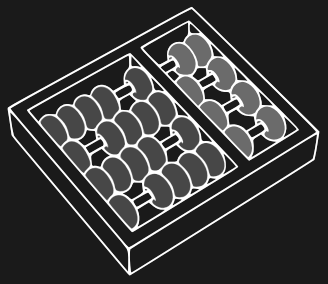


UNICAMP

Sincronização (Parte 4)

Prof. Carlos A. Astudillo

castudillo@ic.unicamp.br

Sistemas Operacionais (MC504A)

2º Semestre 2023

Adaptado de Dave Eckhard

Recapitulando

- Duas formas principais de construir programas com threads
  - **Execução curta de instruções atômicas**
  - Desagendamento (*unscheduling*) voluntaria
- Três requisitos para seções críticas
  - Exclusão mutua
  - Progresso (em relação ao algoritmo de exclusão)
  - Espera acotada
- Algoritmos que não se utilizam para seções críticas
- Implementação das seções críticas com algoritmos que sim se utilizam
  - Mutex (locks)

Mutex

- É uma sequência de instruções atômicas para estabelecer a região crítica
  - Ninguém pode entrelaçar-se na sequência atômica
  - Implementações através do hardware (distinção entre mono e multiprocessador)
- Especificar as sequências que interferem (seção crítica) através de um objeto `mutex`
- Dentro do mutex
  - Condições de corrida
  - Troca atômica, compare and swap, test and set, etc.
  - Divisão em multiprocessadores (load-linked, store-conditional)
  - Os altos e baixos de este tipo de software

Objetivos da aula

- Desagendamento voluntario (*unscheduling*)
- Como? A través de variáveis de condição (condition variables)
  - Ao igual que os mutex, vamos ver que não existe a magia (como se implementam)
  - O problema para dormir de maneira atômica
- Semáforos, monitores —visão general

Desagendamento voluntario

- Qual é a situação?
  - Logramos passar a entrada à seção crítica, e temos o seguro (`lock`)
  - Mas, não necessariamente está no modo correto
  - Outros threads podem corrigir o estado, mas como podem entrar se nós temos o seguro?
  - *Deadlocks*
- Passos a seguir
  - Devemos liberar o recurso compartilhado
  - E devemos garantir que nos acordem ao cumprir-se a condição do recurso
  - Devemos bloquear-nos até que o recurso esteja no estado correto

O que não fazer?

```
// seguramos o recurso e o liberamos num ciclo infinito
    esperando o día do julgamento
while (!reckoning) {
  mutex_lock(&lock);
  if ( date >= 2019-11-17 && hour >= 12 )
    reckoning = true;
  else
    mutex_unlock(&lock);
}
wreak_general_havoc();
mutex_unlock(&lock);
```

Quais são os problemas desta solução?

- Ciclos e esperas inecessárias
  - Não apenas as nossas (nosso thread)
  - Também bloqueamos outros threads
- Em ambiente mono processador?
  - Não tem sentido
- E em multiprocessador?
  - Talvez dependendo da solução
  - Não é generalizável, nem escalável

Solução menos ruim

```
// seguramos o recurso e o liberamos num ciclo infinito
    esperando o día do julgamento
while (!reckoning) {
  mutex_lock(&lock);
  if ( date >= 2019-11-17 && hour >= 12 )
    reckoning = true;
  else {
    mutex_unlock(&lock);
    sleep(1);
  }
}
wreak_general_havoc();
mutex_unlock(&lock);
```

Solução menos ruim, ainda não funciona

- Por que não utilizar `sleep(1)`?
  - Em general $n - 1$ vezes, não será suficiente
  - A n-ésima vez foi demasiado
  - Sempre nos equivocamos, para que usá-la então?
- Mas, qual é o problema?
  - Não queremos esperar por um tempo determinado (porque não sabemos quanto vai ser esse período)
  - Queremos esperar o suficiente: quando a condição mude

