Sistemas Operacionais A, 1^0 semestre/2017

Experimento #1

1. Introdução

O primeiro experimento permite o contato com dois assuntos importantes em Sistemas Operacionais: a criação de processos e o conceito de tempo. Este experimento engloba vários fatores importantes que precisam ser percebidos sobre a duração da execução de um programa em um ambiente multitarefa.

Este exercício foi definido a partir dos experimentos existentes em http://www.rt.db.erau.edu/oldrtlab/ que pertencem ao Laboratório Embry-Riddle de tempo-real.

2. Objetivos

A seguir estão os objetivos do primeiro experimento com relação ao aluno:

a. Permitir sua familiarização com o tipo de experimentos que serão desenvolvidos durante o semestre.
b. Mostrar como ocorre a criação de processos em Sistema Operacionais (SOs) Unix.
c. Mostrar um modo para a medição da duração de um processo ou de parte dele.
d. Permitir a percepção do conceito de multitarefa (*multitasking*) e de seus efeitos sobre a execução de um determinado programa.

3. Tarefas

Cada experimento constitui uma atividade que precisa ser completada através de tarefas básicas. A primeira se refere à compilação e entendimento de um programa exemplo que trata de assuntos cobertos em sala de aula e na teoria. Uma segunda se refere à implementação de uma modificação sobre esse programa.

Primeira Tarefa

A primeira tarefa é compilar o programa exemplo. Se você não sabe como fazer isso, procure orientação com algum colega ou use o comando *man*. O compilador a ser usado deve ser o gcc em ambiente Linux.

Pergunta 1: Apresente a linha de comando para compilar o programa exemplo, de tal maneira que o executável gerado receba o nome de experim2 (sem extensão).

Após compilado sem erros, executar o programa dez (10) vezes. Procure variar a *carga* do computador, aumentando-a a cada execução, ou seja, execute

“simultaneamente” um mesmo programa que consuma CPU e veja se o desvio aumenta. Para isso, coloque para executar esse mesmo programa, várias vezes, a primeira sem carga, a segunda cinco, a terceira dez e assim sucessivamente, de cinco em cinco.

Considerando que o programa que consome CPU tem nome carga, utilize as seguintes linhas de comando:

- ./carga
- ./carga & ./carga

Pergunta 2: Descreva o efeito da diretiva &.

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

Utilize um programa que efetivamente use CPU. Lembre-se que durante um acesso a disco ou à rede ou mesmo a espera da entrada de algum dado (interação) pelo usuário não consome tempo de CPU.

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

Para a apresentação dos três resultados das execuções do programa exemplo, crie um quadro semelhante ao apresentado a seguir:

Filho/Carga	Total (seg)	Média (seg)
1/0	0.278	0.000003
2/0	0.319	0.000004
...	...	...
1/10	0.401	0.000004
...	...	...

Analise os resultados e tente achar um padrão. Utilize gráficos com os resultados obtidos, uma curva para cada processo filho.

Tente entender o que está acontecendo dentro da máquina!

Uma lição importante deste experimento é aprender que, quando há dormência de processos, o tempo de resposta dos processos não é previsível.

Altere, pelo menos, duas das constantes e veja como elas afetam os resultados. **Não se esqueça de registrar essas experiências no relatório**.

Segunda Tarefa

Para a segunda tarefa, com base no programa exemplo, altere-o de maneira que:

a. O número de processos filhos seja igual a cinco;

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()*;

c. O número de microsegundos 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

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

Execute o programa modificado dez (10) vezes, mantendo a carga da máquina (sem outro programas em execução).

Crie um quadro adequado para apresentar os resultados. Analise os resultados obtidos nas dez execuções. Não se esqueça de, também, analisar essas execuções com relação aos resultados das execuções do programa exemplo.

Relatório

Todo experimento deve ser acompanhado de um relatório com algumas partes obrigatórias (veja o exemplo de relatório), no caso do primeiro experimentos são necessárias:

a. *Introdução, indicando em não mais do que 20 linhas o que fazem o programa exemplo e o programa modificado;*
b. *Apresentação de erros de sintaxe e/ou lógica que o programa exemplo possa ter, juntamente com a sua solução;*
c. *Respostas às perguntas que se encontram dispersas pelo texto do experimento e pelo código fonte exemplo;*
d. *Resultados da execução do programa exemplo;*
e. *Resultados da execução do programa modificado;*
f. *Análise dos resultados (deve-se explicar o motivo da desigualdade ou igualdade de resultados);*
g. *Programa usado para aumentar a carga do computador;*
h. *Conclusão indicando aquilo que foi aprendido com o experimento.*

Entrega

A entrega do primeiro experimento deve ser feita em seu escaninho no AVA, em uma pasta com o nome “Experimento1”, de acordo com o cronograma previamente estabelecido, **até a meia-noite** do dia anterior à respectiva apresentação.

