PONTIFÍCIA UNIVERSIDADE CATÓLICA DE CAMPINAS

CENTRO DE CIÊNCIAS EXATAS, AMBIENTAIS E DE TECNOLOGIA

CHRISTOPHER OLIVEIRA	**RA:18726430**
GIULIANO SANFINS	**RA:17142837**
MATHEUS MORETTI	**RA:18082974**
MURILO ARAUJO	**RA:17747775**
VICTOR REIS	**RA:18726471**

SISTEMAS OPERACIONAIS A - EXPERIMENTO 1

CAMPINAS
2020

SUMÁRIO

1. INTRODUÇÃO

O experimento tem como objetivo aprender e se familiarizar com ambiente Linux, como usar o compilador GCC, corrigindo o código que foi entregue tanto logicamente, quanto sintaticamente e o executando. Tendo seu foco na compreensão do funcionamento e criação de processos pai e filhos e o que sua existência afetam o desempenho do programa, através de uma análise temporal de um trecho de código especifico, além de ver quais os fatores que podem interferir na sua medição, usando funções como *fork()*, *exec(), wait()* e *exit().*

Após o entendimento do conteúdo que envolve de forma direta o experimento e de modificar o código, foi necessário realizar uma série de testes em relação ao tempo de execução da função *usleep()*. Precisando registrar e documentar todos os resultados e analisa-los de forma coerente.

2. DISCUSSÃO

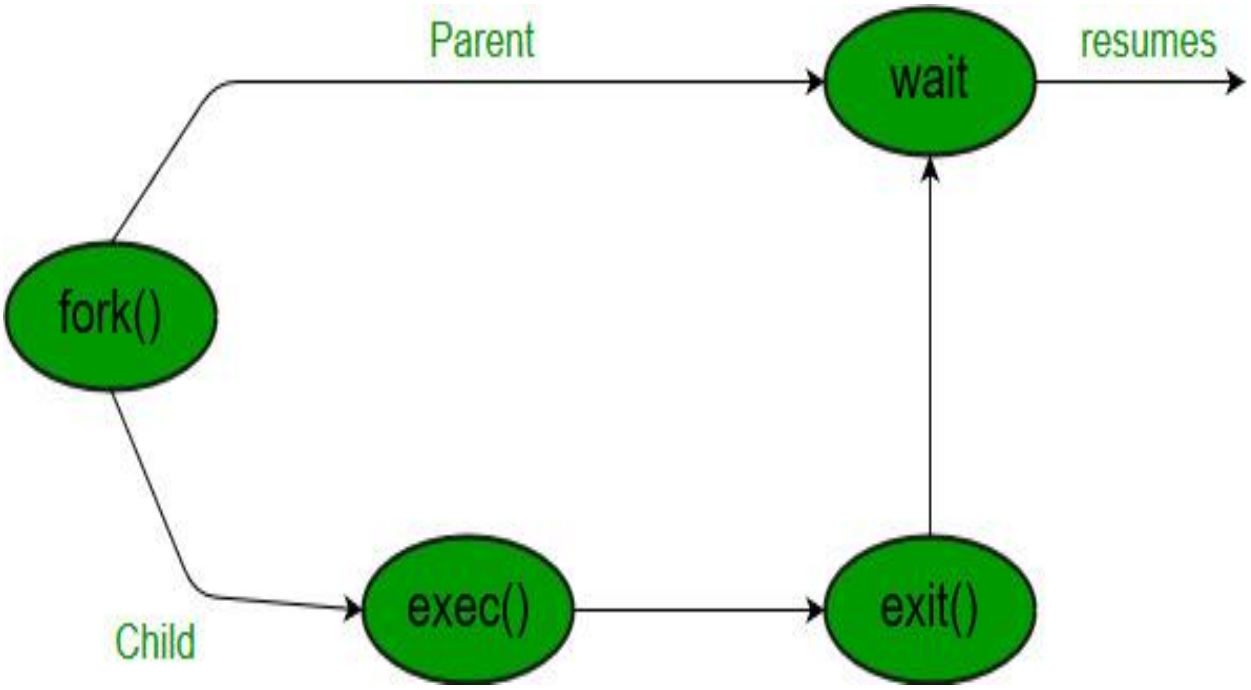

Figura 1: Explicação da função ***fork()*** (*Fonte: GeeksForGeeks)*

A função ***fork()*** cria um processo filho sendo uma cópia idêntica do processo pai. Após o processo de criação do filho é completado o programa rodará normalmente na linha de código depois do ***fork().***

Fork pode receber 3 retornos:

- Valor negativo: processo de criação do filho não foi bem-sucedido.
- Zero: retorna para o processo filho mais novo criado.
- Valor positivo: retorna para o pai o valor do PID do filho criado.

A função ***exec()*** faz com que o processo filho criado, ou qualquer processo em que a função é chamada, não precise mais rodar o programa no qual estava vinculado, ele permite que um processo execute qualquer outro programa de escolha do programador, o espaço antes reservado para um determinado programa é substituído pelo conteúdo do programa escolhido.

A função ***exit()*** termina o programa e todo seu contexto é finalizado.

A função ***wait()*** faz com que o processo pai espere o termino de seus processos filhos, ela devolverá o status de retorno de qualquer processo filho que seja finalizado, o pai apresentará um dos três seguintes comportamentos:

-Bloquear sua execução até que algum processo filho termine.

-Retornar imediatamente caso haja algum erro.

-Retornar imediatamente com o status de término de um processo filho caso o filho tenha já terminado (zumbi). Quando um filho termina, o sistema operacional enviar um sinal para o pai.

