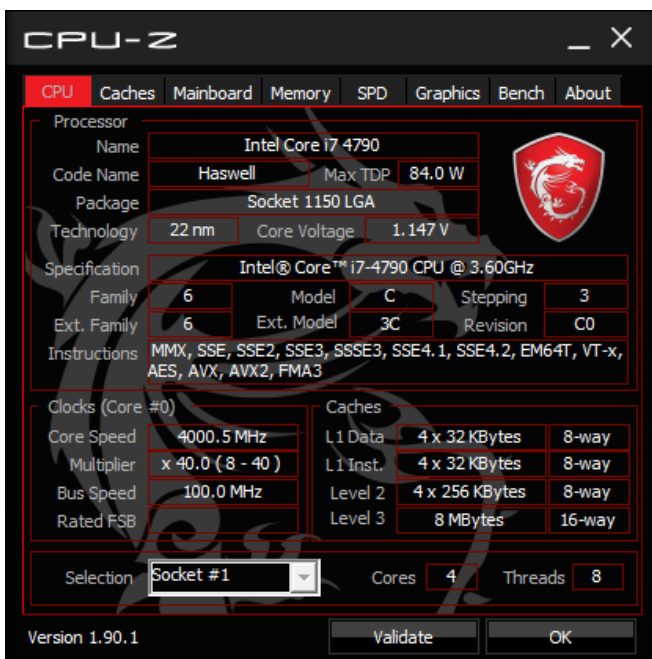


Figura 1 - Detalhes da CPU (desktop)

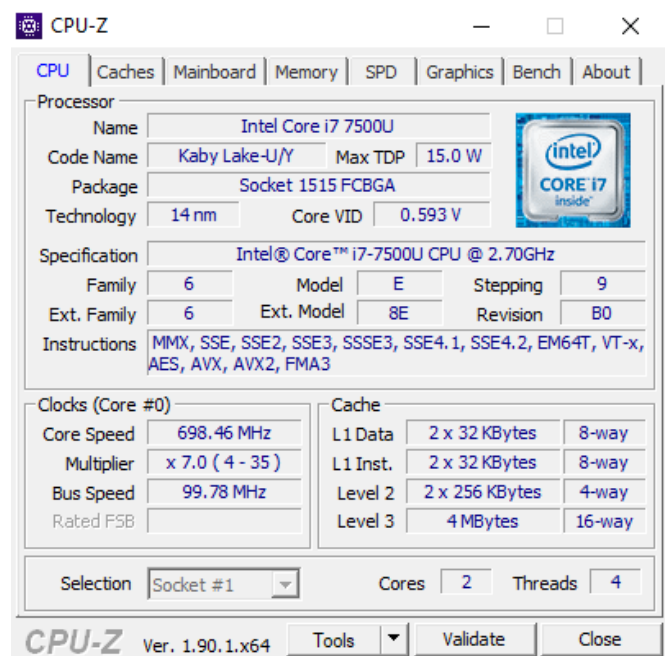


Figura 2 - Detalhes da CPU (notebook)

Com as três máquinas em “jogo”, iremos além da proposta: compararemos também o resultado das medições de tempo entre um S.O. em máquina virtual vs dual boot bem como entre uma distribuição de Linux e outra, de maneira a analisar possíveis diferenças e os potenciais explicativos.

### 3. PERGUNTAS

#### 3.1. REFERENTES AOS SISTEMAS UNIX

**Pergunta 1:** Apresente a linha de comando para compilar o programa exemplo, de tal maneira que o executável gerado receba o nome de “experimento1” (sem extensão).

**Resposta:**

- Comando para compilar: “gcc exemplo1.c -o exemplo1”
- Comando para executar: “./exemplo1”

**Pergunta 2:** Descreva o efeito da diretiva &:

**Resposta:** O operador **& (Ampersand)** é utilizado para colocar o comando anterior a ele em background (segundo plano) e, se houver um comando posterior, ele será executado independente do resultado do comando anterior.

**Pergunta 3:** Qual é o motivo do uso do “./”? Explique porque a linha de comando para executar o gcc não requer o “./”.

**Resposta:** O “./” indica ao shell que o comando ou arquivo seguido após a barra encontra-se no diretório em que o Terminal está sendo executado, de modo que “.” denota o diretório atual e a “/” especifica ao shell o caminho (\$PATH) para o arquivo. No caso da execução do GCC, o mesmo trata-se de um compilador pré-instalado em sistemas Unix, portanto, ao executarmos o comando “gcc” no Terminal seguido do nome direto do arquivo, o próprio gcc se encarrega de executar o comando para a busca do caminho do arquivo, dispensando o uso de “./” por parte do usuário. Concluindo que cada processo/programa que roda nos sistemas Linux (por exemplo) tem uma maneira pré-definida de fazer calls para outros programas, módulos, processos com a possibilidade de passar através de parâmetros outros módulos para a execução do programa desejado.

**Pergunta 4:** Apresente as características da CPU do computador usado no experimento.

Resposta presente no tópico anterior [2].

**Pergunta 5:** O que significa um processo ser determinístico?

**Resposta:** conjunto de instruções e/ou estados que são pré-determinados, assumindo valores e ações previsíveis com base nos estados atuais, podendo ser ou não finito, e tendo sempre o mesmo fim.

**Pergunta 6:** Como é possível conseguir as informações de todos os processos existentes na máquina, em um determinado instante?

**Resposta:** Através do comando “ps -aux”, que faz uma listagem de todos os processos rodando num determinado instante, trazendo diversas informações referentes a cada um. O funcionamento dessa linha de comando se dá pelo uso do sistema de arquivos “proc” presente no Unix, que armazena numa pasta específica para cada processo no diretório proc/PID as informações referentes a estes, e a cada syscall, tanto para revezamento de processos no tempo de CPU quanto para demais interrupções, realiza-se a busca neste diretório.

### 3.2. REFERENTES AO PROGRAMA EXEMPLO

**Pergunta 1:** O que o compilador gcc faz com o arquivo .h, cujo nome aparece após o include?

**Resposta:** Durante o período de compilação, o pré-processador da linguagem faz a linkagem das bibliotecas (“.h”), de modo que as funções presentes no código derivadas de tais bibliotecas possam ser reconhecidas e, então, traduzidas para linguagem de montagem no período de execução. Desta forma, com os códigos.h sendo interpretados pelo compilador GCC, as requisições do programa passam a ser atendidas.

**Pergunta 2:** Apresentar (parcialmente) e explicar o que há em <stdio.h>.

**Resposta:** A lib `stdio.h` ou standard input-output header (cabeçalho padrão de entrada/saída) define três tipos de variáveis (`size_t`, `ARQUIVO`, `fpos_t`), várias macros e várias funções para executar entrada e saída. Dentre as funções de execução mais utilizadas estão a `printf` (envia saída formatada para `stdout`), `scanf` (este comando efetua uma leitura do teclado), `fflush` (Liberta o buffer de saída de um fluxo), `FILE` e `fopen` (realiza syscall para abertura de arquivos).