Menção honrosa

```
// seguramos o recurso e o liberamos num ciclo infinito
    esperando o día do julgamento
while (!reckoning) {
  mutex_lock(&lock);
  if ( date >= 2019-11-17 && hour >= 12 )
    reckoning = true;
  else {
    mutex_unlock(&lock);
    yield(); // melhor que sleep, entregamos o ciclo de CPU (
        not enough!)
  }
}
wreak_general_havoc();
mutex_unlock(&lock);
```

Nos falta algo

- Resolvemos o estado protegido dos objetos compartilhados
  - Utilizamos um objeto mutex (lock)
  - Encapsulamos o código que interfere com os recursos
  - Mas, em algoritmos que realizam tarefas complementares temos problemas (deadlocks)
- Como resolvemos o problema de bloquear um thread pela duração exata?
  - Obtemos a informação de quem sabe
  - Encapsulemos a "duração correta"
  - Introduzindo: variáveis de condição

Requerimentos das variáveis de condição

- Manter a trilha dos threads que estão bloqueados esperando que certa condição mude
- Permitir que os threads que notificam possam desbloquear a outros threads
- Deve ser seguro ao executar-se entre threads (garantir a exclusão mutua)

Principais métodos

- `condition_wait(lock)`
  - libera o lock atomicamente
  - suspende execução do thread chamante
  - coloca o thread chamante na lista de espera da CV
- `condition_signal()`
  - remove o thread da lista de espera da CV
  - marca o thread como executável (pronto)
  - se não tem threads na lista de espera, não se realiza nada
- `cond_broadcast()`
  - Remove todos os threads da lista de espera da CV
  - marca todos os thread como executáveis (prontos)
  - se não tem threads na lista de espera, não se realiza nada

Assinatura das funções

- `condition_wait(&cv, &mutex);`
- O `mutex` é usado para revisar o estado da região crítica
  - Se encontramos um estado não segurado sabemos que está em processo de mudança
- Quem nos acorde precisará o seguro para poder trabalhar
  - Devemos entregar o `mutex` quando dormimos
- A quem acordemos devemos entregar o `mutex` de volta
  - Deve ser conveniente que `condition_wait()` libere e segure de novo a região crítica

Tentemo-lo uma vez mais

```
mutex_lock(&lock);
while ( cv = wait_on() ) { // aqui está a magia
  cond_wait(&cv, &lock);
}
wreak_general_havoc(); // ainda temos o seguro!
mutex_unlock(&lock);
```

wait_on()

```
if (year < 2019)
  return (&new_year);
else if (month < 11)
  return (&new_month);
else if (day < 17)
  return (&new_day);
else if (hour < 12)
  return (&new_hour);
else
  return (0); // aqui terminamos
```

- Este código é conceitual
- As implementações reais podem variar

Quem nos acorda?

```
for (year = 1900; year < 3000; year++)
  for (month = 1; month <= 12; month++)
    for (day = 1; day <= days(month); day++)
      for (hour = 1; hour <= 24; hour++)
        // avançamos as variáveis
        cond_broadcast(&new_hour);
      cond_broadcast(&new_day);
    cond_broadcast(&new_month);
  cond_broadcast(&new_year);
```

- Este código é conceitual
- Em alguma parte em outro thread executamos o código que muda as condições e nos acordará

Dentro da variável de condição

- `cv->queue`
  - Fila de todos os threads bloqueados
  - FIFO, ou algo mais exótico (depende da implementação)
- `cv->mutex` (`lock` ou similares)
  - Protege a fila contra sinais simultâneas (`wait()`/`signal()`)
  - Não é o mutex dos threads que realizam a chamada, é um mutex interno da variável de condição que protege a fila
  - Está encapsulado dentro da implementação

