Aula 4 - Comunicação entre Processos

Sistemas Operacionais
Ciência da Computação
IFB - Campus Taguatinga

Professor João Victor de A. Oliveira

Na aula passada

- Processos
- Threads

Hoje

- Comunicação de Processos
  - Condições de corrida
  - Regiões críticas
  - Exclusão mútua com espera ocupada
  - Exclusão mútua sem espera ocupada
    - Dormir e acordar
    - O problema do produtor-consumidor
    - Semáforos
- Comunicação entre processos UNIX

Comunicação entre processos

- **IPC - *Interprocess Communication***
  - **Três questões em comunicações entre processos**
    i. **Como um processo passa informações para o outro?**
    ii. **Como certificar-se de que dois processos não se atrapalhem?**
      - Ex.: última vaga de um avião disputada por dois passageiros...

    iii. **Qual o sequenciamento adequado quando as dependências entre processos estão pendentes?**
      - Se o processo A produz os dados e o processo B imprime,
        - O processo B tem de esperar até que...
        - O processo A tenha produzido alguns dados antes de imprimir

Comunicação entre processos

- **Vale notar que essas questões também aplicam-se aos threads**
  - **Transferência de dados/informações**
    - Mais fácil para as threads (por quê?)
  - **Duas outras questões**
    - Manter um thread afastado do outro e o sequenciamento correto - aplicam-se bem às threads
- Nesta aula iremos avaliar essas questões para processos
  - Note que os mesmos problemas podem ser aplicados aos threads

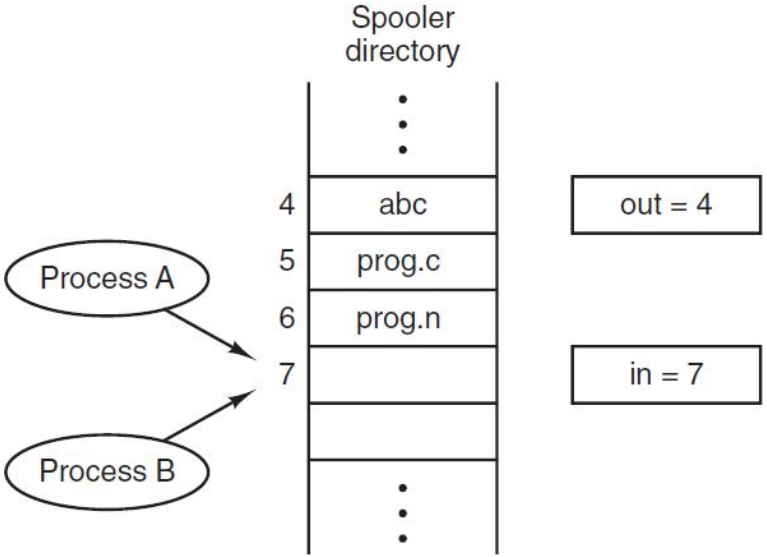
Memória Compartilhada

- Em alguns SOs, processos podem compartilhar alguma memória comum, onde cada processo pode ler e escrever
  - **Memória compartilhada** pode estar na memória principal
    - Possivelmente em uma **Estrutura de Dados de núcleo**
    - ou um **arquivo compartilhado**

Memória Compartilhada

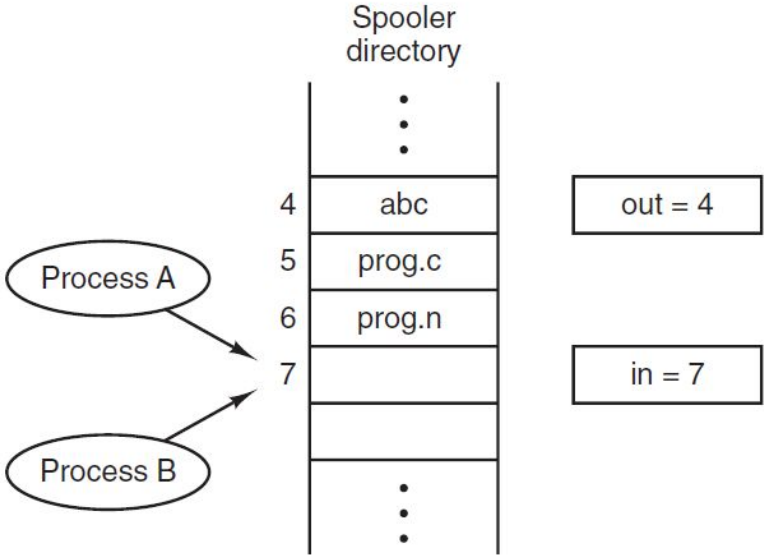
- Problema do *spool* de impressão
  - Quando um processo quer imprimir um arquivo, ele entra com o nome do arquivo em um **diretório spool** especial
    - Outro processo, o ***daemon* de impressão**, confere periodicamente para ver se há quaisquer arquivos a serem impressos
      - Se houver, imprima-os e, então, remova seus nomes do diretório
  - Imagine que nosso diretório de spool tem um número muito grande de vagas, enumeradas em 0,1,2,...,n
    - Cada vaga armazena um nome de arquivo
  - Imagine também duas variáveis compartilhadas:
    - ***in:*** *Aponta para a próxima vaga livre no diretório*
    - ***out:*** *Aponta para o próximo arquivo a ser impresso*