**Pergunta 3:** Qual é a função da diretiva `include` (linha que começa com `#`), com relação ao compilador?

**Resposta:** Indicar ao compilador as dependências do programa de modo que este possa fazer o pré-processamento corretamente e incluir tais dependências, sejam elas, declarações de estruturas de dados (classes, estruturas, enumerações), constantes, cabeçalho de funções/métodos, macros e eventualmente algum código quando se deseja que uma função seja importada direto no fonte ao invés de ser chamada (*source inline*), o que seria as chamadas “bibliotecas dinâmicas”.

**Pergunta 4:** O que são e para que servem `argc` e `argv`? Não esqueça de considerar o `*` antes de `argv`.

**Resposta:** O **`argc`** recebe um inteiro cujo valor especifica o número de argumentos com que a `main()` foi chamada na linha de comando, já o **`argv`** – é um vetor de char que contém os argumentos, um para cada string passada na linha de comando. E para podermos saber quantos elementos temos em **`argv`** é utilizado o **`argc`**. A questão do `*` indica array de ponteiros para caracteres". Dessa forma **`argv`** fica sendo um array em que cada elemento contém o endereço de uma posição de memória na qual o sistema operacional dispôs o primeiro caráter dos argumentos passados à chamada `execv()` ou `execvp()` que inicia o programa.

**Pergunta 5:** Qual a relação: entre `SLEEP_TIME` e o desvio, nenhuma, direta ou indiretamente proporcional?

**Resposta:** Direta, o `SLEEP_TIME` calcula o "delay" ocorrido na CPU durante o processo de compilação do programa. Geralmente esse delay ocorre em razão da

CPU estar ocupada atendendo outra chamada do SO como por exemplo de E/S. Dessa forma o **desvio** calcula a ocorrência do curto período de espera para reprodução em relação ao tempo total de reprodução do programa.

#### 4. O PROGRAMA EXEMPLO

O programa exemplo, já explicado superficialmente na introdução, continha uma série de erros propositais que eram parte do desafio, dentre eles, erros de lógica e sintaxe. Antes de seguirmos para a identificação e solução de tais erros, trataremos o conceito de algumas funções implementadas no código original e o seu propósito.

##### 4.1. CONCEITOS

**Função “fork()”:** Presente nos sistemas Unix, o fork() é um system call que permite a “clonagem” de um processo em execução, criando um processo “filho” idêntico ao que o chamou (o pai), fazendo uma cópia de toda a memória virtual (incluindo o contador de programas) em outro espaço de memória. Apesar de idênticos, o que acontece em um não interfere no outro por terem PIDs e espaços de memória distintos. A função não recebe parâmetro e seu retorno consiste em:

Valor negativo – criação do processo filho não sucedida;

Zero – retorna para o filho recém criado;

Valor positivo – retorna para o pai, o valor contém o PID do processo filho.

**Função “wait()”:** A função wait() é um system call que permite a suspensão (bloqueio) de um processo em execução até que se tenha a finalização da atividade de um processo filho ou um sinal seja recebido. Quando o processo filho terminar a sua execução o processo do “pai” continua a sua execução após a instrução de espera. A função recebe um parâmetro inteiro e esse argumento é preenchido com o código de saída do processo filho (se o argumento não for NULL). Os valores de retorno podem ser:

ID do processo filho – em caso de sucesso execução.



Menos um (-1) – em caso de falha de não haver nenhum processo filho em execução.

**Função “execvp()”:** Faz parte de uma família de funções que permitem que haja a substituição do processo atual por outro. A `execvp()` cria a execução de um processo filho que, por sua vez, faz a execução de um arquivo executável comum cujo caminho é especificado no primeiro parâmetro desta. O segundo parâmetro, porém, são os argumentos do programa, passados como uma matriz de ponteiro de caracteres. Não faz nenhum retorno.

As demais funções presentes no programa não possuem necessidade de explicação devido serem comuns e já conhecidas anteriormente. No mais, existe a função “`kill(pid_t pid, int sig)`” requisitada para o programa modificado que recebe o PID de um processo no primeiro parâmetro e envia o sinal especificado no segundo para ele.

#### 4.2. OS ERROS

A seguir, os trechos contendo os erros e a nossa solução.

##### Trecho #1:

```
/*
 * Criacao dos processos filhos
 */
rtn = 1;
for (count = 0; count < NO_OF_CHILDREN; count + -) {
    if (rtn == 0) {
        rtn = fork();
    } else {
        break;
    }
}
```

Neste trecho temos três erros: o primeiro encontra-se no incremento do contador, o uso do operador “+” com o “-” junto neste contexto ocasiona um looping infinito. O segundo, na comparação do if. A variável `rtn` é inicializada com o valor 1, o que implica que a condicional if nunca seja verdadeira, caindo no else e

interrompendo o laço de repetição, deixando de criar um processo filho, e o último...  
A variável "rtn" não foi declarada anteriormente no programa.

### Solução:

```
int rtn = 1;
for (count = 0; count < NO_OF_CHILDREN; count++) {
    if (rtn != 0)
```

Para entendermos a solução, devemos ter em mente que a variável "rtn" recebe o retorno do fork(), quando um filho é invocado ele retorna ao seu próprio rtn o valor 0, saindo desse laço de repetição, enquanto o pai permanece gerando filhos e recebendo do fork() o PID (id de processo) destes até o número determinado de iterações chegar ao fim (iteraões estas definidas por NO\_OF\_CHILDREN).

Tendo isto em mente, corrigimos o erro de incremento do contador para que as iterações fossem realizadas conforme a condição e trocamos o operador da condicional para **diferente** de zero, fazendo com que somente o processo pai a satisfaça, considerando que os processos filhos estarão com *rtn* = 0.

### Trecho #2:

```
/*
 * Exibe os resultados
 */
printf("Filho #%d - desvio total: %.3f - desvio medio: %.1f\n", child_
no, drift - NO_OF_ITERATIONS * SLEEP_TIME / MICRO_PER_SECOND,
(drift - NO_OF_ITERATIONS * SLEEP_TIME / MICRO_PER_SECOND) /
NO_OF_ITERATIONS);
```

A medição é feita em microssegundos, porém o print está mostrando apenas três casas decimais para o desvio total e somente uma para o desvio médio, o que não faz sentido quando queremos precisão de resultado.

### Solução:

```
printf("\nFilho #%d -- desvio total: %.8f -- desvio medio: %.8f\n"
```

Apenas alterar o número de casas decimais mostradas, neste caso, é suficiente.

### Trecho #3

```
/*
 * Sou pai, aguardo o termino dos filhos
 */
for (count = 0; count > NO_OF_CHILDREN; count++) {
    wait(NULL);
}
```

Se o intuito deste trecho é fazer o pai aguardar o término dos filhos, então a condição do **for** está incorreta. Utilizar o comparador “>” como condição neste contexto implica em nunca entrar no laço, visto que a variável “count” começa em 0, não sendo maior que o NO\_OF\_CHILDREN definido como 3 no programa exemplo.

