Sistemas Operacionais

Experimento #4

1. Introdução

Nos experimentos prévios foram utilizados mecanismos de IPC (*Inter-Process Communication*) baseados em fila de mensagens, memória compartilhada e semáforos. Usando estes mecanismos, foi possível compartilhar dados entre diferentes processos.

Processos, naturalmente, são independentes entre si em relação aos seus espaços de endereçamento de dados e aos seus contextos, enquanto compartilham uma ou mais CPUs do sistema, de maneira concorrente. Para compartilhar dados, processos necessitam de mecanismos de IPC para seu acesso e também para que esse acesso ocorra sob exclusão mútua, se houver condição de corrida.

Uma outra maneira de haver concorrência é através do uso de *threads*. Uma *thread* é definida como uma linha (sequência) de execução que pode ser concorrente a outras linhas, dentro de um mesmo processo, ao mesmo tempo em que é capaz de acessar recursos comuns a todas as linhas desse processo.

Todo processo (programa em execução) é por definição uma *thread*, que correspondente à rotina *main*(). Uma vez em execução, esta primeira *thread* pode criar outras *threads* que concorrerão com ela pela(s) CPU(s). Uma *thread* deverá corresponder à execução de uma rotina declarada no programa e disparada através de chamada ao sistema apropriada.

U m a *thread*, também denominada de processo-leve, passa a ser a unidade de escalonamento dentro do SO, com uma possível e condicional vantagem sobre o escalonamento baseado em processos: a de uma troca de contexto mais simples e, consequentemente, mais rápida. Além disso, o fato das *threads* de um mesmo processo poderem acessar os recursos comuns não requer, necessariamente, o uso de mecanismos de IPC para uma comunicação entre as mesmas.

Pergunta 1: Explique por que a vantagem do uso de *threads* é condicional (a resposta se encontra neste documento).

Este experimento deverá permitir ao aluno familiarizar-se com *threads*, além de permitir diferenciá-las em relação aos processos. Neste experimento também são abordados os problemas de produtores e consumidores, e do jantar das filósofas.

Este exercício foi definido especialmente para as aulas de Sistemas Operacionais, na Faculdade de Engenharia de Computação, na PUC-Campinas, com a ajuda do bolsista alemão Florian Weizenegger.

2. Objetivos

A seguir estão os objetivos deste experimento com relação ao aluno:

- Entender o conceito de *thread*.
- Observar como criar e manipular *threads*.

- Perceber como se dá a execução concorrente de *threads*.
- Entender a diferença entre *thread* e processo, explorando o acesso a uma estrutura de memória local ao processo.
- Explorar o problema dos produtores e consumidores, usando um *buffer* circular.
- Explorar o problema do jantar das filósofas usando *threads.*
- Permitir observar a necessidade de exclusão mútua também entre *threads* concorrentes.

3. 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;
- Conclusão indicando o que foi aprendido com o experimento.

Entrega

A entrega do experimento deve ser feita em seu escaninho no AVA, em uma pasta com o nome “Experimento4”, 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. 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. Tarefas

São duas as tarefas para este experimento:

- Compilar e executar o programa exemplo, sem erros de lógica. Cuidado especial é requerido nos comandos em que ocorrem testes de *buffer* cheio e *buffer* vazio, ou seja, é necessário acertar a forma como está no exemplo. Não é para se preocupar com (tentar resolver) as condições de corrida que existam. Apresentar em um quadro os resultados obtidos ao lado dos resultados esperados. Use pthread_join() na *thread* principal para aguardar as *threads* filhas. A modificação também deve permitir responder às questões que seguem. Apresentar em um quadro os resultados obtidos ao lado dos resultados esperados.
  - Quantas vezes, por *thread* produtora (identificar), não foi possível armazenar valores no *buffer*?
  - Quantas vezes, por *thread* consumidora (identificar), não foi possível consumir valores do *buffer*?
- Programe a solução do jantar das filósofas, fazendo com que cada filósofa seja uma *thread*. Para estabelecer exclusão mútua use *mutex*. Para isso, utilize as chamadas pthread_mutex_init(), pthread_mutex_lock(), pthread_mutex_unlock() e pthread_mutex_destroy(). No lugar de pensando programe uma espera de 25 microsegundos. A cada teste para saber se pode comer, apresentar qual é a filósofa que chamou testa e como se encontram os estados das cinco filósofas. Cada filósofa deve terminar depois de ter comido 365 vezes. Use pthread_join() para aguardar o término das filósofas.

