Sincronização

Parte II – Programação Concorrente

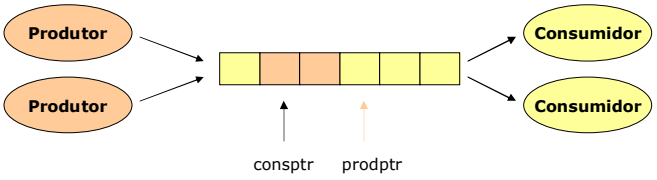
Objectivos da aula de hoje

- Como resolver problemas mais complexos de sincronização
  - Produtores-consumidores, leitores-escritores, jantar dos filósofos
- Como evitar míngua e interblocagem nos nossos programas concorrentes
- Discutir o problema I.A do projecto

Problemas clássicos de sincronização

- O algoritmo dos Produtores/Consumidores
  - tarefas que produzem informação para um buffer e tarefas que lêem a informação do buffer
- O algoritmo dos Leitores/Escritores
  - tarefas que pretendem ler uma estrutura de dados e tarefas que actualizam (escrevem) a mesma estrutura de dados
- O jantar dos filósofos
- E muitos outros
  - Ver aulas 4 e 5 de laboratório, exames antigos e livro

Exemplo de Cooperação entre Processos: Produtor - Consumidor




```
/* ProdutorConsumidor */
int buf[N];
int prodptr=0, consptr=0;
produtor()
{
  while(TRUE) {
    int item = produz();
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
  }
}
```

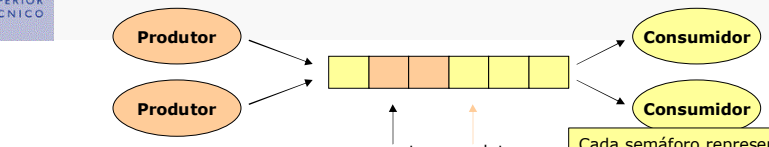
```
consumidor()
{
  while(TRUE) {
    int item;
    item = buf[consptr];
    consptr = (consptr+1) % N;
    consome(item);
  }
}
```

Que acontece se não houver itens no buffer ?

Que acontece se o buffer estiver cheio ?

consptr e prodptr podem ser escritas/lidas concorrentemente… Problema?

Exemplo de Cooperação entre Processos: Produtor - Consumidor

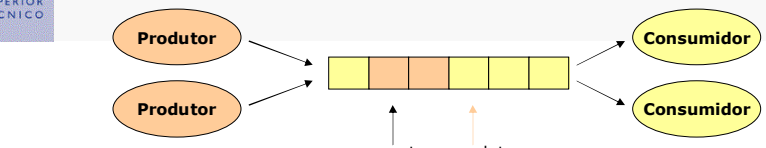



```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco;
semaforo_t pode_prod = criar_semaforo(N),
           pode_cons = criar_semaforo(0);

produtor()
{
  while(TRUE) {
    int item = produz();
    esperar(pode_prod);
    fechar(trinco);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    abrir(trinco);
    assinalar(pode_cons);
  }
}
```

Cada semáforo representa um recurso:
pode_produzir: espaços livres, inicia a N
pode_consumir: itens no buffer, inicia a 0

```
consumidor()
{
  while(TRUE) {
    int item;
    esperar(pode_cons);
    fechar(trinco);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    abrir(trinco);
    assinalar(pode_prod);
    consome(item);
  }
}
```


Exemplo de Cooperação entre Processos: Produtor - Consumidor

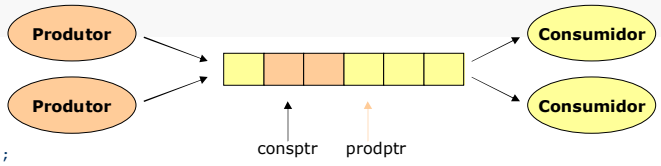



```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco;
semaforo_t pode_prod = criar_semaforo(N),
           pode_cons = criar_semaforo(0);

produtor()
{
  while(TRUE) {
    int item = produz();
    esperar(pode_prod);
    fechar(trinco);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    abrir(trinco);
    assinalar(pode_cons);
  }
}
```

```
consumidor()
{
  while(TRUE) {
    int item;
    fechar(trinco);
    esperar(pode_cons);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    abrir(trinco);
    assinalar(pode_prod);
    consome(item);
  }
}
```

Problema?