### Solução:

```
for (count = 0; count < NO_OF_CHILDREN; count++)
```

A solução dispensa maior explicação, apenas se inverte o comparador.

## 5. RESULTADOS

Foram realizados uma série de testes com o processador em estado “ocioso” e a cada rodada aumentamos em 5 o número cargas no processador, cargas estas executadas por este código já fornecido a nós:

```
#include <stdio.h>

int main() {
    long int count = 0;

    while (1) {
        count++;
        count--;
    }
    return 0;
}
```

Consiste em um looping infinito que incrementa e decrementa uma variável. O objetivo de tal carga é para entendermos como funciona o revezamento de processos

que os SOs baseados em Unix fazem sobre o processador e como isso impacta na performance e tempo de execução.

### 5.1. REFERENTES AO PROGRAMA EXEMPLO

Foram feitos testes de duas distribuições de Linux: Ubuntu 18.04 LTS e Deepin 15.11, ambos baseados no Debian.

#### Ubuntu:

Sistema							
Ubuntu 18.04 LTS (Dual Boot) com CPU Core i7 4790 3.6GHz							
Rodada	Nº de Cargas	Filho #1		Filho #2		Filho #3	
		Desvio Total	Desvio Médio	Desvio Total	Desvio Médio	Desvio Total	Desvio Médio
1	0	0,14249396	0,00014249	0,14241898	0,00014242	0,14234805	0,00014235
2	5	0,05362999	0,00005363	0,05352604	0,00005353	0,05358005	0,00005358
3	10	0,05260098	0,00005260	0,05259502	0,00005260	0,05273604	0,00005274
4	15	0,05280399	0,00005280	0,05280995	0,00005281	0,05280697	0,00005281
5	20	0,05884099	0,00005884	0,05268502	0,00005269	0,06564200	0,00006564
6	25	0,05281401	0,00005281	0,05992496	0,00005992	0,05973601	0,00005974
7	30	0,05281401	0,00005281	0,05849004	0,00005849	0,05260706	0,00005261
8	35	0,05554700	0,00005555	0,05554903	0,00005555	0,05848908	0,00005849
9	40	0,05555403	0,00005555	0,05260003	0,00005260	0,06144297	0,00006144
10	45	0,05849099	0,00005849	0,07257700	0,00007258	0,05847204	0,00005847

Figura 3 – Resultado dos testes do Ubuntu em dual boot

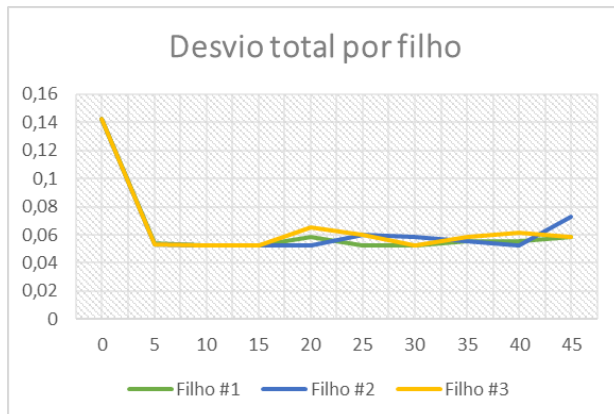


Figura 4 - Gráfico dos desvios totais

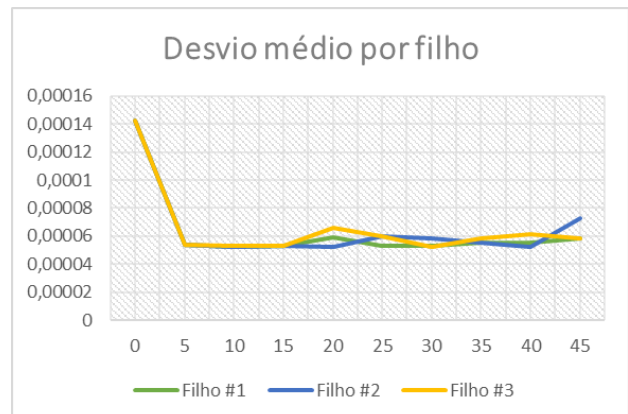


Figura 5 - Gráfico dos desvios médios

Sistema							
Ubuntu 18.04 LTS (Máquina Virtual) com CPU Core i7 4790 4.0GHz							
Rodada	Nº de Cargas	Filho #1		Filho #2		Filho #3	
		Desvio Total	Desvio Médio	Desvio Total	Desvio Médio	Desvio Total	Desvio Médio
1	0	2,06964731*	0,00206965*	2,06895136*	0,00206895*	2,06678501*	0,00206678*
2	5	0,45938802	0,00045939	0,52385700	0,00052386	0,45325100	0,00045325
3	10	0,51935506	0,00051936	0,48389006	0,00048389	0,47747803	0,00047748
4	15	0,56420708	0,00056421	0,55220604	0,00055221	0,56262803	0,00056263
5	20	0,58299804	0,00058300	0,57283807	0,00057284	0,57232702	0,00057233
6	25	0,45626402	0,00045626	0,46234393	0,00046234	0,48864794	0,00048865
7	30	0,46439099	0,00046439	0,46742499	0,00046742	0,49324393	0,00049324
8	35	0,50395501	0,00050396	0,51582599	0,00051583	0,51285899	0,00051583
9	40	0,57012105	0,00057012	0,58639097	0,00057147	0,57146597	0,00057147
10	45	0,46928596	0,00046929	0,48878598	0,00048879	0,47002196	0,00047002