Implementação

```
// estes parâmetros estão definidos como ponteiros já
condition_wait(cv, mutex){
  lock(cv->mutex);
  enqueue(cv->queue, my_thread_id());
  unlock(mutex); // este é o de fora da VC
  ATOMICALLY {
    unlock(cv->mutex);
    kernel->pause(my_thread_id());
  }
  lock(mutex);
}
```

- O que é ATOMICALLY?
- Lembremos que aconteceu com a implementação dos mutex

O que esperamos?

cond_wait(cv, l)	cond_signal(cv)
enqueue(cv->queue, me);	
unlock(l);	
unlock(cv->lock);	
kernel->pause(me);	
lock(l);	
	lock(cv->lock);
	id = dequeue(cv->queue);
	kernel->wake(id);
	unlock(cv->lock)

O que pode acontecer (execução com problemas)

cond_wait(cv, l)	cond_signal(cv)
`enqueue(cv->queue, me);`	
`unlock(l);`	
`unlock(cv->lock);`	
	`lock(cv->lock);`
	`id = dequeue(cv->queue);`
	`kernel->wake(id);`
	`unlock(cv->lock)`
`kernel->pause(me);`	
`lock(l);`	

- Erro no kernel `ERR_NOT_SLEEP`: `kernel->wake(id)`

Logrando atomicidade no `wait()`

- Regras do jogo
  - Não existe uma primitiva (no hardware) para fazer `unlock_and_block()`
  - Temos o `unlock()`, e `block()`, e outras instruções
  - Fora de `condition_wait()` e de `condition_signal()` devemos parecer que estamos fazendo um `unlock` e `block` de maneira atômica
- Soluções
  - Desabilitar interrupções (se estamos no kernel)
  - Dependemos do sistema operacional para implementar variáveis de condição
    - É uma boa ideia?
  - Ter melhores interfaces de bloqueio de threads
  - Mais mutexes?
  - Varias implementações

Resumindo variáveis de condição

- As variáveis de condição permitem entregar o controle da CPU
  - Operações: `wait` e `signal` (`broadcast`)
- Mantemos (virtualmente) a região crítica
- Importante a implementação em `wait`
  - Precisamos bloquear aos processos dentro da variável de condição
  - Precisamos liberar o mutex interno da variável de condição
  - Devemos fazer ambas atomicamente
  - Como?

Conceito de semáforos

- Um semáforo é outro objeto de encapsulamento
  - Pode produzir exclusão mutua
  - Pode produzir o comportamento de "bloquear até que seja o momento"
- Intuitivamente podemos pensar no semáforo como uma conta de recursos
  - Um inteiro representa o número de recursos disponíveis
    - Número de buffers, número de lugares disponíveis, etc.
    - O semáforo se inicia com algum número específico
  - O thread se bloqueia até que exista uma instância disponível

Operações do semáforo

- `wait()` ou `P()`, em holandês *probeer te verlagen* ou *tentar decrementar*
  - Esperar até que o valor seja maior que cero
  - Depois decrementamos o valor (*tomamos* uma instância do recurso)
- `signal()` ou `V()`, em holandês *verhogen* ou *incrementar*
  - Incrementamos o valor do semáforo (*liberamos* a instância do recurso)
- E novamente `wait()` e `signal()` devem ser atômicas

Semáforo como um mutex

```
semaphore m = 1;

do {
  wait(m); // mutex_lock()
  // seção crítica
  signal(m); // mutex_unlock()
  // outras operações
} while (1);
```

Semáforo como uma condição

Thread 0	Thread 1
	`wait(c);`
`result = 42;`	
`signal(c);`	
	`use(result);`

- Funciona como uma variável de condição

Condição com memória

- Os semáforos têm memória dos eventos que foram avisados (`signal()`)
- Por exemplo, um bit de vazio ou cheio
- A diferença das variáveis de condição (senão estão escutando perdem as sinais)

Thread 0	Thread 1
`result = 42;`	
`signal(c);`	
	`wait(c);`
	`use(result);`

Semáforo vs. Mutex–Condição

- O semáforo é uma construção de alto nível
- Boas noticias
  - Integra a exclusão mutua e a espera
  - Evita erros comuns na API dos mutex–condição
    - Uma `signal()` que chega antes do `wait()` se perde
- Más noticias
  - Integra a exclusão mutua e a espera
    - Alguns semáforos são como um mutex
    - Alguns semáforos são como uma condição
    - Como deve a biblioteca saber o que fazer? Esperar ou não?