5. 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:

- Os códigos-fonte,
- A introdução e a conclusão,
- Os erros e soluções no programa exemplo,
- As respostas às questões,
- Os resultados e sua análise.

Recomenda-se fortemente a preparação para apresentação.

6. 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.

Pode não ser claro o porquê da distinção entre processos e *threads*, principalmente quando se considera que a concorrência pode ser obtida através do uso de comandos *fork*() para a criação de novos processos.

Threads, no entanto, constituem uma interessante alternativa para a geração de concorrência. Desde que seja possível a sua utilização, *threads* apresentam vantagens sobre o uso de processos.

Threads são recomendáveis em situações em que pedaços de um mesmo processo podem ser executados concorrentemente. Caso uma solução resulte em um programa estritamente sequencial, não há motivo para o uso de *threads*. No entanto, se houver a necessidade da realização concorrente de tarefas, como é o caso da solução do problema clássico dos produtores e consumidores ou do jantar das filósofas, então *threads* permitem o compartilhamento das estruturas, variáveis e outros recursos globais que sejam declarados no processo, sem a necessidade de mecanismo de IPC.

Todas as *threads,* dentro de um mesmo processo, podem acessar um mesmo espaço de endereçamento. Toda mudança nos recursos globais é visível por todas as *threads* do processo. Por exemplo, o fechamento de um arquivo global ou a atribuição de um valor a uma variável global.

Threads são menores que processos em termos do seu contexto, por serem parte de um todo. Assim, sua criação e manipulação são relativamente mais baratas em termos de uso de CPU. Todo processo tem recursos, como arquivos, mecanismos e dados. As *threads*, podendo compartilhar esses recursos, não exigem quase nada de memória. Cada *thread* tem seu próprio fluxo (linha) de execução com sua pilha, ponteiros e registradores, e acessa o diretório de trabalho, a área de *heap* e os descritores de arquivos do processo.

Pergunta 2: Apresente um quadro comparativo com, pelo menos, três aspectos para processos e *threads*.

Pergunta 3: O que é a área de *heap*?

Outra razão para a popularidade das *threads* é a velocidade com que podem ter seus contextos trocados durante o escalonamento, desde que a *thread* que passa a ser executada pertença ao mesmo processo da *thread* que deixou a CPU. O *time-slice* de um processo não é longo, tipicamente algo em torno de 100 mili segundos. Uma vez que um processo tenha sido escalonado, sua carga na CPU pelo *dispatcher* é uma atividade demorada.

Pergunta 4: Quais são as funções do *dispatcher*?

Uma troca de contexto para processos, que envolve não somente o salvamento do contexto do processo anterior como a carga do contexto do novo, também pode acarretar na necessidade de esvaziamento (*flush*) da memória *cache* e da TLB (*Translation Lookaside Buffer*) e de seu preenchimento com as informações do novo processo. TLBs são usadas para afetar o endereçamento físico de memória a partir de um endereço lógico (assunto para aulas de Gerência de Memória), onde ficam armazenados

endereçamentos anteriores que tiveram que ser computados, de maneira a permitir acessos mais rápidos à memória.

Pergunta 5: O que vem a ser a memória *cache*?

Caso um processo seja retirado da memória, se esta se encontra cheia e há necessidade de liberar espaço (*swap out*), muito provavelmente esse processo deverá retornar (*swap in*) em outro momento. Este procedimento pode não ser muito demorado (possivelmente algumas dezenas de mili segundos) em um rápido computador ou demorado se acessos a disco forem necessários. Com a troca de contexto se realizando frequentemente devido a um número elevado de processos, o tempo dedicado ao escalonador imporá mais lentidão, devido à gerência de memória cheia.

