Projeto de Programação 2

Cada aluno vai implementar o programa individualmente. O programa tem várias etapas com certa independência. A primeira etapa é a mais trabalhosa, mas é direta.

Os fontes e executáveis devem ser deixados na máquina virtual disponível em um diretório, de forma organizada. O professor precisa identificar as etapas 3, 4, 5 e 6.

Entrega: já houve tempo suficiente para a entrega da 1ª etapa. Assim, a partir do ia 15/Out, cada etapa deve ser entregue a cada duas semanas.

Etapa 1: criação de processos, fork/wait, geração de números aleatórios, tempo, dormir

- O processo pai:
  - Cria 3 processos-filho através da *system call* `fork()`. Sugestão: armazenar o pid de cada filósofo em uma posição de um *array* de *int* no pai.
  - Em seguida, aguarda o término de cada um dos filhos (`wait ()`).
  - A cada filho que termina o processo pai imprime na tela uma mensagem do tipo, "Filho X, pid Y, terminou".
  - Depois que os três filhos terminarem o pai imprime na tela, "Todos os filhos terminaram. Processamento teve a duração de Z ms".
  - No caso dos processos criados através de *fork ()* vamos utilizar os sinais (*signal*) como mecanismo do Unix de comunicação entre processos filhos com o pai. Ou seja, o processo filho vai enviar um *signal* para o processo pai - o processo pai já deve estar "aguardando" (bloqueado) este evento através de um *wait()*, como já definido anteriormente.

- Cada processo:
  - executa a sua rotina 100 vezes.
  - Após cada paço da rotina o filho deve exibir uma mensagem na tela, com as informações de seu número (0, 1 ou 2), pid, paço onde se encontra (e outras informações que se entenda por relevantes); e aguarda um tempo aleatório;
  - A cada "dormida" (variando entre 5 e 100 ms) o tempo esperado aleatório em milissegundos também deve ser apresentado na tela.
  - Cada processo deve cronometrar/calcular o tempo gasto médio de execução da rotina, e o desvio padrão. Exibir este valor na tela antes de terminar.
  - Quando a rotina terminar, cada processo "*avisa"* ao processo "pai" que *já terminou*. O pai deve avisar que esta ciente de que o filho X terminou.
- Lembre-se: os processos devem ser programados de forma que se imponha o "intercalamento" aleatório dos mesmos (caso contrário eles podem executar em seqüência). Ou seja, o "dormir" de cada processo deve ser aleatório (usar o *random* e *srand* corretamente).

Etapa 2: memória compartilhada, variáveis compartilhadas.

- O processo pai:
  - Cria uma área de memória compartilhada (*shmem*). A área de memória compartilhada deve ser suficiente para conter duas variáveis inteiras (int);
  - Uma variável int (vamos chamar de primeira variável) deve ser inicializada com 0 e a segunda com 300;
  - Em seguida o pai libera os recursos e termina também. Mas agora também exibe o valor final das duas variáveis compartilhadas.

- Cada processo:
    - Ao exibir as informações, acrescenta o valor das variáveis compartilhadas;
    - Arescentar a rotina abaixo e o tempo de espera agora via ser espalhado:
        - Lê a primeira variável compartilhada
        - Copia para uma variável local
        - Incrementa a variável local
        - Dorme um tempo
        - Copia "de volta" para a primeira variável compartilhada
        - Decremente a variável compartilhada (diretamente na memória compartilhada)
        - Dorme mais um tempo.
- Observe que a memória compartilhada e os semáforos (nos próximos passos) precisam ser liberados após o seu uso.

Etapa 3: semáforos

Na etapa 1 deve ter sido identificado um problema de inconsistência. Pelo menos na primeira variável compartilhada: em várias rodadas, o valor final deve ter sido diferente de 300.

Vamos solucionar o problema com o uso de semáforos (pense em como isso será feito e implemente).

Lembre-se que no processo pai, antes de criar os processos filhos o pai deve criar o semáforo (na verdade o *array* de semáforos) e iniciar o(s) semáforo(s) a ser(em) utilizado(s) nos processos filho.

Lembre-se também de liberar o *array* de semáforos antes de pai terminar.