- Problema do *spool* de impressão
  - Imagine que os processos A e B querem colocar diferentes arquivos na fila de impressão
  - Aplicando a Lei de Murphy:
    - **O processo A** lê de in e armazena o valor, 7, em uma variável local chamada *next_free_slot*
    - Em seguida ocorre uma **interrupção** e troca para o processo B
    - **Processo B** também lê in e recebe o valor 7 na sua variável global *next_free_slot*
    - **Processo B** continua a execução e armazena o nome do seu arquivo na vaga 7 e atualiza in para 8



Memória Compartilhada

- Problema do *spool* de impressão
  - **Processo A** executa novamente
  - Olha *next_free_slot* e escreve o nome do seu arquivo na vaga 7, apagando o nome que o processo B colocou ali.
  - ...
  - ***Daemon* de impressão** não notará nada de errado!
  - Note que o **processo B** **jamais receberá qualquer saída!!!**
- **Situações como essa são chamadas de condições de corrida**



Condição de Corrida

- Situação onde dois ou mais processos estão lendo ou escrevendo alguns dados compartilhados e o resultado final depende de quem executa e quando é executado
  - Com o aumento de núcleos no processador, as condições de corrida estão se tornando cada vez mais comuns

https://hacks.mozilla.org/2017/06/avoiding-race-conditions-in-sharedarraybuffers-with-atomics/

Regiões críticas

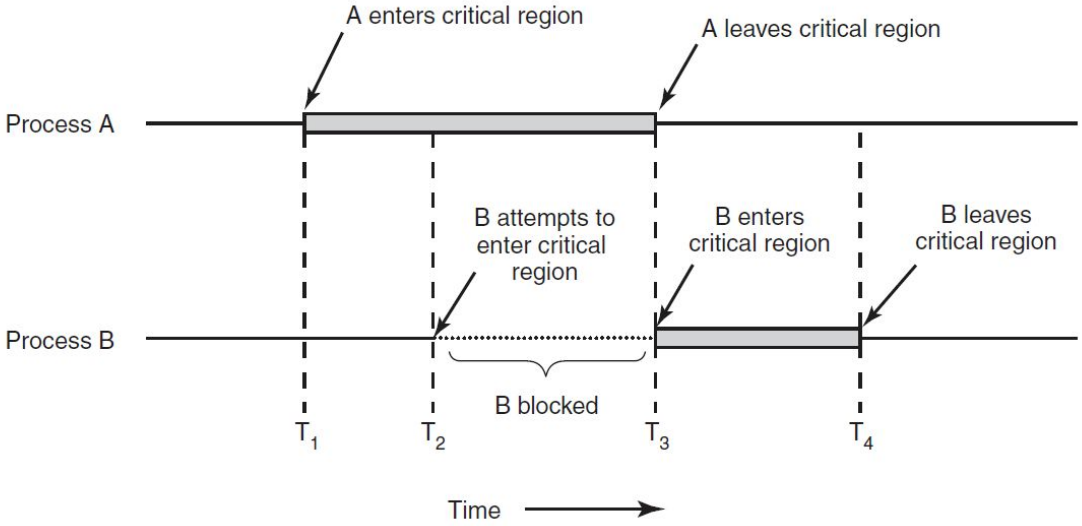
- Como evitar as condições de corrida?
  - **Proibir mais de um processo ler e escrever os dados compartilhados ao mesmo tempo**
  - Ou seja, precisamos de uma **exclusão mútua**
    - Uma maneira de certificar-se que **se** um processo está usando um arquivo ou variáveis compartilhadas, **então** os outros serão impedidos de realizar a mesma coisa ao mesmo tempo
    - Parte do programa onde a memória compartilhada é acessada é chamada de **região crítica ou seção crítica**