Em todos os arquivos entregues deve constar **OBRIGATORIAMENTE** o nome e o RA dos integrantes do grupo.

Devem ser entregues os seguintes itens:

i. *os códigos fonte* (são quatro os programas),
ii. *os executáveis,*
iii. *o relatório final do trabalho, em formato pdf.*

Solicita-se que **NÃO** sejam usados compactadores de arquivos.

Não serão aceitas entregas após a data definida. A não entrega acarreta em nota zero no experimento.

4. Teoria

A seguir são apresentados conceitos diretamente relacionados com o experimento, sua leitura é importante e serão cobrados quando da apresentação. Caso o entendimento desses conceitos não se concretize, procure reler e, se

necessário, realizar pequenas experiências de programação até que ocorra o entendimento (procure o professor se for necessário).

Duração da execução de um trecho de programa

A duração da execução de um trecho de programa não pode ter sua medição realizada com perfeição, se realizada dentro do computador. Primeiramente, um computador é uma máquina digital, então, sua realidade é a execução de uma série de passos simples e *determinísticos.*

Pergunta 5. O que significa um processo ser determinístico?

Mesmo supondo que estes passos sejam realizados em um microsegundo ou em um nanosegundo, os mecanismos disponíveis para medição de tempo em um computador sempre vão apresentar alguma limitação. Além do mais, o próprio conjunto de passos necessários para medir o tempo causa um profundo efeito no tempo que está sendo medido. Os atos de "ler" o tempo do relógio do sistema e copiar em uma área de memória local, onde o processo pode ter acesso, requer uma quantidade de tempo, pois são realizados através da execução de comandos (passos).

Os projetistas do SO podem implementar algum esquema para compensar esse tempo adicional, mas isso é improvável, pois, como é apresentado a seguir, a duração desse tempo não é previsível.

Multitarefa

O conceito de multitarefa não é novo. Olhando um sistema com um único processador (ou um multiprocessador que executa processos múltiplos), no processamento multitarefa, diferentes processos conseguem ter suas instruções executadas durante uma parte do tempo disponível do processador. Por exemplo, se três processos precisam executar, o sistema operacional deixará um processo ser executado por uma pequena quantia de tempo e o próximo processo pegará o processador por uma pequena quantia de tempo, e, então, o terceiro processo pegará por mais uma quantia pequena de tempo (este período pequeno de tempo é chamado fatia de tempo de processamento ou *time slice*). Após cada processo acabar de usar o processador durante sua fatia de tempo, o ciclo se repetirá. Isto dará a ilusão que todos os processos estão executando paralelamente, devido à velocidade de execução da CPU.

Comandos comuns para trabalhar com processos: *ps* e *kill*

Um primeiro comando muito importante no Unix é o comando *man* (de manual), através do qual pode-se obter auxílio a respeito de outro comando. Por exemplo, tente chamar *man gettimeofday* (*gettimeofday é uma função que é discutida a seguir e serve para a leitura de um relógio).*

Antes de examinar as chamadas de sistema usadas por um SO UNIX para executar operações sobre processos, pode ser útil conhecer vários comandos que serão úteis na execução deste e dos experimentos restantes.

Primeiro, o comando *ps* permite ver informações sobre os processos que estão em execução naquele instante na máquina. Digitando *ps* em uma janela, aparecerá uma lista de processos. Embora a maioria dos SOs Unix tenha este comando para relacionar os processos ativos, o mesmo pode ser diferente em cada SO. Execute o comando *man ps* para aprender mais sobre o comando que lista informações de processos.

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

O segundo comando importante é o comando *kill* que é usado para enviar *sinais* a outros processos. Para usar este comando digite *kill pid*, onde *pid* é a identificação numérica associada a um processo que se quer sinalizar. O *pid* de um processo pode ser obtido através de vários métodos, mas o mais comum é usar o comando *ps.* Tenha cuidado com os *pids* submetidos ao comando kill, pode-se terminar a execução, por exemplo, do *shell*, que é o programa que processa os comandos Unix submetidos após o *logon*.