Figura 6 - Resultado dos testes do Ubuntu em máquina virtual

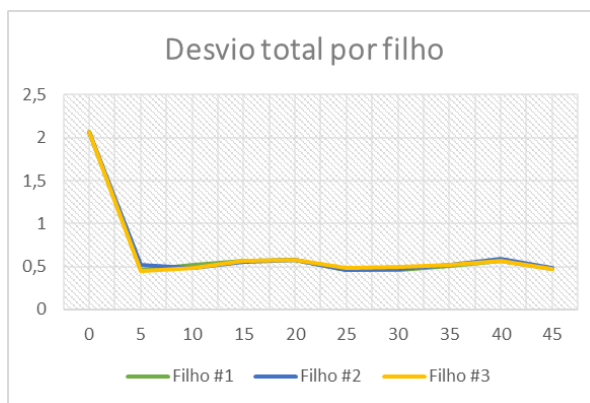


Figura 7 - Gráfico dos desvios totais

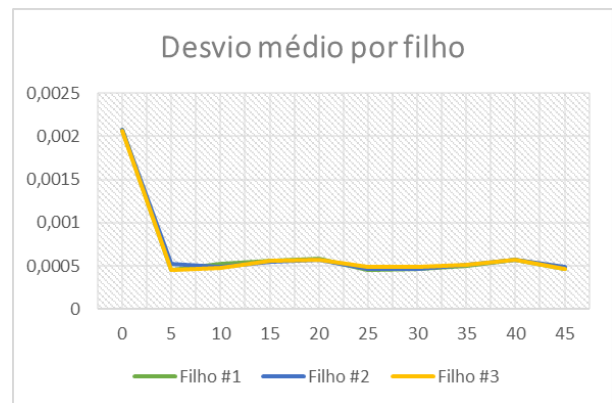


Figura 8 - Gráfico dos desvios médios

## Deepin:

Sistema							
Deepin 15.11 (Dual Boot) com i7 7500U 2.7GHz							
Rodada	Nº de Cargas	Filho #1		Filho #2		Filho #3	
		Desvio Total	Desvio Médio	Desvio Total	Desvio Médio	Desvio Total	Desvio Médio
1	0	0,16177297	0,00016177	0,16166496	0,00016166	0,16036999	0,00016037
2	5	0,05308700	0,00005309	0,07635605	0,00007636	0,05320203	0,00005320
3	10	0,05274701	0,00005275	0,07512403	0,00007512	0,06965899	0,00006966
4	15	0,05418396	0,00005418	0,07302797	0,00007303	0,05390203	0,00005390
5	20	0,05388403	0,00005388	0,06499505	0,00006500	0,05362797	0,00005363
6	25	0,05421102	0,00005421	0,07911801	0,00007912	0,05519104	0,00005519
7	30	0,05416405	0,00005416	0,07828796	0,00007829	0,05427098	0,00005427
8	35	0,05284703	0,00005285	0,06905699	0,00006906	0,05298102	0,00005285
9	40	0,05285501	0,00005286	0,07480597	0,00005299	0,05298901	0,00005299
10	45	0,05356205	0,00005356	0,06931400	0,00006931	0,05368400	0,00005356

Figura 9 - Resultado dos testes do Deepin em dual boot

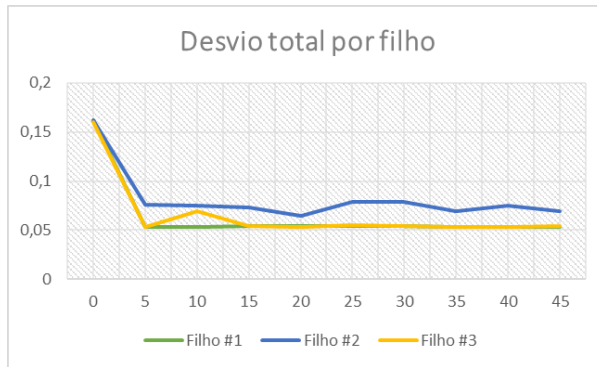


Figura 10 - Gráfico dos desvios totais

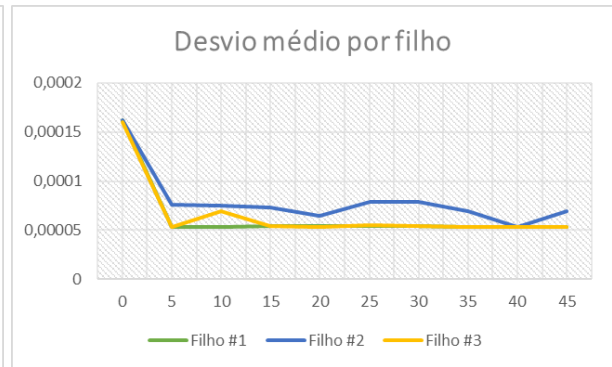


Figura 11 - Gráfico dos desvios médios

Sistema							
Deepin 15.11 (Máquina Virtual) com CPU Core i7 4790 4.0GHz							
Rodada	Nº de Cargas	Filho #1		Filho #2		Filho #3	
		Desvio Total	Desvio Médio	Desvio Total	Desvio Médio	Desvio Total	Desvio Médio
1	0	1,06222597*	0,00106222*	1,05599036*	0,00275206*	1,05690201*	0,00105690*
2	5	0,37018704	0,00037019	0,37919402	0,00037919	0,37882400	0,00037882
3	10	0,44217205	0,00044217	0,44098592	0,00044099	0,43538499	0,00043538
4	15	0,43150401	0,00043150	0,46655607	0,00046656	0,43698502	0,00043699
5	20	0,49202800	0,00049203	0,45608807	0,00045609	0,45700002	0,00045700
6	25	0,35119605	0,00035120	0,36763501	0,00036764	0,34967995	0,00034968
7	30	0,44263196	0,00044263	0,44610095	0,00044610	0,44362199	0,00044362
8	35	0,44130402	0,00044713	0,47332001	0,00047332	0,44680095	0,00044680
9	40	0,43110502	0,00043111	0,43623400	0,00043623	0,43648899	0,00043649
10	45	0,47251499	0,00047251	0,47692704	0,00047693	0,47247899	0,00047248

Figura 12 - Resultado dos testes do Deepin em máquina virtual

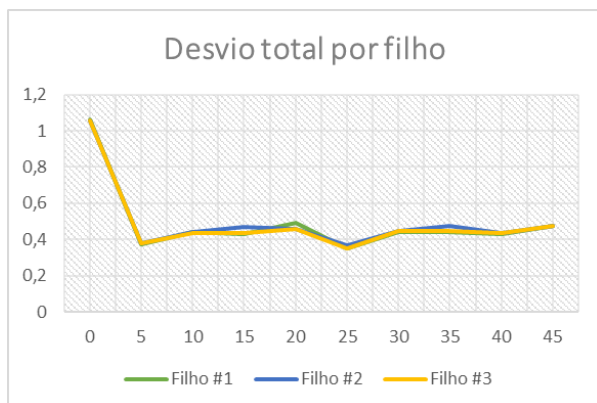


Figura 12 - Gráfico dos desvios totais

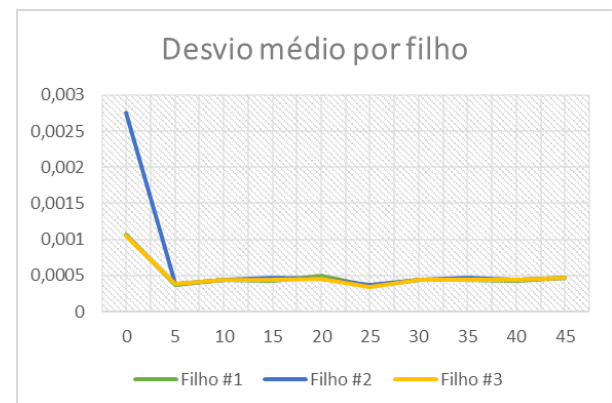


Figura 13 - Gráfico dos desvios médios

## 5.2. REFERENTES AO PROGRAMA MODIFICADO

Os testes do programa modificado foram executados apenas na distro Deepin 15.11, em máquina virtual e em dual boot, em máquinas distintas. Na análise dos resultados, pontuaremos algo curioso.

```

erick@erick-PC: ~/Desktop
erick@erick-PC:~/Desktop$ gcc exp.c -o experimento1_mod
exp.c: In function 'main':
exp.c:78:3: warning: implicit declaration of function 'exit' [-Wimplicit-function-declaration]
    exit(0);
    ^~~~
exp.c:78:3: warning: incompatible implicit declaration of built-in function 'exit'
exp.c:78:3: note: include '<stdlib.h>' or provide a declaration of 'exit'
erick@erick-PC:~/Desktop$ gcc filho.c -o filho
filho.c: In function 'main':
filho.c:25:17: warning: implicit declaration of function 'atoi' [-Wimplicit-function-declaration]
    int SLEEP_TIME=atoi(argv[0]);
                  ^~~~~
erick@erick-PC:~/Desktop$