Exclusão mútua

- Exclusão mútua
  - Durante uma comunicação entre processos, jamais dois processos podem estar em suas regiões críticas ao mesmo tempo
    - Com isso evitamos as condições de corrida?
      - Infelizmente a resposta é não! Por quê?
      - Precisamos que 4 condições se mantenham para garantirmos uma exclusão mútua eficaz

Regiões críticas

- 4 condições para uma boa solução de exclusão mútua
  1. Dois processo jamais podem estar **simultaneamente dentro de suas regiões críticas**
  2. Nenhuma suposição pode ser feita a respeito de **velocidades ou do número de CPUs**
  3. Nenhum processo **executando fora de sua região crítica** pode bloquear qualquer processo
  4. Nenhum processo deve ser obrigado a **esperar eternamente** para entrar em sua região crítica



Exclusão mútua com espera ocupada

Veremos as seguintes técnicas:

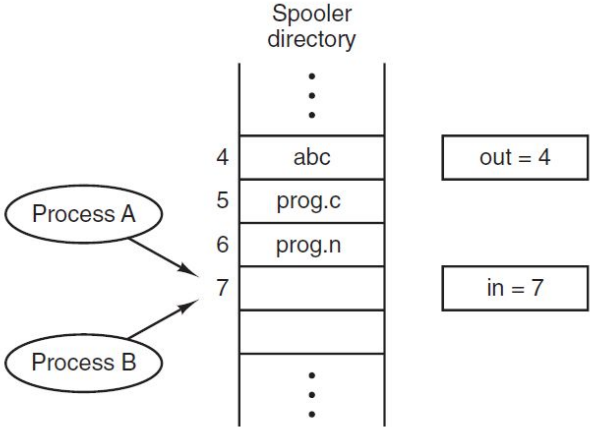
1. Variáveis do tipo trava
2. Alternância explícita
3. Solução de Peterson
4. Instrução TSL

Variáveis do tipo trava

- **Considere ter uma única variável (de trava) compartilhada, inicialmente com valor igual a zero**
  - Quando um processo entrar em sua região crítica, ele primeiro testa a trava
    - **Se a trava é 0,** o processo a configura para 1 e entra na região crítica
    - **Se a trava já é 1,** o processo apenas espera até que ela se torne 0
  - Desse modo uma trava em 0 significa que **nenhum processo está em sua região crítica**
  - Já uma trava em 1 significa que **algum processo está em sua região crítica**

Variáveis do tipo trava

- **Infelizmente,** variáveis do tipo trava possuem a mesma falha fatal que vimos no problema do diretório de spool
  - Suponha que um processo lê a trava e vê que ela é 0
  - Antes de ela puder configurar a trava para 1, outro processo é escalonado, executa e configura para a trava 1
    - Os dois processos entrarão na sua região crítica!!!



Alternância Explícita

- Variável do tipo turn inicialmente igual a zero, controla de quem é a vez ao entrar na região crítica
  - **O processo (a)** inspeciona turn e descobre que ele é 0 e entra na sua região crítica
  - **O processo (b)** também encontra o valor 0 e espera em um laço fechado até que que o processo (a) saia da região crítica e mude turn para 1
- Testar continuamente uma variável até que algum valor apareça é chamado de **espera ocupada**
  - **Deve ser evitada, pois desperdiça tempo de CPU!!!**
  - **Uma trava que usa a espera ocupada é chamada de trava giratória (*spin lock*)**

```
while (TRUE) {
    while (turn != 0)          /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)          /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Alternância Explícita

- Caso um processo seja muito mais lento que o outro não é uma boa ideia usar espera ocupada
  - Imagine que o processo (b) termine sua região crítica e não modifique de primeira turn=1
    - Ex.: ocorrência de um tratamento de exceção
    - Viola a condição 3 de exclusão mútua: um processo não pode estar bloqueado por outro que não esteja em sua região crítica!