O programa abaixo tem por objetivo pré-definido agir como carga para aumentar o consumo do processador. Foi utilizado na primeira parte do exercício, onde era necessário colocar até 45 cargas simultâneas, fazendo com que elas competissem entre si pelo tempo de CPU.

```
#include <stdio.h>

int main()
```

```
{
long long int count=0;

while(1){
count++;
}

return 0;
}
```

3. SOLUÇÃO DE ERROS DE SINTAXE

3.1 Ausência da declaração da variável rtn

Int rtn;

3.2 Utilização de +- dentro uma repetição (FOR)

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

3.3 Inversão de um operador lógico no FOR do Pai.

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

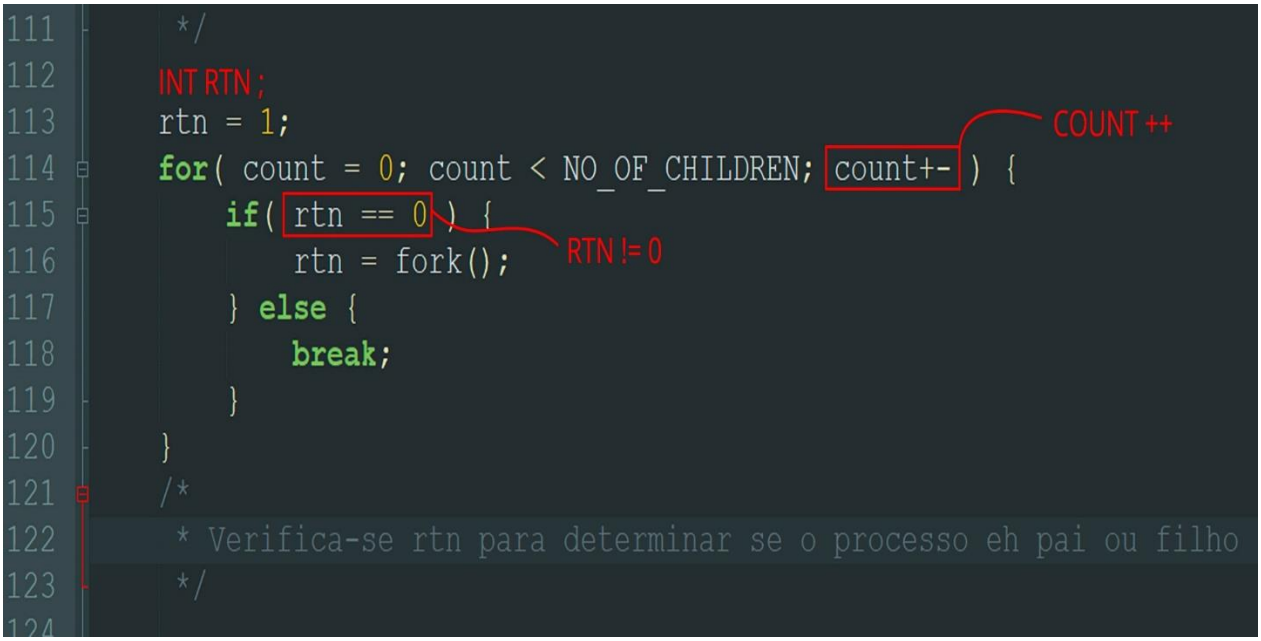

Figura 2: Primeiras alterações do código

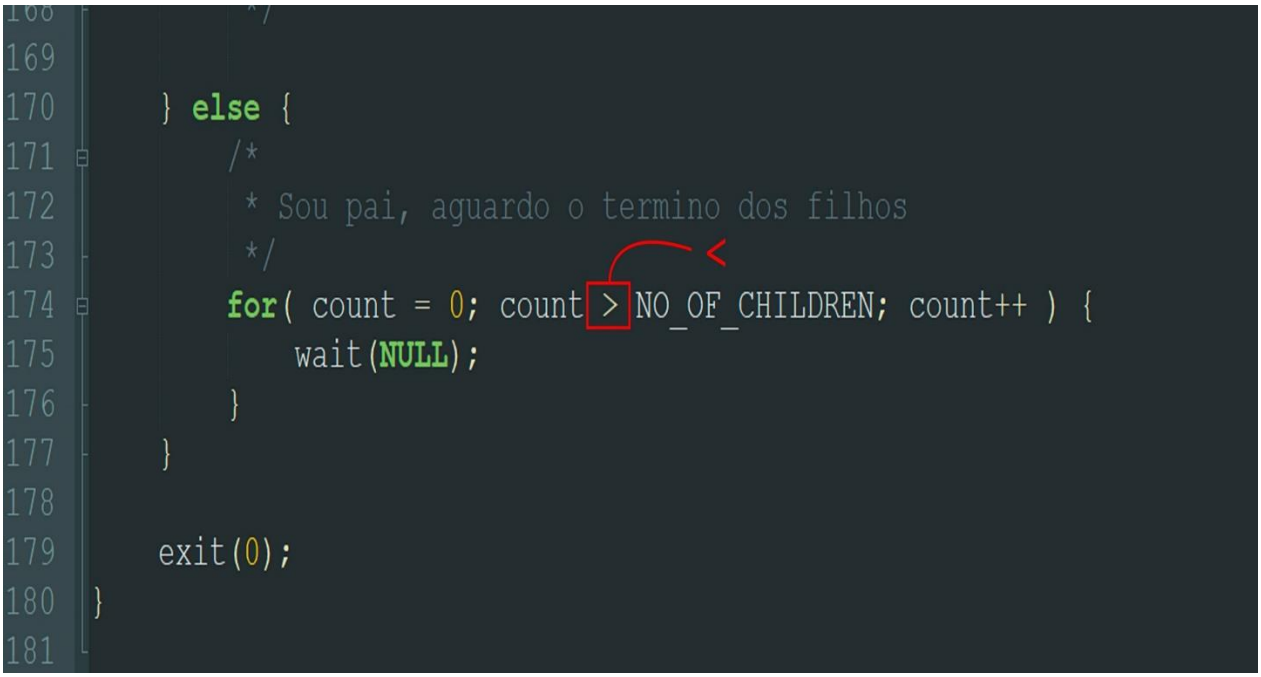

Figura 3: Última alteração do código

4 PERGUNTAS

4.1 Pergunta 1

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

R: Os arquivos fonte contém diretivas do tipo *#include* e *#define* que são processadas pelo pré-processador ***cpp***, que faz a expansão dos macros, a inclusão dos arquivos com cabeçalhos, e remove os comentários. A saída do ***cpp*** é entregue ao compilador ***gcc***, que faz a tradução de C para linguagem de montagem.

4.2 Pergunta 2

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

R: *stdio.h* é um cabeçalho da biblioteca padrão do C, "cabeçalho padrão de entrada/saída". Possui definições de sub-rotinas relativas às operações de entrada/saída, como leitura de dados digitados no teclado e exibição de informações na tela do programa de computador. Também possui numerosas definições de constantes, variáveis e tipos.