```

Figura 14 - Compilação do programa modificado

Sistema											
Deepin 15.11 (Dual Boot) com CPU Core i7 7500U 2.7GHz											
Rodada	Tempo de dormência	Filho #1		Filho #2		Filho #3		Filho #4		Filho #5	
		Desvio Total	Desvio Médio	Desvio Total	Desvio Médio	Desvio Total	Desvio Médio	Desvio Total	Desvio Médio	Desvio Total	Desvio Médio
1	400	0,53226602	0,00053227	0,53223598	0,00053224	0,53221500	0,00053221	0,53202701	0,00053203	0,53211904	0,00053212
2	600	0,73225099	0,00073225	0,73223400	0,00073223	0,73221397	0,00073221	0,73269701	0,00073270	0,73269701	0,00073270
3	800	0,93378800	0,00093379	0,93395698	0,00093396	0,93374300	0,00093374	0,93353200	0,00093353	0,93353200	0,00093353
4	1000	0,16060901	0,00016061	0,16041899	0,00016042	0,16056800	0,00016057	0,16130602	0,00016131	0,16037107	0,00016037
5	1200	0,36559701	0,00036560	0,36557901	0,00036558	0,36556196	0,00036556	0,36541498	0,00036541	0,36558104	0,00036558
6	1400	0,56776702	0,00056777	0,56755197	0,00056755	0,56753397	0,00056753	0,56737900	0,00056738	0,56745899	0,00056746
7	1600	0,76993895	0,00076994	0,76994205	0,00076994	0,76993799	0,00076994	0,76975000	0,00076975	0,77006602	0,00077007
8	1800	0,96759903	0,00096760	0,96758795	0,00096759	0,96749794	0,00096750	0,96781301	0,00096781	0,96743906	0,00096744
9	2000	0,14822197	0,00014822	0,14811492	0,00014811	0,14809990	0,00014810	0,14795303	0,00014795	0,14802790	0,00014803
10	2200	0,34437108	0,00034437	0,34433603	0,00034434	0,34432006	0,00034432	0,34430695	0,00034431	0,34423995	0,00034424

Figura 15 - Resultado dos testes do Deepin em dual boot (zoom para visualizar)

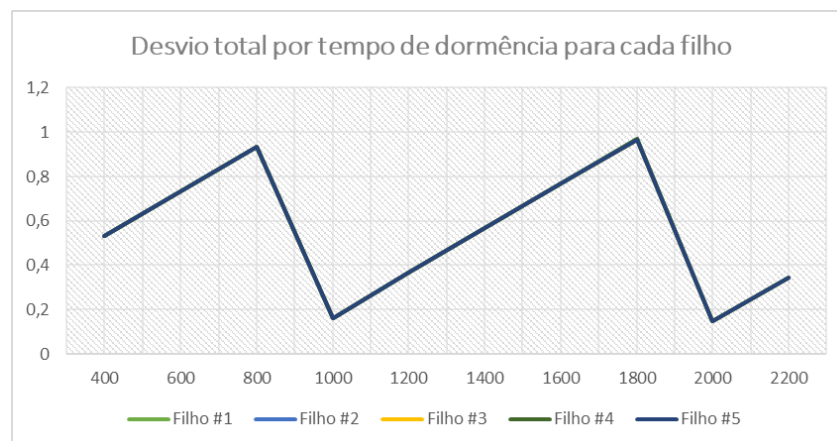


Figura 16 - Gráfico de desvio total dos resultados da tabela anterior



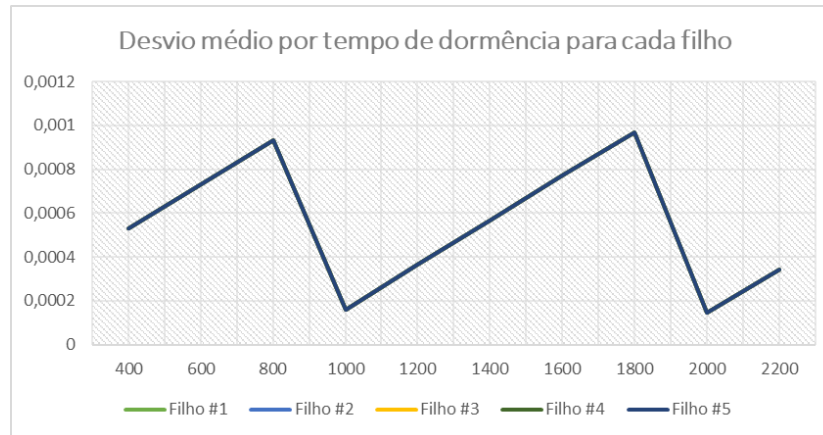


Figura 17 - Gráfico de desvio médio dos resultados da tabela anterior

Sistema											
Deepin 15.11 (Máquina Virtual) com CPU Core i7 4790 4.0GHz											
Rodada	Tempo de dormência	Filho #1		Filho #2		Filho #3		Filho #4		Filho #5	
		Desvio Total	Desvio Médio	Desvio Total	Desvio Médio	Desvio Total	Desvio Médio	Desvio Total	Desvio Médio	Desvio Total	Desvio Médio
1	400	0,98674399	0,00098674	0,98748201	0,00098748	0,98838902	0,00098839	0,98943800	0,00098944	0,98609900	0,00098610
2	600	1,04123604	0,00104124	1,05106199	0,00105106	1,05332303	0,00105332	1,04575801	0,00104576	1,04960299	0,00104960
3	800	1,07940900	0,00107941	1,09222996	0,00109223	1,09042704	0,00109043	1,09275305	0,00109275	1,09172797	0,00109173
4	1000	0,63208294	0,00063208	0,62322092	0,00062322	0,63627100	0,00063627	0,62286305	0,00062286	0,62468896	0,00062469
5	1200	0,95797300	0,00095797	0,96397901	0,00096398	0,96179605	0,00096180	0,96358395	0,00096358	0,96395802	0,00096396
6	1400	1,09002209	0,00109002	1,07982588	0,00107983	1,08203697	0,00108204	1,07144403	0,00107144	1,08979511	0,00108980
7	1600	1,11882114	0,00111882	1,10995102	0,00110995	1,10311294	0,00110311	1,10178995	0,00110179	1,08292294	0,00108292
8	1800	1,27794790	0,00127795	1,26893306	0,00126893	1,28059196	0,00128059	1,27338696	0,00127339	1,26555204	0,00126555
9	2000	0,79259300	0,00079259	0,80524898	0,00080525	0,80201387	0,00080201	0,80468106	0,00080468	0,80466604	0,00080467
10	2200	1,09566903	0,00109567	1,09563589	0,00109564	1,08969498	0,00108970	1,09227910	0,00108923	1,08922791	0,00108923