Uma troca de contexto envolvendo *threads*, considerando *threads* de um mesmo processo, é muito mais rápida que uma troca de contexto entre processos. O que passa a ser necessário é a recomposição do estado da CPU. Para *threads* de um mesmo processo, tanto a *cache* como a TLB continuam válidas e não necessitam ter seu conteúdo trocado. A possibilidade de substituir uma solução baseada em processos para outra baseada em *threads* oferece a oportunidade para um ganho de tempo considerável (*Difference between Processes and Threads* - http://www.differencebetween.net/miscellaneous/difference-between-thread-and-process/).

Criando e Gerenciando Threads em C

Existe uma interface especificada pela IEEE Posix para *threads* na linguagem C que deu origem à biblioteca *pthread.h*.

Para a criação de *threads*, inicialmente é necessária a declaração de uma variável do tipo *pthread_t*. Este tipo permite a declaração de uma variável que recebe um descritor quando a *thread* é criada. Posteriormente, esse descritor pode ser usado em comandos de controle para *threads*. Duas das principais chamadas ao sistema para respectivamente iniciar e terminar *threads* são:

pthread_create() permite a criação de uma *thread* que terá a execução da rotina passada como parâmetro de maneira concorrente à *thread* que está em execução.

pthread_exit() indica o término da *thread* que está em execução e pode ser usada para terminar a *thread*.

Programa Exemplo

O programa exemplo oferece uma tentativa de solução para o conhecido problema dos produtores e consumidores, que compartilham um *buffer* que é percorrido de maneira circular. Na realidade, o *buffer* é um vetor onde produtores armazenam o que produzem e de onde consumidores retiram o que consomem. Elementos que sejam retirados do *buffer* deixam de existir (teoricamente). Elementos que são armazenados no *buffer*, eventualmente, podem deixá-lo cheio, sem espaço para novos armazenamentos.

Existem dois ponteiros que percorrem o *buffer*: (*rp*) que aponta para a posição do *buffer* onde deve ocorrer a próxima retirada (*reading pointer*); e (*wp*) que aponta para a posição do *buffer* onde deve ocorrer o próximo armazenamento (*writing pointer*).

Cada um dos ponteiros tem seu valor incrementado, de maneira a poder apontar para a próxima posição do *buffer*, exceto se estiver na última posição, situação em que é alterado para apontar para a primeira posição, ocasionando um efeito circular.

Duas rotinas são usadas para armazenar e retirar elementos do *buffer*: *myadd*() e *myremove*().

```
int myadd(int toAdd) {
    //verificacao se o buffer nao esta cheio
    if ((rp != (wp+1)) && (rp + SIZEOFBUFFER – 1 != wp)) {
        wp++;
        //verificacao se wp chegou a ultima posicao do buffer
        if (wp == (start + SIZEOFBUFFER)) {
            wp = start;     /* realiza a circularidade no buffer */
        }
        *wp = toAdd;
        return 1;
    } else return 0;
}
```

A rotina *myadd*(*int toAdd*) verifica primeiro se o *buffer* se encontra cheio através de duas condições. Em um destes dois casos ((*rp == (wp+1)) or (wp == rp + SIZEOFBUFFER - 1)*), é retornado 0 sem haver o armazenamento do elemento.

Caso o *buffer* não seja considerado cheio, *wp* é incrementado. Após isto, é verificado se *wp* já atingiu o fim do *buffer* e, em caso positivo, este é posicionado no início. Em seguida, o elemento é armazenado na posição apontada por *wp*.

A verificação de buffer estar cheio é falha. Perceba que o *buffer* estando vazio, se um produtor chegar ao buffer primeiro, wp terá como conteúdo o valor de rp + SIZEOFBUFFER – 1 (valor este de inicialização na main).