É um dos cabeçalhos mais populares da linguagem de programação C, intensivamente utilizado tanto por programadores iniciantes como por experientes.

Abaixo temos 4 funções desta biblioteca que são muito utilizadas:

-*printf()* Função usada para imprimir dados na tela.

-*scanf()* Função usada para capturar dados do usuário.

-*fprintf()* Função usada para imprimir dados em arquivo.

-*fscanf()* Função usada para ler dados de arquivos.

4.3 Pergunta 3

Qual é a função da diretiva include (linha que começa com #), com relação ao compilador?

R: O que o compilador faz é simplificadamente o mesmo que você copiar e colar o texto que está dentro do include para dentro do texto que está na fonte principal. Ele está importando todo seu conteúdo. Normalmente esses conteúdos são apenas 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 na fonte ao invés de ser chamada (*source inline*). O mais comum é ele não ter código. Normalmente ele é usado para declarar as estruturas de dados, mas não definir os algoritmos.

4.4 Pergunta 4

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

R: ***argc*** – é um valor inteiro que indica a quantidade de argumentos que foram passados ao chamar o programa.

argv – é um vetor de char que contém os argumentos, um para cada *string* passada na linha de comando.

4.5 Pergunta 5

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

R: O desvio é um método para medir a dispersão e analisar o quanto os valores de um conjunto se afastam de uma regularidade. E verificar o quanto os elementos de

um conjunto respeitam um padrão. Em nosso experimento é analisado a partir do tempo de execução, ou seja, o *sleep* está diretamente ligado a este valor, aumentando este ou diminuindo afeta proporcionalmente o desvio, em um caso isolado, porém ele não é o único fator implicante no desvio, há fatores externos na própria máquina, como outros processos utilizando do processador, causando ainda mais sobrecarga afetando assim ainda mais o valor do desvio no final, portanto pode se concluir que o *sleep* é apenas um fato diretamente ligado ao desvio, mas não determinante.

4.6 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).

R: GCC exemplo.c -o experimento1.

Após a execução do programa sem nenhum erro, foi executado o programa 10 vezes, variando a carga em cinco em cinco, começando inicialmente com nenhuma carga e finalizando a análise da execução com 45 cargas. Abaixo pode ser observado uma tabela demonstrativa que indica a variação das cargas e o valor do desvio total e médio em segundos de cada filho do processo.

	Filho 1		Filho 2		Filho 3	
Carga	**Total**	**Médio**	**Total**	**Médio**	**Total**	**Médio**
0	0,12931	0,00012931	0,12923	0,00012923	0,12915	0,00012915
5	0,05359	0,00005359	0,05687	0,00005687	0,05601	0,00005601
10	0,05796	0,00005796	0,05849	0,00005849	0,06073	0,00006073
15	0,05949	0,00005949	0,0526	0,0000526	0,05281	0,00005281
20	0,05268	0,00005268	0,05914	0,00005914	0,05281	0,00005281
25	0,05704	0,00005704	0,05636	0,00005636	0,05341	0,00005341
30	0,058	0,000058	0,05281	0,00005281	0,05281	0,00005281
35	0,05537	0,00005537	0,05261	0,00005261	0,05393	0,00005393
40	0,05738	0,00005738	0,05378	0,00005378	0,0528	0,0000528
45	0,05395	0,00005395	0,05371	0,00005371	0,0528	0,0000528

Tabela 1: Desvio total e médio de cada filho em consequência da variação das cargas em cinco em cinco

Depois da execução do programa e analisar os resultados obtidos dessa variação, foi realizado a confecção de dois gráficos para estar sendo analisado melhor a variação dos três filhos, pode ser observado abaixo esses dois gráficos de dispersão.

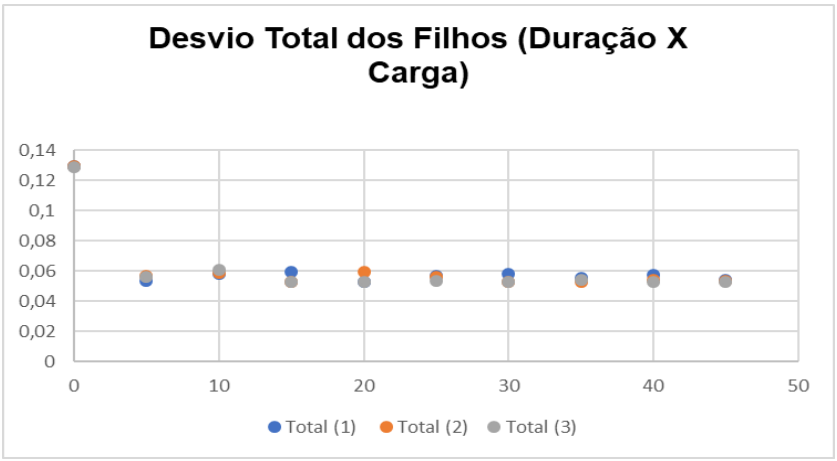

Gráfico 1 de dispersão: Duração X Carga do desvio total dos 3 filhos

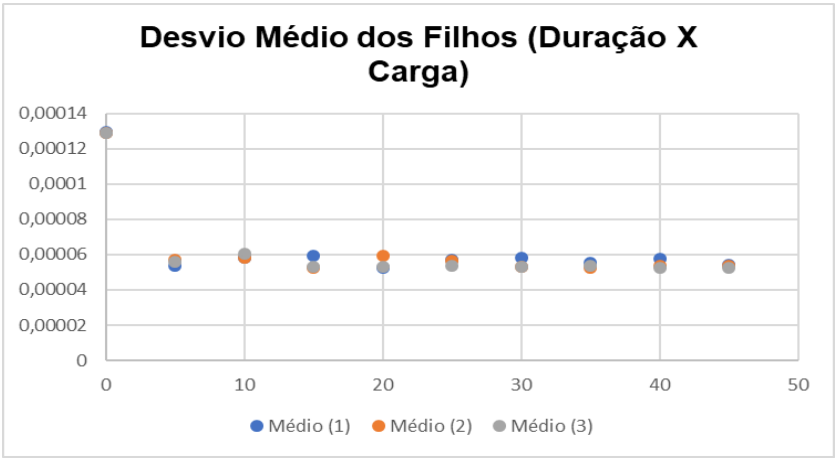

Gráfico 2 de dispersão: Duração X Carga do desvio médio dos 3 filhos

4.7 Pergunta 2

Descreva o efeito da diretiva &.