Exemplo de Cooperação entre Processos: Produtor - Consumidor

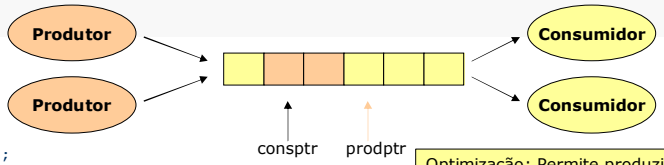



```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco;
semaforo_t pode_prod = criar_semaforo(N),
           pode_cons = criar_semaforo(0);

produtor()
{
  while(TRUE) {
    int item = produz();
    esperar(pode_prod);
    fechar(trinco);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    abrir(trinco);
    assinalar(pode_cons);
  }
}
```

```
consumidor()
{
  while(TRUE) {
    int item;
    esperar(pode_cons);
    fechar(trinco);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    assinalar(pode_prod);
    abrir(trinco);
    consome(item);
  }
}
```

Problema?

Exemplo de Cooperação entre Processos: Produtor - Consumidor




```
int buf[N];
int prodptr=0, consptr=0;
trinco_t trinco_p, trinco_c;
semaforo_t pode_prod = criar_semaforo(N),
           pode_cons = criar_semaforo(0);

produtor()
{
  while(TRUE) {
    int item = produz();
    esperar(pode_prod);
    fechar(trinco_p);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    abrir(trinco_p);
    assinalar(pode_cons);
  }
}
```

Optimização: Permite produzir e consumir ao mesmo tempo em partes diferentes do buffer

```
consumidor()
{
  while(TRUE) {
    int item;
    esperar(pode_cons);
    fechar(trinco_c);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    abrir(trinco_c);
    assinalar(pode_prod);
    consome(item);
  }
}
```

Problema dos Leitores - Escritores

- Pretende-se gerir o acesso a registo partilhado em que existem duas classes de processos:
  - Leitores – apenas lêem a estrutura de dados
  - Escritores – lêem e modificam a estrutura de dados
- Condições
  - Os escritores só podem aceder em exclusão mútua
  - Os leitores podem aceder simultaneamente com outro leitores mas em exclusão mútua com os escritores
  - Nenhuma das classes de processos deve ficar à mingua

Problema dos Leitores - Escritores

```
leitor() {
  while (TRUE) {
    inicia_leitura();
    leitura();
    acaba_leitura();
  }
}

escritor() {
  while (TRUE) {
    inicia_escrita();
    escrita();
    acaba_escrita();
  }
}
```

Problema dos Leitores – Escritores: hipótese 1

```
leitor() {
  while (TRUE) {
    fechar(mutex);
    leitura();
    abrir(mutex);
  }
}

escritor() {
  while (TRUE) {
    fechar(mutex);
    escrita();
    abrir(mutex);
  }
}
```

Demasiado forte!
É possível permitir mais paralelismo!

Leitores – Escritores: Condições de bloqueio mais complexas

- Leitor pode ler em simultâneo com outros leitores, mas não pode ler enquanto alguém escreve
- Escritor não pode escrever se houver alguém a ler ou escrever

Leitores-Escritores: esboço da solução

O que acontece se, após um leitor começar a ler, não pararem de chegar leitores?

Escritores entram em míngua! Solução?