Figura 18 - Resultado dos testes do Deepin em máquina virtual (zoom para visualizar)

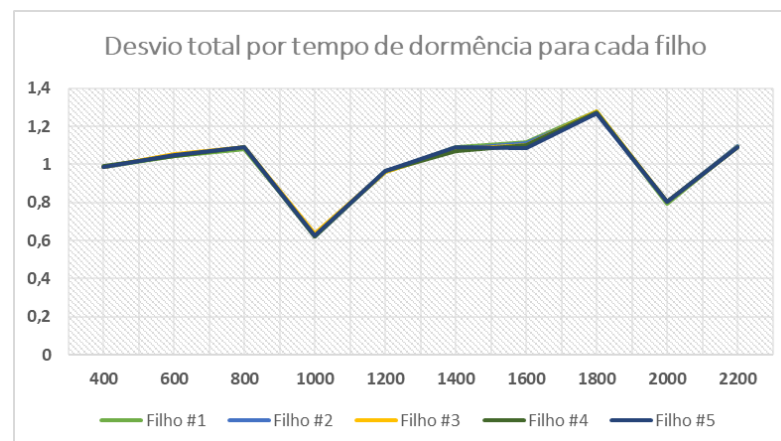


Figura 19 - Gráfico de desvio total dos resultados da tabela anterior

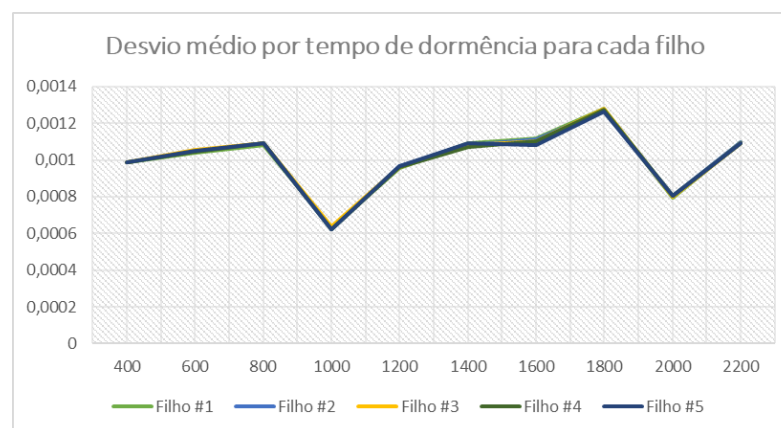
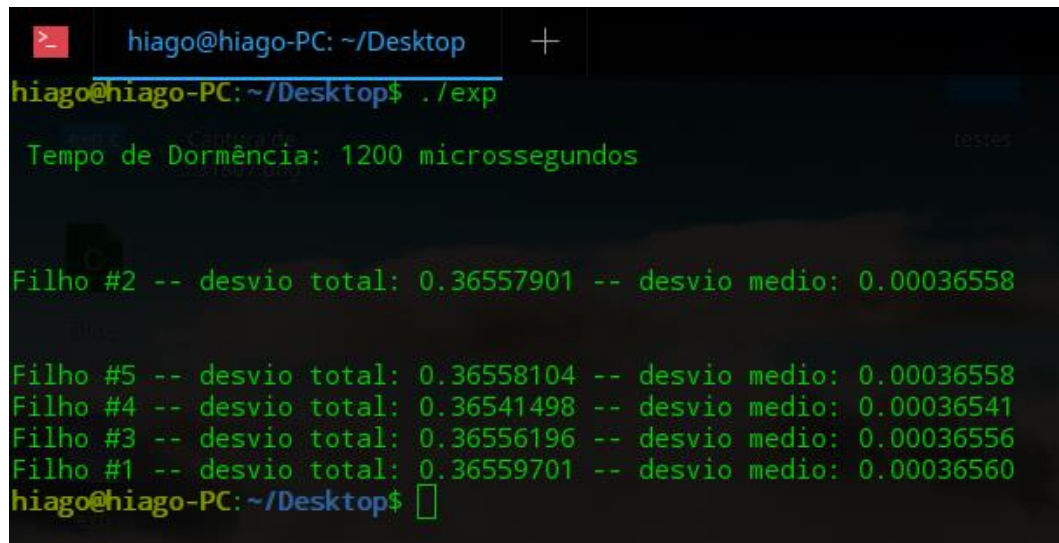


Figura 20 - Gráfico de desvio médio dos resultados da tabela





```

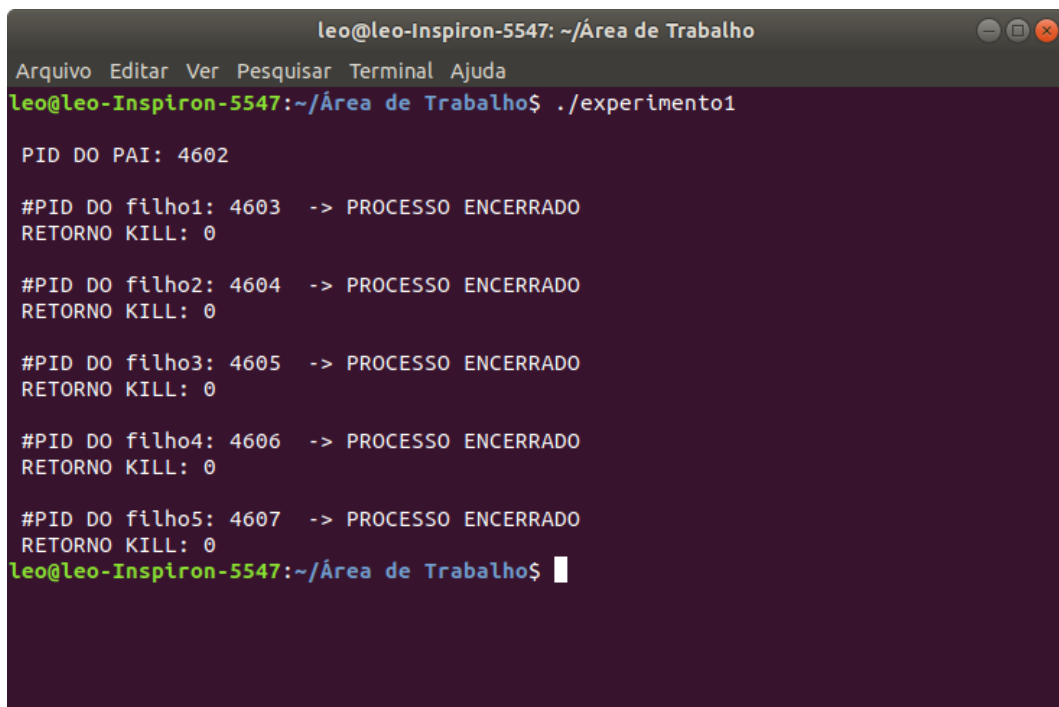
hiago@hiago-PC: ~/Desktop
hiago@hiago-PC:~/Desktop$ ./exp
Tempo de Dormência: 1200 microssegundos

Filho #2 -- desvio total: 0.36557901 -- desvio medio: 0.00036558

Filho #5 -- desvio total: 0.36558104 -- desvio medio: 0.00036558
Filho #4 -- desvio total: 0.36541498 -- desvio medio: 0.00036541
Filho #3 -- desvio total: 0.36556196 -- desvio medio: 0.00036556
Filho #1 -- desvio total: 0.36559701 -- desvio medio: 0.00036560
hiago@hiago-PC:~/Desktop$

