Sistemas Operacionais

Experimento #2

1. Introdução

O experimento #1 deve ter permitido o aprendizado sobre criação de processos concorrentes no Sistema Operacional (SO) Unix. No entanto, existem aplicações que requerem algum método que permita o compartilhamento de dados (informação) entre processos.

IPC (*Inter-Process Communication*), ou comunicação entre processos, consiste em um conjunto de métodos que permitem que processos em um SO Unix possam se comunicar.

Como existem muitas versões de Unix, existem métodos diferentes para realização de IPC. A variante Berkeley do Unix usa *sockets*. A variante da AT&T, o System V, usa filas de mensagens e compartilhamento de memória. Além disso, no POSIX.1b, filas de mensagens e memória compartilhada são definidas de modo diferente que no System V. *Pipes* constituem outro tipo de mecanismo para IPC. Até mesmo um arquivo pode ser usado para comunicação entre processos. Apesar da variedade de métodos diferentes para IPC, é importante entender e perceber como processos podem se comunicar.

Pergunta 1: Esclarecer o que são: Berkeley Unix, System V, POSIX, AT&T, *socket*, fila de mensagem, memória compartilhada e *pipes*.

Neste experimento são exploradas algumas das chamadas para mecanismos de IPC existentes no System V, ou seja, aquelas para manipulação de filas de mensagens.

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

2. Objetivos

Este experimento deve permitir ao aluno:

- Perceber o por quê de processos precisarem se comunicar;
- Entender o método de troca de mensagens, para que processos possam se comunicar.

3. Tarefas

A primeira parte da tarefa é compilar e executar o programa exemplo.

Após compilado sem erros, executar o programa 10 (dez) vezes. Procure *carregar* mais o computador a cada execução, ou seja, aumentar a sua carga através de um número maior de processos que realmente disputem a CPU. Tente observar se o desvio aumenta.

Tente perceber que, diferentemente do que ocorreu no experimento #1, as tomadas de tempo são realizadas por diferentes processos.

Apresente os dez resultados obtidos e qual a carga usada. Tente entender o que está acontecendo dentro da máquina.

Para a apresentação dos resultados do programa exemplo, crie um quadro semelhante ao apresentado a seguir. Analise os resultados e tente achar um padrão (faça uso de gráficos). Procure inicializar a carga e o experimento no mesmo console e inclua no relatório *printscreens* dos pids dos processos iniciados.

Execução	Médio (seg)	Máximo (seg)	Carga	No mesmo console
1	0.003	0.278	5	Sim
2	0.003	0.519	10	Sim
3	0.003	0.332	15	Sim
4	0.004	42.705	20	Sim
...	...	...	...	...

No relatório, apresentar os resultados do comando *ipcs* após cada execução. Explicar as características que são observáveis.

Para a segunda tarefa, um dos fatores que pode determinar a demora para uma mensagem ser transferida entre dois processos é o tamanho da mensagem. A outra é a carga da máquina. Altere o programa exemplo de maneira que:

- um valor numérico inteiro seja lido a partir do teclado, dentro do intervalo 1 a 10. Este deve ser multiplicado por 512 e o resultado deverá ser o tamanho da mensagem a ser enviada;
- haja a declaração de uma segunda fila de mensagens, que junto com a primeira fila de mensagem, deve ser declarada adequadamente nos processos filhos e não mais no corpo principal do pai (**não haverá herança**);
- o filho receptor continua calculando os tempos médio e máximo, e deve, além disso, calcular o tempo total (somatória de todos os tempos) e o tempo mínimo, mas não os exibe;
- a segunda fila seja usada para que um terceiro filho seja capaz de receber e exibir os cálculos feitos pelo irmão receptor (ou seja, durante um breve instante o receptor será transmissor);
- estabeleça um esquema para remover as filas de mensagens.

Execute o programa modificado dez vezes com o mesmo número de iterações. Para cada execução modifique o valor usado para determinar o tamanho da mensagem, começando

em 1, aumentando de 1, até 10. Crie um quadro adequado para apresentar os resultados. Analise os resultados obtidos e explique as diferenças (faça uso de gráficos).

4. Resultado

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 o exemplo.

Este experimento deve ser acompanhado de um relatório com as seguintes partes obrigatórias:

- Introdução, indicando em não mais do que 20 linhas o que fazem o programa exemplo e os programas modificados;
- Apresentação de erros de sintaxe e/ou lógica que o programa exemplo possa ter, juntamente com a sua solução;
- Respostas às perguntas que se encontram dispersas pelo texto do experimento e pelo código fonte exemplo;
- Resultados da execução do programa exemplo;
- Resultados da execução do programa modificado;
- Análise dos resultados (deve-se explicar o motivo da igualdade ou desigualdade de resultados, usando como suporte gráficos com dados);
- Conclusão indicando o que foi aprendido com o experimento.