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;

inicia_leitura()
{

  if (em_escrita) {
    leitores_espera++;
    leitores_espera--;

  }
  nleitores++;

}
acaba_leitura()
{

  nleitores--;

}
```

Bloquear até não haver ninguém a escrever

Desbloquear quem esteja à espera para escrever

```
inicia_escrita()
{

  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    escritores_espera--;
  }
  em_escrita = TRUE;

}
acaba_escrita()
{

  em_escrita = FALSE;

}
```

Bloquear até não haver ninguém a escrever ou a ler

Desbloquear quem esteja à espera para ler ou para escrever

Departamento de Engenharia Informática

Leitores-Escritores: esboço da solução

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;

inicia_leitura()
{

  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    leitores_espera--;

  }
  nleitores++;

}
acaba_leitura()
{

  nleitores--;

}
```

Bloquear até não haver ninguém a escrever

Desbloquear quem esteja à espera para escrever

```
inicia_escrita()
{

  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    escritores_espera--;
  }
  em_escrita = TRUE;

}
acaba_escrita()
{

  em_escrita = FALSE;

}
```

Bloquear até não haver ninguém a escrever ou a ler

Desbloquear quem esteja à espera para ler ou para escrever

Leitores-Escritores

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;

inicia_leitura()
{
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;

    esperar(leitores);

    leitores_espera--;


  }
  nleitores++;
}
acaba_leitura()
{
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0)
    assinalar(escritores);
}
```

```
semaforo_t leitores=0, escritores=0;


inicia_escrita()
{
  if (em_escrita || nleitores > 0) {
    escritores_espera++;

    esperar(escritores);

    escritores_espera--;
  }
  em_escrita = TRUE;
}
acaba_escrita()
{
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++)
      assinalar(leitores);
  else if (escritores_espera > 0)
    assinalar(escritores);
}
```

Não existem secções críticas??

Departamento de Engenharia Informática

Leitores-Escritores

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;

inicia_leitura()
{
  fechar(m);
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;

    esperar(leitores);

    leitores_espera--;


  }
  nleitores++;
  abrir(m);
}
acaba_leitura()
{
  fechar(m);
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0)
    assinalar(escritores);
  abrir(m);
}
```

```
semaforo_t leitores=0, escritores=0;
trinco_t m;

inicia_escrita()
{
  fechar(m);
  if (em_escrita || nleitores > 0) {
    escritores_espera++;

    esperar(escritores);

    escritores_espera--;
  }
  em_escrita = TRUE;
  abrir(m);
}
acaba_escrita()
{
  fechar(m);
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++)
      assinalar(leitores);
  else if (escritores_espera > 0)
    assinalar(escritores);
  abrir(m);
}
```

Leitores-Escritores

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;

inicia_leitura()
{
  fechar(m);
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    abrir(m);
    esperar(leitores);
    fechar(m);
    leitores_espera--;

  }
  nleitores++;
  abrir(m);
}
acaba_leitura()
{
  fechar(m);
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0)
    assinalar(escritores);
  abrir(m);
}
```

```
semaforo_t leitores=0, escritores=0;
trinco_t m;

inicia_escrita()
{
  fechar(m);
  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    abrir(m);
    esperar(escritores);
    fechar(m);
    escritores_espera--;
  }
  em_escrita = TRUE;
  abrir(m);
}
acaba_escrita()
{
  fechar(m);
  em_escrita = FALSE;
  if (leitores_espera > 0)
    for (i=0; i<leitores_espera; i++)
      assinalar(leitores);
  else if (escritores_espera > 0)
    assinalar(escritores);
  abrir(m);
}
```

Problema: E se uma nova tarefa obtém acesso antes das tarefas assinaladas?

Leitores-Escritores