```
while (TRUE) {
    while (turn != 0)          /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)          /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Solução de Peterson

[Peterson, 1981] descobriu uma maneira simples de realizar uma exclusão mútua

- Antes de usar as variáveis compartilhadas (de entrar em uma região crítica), cada processo chama ***enter_region*** com seu número de processo (0 ou 1)

- Após terminado de usar as variáveis compartilhadas, o processo chama ***leave_region*** para indicar que ele terminou e permitir que outro processo execute

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Solução de Peterson

Funcionamento

- Inicialmente nenhum processo está na sua região crítica

- Agora o processo 0 chama ***enter_region***, indica seu interesse, alterando o valor de seu elemento de arranjo e alterando turn para 0

- Se o processo 1 fizer agora uma chamada para **enter_region** ele esperará ali até que interested[0] seja false

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Solução de Peterson

Funcionamento

- Considere o caso em que ambos os processos chamam **enter_region** simultaneamente

- Ambos armazenarão o seu número de processo em turn

- O último a armazenar é o que conta

- Note que ainda temos uma espera ocupada

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Solução de Peterson - Exemplo de execução

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
    int other;                          /* number of the other process */

    other = 1 – process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE)  /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Instruções TSL

- *Instrução TSL - Test and Set Lock*
  - *TSL RX, LOCK*
    - Lê o conteúdo da palavra **lock** da memória para o registrador **RX** então armazena um valor diferente de zero no endereço de memória lock
      - Leitura e armazenamento da palavra são **indivisíveis** (nenhum outro processo pode acessar a palavra na memória até que a instrução tenha terminado)
    - Quando **LOCK** está em 0, qualquer processo pode configurá-lo para 1 usando a instrução TSL e, então, ler ou escrever na memória compartilhada
    - Quando terminado, o processo configura lock de volta para zero usando uma instrução **move** comum

Instruções TSL

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was not zero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Instrução XCHG

- *Instrução XCHG*
  - Troca os conteúdos de duas posições atômicamente (indivisível)
    - ex.: Entre um registrador e uma palavra de memória
  - Todas as **CPUs intel x86** usam a instrução XCHG para sincronização de baixo nível

```
enter_region:
        MOVE REGISTER,#1          | put a 1 in the register
        XCHG REGISTER,LOCK        | swap the contents of the register and lock variable
        CMP REGISTER,#0           | was lock zero?
        JNE enter_region          | if it was non zero, lock was set, so loop
        RET                       | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0              | store a 0 in lock
        RET                       | return to caller
```

Exclusão mútua sem espera ocupada

- **Dormir e Acordar**
  - **Objetivo**: evitar a espera ocupada
    - **Dormir e Acordar:** Bloqueia o processo em vez de desperdiçar tempo de CPU quando não é autorizado a entrar nas suas regiões críticas
      - Ex.: par de primitivas ***sleep*** e ***wakeup***
        - ***sleep***: faz que o processo que a chamou bloqueie, isto é, seja suspenso até que outro processo o acorde
        - ***wakeup***: recebe como parâmetro o processo a ser desperto

Problema do Produtor-consumidor

- Também conhecido como problema do **buffer limitado**
  - **Produtor:** insere informações no buffer de tamanho fixo comum
  - **Consumidor:** retira essas informações do buffer
- Problemas
  - Produtor quer colocar um item novo no buffer, mas ele está cheio
    - Solução: produtor deve dormir até que o consumidor tenha removido um ou mais itens
  - Consumidor quer remover um item no buffer, mas o buffer está vazio
  - **Condições de corrida similares às anteriores**

Problema do Produtor - Consumidor

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Problema do Produtor-consumidor

Veja o código ao lado

Note que pode haver uma condição de corrida, porque o acesso a ***count*** *é irrestrito*

Ex.:

- O buffer está vazio e o consumidor acabou de ler **count** para ver se é zero.
- É iniciado a execução do produtor, que insere um item no buffer e incrementa o **count**. Note que **count** está em 1 e envia o sinal de acordar o consumidor

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item();                /* generate next item */
          if (count == N) sleep();              /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep();              /* if buffer is empty, got to sleep */
          item = remove_item();                 /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Problema do Produtor-consumidor

Ex.:

- Infelizmente, o consumidor ainda não está dormindo, logo não recebe o sinal de despertar. O consumidor tinha lido count = 0, logo irá dormir
- Cedo ou tarde o produtor preencherá todo o buffer e vai dormir também
- **Ambos irão dormir** (deadlock)

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item();                /* generate next item */
          if (count == N) sleep();              /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep();              /* if buffer is empty, got to sleep */
          item = remove_item();                 /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Problema do Produtor-consumidor