A rotina *myadd* retorna 1 para indicar que o armazenamento foi realizado.

```
int myremove() {
    //verificacao se o buffer nao esta vazio
    if (wp != rp) {
        int retValue = *rp;
        rp++;
        //verificacao se rp chegou aa ultima posicao do buffer
        if (rp == (start + SIZEOFBUFFER)) {
            rp = start;      /* realiza a circularidade no buffer */
        }
        return retValue;
    } else return 0;
}
```

A rotina *myremove*(), diferente da rotina *myadd*(), verifica primeiro se o *buffer* encontra-se vazio. Estará vazio, caso *rp* esteja apontando para a mesma posição que *wp* (*wp == rp*). Neste caso é retornado 0 sem haver a retirada de valor.

A verificação de buffer não estar vazio é falha. Perceba que o *buffer* estando vazio, se um consumidor chegar ao buffer primeiro, wp terá como conteúdo o valor de rp +

SIZEOFBUFFER – 1 (valor este de inicialização na main, ou seja, buffer vazio) e wp!=rp, vai ser verdadeiro, deixando consumir o que não existe.

A rotina *myremove*(), uma vez retirado o valor do *buffer*, incrementa *rp* e verifica se o mesmo atingiu o fim do *buffer*, para em seguida posicioná-lo no seu início. Esta rotina retorna o valor retirado caso tenha tido sucesso.

Dando uma olhada na rotina *main*(), percebe-se que pode haver a criação de diversas *threads*:

```
int main(int argc, char *argv[]) {
    int tp, tc;
    int i;

    start = &buffer[0];
    wp = start + SIZEOFBUFFER - 1;
    rp = start;

    for (i=0;i<NUM_THREADS;i++) {
        // tenta criar uma thread consumidora
        tc = pthread_create(&consumers[i], NULL, consume, (void *)i+1);
        if (tc) {
            printf("ERRO: impossivel criar uma thread consumidora\n");
            exit(-1);
        }
        // tenta criar uma thread produtora
        tp = pthread_create(&producers[i], NULL, produce, (void *)i+1);
        if (tp) {
            printf("ERRO: impossivel criar uma thread produtora\n");
            exit(-1);
        }
    }
    printf("Terminando a thread main()\n");
    pthread_exit(NULL);
}
```

Inicialmente, dois ponteiros para o *buffer* circular são inicializados para o endereço de início do *buffer* (*rp* e *start,* este último sempre aponta para o início). Em seguida há um comando de repetição que depende do número de *threads,* NUM_THREADS, para criação de rotinas para produtores e consumidores (um de cada, no exemplo). A *thread* de produção é criada com a rotina *produce*() como parâmetro, enquanto a de consumo, com a rotina *consume*(). O resultado da chamada *pthread_create*() indica se a *thread* foi criada com sucesso (0) ou não (1).

Após a criação das *threads*, a rotina *main*() termina, através da chamada *pthread_exit*(). Como não há qualquer tipo de tratamento para status de fim de *thread*, NULL é usado como parâmetro.

```
void *produce(void *threadid)
{
    int sum = 0;
    int ret = 0;
```

```
    printf("Produtor #%d iniciou...\n", threadid);

    while (cont_p < NO_OF_ITERATIONS) {
        ret = myadd(10);
        if (ret) {
            cont_p++;
            sum += 10;
        }
    }
    printf("Soma produzida pelo Produtor #%d : %d\n", threadid, sum);
    pthread_exit(NULL);
}
```

Na rotina *produce*() ocorrem tentativas de armazenar o valor 10 no *buffer*. Algumas dessas tentativas podem não dar certo. De qualquer maneira, NUM_OF_ITERATIONS vezes, será armazenado o valor 10. O contador *cont_p* é usado para controlar que ocorram NUM_OF_ITERATIONS manipulações bem sucedidas no *buffer*. A rotina *consume*() segue uma lógica semelhante.

Dica para compilação

ATENÇÃO: A biblioteca *pthread* necessita ser adicionada quando da compilação dos programas que manipulam *threads*. Para isso, use a seguinte opção de chamada do compilador *gcc*:

gcc experiment.c -o experiment -pthread