```
int nleitores=0;
boolean_t em_escrita=FALSE;
int leitores_espera=0, escritores_espera=0;


inicia_leitura()
{
  fechar(m);
  if (em_escrita || escritores_espera > 0) {
    leitores_espera++;
    abrir(m);
    esperar(leitores);
    fechar(m);
  }
  else
    nleitores++;
  abrir(m);
}
acaba_leitura()
{
  fechar(m);
  nleitores--;
  if (nleitores == 0 && escritores_espera > 0){
    assinalar(escritores);
    em_escrita=TRUE;
    escritores_espera--;
  }
  abrir(m);
}
```

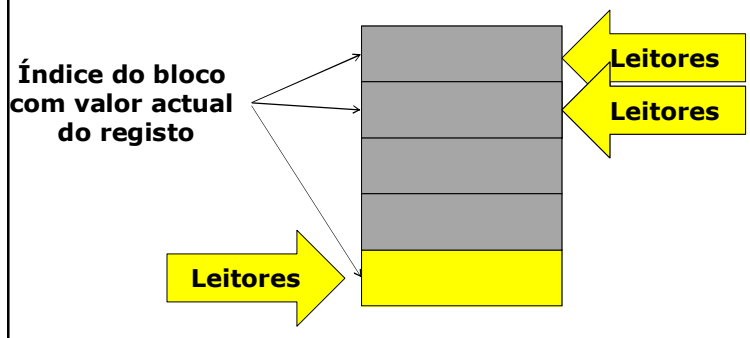
```
semaforo_t leitores=0, escritores=0;
trinco_t m;

inicia_escrita()
{
  fechar(m);
  if (em_escrita || nleitores > 0) {
    escritores_espera++;
    abrir(m);
    esperar(escritores);
    fechar(m);
  }
  else
    em_escrita = TRUE;
  abrir(m);
}
acaba_escrita()
{
  fechar(m);
  em_escrita = FALSE;
  if (leitores_espera > 0) {
    for (i=0; i<leitores_espera; i++) {
      assinalar(leitores);
      nleitores++;
    }
    leitores_espera -= i;
  }
  else if (escritores_espera > 0) {
    assinalar(escritores);
    em_escrita=TRUE;
    escritores_espera--;
  }
  abrir(m);
}
```

Problema?

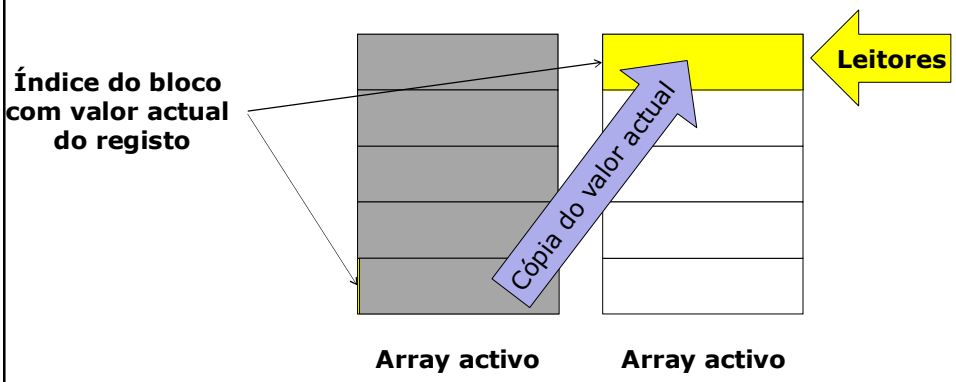
Consultar solução no livro

Possível permitir que escrita progrida em paralelo com leituras?

Hipótese 3: Leitores/escritores baseados em log



Hipótese 3: Leitores/escritores baseados em log



Problema I.A do Projecto

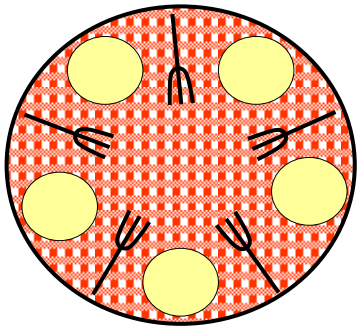
- Resolver leitores/escritores baseados em log
  - Com suporte a múltiplos registos
- Deverão usar a API definida em include/sthreads.h
  - Suporta threads, mutexes, semáforos, trincos leitura/escrita
  - Usando opção de compilação USEPTHREADS, a API é implementada por primitivas das pthreads
  - Mais tarde, tirar a opção USEPTHREADS permitirá usar vossa implementação das mesmas primitivas

Sugestão de estudo autónomo: Exercícios suplementares da aula 5 de laboratório

Jantar dos Filósofos

- Cinco Filósofos estão reunidos para filosofar e jantar spaghetti. Para comer precisam de dois garfos, mas a mesa apenas tem um garfo por pessoa.
- Condições:
  - Os filósofos podem estar em um de três estados : *Pensar*; *Fome*; *Comer*.
  - O lugar de cada filósofo é fixo.
  - Um filósofo apenas pode utilizar os garfos imediatamente à sua esquerda e direita.

Jantar dos Filósofos

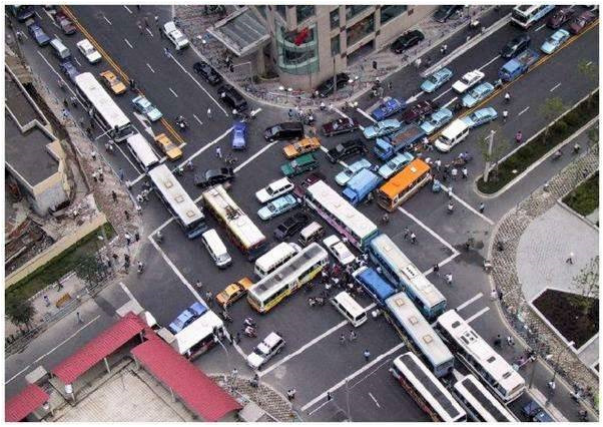


Jantar dos Filósofos com Semáforos, versão #1