Solução parcial: Bit de espera pelo sinal de acordar

- Quando for adormecer, se o bit de espera pelo sinal de acordar estiver ligado, ele será desligado, mas o processo permanecerá desperto

Infelizmente com três ou mais processos, o bit de espera pelo sinal de acordar seria insuficiente

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item();                /* generate next item */
          if (count == N) sleep();              /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep();              /* if buffer is empty, got to sleep */
          item = remove_item();                 /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Semáforos

- **[Dijkstra, 1965]** Propõe um novo tipo de variável chamada **semáforo**
  - Sugere usar uma variável inteira para contar o número de sinais de acordar salvos para uso futuro
    - **Assume valor 0**, indicando que nenhum sinal de despertar fora salvo ou
    - **Assume valor diferente de zero,** se um ou mais sinais de acordar estiverem pendentes
  - **Duas operações em semáforos**
    - **down** (similar ao sleep)
    - **up** (similar ao wakeup)

Operações em Semáforos

- **down** (similar ao sleep)
  - Chamada originalmente de P (*proberen* - tentar)
  - Confere para ver se o valor é maior do que 0. Se for, ele irá decrementar o valor e apenas continuará.
  - Caso seja 0, o processo é colocado para dormir, sem completar o down
- **up** (similar ao wakeup)
  - Chamada originalmente de V (*Verhogen* - erger)
  - Incrementa o valor de um determinado semáforo
  - Se um ou mais processos estiverem dormindo naquele semáforo, incapaz de completar uma operação down anterior, um deles é escolhido pelo sistema e é autorizado a completar seu down
- **Essas operações são atômicas!**

Produtor-Consumidor usando Semáforos

- Neste exemplo usamos semáforos de duas maneiras:
  - Mutex - para exclusão mútua
  - Full e Empty - para sincronização

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

Mutexes

- Quando a capacidade do semáforo fazer contagem não é necessária, podemos utilizar uma versão simplificada, chamada **mutex**
  - **Úteis apenas para gerenciar a exclusão mútua de algum recurso ou trecho de código compartilhado**
    - Fácil de implementar e úteis em pacotes threads em espaço de usuário
  - **Mutex** é uma variável compartilhada que pode estar em um de dois estados: **Destravado ou travado**

Mutexes

- Quando um thread (ou processo) precisa de acesso a uma região crítica, ele chama ***mutex_lock***
  - Se o mutex já estiver travado o thread que chamou será bloqueado até que o thread na região crítica tenha concluído e chame **mutex_unlock**
  - **Se múltiplos threads estiverem bloqueados no mutex, um deles será escolhido ao acaso e liberado para adquirir a trava**

Mutexes em pthreads

- **Pthreads** oferece uma série de funções que podem ser usadas para sincronizar threads

Chamada de thread	Descrição
Pthread_mutex_init	Cria um mutex
Pthread_mutex_destroy	Destrói um mutex existente
Pthread_mutex_lock	Obtém uma trava ou é bloqueado
Pthread_mutex_trylock	Obtém uma trava ou falha
Pthread_mutex_unlock	Libera uma trava

Mutexes em pthreads

- Pthreads também oferecem mecanismos de sincronização: **Variáveis de Condição**
  - *Permitem que threads* sejam bloqueados devido a alguma condição não estar sendo atendida

Chamada de thread	Descrição
Pthread_cond_init	Cria uma variável de condição
Pthread_cond_destroy	Destrói uma variável de condição
Pthread_cond_wait	É bloqueado esperando por um sinal
Pthread_cond_signal	Sinaliza outro thread e o desperta
Pthread_cond_broadcast	Sinaliza para múltiplos threads e desperta todos eles

Problema Produtor-consumidor com pthreads

```
#include <stdio.h>
#include <pthread.h>

#define MAX 1000000000                  /* how many numbers to produce */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;            /* used for signaling */
int buffer = 0;                         /* buffer used between producer and consumer */

void *producer(void *ptr)               /* produce data */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                        /* put item in buffer */
                pthread_cond_signal(&condc);       /* wake up consumer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}
```

```
void *consumer(void *ptr)               /* consume data */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);    /* get exclusive access to buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                        /* take item out of buffer */
                pthread_cond_signal(&condp);       /* wake up producer */
                pthread_mutex_unlock(&the_mutex);/* release access to buffer */
        }
        pthread_exit(0);
}

int main(int argc, char **argv)
{
        pthread_t pro, con;
        pthread_mutex_init(&the_mutex, 0);
        pthread_cond_init(&condc, 0);
        pthread_cond_init(&condp, 0);
        pthread_create(&con, 0, consumer, 0);
        pthread_create(&pro, 0, producer, 0);
        pthread_join(pro, 0);
        pthread_join(con, 0);
        pthread_cond_destroy(&condc);
        pthread_cond_destroy(&condp);
        pthread_mutex_destroy(&the_mutex);
}
```