Entrega

A entrega do experimento deve ser feita 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;*

ii. 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.

5. Teoria

A seguir são discutidos conceitos diretamente relacionados com o experimento. Sua leitura é importante e será cobrada 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.

Este experimento tem como foco um método primário para IPC, filas de mensagens. Antes de apresentar como é possível criar e remover esse tipo de construção na linguagem C, dois novos comandos são introduzidos, pois deverão ser muito úteis ao longo deste e dos futuros experimentos. Primeiro, o comando ipcs pode ser usado para listar os recursos IPC alocados. Segundo, o comando ipcrm pode ser usado para remover os recursos IPC alocados.

Procure mais informação sobre esses comandos usando o comando man.

Pergunta 2: As chamadas ipcs e ipcrm apresentam informações sobre quais tipos de recursos?

Filas de mensagem

Uma fila é uma estrutura de dados que permite atendimento FIFO (*First In, First Out* - primeiro que chega, primeiro que sai) – atenção especial ao “permite”. Em uma fila de mensagens, a primeira mensagem que é colocada na fila pode ser a primeira mensagem a ser lida da fila, ocorrendo um sincronismo entre origem e destino, pois as mensagens são lidas na ordem que foram enviadas. O oposto a isto é o assincronismo, onde a ordem recebida pode ser diferente da ordem enviada. Há quatro chamadas de sistema associadas com filas de mensagem:

- msgget()
- msgctl()
- msgsnd()
- msgrcv()

O seguinte esquema ilustra como usar uma fila de mensagens:

- Use msgget() para obter o número ID correspondente a uma chave única, criando a fila de mensagens se necessário.

Pergunta 3: Qual a diferença entre o *handle* devolvido pela chamada msgget e a chave única?

- Use msgsnd() e msgrcv(), respectivamente, para transferir dados para e retirar dados da fila de mensagens identificada pelo número ID previamente obtido.
- Use msgctl() para remover a fila de mensagens do sistema.

Filas de mensagens são relativamente simples de usar. O SO controla os detalhes internos de comunicação. Quando se envia uma mensagem através da fila, qualquer

processo que espera por uma mensagem naquela fila é alertado, dependendo dos parâmetros da chamada msgrcv().

O SO verifica a integridade da fila e não permite que processo algum tenha acesso a uma fila de modo destrutivo. Embora as filas de mensagens tenham estas vantagens, elas têm duas desvantagens distintas. Filas de mensagens são lentas em transferir grandes quantias de dados e há um limite claro para o tamanho do pacote de dados que pode ser transferido. Então, filas de mensagens são melhores quando taxas lentas de transferência de dados podem ser usadas (com *bandwidth* limitado). O mecanismo de filas de mensagens é um excelente modo de IPC para processos passarem informação de "controle" para outro processo.

Pergunta 4: Há tamanhos máximos para uma mensagem? Quais?

Pergunta 5: Há tamanhos máximos para uma fila de mensagens? Quais?

Programa Exemplo

O programa exemplo para recursos compartilhados procura estabelecer quanto demora para se transferir uma mensagem através de uma fila de mensagens. É um programa simples, mas apresenta algumas técnicas interessantes que podem ser usadas em uma variedade de aplicações diferentes. Aqui está o algoritmo básico para o programa:

- O pai cria a fila de mensagens.
- O pai cria dois filhos.
- O primeiro filho vai:
    - Receber uma mensagem da fila que contém um valor de tempo.
    - Chamar gettimeofday() para adquirir a hora atual.
    - Usando o tempo que se encontra na mensagem e aquele que acabou de adquirir, calcular a diferença entre os mesmos.
    - Repetir os passos acima um número determinado de vezes.
    - Exibir os resultados.
- O segundo filho vai:
    - Chamar gettimeofday () para adquirir a hora atual.
    - Colocar o tempo obtido em uma mensagem.
    - Colocar a mensagem na fila.
    - Dormir para permitir que o irmão possa ser executado.
    - Repetir os passos acima um número determinado de vezes.
- O pai espera os filhos terminarem.

Observe como a fila de mensagens é criada:

```
if ((queue_id = msgget(key, IPC_CREAT | 0666)) == -1) {
      fprintf(stderr, "não foi possível criar fila de mensagem");
      exit(1);
}
```

No código acima, a chamada msgget() **pode** criar uma fila de mensagens. Os argumentos são uma chave única *key* (que foi escolhida com valor MESSAGE_QUEUE_ID, se considerarmos que foi declarado anteriormente; este valor é como um nome para a fila de mensagens; qualquer número que não está sendo atualmente usado pode ser usado) e um conjunto de *flags* que neste caso são IPC_CREAT e 0666 em conjunto. IPC_CREAT diz que se quer criar a fila se ela não existe e 0666 são as permissões de acesso do Unix (permissão de leitura e escrita para todos). O valor de retorno é queue_id que será usado para outras chamadas de funções que manipulam mensagens. Note que, criando a fila antes de criar os filhos com fork, todos os filhos herdam o queue_id e assim a chamada msgget () só precisa ser feita uma vez.