```
semaforo_t garfo[5] = {1, 1, 1, 1, 1};

filosofo(int id)
{
    while (TRUE) {
        pensar();
        esperar(garfo[id]);
        esperar(garfo[(id+1)%5]);
        comer();
        assinalar(garfo[id]);
        assinalar(garfo[(id+1)%5]);
    }
}
```

- Problema?

Problema? Interblocagem!



Como prevenir a interblocagem?

Possível solução:

Adquirir os semáforos sempre pela mesma ordem (ordem crescente de número de semáforo)

Jantar dos Filósofos com Semáforos, versão #2

```
semaforo_t garfo[5] = {1, 1, 1, 1, 1};

filosofo(int id)
{
    while (TRUE) {
        pensar();
        if (id == 4) {
            esperar(garfo[(id+1)%5]);
            esperar(garfo[id]);
        } else {
            esperar(garfo[id]);
            esperar(garfo[(id+1)%5]);
        }
        comer();
        assinalar(garfo[id]);
        assinalar(garfo[(id+1)%5]);
    }
}
```

- Solução preventiva genérica para interblocagem
  - Mas em geral é difícil de assegurar

Como prevenir a interblocagem?

Possível solução:

Trancar todos os recursos necessários no início, caso todos estejam disponíveis.

Caso algum não esteja disponível, libertar todos os recursos.

Tentar de novo depois.

Jantar dos Filósofos com Semáforos, versão #3

- Bloquear filósofo com fome caso não haja condições para ele comer
  - Ou seja, parceiros de ambos os lados estarem sem fome
- Usar um semáforo para representar a condição de bloqueio (e não o estado dos garfos/filósofos),
- Representar o estado dos garfos/filósofos com variáveis acedidas numa secção crítica
  - 3 Estados: PENSAR, FOME, COMER

Departamento de Engenharia Informática

Jantar dos Filósofos com Semáforos, versão #3

```
#define PENSAR 0
#define FOME 1
#define COMER 2
#define  N 5
int estado[N] = {0, 0, 0, 0, 0};
semaforo_t semfilo[N] = {0, 0, 0, 0, 0};
trinco mutex;