R: Permite que o programa execute em segundo plano (*Background*), deixando o prompt de comando livre para outros comandos.

4.8 Pergunta 3

Qual é o motivo do uso do "./"? Explique porque a linha de comando para executar o *gcc* não requer o "*./*".

R: Usar "./" não tem a ver com o *shell script*. O Linux procura executáveis (*shell scripts* ou quaisquer outros) na variável de ambiente PATH. Então, se seu executável não estiver no PATH, o Linux não executa. O "ponto-barra" serve para indicar o caminho do executável: o ponto refere-se ao diretório atual e a barra, naturalmente, separa o diretório do nome do arquivo.

Então, a regra é: ao executar qualquer programa, se o mesmo estiver no PATH, basta usar o nome do executável. Caso contrário, é necessário indicar o caminho completo que, no caso do diretório atual, é *"./<Executável>"*.

4.9 Pergunta 4

Apresente as características da CPU do computador usado no experimento.

R: Abaixo pode-se observar as características da CPU do computador usado no experimento:

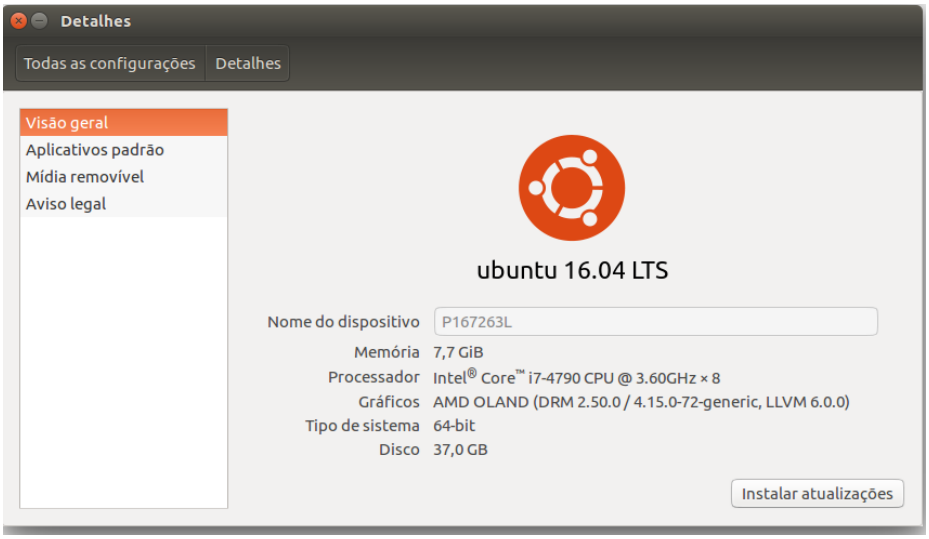

Figura 4: Característica da CPU

Abaixo pode estar sendo observado o quadro semelhante que foi elaborado para análise dos dados, extraídos da execução do programa e das confecções dos gráficos solicitados para devidas comparações.

Filho / Carga	Total (seg)	Média (seg)
1/0	0,12931	0,00012931
2/0	0,12923	0,00012923
3/0	0,12915	0,00012915
1/5	0,05359	0,00005359
2/5	0,05687	0,00005687
3/5	0,05601	0,00005601
1/10	0,05796	0,00005796
2/10	0,05849	0,00005849
3/10	0,06073	0,00006073
1/15	0,05949	0,00005949
2/15	0,0526	0,0000526
3/15	0,05281	0,00005281
1/20	0,05268	0,00005268
2/20	0,05914	0,00005914
3/20	0,05281	0,00005281
1/25	0,05704	0,00005704
2/25	0,05636	0,00005636
3/25	0,05341	0,00005341
1/30	0,058	0,000058
2/30	0,05281	0,00005281
3/30	0,05281	0,00005281
1/35	0,05537	0,00005537
2/35	0,05261	0,00005261
3/35	0,05393	0,00005393
1/40	0,05738	0,00005738
2/40	0,05378	0,00005378
3/40	0,0528	0,0000528
1/45	0,05395	0,00005395
2/45	0,05371	0,00005371
3/45	0,0528	0,0000528

Tabela 2: Análise do desvio total e médio, em relação a filho / carga

Abaixo pode ser observado 3 gráficos elaborados, para observar o comportamento de cada filho em relação ao aumento do número de cargas.