Observe a seguir duas estruturas importantes que são usadas para transferir dados através das filas de mensagens.

A primeira estrutura é usada para o envio de dados pela fila de mensagens. É uma estrutura padrão que permanece a mesma para qualquer aplicação de fila de mensagens, com o tamanho de *mtext* que muda de acordo com o tamanho dos dados que precisam ser transferidos. Esta contém dois pedaços de informação. O primeiro é o tipo da mensagem (*mtype*), que é escolhido pelo usuário. Para este experimento, fica sempre o mesmo. O próximo é um *array* de caracteres de tamanho igual ao tamanho dos dados que precisam ser transferidos. A estrutura de dados é definida abaixo.

```
typedef struct {
      long mtype ;
      char mtext[sizeof(data_t)];
} msgbuf_t;
```

Pergunta 6: Para que serve um typedef?

Pergunta 7: Onde deve ser usado o que é definido através de um typedef?

A estrutura seguinte é usada para os dados que precisam ser transferidos pela fila. Contém o número da mensagem (msg_no – dentro do *loop*) e o tempo obtido antes do envio (send_time). Esta estrutura é definida pelo usuário e muda de fila de mensagens para fila de mensagens, dependendo do que o usuário desejar enviar.

```
typedef struct {
      unsigned msg_no;
      struct timeval send_time;
} data_t;
```

O seguinte esquema é usado para enviar uma mensagem pela fila:

- Declare uma estrutura do tipo msgbuf_t e um ponteiro do tipo data_t (que pode apontar para uma estrutura do tipo data_t).
- Associe o ponteiro data_t ao endereço de mtext na estrutura msgbuf_t.
- Usando o ponteiro data_t, preencha os dados a serem enviados.
- Chame msgsnd() para enviar a mensagem na fila.

O seguinte esquema é usado para receber uma mensagem da fila:

- Declare uma estrutura do tipo msgbuf_t e um ponteiro do tipo data_t.
- Associe o ponteiro data_t ao endereço de mtext na estrutura msgbuf_t.
- Chame msgrcv() para receber a mensagem pela fila.
- Usando o ponteiro data_t, manipule os dados que foram recebidos.

A função sender() é examinada (a análise da função receiver() permanece como um exercício para o leitor). Note que a função sender() é chamada por um dos filhos, enquanto a função receiver() é chamada pelo outro filho. Primeiro, na função sender() declaram-se as estruturas necessárias e associa-se o ponteiro como descrito anteriormente.

```
msgbuf_t message_buffer;
data_t *data_ptr = (data_t *) (message_buffer.mtext);
```

Na função sender() entra-se em um "loop", de forma que uma mensagem é enviada. Para este experimento, sender() repete um loop NO_OF_ITERATIONS vezes. Dentro do loop, na função sender(), chama-se gettimeofday() para obter o tempo em que a mensagem foi colocada na fila. O dado é então copiado na estrutura.

```
gettimeofday(&send_time,NULL);
message_buffer.mtype = MESSAGE_MTYPE;
data_ptr->msg_no = count;
data_ptr->send_time = send_time;
```

Em seguida, na função sender(), coloca-se os dados na fila através da chamada msgsnd(). Nesta chamada, queue_id corresponde ao valor de retorno da chamada msgget(). O segundo argumento é um ponteiro para a estrutura que será transferida, neste caso message_buffer. O terceiro argumento é o tamanho dos dados contidos dentro da estrutura. O argumento final é uma flag que está fixada em zero.

```
if (msgsnd(queue_id,(struct msgbuf *)&message_buffer,
    sizeof(data_t),0) == -1) {
      fprint(stdrerr, "impossivel enviar mensagem!\n");
      exit(1);
}
```

Pergunta 8: Na chamada msgsnd há o uso de *cast*, porém agora utiliza-se um “&” antes de message_buffer. Explicar para que serve o “&” e o que ocorreria se este fosse removido.

Em seguida, chamando usleep() em sender(), espera-se por alguns micro segundos para que a rotina receiver() possa ser executada em outro processo.

6. Apresentação

Os resultados do experimento serão apresentados em sala no dia de aula prática da semana marcada para a entrega, 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 preparar para a apresentação, em formato digital:

- Uma introdução sobre o experimento (apresentando diagramas e imagens);
- Os resultados e sua análise (apresentando imagens, tabelas e gráficos);
- Os códigos-fonte e as partes relevantes do código;
- Uma conclusão sobre o experimento.

Recomenda-se fortemente a preparação para apresentação.