Monitores

- Uso de semáforos e mutexes de forma inadequada podem gerar **deadlocks**
  - Erros usando essas técnicas têm comportamentos **imprevisíveis e irreproduzíveis**
  - Para facilitar isso, **Brinch Hansen (1973)** e **Hoare (1974)** propuseram uma primitiva de sincronização de nível mais alto chamado de **monitor**
- Monitor
  - Coleção de rotinas, variáveis e estruturas de dados que são reunidas em um tipo especial de módulo ou pacote
  - **Propriedade importante:** apenas um processo pode estar ativo em um monitor em qualquer dado instante

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .       .       .
        end;
end monitor;
```

Monitores

- Monitores são uma construção de uma linguagem de programação
  - Cabe ao compilador implementar a exclusão mútua nas entradas do monitor
    - Menos provável que algo dê errado
  - Variáveis de condição (sincronização)
    - **Wait e signal**
      - Quando uma rotina de monitor descobre que não pode continuar, ela realiza um **wait** em alguma variável de condição
      - No outro processo, pode-se despertar o parceiro adormecido realizando um **signal**

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .       .       .
        end;
end monitor;
```

Monitores

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;

        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Comunicação entre Processos Unix

Pipes

- Extensamente usados no shell do Unix
  - Exs.:
    - Listar o histórico de comandos usados em determinado arquivo
      - history | grep “.py”
      - Neste caso, a saída do comando history é a entrada do programa grep
- O mesmo conceito pode ser usado na programação
  - **pipes são buffers protegidos em memória,** acessados segundo a política FIFO (First in, First out)

Criação de Pipes

int pipe(int fd[2]);

- Cria um pipe composto de dois descritores de arquivos
  - fd[0] para leitura
  - fd [1] para escrita
- Retorna 0, em caso de sucesso e 1 em caso de erro

Leitura e Escrita de Pipes

int write(int fd, char *buff, int nbyte)

- Escreve **nbyte** bytes no buffer, apontado por **buff**, no arquivo descrito em **fd**
- Retorna o número bytes escritos com sucesso e, caso apresente erro, retorna -1

int read(int fd, char *buff, int nbyte)

- Lê **nbyte** bytes do arquivo **fd** para o buffer apontado por **buff**
- Retorna o número de bytes lidos com sucesso e, caso apresente erro, retorna 1

Funcionamento geral

1. Criar o pipe
2. Criar processo filho (Fork)
    a. Fork duplicará os descritores de arquivos, sendo assim, o pipe fica disponível para ambos processos pai e filho
3. Um processo lê do pipe (fd[0]) e outro escreve em pipe fd[1]

Ex.: progpipe.c

Filas de Mensagem

- **Mecanismo de comunicação protegido**
  - Podem estar em memória ou disco e apenas processos autorizados têm acesso à fila
- São permanentes
  - Não são destruídas quando o processo que as criou morre (remoção deve ser explícita)
- **Funcionamento**
  1. Criar Fila
  2. Obter identificador da fila
  3. Ler ou escrever na fila
  4. Remover a fila (ou não)

Criação de Filas de Mensagem

int msgget(key_t key, int IPC_CREAT | msgflg**)**

- Bibliotecas necessárias: (Basta acessar o manual da função no unix: ***man msgget***
- **Função:** Cria uma fila com a chave **key**
  - **key_t key:** é a chave indicando uma constante numérica representando a fila de mensagens e pode receber dois valores
    - IPC_PRIVATE (=0): a fila de mensagens não tem chave de acesso e somente o proprietário ou o criador da fila poderão ter acesso à fila;
    - Um valor desejado para a chave de acesso da fila de mensagens.
  - **msgflg**.: permite estabelecer direitos de acesso e comandos de controle
    - Caso **IPC_CREAT** seja especificado (combinado) em msgflg, é criada uma nova fila
    - As permissões nesse caso ficam nos bits menos significativos de msgflg
      - Exemplo: msgflg = 0600 ⇒ 110 000 000 , ou seja, apenas o dono da processo pode ler e escrever na fila de mensagem