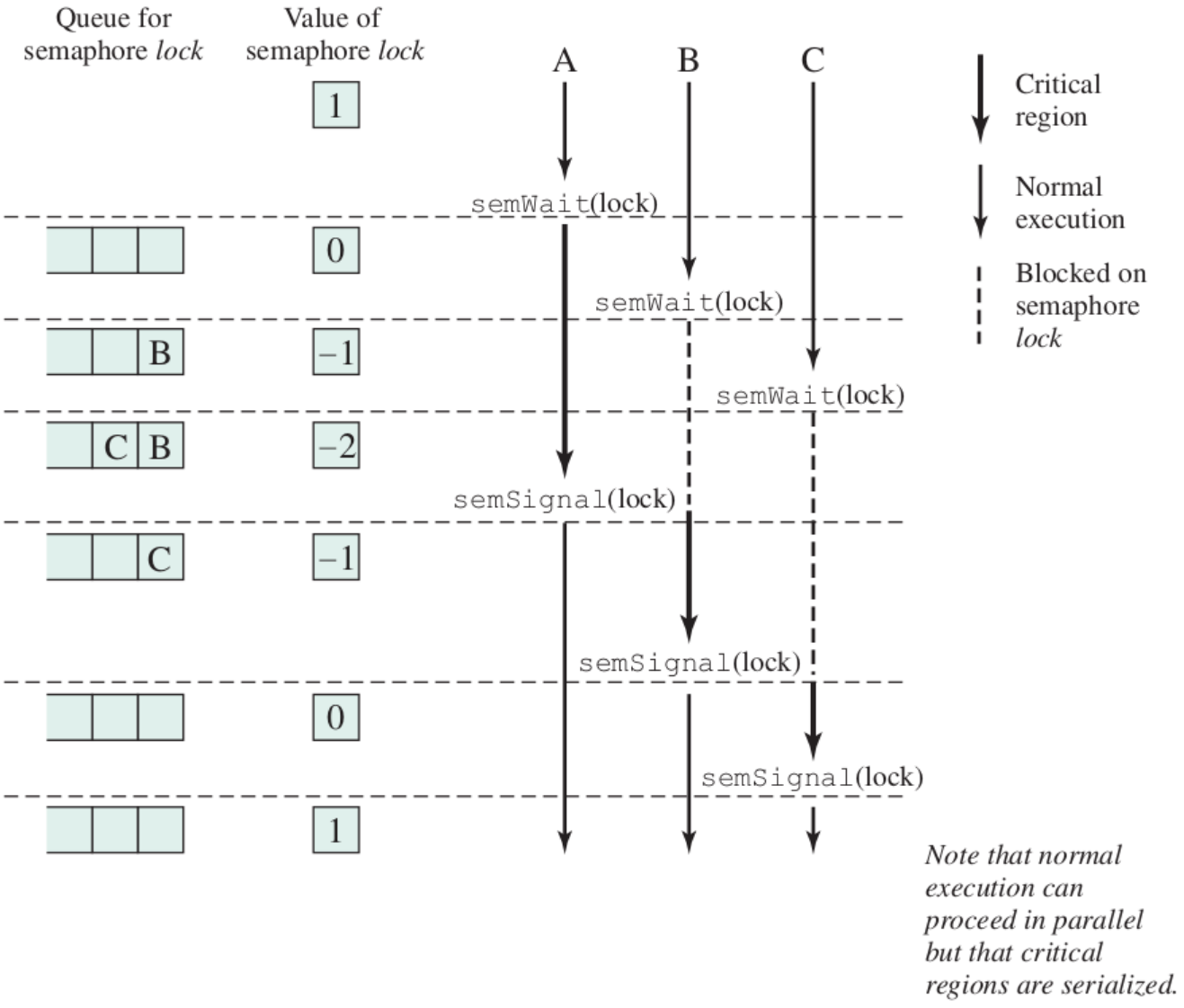
Distintos tipos de semáforos

- Semáforo binário
  - Conta de 0 a 1 (disponível, não disponível)
  - Consideremo-lo como um indicador do quem o implementa (um mutex)
- Semáforos que não se bloqueiam
  - `wait(semaphore, timeout);`
- Semáforos que evitam os deadlocks
  - Vamos vê-los na aula de deadlocks

Primitivas do Semáforo Binário

```
struct binary_semaphore {
        enum {zero, one} value;
        queueType queue;
};
 void semWaitB(binary_semaphore s)
{
        if (s.value == one)
            s.value = zero;
        else {
                    /* place this process in s.queue */;
                    /* block this process */;
        }
}
 void semSignalB(semaphore s)
{
        if (s.queue is empty())
            s.value = one;
        else {
                    /* remove a process P from s.queue */;
                    /* place process P on ready list */;
        }
}
```

Exemplo de semáforos



Avaliação de semáforos

- Pode parecer simples e intuitivo
  - Mas tem demasiadas variações que manter e entender
- Segundo Andrew Tanenbaum: "O bom dos estândares é que têm vários donde escolher"
- Conceptualmente mais simples que ter dois objetos
  - Um para exclusão mutua
  - Um para esperar
  - Depois de que entendemos o que estava acontecendo para garantir a exclusão mutua

wait()

```
wait(semaphore s) {
  // Obtemos acesso exclusivo
  --s->count;
  if (s->count < 0) {
    enqueue(s->queue, my_id());
    ATOMICALLY {
      // liberamos o acesso exclusivo
      thread_block();
    }
  } else
    // liberamos o acesso exclusivo
}
```

signal()

```
signal(semaphore s) {
  // Obtemos acesso exclusivo
  ++s->count;
  if (s->count <= 0) {
    tid = dequeue(s->queue);
    thread_unblock(tid);
  } else
    // liberamos o acesso exclusivo
}
```

- O que temos aqui?
- Um algoritmo de exclusão similar ao mutex, ou
- Um sistema do sistema operacional de desagendamento (para acordar)