Criando e Administrando Processos em C: *fork*, *exec* e *wait*

Existem três chamadas de sistema comuns, fornecidas pelo Unix, para criar e administrar processos: *fork*, *exec* e *wait*. A primeira chamada é usada para criar processos, a segunda para carregar uma *imagem* de processo (programa), diferente da que está em execução, e a última para esperar pelo término de um processo. Caso não entenda alguma desta terminologia, não se preocupe, ela deverá ficar clara ao final do experimento. Procure utilizar o comando *man* sobre estes comandos para informação mais específica.

Como exemplo de criação de um processo observe o que acontece quando se executa um *ls* em um *shell*:

O *shell* é um processo cujo programa foi escrito para que pedidos do usuário sejam executados (no caso *ls*).
O processo *shell* interpreta a linha de comando, através de um *read*, e determina o que fazer.
Uma chamada *fork ()* no programa *shell*, quando executada, ocasiona a criação de duas instâncias do processo *shell,* uma nova e uma que já existia.
Ambas as instâncias continuam suas execuções na instrução que segue a chamada *fork ().*
A 1ª instância do *shell*, então, realiza a chamada *wait ()* para esperar pelo término da segunda.
A 2ª instância do *shell* chama exec () para carregar o programa *ls*.
O programa *ls*, uma vez carregado, é executado até o seu término.
A 1ª instância do *shell* sai do *wait ()* e continua a sua execução.

A chamada *fork* cria um processo novo. Este processo é idêntico ao original com exceção de uma diferença secundária, o valor de retorno da chamada *fork ()* (há outras diferenças, mas neste momento não têm importância). Acesse a seguinte página *fork.swf*. Observe a animação e relacione-a com o seguinte trecho de código:

```
int rtn;

... algum codigo ...

rtn = fork();

if( rtn == 0 ) {

... codigo para o processo filho ...

}
```

```
else {

... codigo para o processo pai ...

}
```

Perceba que a chamada *fork()* retorna um valor que é armazenado na variável *rtn*.

Lembre-se que, após a execução do *fork(),* existirão dois processos distintos. Cada um deles executando a atribuição do valor retornado por *fork()* em *rtn*.

Considerando que um desses dois processos esteja em execução, se o valor de *rtn* é nulo, o processo que está em execução é o processo novo e é chamado de filho. Se o valor de *rtn* é não-nulo, o processo é o processo original ou pai e o valor de *rtn* corresponde ao *pid* do filho.

Outra maneira de entender: imagine que a chamada *fork ()* realiza uma clonagem. Antes de chamar *fork ()* havia um processo, depois são dois! Idênticos, exceto pelas áreas de memória usadas e pelo *pid*.

Criação de processos no Unix é um conceito importante que necessita ser entendido. Se não entendeu, procure escrever alguns programas semelhantes que exibam informações do que está acontecendo. Esta técnica pode ser usada para esclarecer o que está ocorrendo.

Uma rotina do SO escolhe qual processo vai ser processado em determinado momento. Tal rotina está baseada em um algoritmo (chamado de escalonamento porque realiza a escolha) com uma política de escolha e com prioridades para determinar qual processo será executado, quando a CPU está disponível. Considere que todos os processos têm uma prioridade igual. Cada processo, então, terá uma fatia de tempo dada pelo SO para ser executado no processador. Ao término da fatia de tempo, o SO interrompe o processo (retira-o da CPU) e aloca uma fatia de tempo para outro processo. Deste modo, cada processo que precisa ser processado vai conseguir o seu intento, durante uma fatia de tempo. Às vezes, processos terminam e a CPU é associada a um outro processo. O ato de retirar um processo correntemente em execução e substituí-lo por outro é denominado **troca de contexto**.

Uma troca de contexto demora uma quantia de tempo. Assim, se um processo está em execução e um segundo processo necessita ser processado, o primeiro processo tem que liberar o processador e uma troca de contexto tem que acontecer.

Considere o seguinte trecho:

```
while(1) {

    read_sensor ();

    usleep(1000);

}
```

Em princípio pode parecer que a execução do trecho vai fazer que o processo permaneça 1000 unidades de tempo bloqueado, após a chamada *read_sensor()* , porém, um exame mais detalhado revela um problema:

> O *loop* requer algum tempo para executar. Assim, o trecho ao invés de ocasionar a leitura do sensor a cada 1 milisegundo, demorará 1 milisegundo mais o tempo que leva para executar os demais comandos

que compõem o *loop*. Porém, não há garantia alguma que a chamada *usleep* () retorne exatamente a cada um milisegundo. Um dos motivos:

> outros processos também podem estar sendo executados e, portanto, podem ocorrer trocas de contexto por conta da fatia de tempo dada ao processo que contém o *loop*. Então, o processo em questão tem que esperar um período a mais de tempo pelo processador. Este período de tempo fará o *loop* demorar mais tempo que o idealmente necessário e, possivelmente, fará a leitura do sensor ficar tardia. Isto é chamado desvio ou *drift* e deve ser observado na execução do programa exemplo. Um desvio acontece quando um evento periódico se afasta levemente do tempo em que é suposto para ocorrer.