Criação de Filas de Mensagem

int msgget(key_t key, int IPC_CREAT | msgflg**)**

- **Retorno:** identificador da fila (msqid) ou erro (-1)
- **Exemplo**
  - if ((msqid = msgget(0x4321, IPC_CREAT | 01FF) == -1) {
    printf(“Erro na criação da fila”); exit (1);

Exemplo: msgget_criacao.c

Obtenção de uma Fila de Mensagem existente

int msgget(key_t key, int msgflg**)**

- **Retorno:** retorna o msqid a partir de uma chave key já definida, caso o processo tenha permissão para isso. Caso contrário retorna erro (-1)
- **Exemplo**
    - if ((msqid = msgget(0x4321,0) == -1) {
      printf(“a fila não existe”); exit (1);

Envio de Mensagem

int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg)

- Permite a inserção de uma mensagem na fila.
- **Retorno:** 0, caso a mensagem seja inserida na fila de mensagem e -1 (erro)
- A estrutura da mensagem é limitada de duas maneiras:
  - Ela deve ser menor que o limite estabelecido pelo sistema
  - Ela deve respeitar o tipo de dado estabelecido pela **função que receberá a mensagem**
    - Esta função recebe três parâmetros em sua chamada:
      - Identificador da fila **msqid**;
      - Ponteiro **msgp** para a **estrutura de tipo msgbuf** que contém a mensagem a ser enviada
      - Inteiro **msg_sz** indicando o tamanho em bytes da mensagem apontada por msgbuf
      - flag **msgflg** que controla o modo de envio da mensagem.

Envio de Mensagem

int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg)

- **msgflg** pode ser usado da seguinte forma
  - **Se seu valor é igual a zero:** provocará o bloqueio do processo chamando msgsnd quando a fila de mensagens estiver cheia
  - Se tem a flag **IPC_NOWAIT**, a função retorna imediatamente sem enviar a mensagem e com erro igual a -1, indicando que a fila está cheia
- Estrutura **msgbuf**
  - Ela é definida em <sys/msg.h> da seguinte maneira:

```
/* Template for struct to be used as argument for
 * `msgsnd' and `msgrcv'.  */
struct msgbuf
  {
    long int mtype;     /* type of received/sent message */
    char mtext[1];      /* text of the message */
  };
```

Envio de Mensagem

int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg)

Exemplo: msgsnd.c

Recebimento de mensagens

int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg)

- Retira uma mensagem da fila
- Recebe 5 parâmetros
  - **msqid:** identificador da fila
  - **msgp:** estrutura de dados que de uma mensagem
  - **msgsz:** tamanho máximo da mensagem a ser recebida
  - **msgtyp:** indica qual mensagem deve ser recebida
  - **msgflg:** modo de execução da recepção da mensagem

Recebimento de mensagens

int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg)

- msgtyp
  - **= 0**, a primeira mensagem da fila será lida, isto é, a mensagem na cabeça da lista será lida;
  - **> 0**, a primeira mensagem que tiver um valor igual a msgtyp deverá ser retirada da fila
  - **< 0**, a primeira mensagem da fila com o mais baixo valor de tipo que é menor ou igual ao valor absoluto de msgtyp será lida;
- msgflg
  - **IPC_NO_WAIT:** retorna imediatamente um código de erro quando a fila não tiver uma mensagem desejada
    - Se não estiver setada, o processo é bloqueado até que haja mensagem do tipo
  - **MSG_NO_ERROR:** a mensagem é truncada com um tamanho máximo **msgsz bytes**, sendo a parte truncada perdida,
    - Caso não estiver setada, msgrcv retorna erro

Recebimento de mensagens

int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long msgtyp, int msgflg)

Exemplo: **msgrcv.c**

Examinando, Alterando e destruindo uma fila de mensagens

int msgctl (int msqid, int cmd, struct msqid_ds *buf)

- Utilizada para examinar e modificar os atributos de uma fila de mensagens
- Três parâmetros:
  - Identificador da fila de mensagens (msqid);
  - Comando a ser efetuado sobre a fila (cmd);
  - Um ponteiro para uma estrutura do tipo msqid_ds (buf).
- **cmd**
  - **IPC_RMID (0):** Destrói a fila
  - **IPC_SET (1):** Altera valores de variáveis da estrutura **msqid_ds**
  - **IPC_STAT (2):** A estrutura msqid_ds apontada por buf refletirá os valores associados à fila de mensagens.