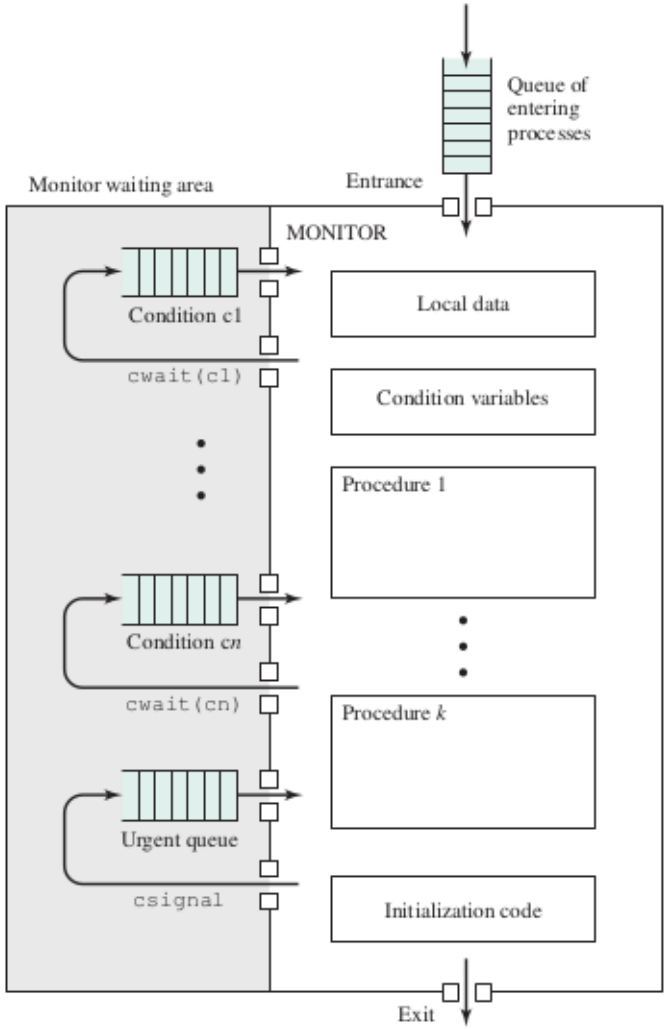
Resumindo semáforos

- Proveem a funcionalidade do mutex e de variáveis de condição num objeto só
- Operações
  - `wait` ou `P`: decrementa o valor
  - `signal` ou `V`: incrementa o valor
- Problemas
  - Faz demasiadas coisas

Conceito básico de monitor

- Problemas existentes
  - O semáforo elimina alguns erros dos mutex–condições
  - Porém, ainda existem erros comuns
    - Trocar `signal()` e `wait()`
    - Acidentalmente omitir algum
  - `signal()` e `wait()` podem aparecer espalhados pelo programa e pode ser difícil ver o efeito nos semáforos que eles afetam
- Uma abstração de alto nível: o monitor
  - É um módulo de linguagens de alto nível
  - Mesma funcionalidade dos semáforos, mas mais facíl de controlar
  - Todos acedem alguns recursos compartilhados
  - O compilador adiciona o código de sincronização
    - Pensem: Java
    - Um thread que se executa num procedimento bloqueia a entrada de todos os outros threads

Estrutura de um Monitor



Exemplo de `monitor` numa loja

```
int cash[STORES] = {0};
int wallet[CUSTOMERS] = {0};

synchronized boolean buy (int cust, int store, int price){
  if (wallet[cust] >= price) {
    cash[store] += price;
    wallet[cust] -= price;
    return (true);
  } else
    return (false);
}
```

E a espera?

- A exclusão mutua automática é boa, mas é forte demais
- As vezes algum thread precisa esperar por outro
  - A exclusão mutua automática não nos permite este comportamento
  - Devemos sair do monitor, e volver a entrar, quando?
- Parece familiar o problema de determinar quando dormir e acordar?

Espera dos monitores

```
synchronized void cash_check(int acc, int chck) {
  while (account[acc].balance < check[chck].value) {
    // tenho que esperar, quanto tempo?
    // como era a solução? :S
  }
  account[acc].balance -= check[chck].value;
}
```

Solução incorreta de espera dos monitores

```
synchronized boolean try_cash_check(int acc, int chck) {
  if (account[acc].balance < check[chck].value) {
    return (false); // não é meu problema, que alguém mais seja
         responsável
  }
  account[acc].balance -= check[chck].value;
  return (true);
}
```

Variáveis de condição de monitor

- Similares às variáveis de condição já vistas em aula
- `condition_wait(cv)`
  - Somente um parâmetro
  - O mutex que temos que compartilhar é implícito (dentro do monitor)
  - Operação
    - Sair temporalmente do monitor —liberamos o mutex
    - Esperamos até que nos enviem uma sinal
    - Re-entramos ao monitor, e seguramos o mutex

Espera do monitor com variáveis de condição

```
void cash_check(int acc, int chck) {
  while (account[acc].balance < check[chck].value)
    condition_wait(account[acc].activity);
  account[acc].balance -= check[chck].value;
}
```

- Quem envia o sinal através de `signal()`?