```

Figura 21 - Teste de dormência de 1200 microssegundos em dual boot (notebook)



```

leo@leo-Inspiron-5547: ~/Área de Trabalho
Arquivo Editar Ver Pesquisar Terminal Ajuda
leo@leo-Inspiron-5547:~/Área de Trabalho$ ./experimento1

PID DO PAI: 4602

#PID DO filho1: 4603 -> PROCESSO ENCERRADO
RETORNO KILL: 0

#PID DO filho2: 4604 -> PROCESSO ENCERRADO
RETORNO KILL: 0

#PID DO filho3: 4605 -> PROCESSO ENCERRADO
RETORNO KILL: 0

#PID DO filho4: 4606 -> PROCESSO ENCERRADO
RETORNO KILL: 0

#PID DO filho5: 4607 -> PROCESSO ENCERRADO
RETORNO KILL: 0
leo@leo-Inspiron-5547:~/Área de Trabalho$

```

Figura 22 - Aplicando kill() nos processos filhos por meio do pai printando o retorno

Para que isso funcionasse, o fizemos por meio da função “kill()”, passando por parâmetro o PID do processo filho e o sinal “SIGKILL”, causando o término imediato da execução do processo.

## 6. ANÁLISE DOS RESULTADOS

Os gráficos e tabelas de ambas as fases do experimento nos revelam detalhes de como o gerenciamento de processos do SO e o uso do multiprocessamento

impactam na performance do programa, bem como no desvio ocasionado pelas chamadas de sistema. Os testes em máquina virtual e em dual boot nos mostraram como um SO sendo simulado dentro de outro pode ter sua performance deliberadamente afetada, isso porque temos um primeiro sistema “chefe” fazendo seu próprio escalonamento e gerenciamento de processos e, ao mesmo tempo, um segundo sistema rodando como por intermédio do primeiro e que também possui seu próprio modo de gerenciar arquivos, processos, memória e etc.

### 6.1. PROGRAMA EXEMPLO

A curva dos processos filhos no gráfico do programa-exemplo mostra uma espécie de “efeito parábola”, onde o contraste entre a execução deste sem carga e com carga mostra que SO quando há poucos processos em execução faz o processador ficar em estado ocioso a maior parte do tempo.

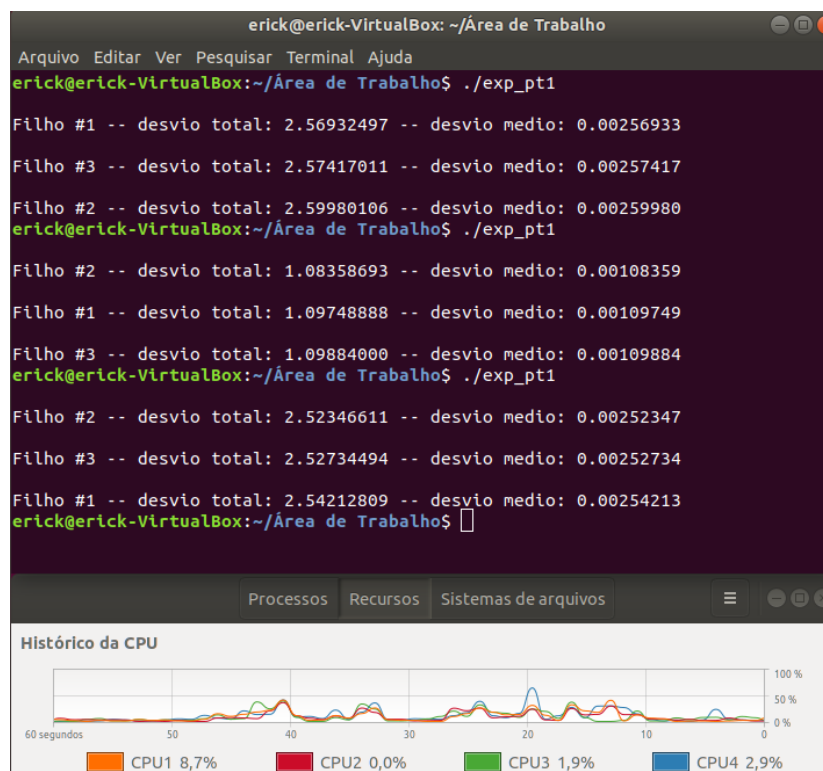


Figura 23 - Programa exemplo sem carga rodando em VM Ubuntu com CPU i7 4790

Na imagem, um exemplo da grande variação quando estamos com ambos os sistemas ociosos (o sistema “pai” e o que está rodando em máquina virtual). Ao que tudo indica, o processador não consegue ser bem aproveitado pelos SOs, tornando o resultado extremamente variável.

Esse comportamento em máquina virtual se torna ainda mais interessante quando comparamos com o teste feito no mesmo sistema e ambiente (Ubuntu c/ i7 4790) rodando o SO nativamente. No primeiro teste já encontramos a primeira diferença: não há uma variação tão grande de resultado a cada vez que se executa o programa, e o segundo se evidencia nas casas decimais do tempo de execução e desvio medidos, quando o sistema rodando nativamente executa a mesma tarefa quase *dez vezes* mais rápido. Os testes mostraram que o programa exemplo só fica estável quando sobrecarregamos a CPU, colocando em funcionamento o escalonamento massivo de processos e o multiprocessamento.

Já entre distribuições diferentes (Deepin e Ubuntu em máquina virtual), é observável pelos gráficos e tabelas exatamente o mesmo fenômeno de variação no teste sem carga e maior estabilidade ao testarmos utilizando o código de carga. A similaridade de resultados que os dois SOs apresentaram é outro destaque, e isso pode ter uma explicação: partem da mesma base, o Debian, o que potencialmente indica que ambos utilizam do mesmo sistema de gerenciamento e escalonamento de processos, o que faria com que apenas detalhes menores impactassem na performance do teste, como a interface gráfica e outros recursos próprios de cada um.