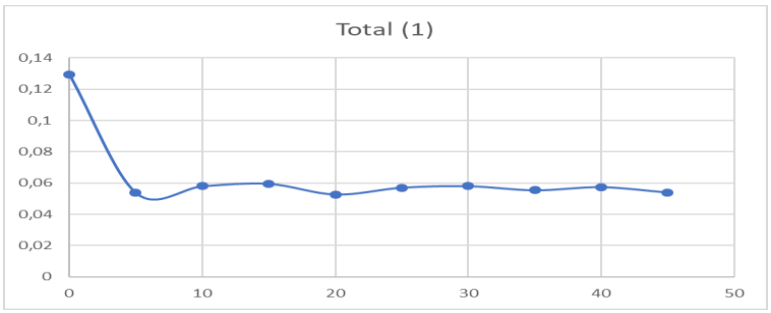

Gráfico 3: Curva Duração X Cargas do desvio total do filho 1

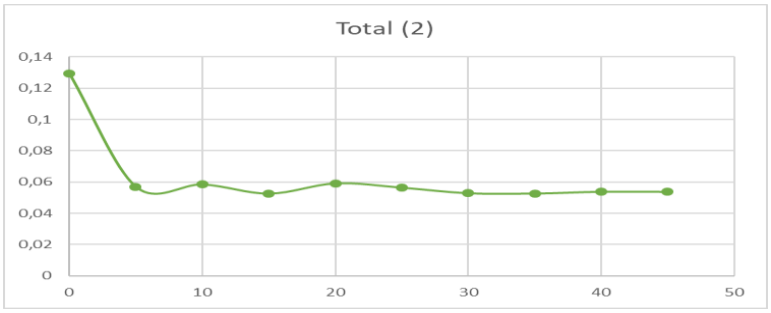

Gráfico 4: Curva Duração X Cargas do desvio total do filho 2

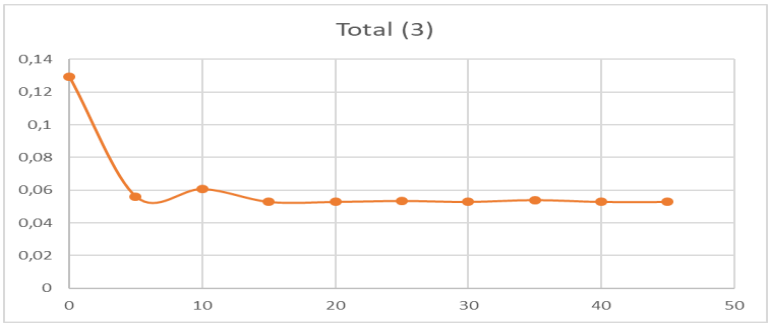

Gráfico 5: Curva Duração X Cargas do desvio total do filho 3

Depois de analisar o comportamento de cada filho isolado em relação ao aumento de cargas, foi feito uma comparação em um único gráfico com 3 curvas, onde cada curva representa o comportamento de um dos três filhos, o primeiro seria em relação ao desvio total e o segundo em relação ao desvio médio.

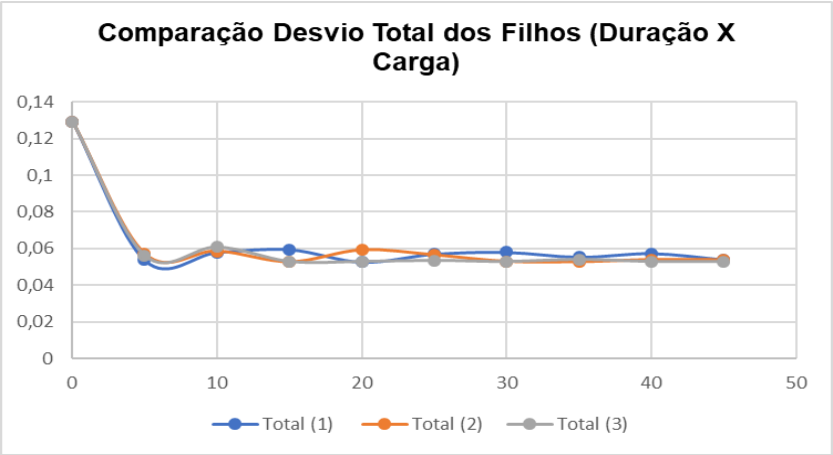

Gráfico 6: Duração X Carga do desvio total dos 3 filhos

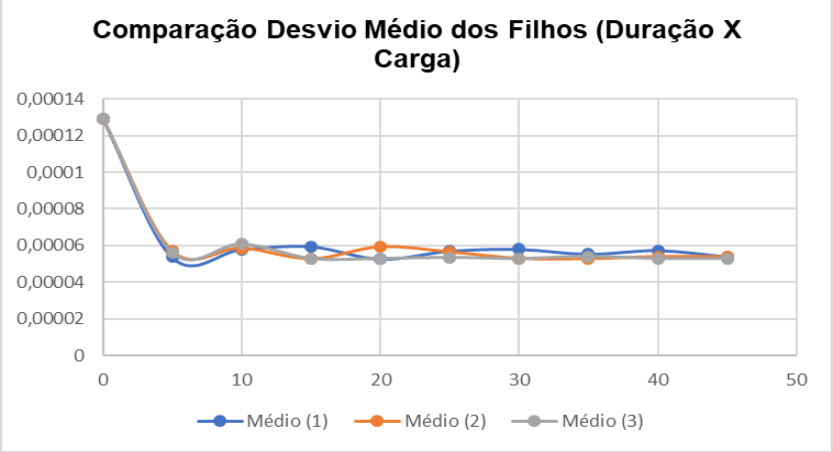

Gráfico 7: Duração X Carga do desvio médio dos 3 filhos

E por fim, foi feito um último modelo de gráfico que seria um ponto médio de cada filho que foi executado com a mesma carga, como são pontos próximos que podem ser observados na tabela acima, foi gerado um ponto médio em relação a esses dos três filhos em cada carga, podendo observar a plotagem dos pontos com legenda no gráfico, observando que o ponto que mais se afasta dos outros, é quando a carga executada nos três filhos é igual a zero.

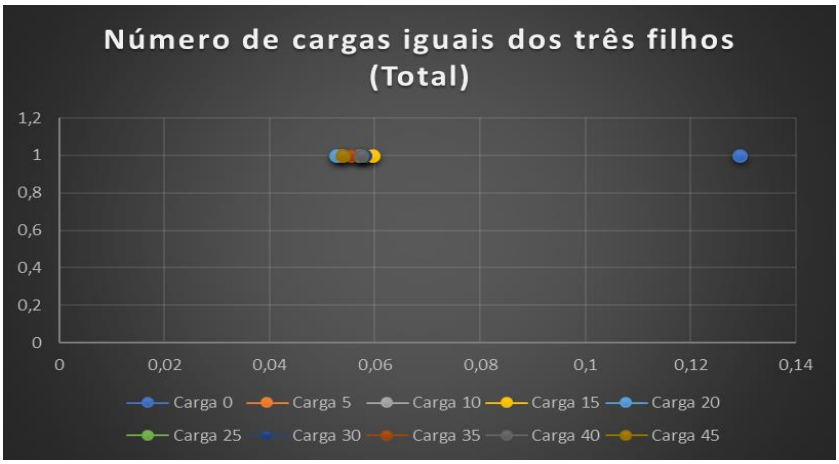

Gráfico 8: Cargas iguais dos três filhos (Cargas iguais X Desvio Total em seg)