Variáveis de condição do monitor

- `signal()`: políticas de execução
- Quem se executa depois?
- Quem chamou?, ou o chamado?
- O que acontece com o seguro do monitor? No sinal, saímos de lá (efeito secundário)
- Diferentes políticas implicam distintos tipos de monitor (problemas similares com os semáforos)
  - Hoare: suspende o thread chamante (coloca numa fila), executa um dos thread esperando na fila da cv.
  - Processo que chamou signal() deve sair do monitor imediatamente (signal deve ser a última sentencia do procedimento).
  - Deixar o processo sinalizador (que chamou signal) continuar a execução e permitir que o proceso chamnte (que chamou wait) se execute depois de que o processo sinalizador saia do monitor.

O Problema do Produtor-Consumidor

- Um o mais produtores geram items para ser usados por um consumidor.
- Utiliza um buffer circular de n posições para compartilhar os items.

Bloquear

- Produtor: insertar no buffer cheio.
- Consumidor: remover de buffer vázio.

Desbloquear

- Produtor: item removido.
- Consumidor: item insertado.

O Problema do Produtor-Consumidor

```
producer:
while (true) {
    /* produce item v */
    while ((in + 1) % n == out)
       /* do nothing */;
    b[in] = v;
    in = (in + 1)  %  n;
}
```

```
consumer:
while (true) {
    while (in == out)
       /* do nothing */;
    w = b[out];
    out = (out + 1) % n;
    /* consume item w */;
}
```

O Problema do Produtor-Consumidor

Solução usando mutex-condição

```
#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);
}
```

O Problema do Produtor-Consumidor

Solução usando mutex-condição

```
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);
}
```

O Problema do Produtor-Consumidor

Solução usando Semáforos

```
/* program boundedbuffer */
      const int sizeofbuffer = /* buffer size */;
      semaphore s = 1, n = 0, e = sizeofbuffer;
      void producer()
      {
            while (true) {
                  produce();
                  semWait(e);
                  semWait(s);
                  append();
                  semSignal(s);
                  semSignal(n);
            }
      }
      void consumer()
      {
            while  (true) {
                  semWait(n);
                  semWait(s);
                  take();
                  semSignal(s);
                  semSignal(e);
                  consume();
            }
      }
```

O Problema do Produtor-Consumidor

Solução usando Monitores

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;                 /* condition variables for synchronization */
void append (char x)

{
      if (count == N) cwait(notfull);           /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;
      /* one more item in buffer */
      csignal (notempty);                       /*resume any waiting consumer */
}
void take (char x)
{
      if (count == 0) cwait(notempty);        /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                  /* one fewer item in buffer */
      csignal (notfull);                      /* resume any waiting producer */
}
{                                                                   /* monitor body */
      nextin = 0; nextout = 0; count = 0;             /* buffer initially empty */
}
```

O Problema do Produtor-Consumidor

Solução usando Monitores

```
void producer()
{
      char x;
      while (true) {
      produce(x);
      append(x);
      }
}
void consumer()
{
      char x;
      while (true) {
      take(x);
      consume(x);
      }
}
void main()
{
      parbegin (producer, consumer);
}
```

Pontos importantes

- Dois operações fundamentais
  - Exclusão mutua para sequências que devem ser atômicas
  - Devemos desagendar atomicamente (e acordar depois)
- Mutex–variáveis de condição (estilo de `pthreads` POSIX)
  - Dois objetos
  - Um para cada operação
- Semáforos e monitores
  - Semáforo: um objeto
  - Monitor: objetos gerados transparentemente pelo compilador
  - As mesmas ideas estão encapsuladas

O que lembrar?

- Devem saber
  - Os problemas e os objetivos dos distintos objetos
  - As técnicas que estão por trás das soluções
  - Como se envolve o *design* na solução de problemas?
- Já temos nossas ferramentas para poder sincronizar
  - Ainda temos que resolver o problema dos *deadlocks*

Leituras extra

- http://www.cs.cornell.edu/courses/cs4410/2012fa/papers/commandments.pdf