Testa(int k){
  if (estado[k] == FOME &&
       estado[(k+1)%N] != COMER &&
       estado[(k-1)%N] != COMER){
       estado[k] = COMER;
       assinalar(semfilo[K]);
  }
}
```

```
filosofo(int id)
{
    while (TRUE) {
         pensar();
         fechar(mutex);
         estado[id] = FOME;
         Testa(id);
         abrir(mutex);
         esperar(semfilo[id]);
         comer();
         fechar(mutex);
         estado[id] = PENSAR;
         Testa((id-1+N)%N);
         Testa((id+1)%N);
         abrir(mutex);
     }
}
```

Sistemas Operativos 2011/12

Jantar dos Filósofos com Semáforos, versão #4

```
semaforo_t garfo[5] = {1, 1, 1, 1, 1};
semaforo_t sala = 4;

filosofo(int id)
{
    while (TRUE) {
        pensar();
        esperar(sala);
        esperar(garfo[id]);
        esperar(garfo[(id+1)%5]);
        comer();
        assinalar(garfo[id]);
        assinalar(garfo[(id+1)%5]);
        assinalar(sala);
    }
}
```

- Limitar o acesso à "sala" a N-1 filósofos (fica sempre pelo menos um garfo livre)

Outras soluções para prevenir interblocagem

- Requisitar recursos ao longo execução com chamada não bloqueante "trylock"
  - Libertar todos se algum não está disponível
- Detectar interblocagem entre tarefas/processos
  - Eliminar interblocagem eliminando pelo menos um dos processos/tarefas em contenção
  - Em vez de prevenir interblocagem, resolve-a quando esta acontece

Objectivos desta aula

- Introdução aos Monitores
- Como prevenir interblocagem
- Introdução à gestão de processos e tarefas
  - Tarefas-núcleo vs. Pseudo-tarefas
  - Introdução à gestão de processos no núcleo

Resumo das dicas até ao momento

- Sempre que uma variável pode ser lida e escrita por diferentes tarefas
  - assegurar exclusão mútua no acesso à variável
- Como assegurar exclusão mútua?
  - Trincos lógicos
  - Semáforos inicializados a 1 também são solução de recurso, mas têm pior desempenho e erros de programação são mais difíceis de detectar
- Se houver muitas tarefas que acedem à secção crítica apenas para ler, pode compensar usar trincos R/W

Resumo das dicas até ao momento

- Secções críticas devem ser o mais curtas possível
- Secções críticas não deverão incluir chamadas bloqueantes
  - Sempre que possível, passar chamada bloqueante para antes ou depois da secção crítica
  - Quando isso não é possível:
    - Abrir mutex antes da chamada bloqueante
    - Fechar mutex depois da chamada bloqueante retornar
    - Assegurar que a condição que se queria preservar verdadeira dentro da secção crítica se mantém quando re-entramos na secção crítica

Departamento de Engenharia Informática

Resumo das dicas até ao momento

- Quando um recurso só está simultaneamente disponível para N tarefas, usar semáforo inicializado a N
  - Atenção que acesso a variáveis partilhadas dentro do recurso continua a exigir exclusão mútua! Logo é necessário mutex
- Quando queremos ter uma tarefa a bloquear-se à espera que outra a desbloqueie
  - Usar semáforo a 0
  - Primeira tarefa espera, a outra tarefa assinala

Conseguimos ter tudo isto garantido numa ferramenta única e fácil de usar?

(Se conseguíssemos, a minha avó poderia programar aplicações concorrentes.)

Monitores

Monitores

- Objectivo
  - Mecanismos de sincronização para linguagens de programação que resolvesse a maioria dos problemas de partilha de estruturas de dados:
    - Garantir implicitamente a exclusão mútua
    - Mecanismos para efectuar a sincronização explícita dos processos em algoritmos de cooperação ou de gestão de recursos

Monitores, tão simples como isto:

```
synchronized void meuMetodo () {
   //Aqui estou na secção crítica
   while (condicaoNecessaria == false)
        wait();
   //Aqui estou na secção crítica e a condição verifica-se