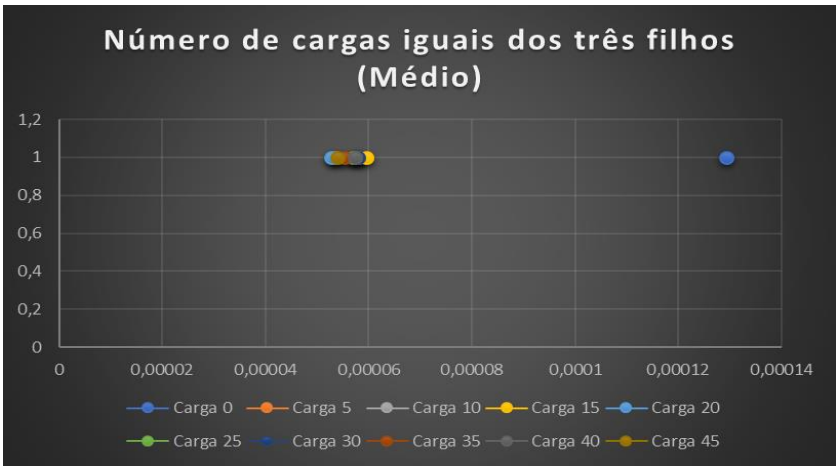

Gráfico 9: Cargas iguais dos três filhos com o desvio médio (Cargas iguais X Desvio Médio em seg)

Depois de analisar os dados obtidos, fazer as devidas comparações e confecções dos gráficos, percebemos que o tempo de resposta dos processos não é algo previsível, podendo ter diversos fatores externos no programa que podem influenciar no tempo de desvio. Após isso, foi alterado duas constantes do programa exemplo, primeiramente foi alterada a constante *"NO_OF_ITERATIONS",* de 1000, para 2000, ou seja, dobrando o seu valor, essa constante se refere ao número de vezes que vai se repetir o processo repetitivo existente em cada futuro processo filho.

A segunda e a última constante que foi alterada, foi a *"SLEEP_TIME",* de 1000, para 2000, ou seja, dobrando o seu valor também e essa constante corresponde a quantidade de tempo para ficar bloqueado. Depois de alterar essas duas constantes,

executamos o programa três vezes, alterando o número de carga de 20 em 20, como podemos observar na tabela abaixo:

	Filho 1		Filho 2		Filho 3	
Carga	**Total**	**Médio**	**Total**	**Médio**	**Total**	**Médio**
0	0,2726	0,0001363	0,27251	0,00013626	0,27244	0,00013622
20	0,10801	0,000054	0,18655	0,00009328	0,10813	0,00005407
40	0,15577	0,00007789	0,15546	0,0000773	0,15451	0,00007725

Tabela 3: Três execuções do programa, depois das alterações de duas constantes

Para analisar a diferença do programa exemplo executada com esse novo programa com as duas constantes alteradas, podemos estar observando a tabela abaixo da execução do programa com as mesmas três cargas que foram inseridas para o programa ser executado.

	Filho 1		Filho 2		Filho 3	
Carga	**Total**	**Médio**	**Total**	**Médio**	**Total**	**Médio**
0	0,12931	0,00012931	0,12923	0,00012923	0,12915	0,00012915
20	0,05268	0,00005268	0,05914	0,00005914	0,05281	0,00005281
40	0,05738	0,00005738	0,05378	0,00005378	0,0528	0,0000528

Tabela 5: Três execuções do programa, sem as alterações das duas constantes

Para realizar a comparação do programa exemplo que não tem nenhuma constante alterada com esse agora que foi alterada as duas constantes citadas, foi utilizada um gráfico de *“Pareto”,* sendo um dos gráficos mais recomendados para essa comparação, por ser tratar de pouco dados e também em relação a variância dos resultados apresentados. No gráfico de *“Pareto”* abaixo, foi plotado a distribuição dos dados da tabela em ordem decrescente de frequência. A linha laranja que está no gráfico, representa uma linha cumulativa em um eixo secundário, ou seja, sendo uma porcentagem do total. A sigla “CA” utilizada na legenda, indica com alteração, que seria das alterações das constantes, e a sigla “SA”, seria sem alterações das constantes, e os

números entre parentes, indica o número do filho que está sendo analisado. Podendo observar a diferença que ocasiona quando dobra essas duas constantes em três cargas.

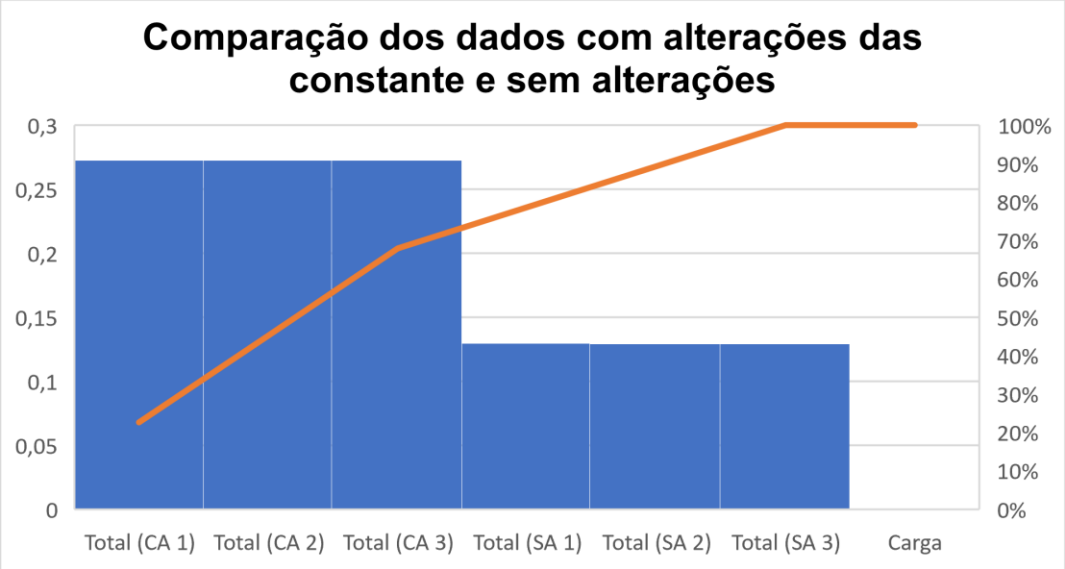

Gráfico 10: Comparação dos dados relacionados as constantes (Duração em seg X Desvio total de cada filho)

Carga	Total (CA 1)	Total (CA 2)	Total (CA 3)	Total (SA 1)	Total (SA 2)	Total (SA 3)
0	0,2726	0,27251	0,27244	0,12931	0,12923	0,12915
20	0,10801	0,18655	0,10813	0,05268	0,05914	0,05281
40	0,15577	0,15546	0,15451	0,05738	0,05378	0,0528

Tabela 5: Utilizada para confecção do gráfico acima

4.10 Pergunta 5

O que significa um processo ser determinístico?