Programa Exemplo

Considere o Código Fonte do Programa Exemplo. Observe o trecho de código onde a chamada *fork ()* acontece:

```
rtn = 1;

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

   if( rtn != 0 ) {

            rtn = fork();

   }

   else {

            break;

      }

}
```

Primeiro, *rtn* é fixado em um. Posteriormente, o *loop* é executado uma quantia de vezes equivalente a NO_OF_CHILDREN, criando os filhos. Lembre que o pai recebe um número não nulo na chamada da função *fork* () e o filho recebe um zero. Assim, o processo pai começa fora do *loop* com *rtn* 1 e depois terá *rtn* com um valor diferente de zero, enquanto os filhos sempre têm *rtn* igual a zero. Ambos, pai e filho, após a execução do *fork* (), estarão executando o comando de atribuição e em seguida o *for.*

A cada execução do *loop*, o pai tem *rtn != 0* avaliado como verdadeiro. Portanto, a chamada *fork* () é executada. Enquanto cada um dos filhos, após sua criação, tem *rtn != 0* avaliado como falso e sai fora do *loop*. Lembre-se que o filho é uma duplicata exata do pai na hora do *fork* (), com exceção do valor de retorno.

Veja no programa exemplo que, em seguida à chamada fork () e à execução do *for*, o comando *if,* fora do *loop,* verifica o valor de *rtn* e decide se o processo é um filho ou se é o pai. Se for o pai, a chamada *wait ()* é realizada para esperar pelo término dos filhos. Os filhos, por sua vez, tentarão medir o desvio que acontece com a chamada *usleep()*, repetida NO-OF_ITERATIONS vezes. Cada filho executa, então, o trecho de código seguinte:

```
[...]

gettimeofday( &start_time, NULL );

for( count = 0; count < NO_OF_ITERATIONS; count++ ) {

    usleep(SLEEP_TIME);

}

gettimeofday( &stop_time, NULL );

[...]
```

gettimeofday () é uma chamada de sistema original da versão BSD4.3 do Unix (o comando *man* pode ser usado para obter mais informação sobre essa chamada de sistema). Essa chamada devolve uma estrutura *timeval* (veja a estrutura abaixo) que contém o número de segundos e micro segundos que decorreram desde 1 de janeiro de 1970 às 00:00 (o momento que é considerado o nascimento do Unix). Pode-se perceber, a partir do formato da estrutura *timeval,* que esta chamada de sistema não apresentará precisão maior que um microsegundo. Porém, é útil saber o quão precisa esta chamada é.

```
Struct timeval

{

Int tv_sec;

Int tv_usec;

};
```

O tempo inicial é retornado usando a chamada *gettimeofday ().* Depois, o *for* é executado até NO_OF_ITERATIONS vezes. A cada vez, o processo para de ser executado por um período equivalente a SLEEP_TIME, permitindo assim que outro processo tenha acesso à CPU.

Ao término do *for*, quando a CPU é novamente associada ao processo original, o tempo final é medido. Teoricamente, o tempo final menos o tempo inicial deve ser equivalente a NO_OF_ITERATIONS multiplicado por SLEEP_TIME. Porém, este não é o caso, devido ao comando *for* demorar um pouco para ser executado (porque as instruções que o compõem requerem algum tempo do processador), e também do tempo necessário para chamar a rotina *usleep ().* Além disso, pode haver outros processos disputando o processador, ocasionando um atraso maior para que o processo volte a conseguir a CPU para continuar sua execução.

5. Apresentação

Os resultados do experimento serão apresentados em sala no dia de aula prática, com a **PRESENÇA OBRIGATÓRIA** de todos os alunos, de acordo com o cronograma previamente estabelecido.

Serão escolhidos alunos para a apresentação e discussão dos resultados.

Todos os alunos que completaram o experimento devem se preparar para a

apresentação no ambiente Linux. Serão cobrados:

Os códigos fonte,

Os executáveis,

A introdução e a conclusão,

Os erros e soluções no programa exemplo,

As respostas às questões,

Os resultados e sua análise.

Recomenda-se fortemente que a preparação para apresentação seja feita individualmente.