   // Quando retornar, liberto a secção crítica
}
```

Caso precise esperar por condição, liberto a secção crítica e bloqueio-me

Monitores - Sincronização

- Exclusão mútua - implícita na entrada no monitor.
  - Tarefa que entre no monitor ganha acesso à secção crítica.
  - Tarefa que sai do monitor liberta a secção crítica.

Departamento de Engenharia Informática

Monitores - Sincronização

- Variáveis condição
  - Declaradas na estrutura de dados
  - Muitas vezes apenas 1 variável por monitor
- Operações:
  - Wait: Liberta a secção crítica. Tarefa é colocada numa fila associada à condição do wait.
  - Signal (ou Notify): assinala a condição. Se existirem tarefas na fila da condição, desbloqueia a primeira.
  - SignalAll (ou NotifyAll): desbloqueia todas as tarefas na fila da condição

Semântica habitual do signal

- Signal desbloqueia uma tarefa da fila associada à condição.
- Mas não liberta a secção crítica.
  - Wait só retorna após a tarefa que se desbloqueou voltar a conseguir entrar na secção crítica
  - Acontece quando?
- Se não existirem tarefas na fila, o efeito perde-se
  - ao contrário dos semáforos, as condições não memorizam os acontecimentos
- Existem outras semânticas

Comparação Semáforos e Monitores

	Semáforos	Monitores
Exclusão Mútua	Mecanismo Básico de Sincronização: *Mutexes* ou Semáforo inicializado a 1	Implícita
Cooperação	Semáforos Inicializados a zero (sem. privados)	Variáveis condição

Monitores: leitores/escritores (em Java)

```
class escritoresLeitores {
    int leitores = 0; int escritores = 0;
     int leitoresEmEspera = 0; int escritoresEmEspera  =  0;

    synchronized void iniciaLeitura () {
       leitoresEmEspera++;
    while (escritoresEmEspera > 0 || escritores > 0) wait();
  leitoresEmEspera--; leitores++;
    }
    synchronized void acabaLeitura () {
       leitores--;
       notifyAll();
    }

    synchronized void iniciaEscrita () {
  escritoresEmEspera++;
  while (leitores > 0 || escritores > 0) wait();
        escritoresEmEspera --;
        escritores ++;
    }

    synchronized void acabaEscrita () {
       escritores--;
       notifyAll();}
    }
```

Esta solução não evita míngua

Monitores: caixa de correio (em Java) (produtores-consumidores)

```
class caixaCorreio {
      int MAX = 10;  int[] tampao = new int[MAX];
      int contador = 0;  int indPor = 0;  int indTirar = 0;

      synchronized void enviar () {
while (contador == MAX) wait();
      tampao[indPor] = mensagem;
indPor++; if (indPor == MAX) indPor = 0; contador ++;
notifyAll();
      }

      synchronized void receber () {
while (contador == 0) wait();
      mensagem = tampao[indTirar] ;
indTirar++; if (indTirar == MAX) indTirar = 0; contador --;
notifyAll();
      }
}
```

Mecanismos Directos de Sincronização

Mecanismos Directos de Sincronização

- Objectivo
  - Suspender temporariamente a execução de subprocessos
- Limitações:
  - A sincronização directa implica o conhecimento do identificador do processo sobre o qual se pretende actuar.
  - Não se pode dar aos programas dos utilizadores a possibilidade de interferirem com outros utilizadores
  - A restrição habitual é apenas permitir o uso de sincronização directa entre processos do mesmo utilizador

Mecanismos Directos de Sincronização

- Funções que actuam directamente sobre o estado dos processos
  - `Suspender (IdProcesso)`
  - `Acordar (IdProcesso)`
- A função de suspensão é também frequentemente utilizada para implementar mecanismos de atraso temporizado que funcionam como uma auto-suspensão
  - `Adormecer (Período)`