R: Dada uma certa entrada, ela produzirá sempre a mesma saída, com a máquina responsável sempre passando pela mesma sequência de estados.

4.11 Pergunta 6

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

R: Executando o comando “*ps -aux (a- all with tty, including other users; u- effective user id orn ame; x- register format)”.* Como podemos observar a execução deste comando na imagem abaixo:

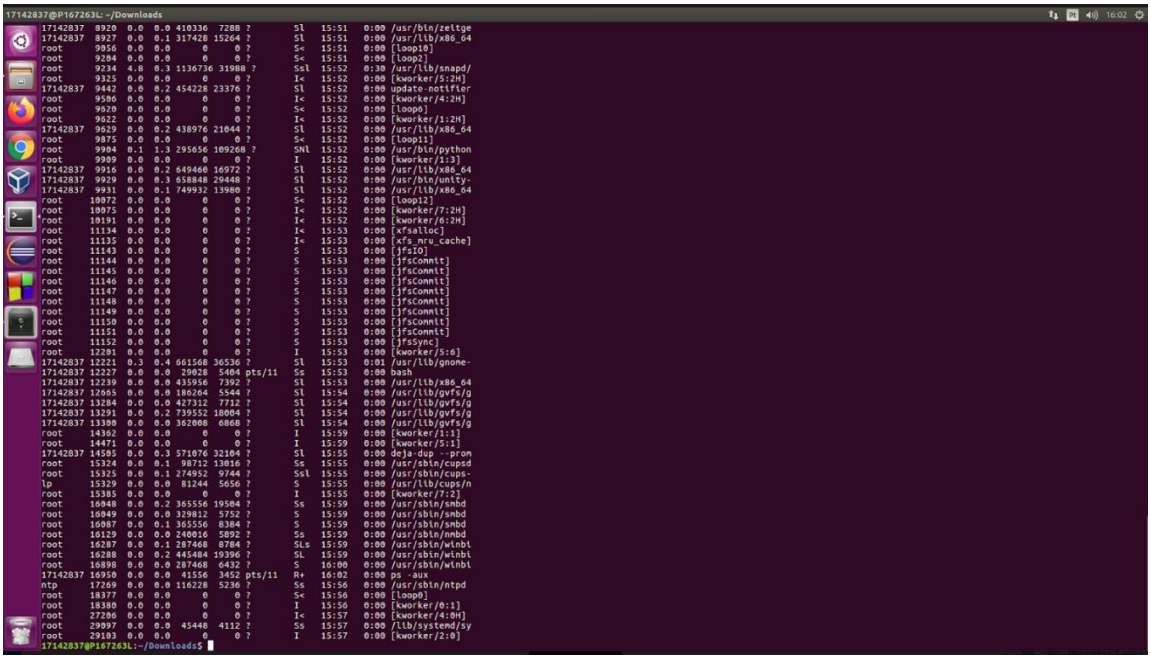

Figura 3: Execução do comando para conseguir informações de todos os processos existentes na máquina

5. SEGUNDA TAREFA

5.1 Letra a.

O número de processos filhos seja igual a cinco;

R: Na segunda tarefa, primeiramente, alteramos o número de processos filhos para cinco.

5.2 Letra b.

Nesse programa modificado, ao invés de ter o código do filho no mesmo programa do código do pai (como acontece fora do loop onde está o *fork()*, crie outro programa, que em execução corresponderá a um processo filho, e chame-o usando uma chamada do tipo *exec()*;

R: Criamos um outro programa com o nome *filho.h* que é chamado através de uma chamada do tipo *exec(),* que é *execvp()*, passando como parâmetro o nome do

executável do filho na primeira posição, e um vetor de argumentos na segunda posição, no qual esse vetor sempre tem NULL na sua última posição. Esse é um exemplo da chamada do comando que foi utilizado *“execvp("./filho",args);”.* Passando todo o código do filho que estava no mesmo programa do pai, para esse novo programa criado que foi chamado no meio da execução, passando as variáveis necessárias da execução do programa filho por parâmetro através do vetor de *string* “*args”*. E para identificar o PID do filho e do pai, para saber de quem estava se tratado nas medições e quando necessário, foi utilizado a função *getpid ().*

5.3 Letra c.

O número de microssegundos para a chamada *usleep()* deve ser igual a um múltiplo de 200, começando em 400. Passe esse valor como parâmetro na chamada do tipo *exec()* e;

R: Foi definido 400 o número de microssegundos da chamada *usleep()* inicialmente e acrescendo em 200 em 200 no total de dez vezes, onde foi analisado o desvio total e médio dos 5 filhos. Abaixo podemos observar a tabela (Tabela 6) demonstrativa em relação a essa análise, não foi confeccionado gráficos para essa análise, pois os resultados são muito próximos de um filho para o outro, sendo praticamente o mesmo ponto no gráfico, dependendo da dimensão do ponto ou curva.

5.4 Letra d.

Use *kill(),* no processo pai, para terminar os processos filhos (cuidado, pois, para terminar um processo efetivamente, este não pode ter terminado).

R: Foi utilizado a função *kill(),* no processo pai para estar terminando os processos filhos. Sendo executado dez vezes e mantendo a carga da máquina sem utilizar outros programas em execução. Não foi realizado a confecção de gráficos comparativos para analisar essas execuções, porque os pontos obtidos de cada filho, em relação a cada tempo, são pontos bem próximos um do outro, dependendo do gráfico de dispersão, tamanho dos pontos ou da curva, pode-se concluir a mesma reta ou pontos no gráfico, ou seja, não obtendo bons gráficos comparativos.