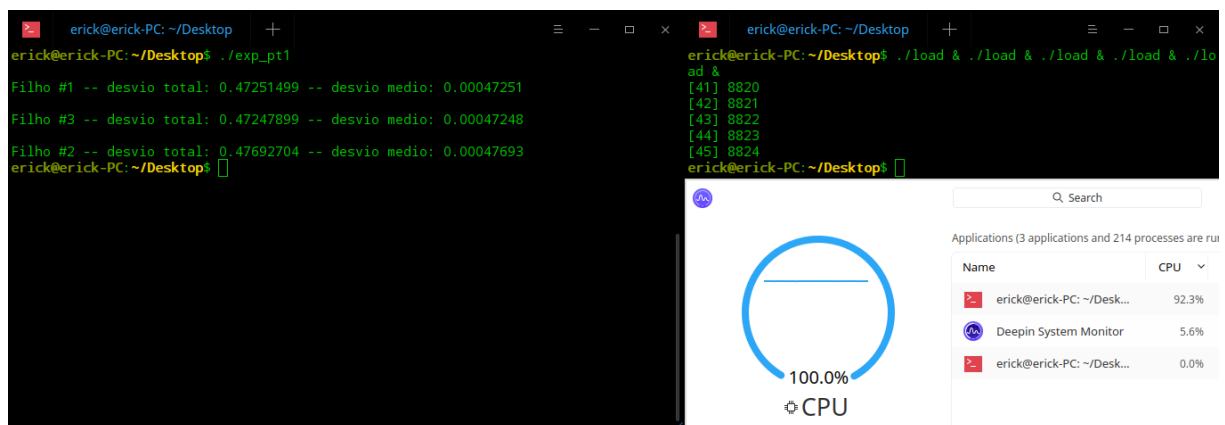


Figura 24 - Programa exemplo com 45 cargas rodando em VM Deepin com CPU i7 4790

A figura acima nos mostra o principal foco do experimento: o consumo de CPU, que quando trabalha incessantemente diminui o tempo de desvio do programa exemplo devido cada processo filho liberar rapidamente a CPU entrando em estado de dormência, fazendo com que esse revezamento seja a principal explicação para a similaridade de resultados que os processos filhos compartilham entre si. Quando temos um tempo maior em alguma das rodadas, a exemplo, a rodada 3 do teste feito em dual boot Deepin, que mostra os três filhos com tempos muito diferentes entre si

(facilmente percebido no gráfico), tornando possível afirmar que esta diferença muito provavelmente foi exercida por alguma atividade não prevista do SO, explicando também o porque o resultado varia tanto em máquinas virtuais, já que isso deve ocorrer a todo instante, fazendo a CPU ser concorrida entre ainda mais processos que surgem de repente.

## 6.2. PROGRAMA MODIFICADO

Os resultados nos revelam dados completamente contrários à nossa expectativa: estávamos crentes de que quanto maior fosse nosso tempo de dormência, maior seria o tempo obtido na execução, o que não foi verdade. Conforme observado na tabela, testes com valores maiores de sleep apresentaram um tempo consideravelmente menor, a exemplo o teste em máquina virtual Deepin com 2000 microssegundos de atraso.

As razões por trás disso podem ser inúmeras, desde a execução em threads distintos e menos concorridos, bem como o “cruzamento” com o timer de processos do SO com a saída ou entrada dos processos filhos na CPU.

Um detalhe super válido de se notar é a diferença esmagadora que há entre os resultados do SO rodando em máquina virtual em comparação a rodando nativamente, principalmente quando consideramos as diferenças entre as máquinas, a rodando nativamente com um i7 dual core com menor memória cache e a outra, rodando em máquina virtual, com um i7 quad core.

Pesquisando em fóruns de hardware e vendo relatos de pessoas rodando experimentos parecidos, percebemos que a concorrência gerada pela VM e a escassez de recursos físicos para permitirem o pleno funcionamento da máquina virtual, acarretam em escolhas diferentes quando seus recursos são esgotados, como o comportamento de processamento e distribuição de carga efetuado bem mais cedo na VM do que num processador rodando de forma nativa com recursos físicos mesmo.

A possível explicação, além da que já foi explicada sobre a máquina virtual no início da análise de resultados da parte 1, remonta o que foi dito sobre a concorrência entre threads, em um processador com mais núcleos, podemos ter filhos sendo executados em um núcleo menos concorrido em detrimento de outro a todo instante sendo retirado da CPU concorrendo com vários processos em outro núcleo.

Na parte 1 se pode sentir o peso das cargas, quando montada a estrutura de distribuição na parte 2 com o programa modificado acreditávamos que esse tempo ia ser muito maior, vemos claramente a disparidade de performance do processador quando comparado com os dados da parte 2, a variância presente na lida com as cargas, assim, se pode extrair de aprendizado que a covariância de processamento de cargas para cada filho é influenciado diretamente pelo término de processamento de processos paralelos distribuídos de forma horizontal nos processos, bem como sua execução depende muito da distribuição dos processos por parte do processador e como ele vai delegar a execução deste nó quando possível, então, geralmente na média obtemos um tempo menor de execução, mas, isso só acontece porque temos a disposição lugares livres ou com possibilidade de rodar de forma “paralela”, uma vez que os hardwares experimentados aqui não permitem um escalamento vertical para melhor aplicabilidade dos testes em processadores de maior porte, não conseguimos demonstrar de maneira eficaz os comportamentos dos programas quando colocados em máquinas maiores e com maior disponibilidade de recursos e maior desempenho de cache por exemplo.

## 7. CONCLUSÃO

Através de análises do primeiro experimento, foi possível identificar diversos comportamentos da CPU durante as fases do experimento. No primeiro requerimento do trabalho que solicita executar o programa “experimento.c” sem carga, podemos identificar que processos simples não fazem muito uso da CPU, e por isso o processador é usado com baixo ciclos de clock a ponto de economizar energia a manter a execução do programa otimizada.

Em seguida, ao colocarmos cargas durante a execução do programa pudemos ver um maior uso do processador. Dessa forma, exigindo mais da CPU e aumentando o clock do processador, para atender todos os processos sem um desvio de tempo muito grande. Também, a visibilidade sobre o revezamento de processos se tornou maior graças ao cálculo dos desvios, pois possibilitou saber a variação de tempo que CPU gasta para atender outro processo enquanto que um está em fase de execução e depois atender este quando estiver terminado.

O experimento permitiu que percebêssemos a influência da gerência do SO no processamento e a atuação, em especial na segunda parte do experimento, do

chamado “clock tick” do kernel do Linux – que pode ser visualizado através do comando `sysconf(_SC_CLK_TCK)`; – que estabelece um tempo mínimo entre a saída de um programa para a entrada de outro no escalonamento de CPU.

Por fim, o experimento permitiu maior compreensão da interação do SO com a CPU bem como dos processos com os dois, tornando possível também o conhecimento de funções disponíveis para a manipulação de processos na linguagem C em distribuições de sistemas baseados em Unix.

## 8. BIBLIOGRAFIA

RAO, DHANANJAI. “**FORK AND EXEC IN LINUX**”. YouTube. Disponível em <<https://youtu.be/nwm7rJG90i8/>>. Acesso em 12/02/2020.

LINDER, MARCELO. “**OS ARGUMENTOS ARGV E ARGV**”. Univasf. Disponível em <[http://www.univasf.edu.br/~marcelo.linder/arquivos\\_pc/aulas/aula19.pdf/](http://www.univasf.edu.br/~marcelo.linder/arquivos_pc/aulas/aula19.pdf/)>. Acesso em 14/02/2020.