Compare os resultados do valor final das variáveis e do tempo de execução dos loops (cada paço) e do tempo total do sistema.

Etapa 4: troca de mensagens

Nesta etapa, vamos substituir o mecanismo de *wait*/*signal* do Unix pela troca de mensagens para garantir a sincronização entre processos filhos e pai.

Cada processo filho, ao final de toda a sua atividade vai enviar (ou depositar) uma mensagem para o pai (na fila de mensagens).

O pai, ao invés de aguardar 3 *signals*, um de cada filho, agora tentará ler 3 mensagens da fila. Não vai mais usar o *wait*.

Certamente, para isso o pai precisará criar uma fila de mensagens, ante de criar os filhos. E liberar esta fila ao final.

Etapa 5: mexendo com prioridade dos processos, starvation

Nesta etapa vamos usar as chamadas de sistema relacionadas ao comando *renice* (uma das chamadas possíveis é *setpriority ()*;).

O objetivo é diminuir a prioridade de um ou dois dos processos filho para "forçar" uma situação de *starvation*. O comportamento do sistema deverá ser tal que o processo que não teve sua prioridade reduzida terminará bem antes dos outros, que tiveram suas prioridades diminuídas.

Compare os temos envolvidos (por exemplo, a média e do desvio padrão de cada iteração do loop década processo filho e do tempo total do sistema.

Etapa 6: resolvendo o problema do starvation

Nesta etapa vamos resolver o problema do *starvation*.

Verifique que tipo de semáforo é provido no Linux (FIFO ou não). Os semáforos usados até o momento deveriam resolver este problema?

Pode utilizar qualquer solução. Por exemplo, o uso de barreiras ou de variáveis de contagem para cada processo filho.

Acrescente semáforos ou variáveis compartilhadas que forem necessárias.

Introduza código no processo pai (para a criação ou inicialização de semáforos e/ou mais área compartilhada) e/ou nos processos filho (fazendo uso das variáveis compartilhadas ou semáforos). Não se preocupe com a elegância da solução.

Pode usar uma estratégia "hard" em que todos os processos precisam fazer um certo número de iterações antes de continuar, ou mais "soft" em que a diferença do número de iterações que cada processo filho já realizou não é maior que um dado número de vezes.

O *array* previsto para ficar na memória compartilhada (além da variável de contagem) pode ser usado para evitar o *starvation*. Ou uma contagem, ou binário (0 e 1). É com você avaliar e aplicar alguma técnica na rotina dos processos.

Observação:
A estratégia do pai criar (ou alocar) os recursos (memória compartilhada, semáforos, filas de mensagens) apenas organiza e facilita a programação. Os filhos criados por fork() "herdam" do pai todos os recursos criados. A alternativa seria o pai passar, de alguma forma, a chave de segurança para cada filho, e cada filho se encarregaria de fazer o reconhecimento ou *attach* dos recursos antes de usá-los.

ETAPA BÔNUS

Refazer os passos 1, 2, 3, 5 com monitores em Java. Mas ... preservando o máximo grau de concorrência (ou seja, usando blocos de sincronização, ao invés de sincronizar o método)

Dica para usar a memória compartilhada acessando campos de uma estrutura:

```
// colocar no início do código

typedef struct shm_type
{
   int varA;
   int varB;
} T_SHM;

key_t chave_memo = 6622;
key_t chave_sema = 6621;

int localA;

int memo_comp,  /* id da shared memory */
struct shmid_ds shmid_ds; /* para conter os resultados de smclt */
T_SHM *shmem; /* aponta para a area de memoria compartilhada */


// cria a shm
memo_comp = shmget (chave_memo, sizeof (T_SHM), (0666 | IPC_CREAT);


// agora associa ao processo
memo_comp = shmget (chave_memo, sizeof (T_SHM), (0666 | IPC_CREAT | IPC_EXCL


// ... agora podemos usar a memória compartilhada

shmem -> varA = 10; // atribuindo valor ao campo varA da memória

localA = shmem -> varA; // variável local localA recebe valor de varA


// desassocia a memória compartilhada
shmdt ((void *)shmem);



/* libera a memória compartilhada (o pai faz isso quando os filhos terminarem */

shmctl (memo_comp, IPC_RMID , &shmid_ds);
```