	Filho 1		Filho 2		Filho 3		Filho 4		Filho 5	
Tempo	**Total**	**Médio**	**Total**	**Médio**	**Total**	**Médio**	**Total**	**Médio**	**Total**	**Médio**
400	0,49296	0,00049296	0,4945	0,0004945	0,49339	0,00049339	0,49296	0,00049296	0,49393	0,00049393
600	0,73365	0,00073365	0,73347	0,00073347	0,73128	0,00073128	0,73345	0,00073345	0,7334	0,0007334
800	0,94744	0,00094744	0,94739	0,00094739	0,94531	0,00094531	0,94438	0,00094438	0,9473	0,0009473
1000	0,12374	0,00012374	0,12485	0,00012485	0,12487	0,00012487	0,12253	0,00012253	0,12481	0,00012481
1200	0,34447	0,00034447	0,34415	0,00034415	0,3439	0,0003439	0,3439	0,0003439	0,34388	0,00034388
1400	0,54515	0,00054515	0,54378	0,00054378	0,54459	0,00054459	0,54463	0,00054463	0,54461	0,00054461
1600	0,74717	0,00074717	0,74693	0,00074693	0,74682	0,00074682	0,74683	0,00074683	0,74665	0,00074665
1800	0,94095	0,00094095	0,94058	0,00094058	0,94038	0,00094038	0,94038	0,00094038	0,94036	0,00094036
2000	0,145	0,000145	0,14364	0,00014364	0,1434	0,0001434	0,14436	0,00014436	0,14339	0,00014339
2200	0,34581	0,00034581	0,34595	0,00034595	0,34548	0,00034548	0,34576	0,00034576	0,34557	0,00034557

Tabela 6: Análise de dez execução do programa e os cinco filhos, variado o tempo de 200 em 200 microssegundos

6. ANÁLISE DOS RESULTADOS

Medir a passagem do tempo é uma tarefa muito exigida em aplicações de diversas áreas, seja para a sincronização de processos, ou para algum tipo específico de aplicação, como programas de avaliação de desempenho.

Geralmente aplicações não computacionais lidam com o tempo de maneira explícita e tratam a passagem de tempo como uma tarefa natural e simples, porém em meios computacionais as medições podem ser afetadas por flutuações de desempenho do sistema, falhas de software, sobrecarga, etc.

Nesse experimento usamos a função *usleep()* para a medição do tempo. A função *usleep()* deveria ter, teoricamente, uma precisão na ordem de microssegundos e nanosegundos, mas tal feito não ocorre normalmente em um processo rodando com prioridade normal, sem privilégios. Sua precisão fica dependente da política de gerenciamento de tempo utilizada pelo Linux, neste caso é feito através dos *ticks* que duram cerca de 10ms.

O kernel do Linux usa o gerenciamento de processos em diversas tarefas internas, e estas acabam interferindo diretamente no comportamento de um processo. Na primeira parte do experimento um dos maiores problemas é que o *clock* é variável, isso acarreta que a *cpu* acaba subindo o *clock* em relação a carga nela.

Tome a seguinte situação como exemplo, querer-se-ia que um processo durma 15 milissegundos, o sistema operacional (Linux) entende que o processo precisa ficar no mínimo 15 *ms* dormente, ele não trata com exatidão. E por não o tratar com exatidão, ele irá buscar cumprir no mínimo o tempo desejado, utilizando-se da ocorrência de *ticks* para determinar o tempo que o processo irá ficar parado. Como cada *tick* ocorre a cada 10ms, o S.O acordara o processo após 2 *ticks* desta forma ele garante que os 15 *ms* tenham passado.

Outro fator que pode aumentar o tempo de dormência, além do desejado, é que a função pode não iniciar exatamente na ocorrência do primeiro *tick*. Por exemplo, é possível que a função seja chamada 7 *ms* antes do próximo *tick*, assim o processo só será acordado após os 2 *ticks* (20ms), que iriam acontecer de qualquer maneira, mais 7 *ms* depois, totalizando 27 *ms* de dormência. Abaixo pode ser observado um exemplo ilustrativo:

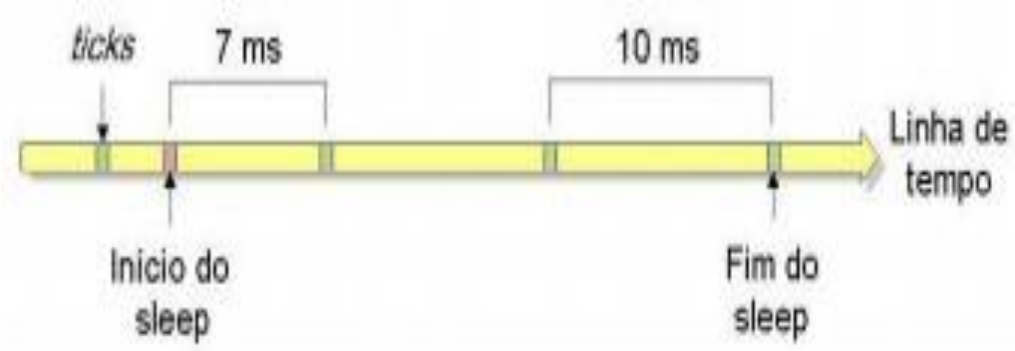

Figura 4: Exemplo do funcionamento dos *ticks*

7. CONCLUSÃO

No experimento, foi solicitado submeter um programa base a uma série testes, na qual trouxe-nos conhecimentos diversos, base para iniciarmos uma breve aprendizagem sobre o ambiente *linux* e o compilador gcc.

Com os testes foram adquiridos conhecimentos sobre a concorrência de processos na CPU, e como estes são afetados por fatores externos (sobrecargas como outros programas e processos na concorrência), assim sendo possível perceber a mudança nos tempos de execução e por seguinte ao analisá-los, foi possível adquirir as variações correspondentes para o estudo, notou-se que a mudança de variáveis e sobrecargas de processos impactavam neste valor. Tomou-se então conhecimento sobre processos pai e filhos, como ocorrem e todo seu comportamento quando submetidos a esses testes simulando um programa comum.

Portanto, este experimento foi de grande proveito para o aprendizado de todos do grupo, mesclando conceitos, antes desatados, que foram ensinados durante todo o curso até o presente, criando assim, um conjunto de ideias que se tornam cada vez mais sólidas.