Examinando, Alterando e destruindo uma fila de mensagens

int msgctl (int msqid, int cmd, struct msqid_ds *buf)

exemplos: msgctl.c e msgctl_destroy.c

Comando no linux: ipcs -q

Memória Compartilhada

- Processos Unix podem compartilhar um segmento de memória protegido pelo kernel
- **Funcionamento:**
  1. Criação do Segmento de memória compartilhada
  2. *Attach* no segmento compartilhado que retorna um **ponteiro para o início da área compartilhada**
  3. Acesso ao segmento através do ponteiro com operações normais de read e write
  4. Remoção explícita do segmento de memória compartilhada é necessário!

Memória Compartilhada - Criação

int shmget(key_t key, int size, int IPC_CREAT | shmflg);

- Cria um segmento de memória compartilhada de tamanho **size** com a **chave key** e as permissões de acesso em **shmflg**.
  - O parâmetro IPC_CREAT determina a criação de segmentos de memória.
  - O uid e gid da memória são o uid e o gid efetivos do usuário dono do processo.
- Retorno:
  - Identificador da memória compartilhada ou erro (-1)

Memória Compartilhada - Attach

char *shmat(int shmid, char *shmaddr, int shmflg)

- mapeia o segmento de memória compartilhada **shmid** no endereço **shmaddr** de seu espaço de endereçamento
  - **shmid** é o descritor de memória compartilhada obtido no shmget
  - **shmaddr** endereço no qual o segmento de memória será mapeado
    - Se shmaddr for 0, o endereço de mapeamento é selecionado pelo sistema
  - **shmflg** determina o modo de acesso à memória compartilhada
    - read_only ou read/write
- Retorno:
  - Endereço do segmento de memória compartilhada ou erro (-1)

Memória Compartilhada - dettach

int shmdt(int shmid);

- Desfaz o mapeamento do segmento de memória compartilhada **shmid**

Retorno: 0, em caso de sucesso, -1 em caso de erro

Memória Compartilhada - Remoção

int shmctl(int shmid, IPC_RMID, struct shmid_ds *buf)

- Remove a segmentação de memória compartilhada **shmid**
- A partir deste momento, nenhum outro processo possui mais acesso a este segmento
- Retorno
  - 0 se for sucesso
  - -1 se for erro

Exemplo: progexsh.c

Semáforos no Unix

- **Semáforos POSIX**
  - Biblioteca semaphore.h (funciona para threads e para processos)
  - Deve-se adicionar os parâmetros **-lpthread -lrt** na compilação
- **Operação Down**
  - int sem_wait(sem_t *sem);
- **Operação UP**
  - int sem_post(sem_t *sem);
- **Inicialização de um semáforo**
  - sem_init(), para processos ou threads
  - sem_open () para comunicação entre processos (IPC)

Semáforos no Unix

- sem_init(sem_t *sem, int pshared, unsigned int value);
  - **sem:** especifica o semáforo a ser inicializado
  - **pshared:** especifica se o novo semáforo é compartilhado entre processos ou entre threads
    - Um valor diferente de zero significa que o semáforo é compartilhado entre processos
    - Um valor igual a zero significa que o semáforo é compartilhado entre threads
  - **Value:** especifica o valor a ser atribuído ao semáforo criado

- sem_destoy(sem_t *mutex);
  - Destrói um semáforo

Exemplo com threads: semthreads.c

Referências

Santos, C. A. S.. As Filas de Mensagens. Disponível em: https://www.dca.ufrn.br/~adelardo/cursos/DCA409/node105.html Acessado em: 19/09/2018

Melo, A. C. M. A.. Notas de aula de Sistemas Operacionais. Disponível em: https://cic.unb.br/~alba/so.htm Acessado em 19/09/2018

The fork() System Call. Disponível em http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html Acessado em 11/09/2019

Mandeep Singh. Difference between fork() and exec(). Disponível em https://www.geeksforgeeks.org/difference-fork-exec/ Acessado em 11/09/2019

Tanenbaum, A. S. e Bos, H.. Sistemas Operacionais Modernos. 4.ed. Pearson/Prentice-Hall. 2016.