Unidade 4 - Sincronização de Processos

4.1 Conceitos Básicos

Conceitos Básicos (1/1)

- Processos cooperantes: podem afetar, ou ser afetados por outros processos
- Acesso concorrente/paralelo a dados partilhados pode afetar a sua consistência
- **exemplo**: acesso concorrente de dois processos A e B a uma variável partilhada v, em que o processo A executa "$v = v + 1$" e o processo B executa "$v = v - 1$"
  - as operações de alto nível "$v = v \pm 1$" traduzem-se nas operações de baixo nível (máquina) "$R = v; R = R \pm 1; v = R$", sendo R um registo da CPU
  - como pode haver mudança de contexto entre instruções-máquina sucessivas, uma operação de alto nível pode não ser realizada completamente antes que outro processo aceda aos mesmos dados, podendo-os tornar inconsistentes:

Instante	Processo	Operação Máquina	Consequência
t_0	A	$R_A = v$	$(R_A = 5)$
t_1	A	$R_A = R_A + 1$	$(R_A = 6)$
t_2	B	$R_B = v$	$(R_B = 5)$
t_3	B	$R_B = R_B - 1$	$(R_B = 4)$
t_4	A	$v = R_A$	$(v = 6)$
t_5	B	$v = R_B$	$(v = 4)$

  - uma situação como esta, em que vários processos modificam os mesmos dados, e o resultado é afetado pela ordem de acesso, é uma **race condition**

Necessários mecanismos que garantam consistência de dados acedidos concorrentemente.

4.2 O Problema da Secção Crítica

O Problema da Secção Crítica (1/2): Caracterização

- Sejam *n* processos, com secções de código, designados por **secções críticas**, onde existe código que não pode ser executado por mais que um processo em simultâneo (e.g. modificação de dados partilhados)
- **Problema da Secção Crítica** (PSC): como desenhar um protocolo que permite a *n* processos cooperantes sincronizarem-se, no acesso a dados partilhados, sem colocarem em causa a consistência destes ?
- Estrutura genérica de um processo no âmbito do PSC:

```
while(TRUE) {
    secção de entrada
        secção crítica
    secção de saída
        secção restante
}
```

  - **secção de entrada**: obtém-se autorização para aceder à **secção crítica**
  - **secção de saída**: devolve-se essa autorização
  - **secção restante**: representa o resto do código do processo
  - a duração indefinida do processo é representada pelo ciclo infinito

O Problema da Secção Crítica (2/2): Requisitos de uma Solução

1. **Exclusão Mútua**: se um processo P_i está a executar dentro da sua **secção crítica**, então nenhum dos outros processos com os quais coopera pode estar a executar dentro das respetivas **secções críticas**

2. **Progresso**: se nenhum processo está a executar dentro da sua **secção crítica** e existem processos que querem entrar nas respetivas **secções críticas**, então apenas os processos que se encontram na **secção de entrada** são candidatos para a seleção dos processos que entram na **secção crítica** (e essa seleção não pode ser adiada indefinidamente)

3. **Espera Limitada**: após um processo ter pedido para entrar na sua **secção crítica** (e antes de ser satisfeito), existe um limite ao nº de vezes que outros processos podem entrar nas suas **secções críticas**

(Assumindo que cada processo está a executar a velocidade não nula, e nada se assumindo sobre a velocidade relativa de cada processo)

Solução de Peterson para 2 Processos (1/6): Algoritmo 1

- **Solução de Peterson**:
  solução clássica baseada em software, para n=2 processos (P_0, P_1)
- **Algoritmo 1** (uma primeira tentativa; ainda não é uma solução)
  - Variáveis partilhadas: `int turn = 0;`
  - Implicações:
    - (`turn == i`) $\Rightarrow$ (P_i pode entrar na sua secção crítica)
    - sendo n=2 então (`i==0` $\Leftrightarrow$ `j==1`) e (`i==1` $\Leftrightarrow$ `j==0`)
  - Processo P_i

```
while(TRUE) {
    while (turn != i) {;}
        secção crítica
    turn = j;
        secção restante
}
```

  - Satisfaz a exclusão mútua e a espera limitada
  - Não satisfaz o progresso: impõe alternância estrita entre P_0 e P_1

Solução de Peterson para 2 Processos (2/6): Algoritmo 1

Algoritmo 1 - demonstração da violação da propriedade do **progresso**:

```
while (1) {
        while (turn != i) ;
        critical section
        turn = j;
        remainder section
}
```

Instant	P0 (i=0, j=1)	P1 (i=1, j=0)	turn (initially 0)
t0	while (turn != i); (False)		0
t1	critical section		0
t2		while (turn != i); (True)	0
t3	critical section		0
t4	turn = 1		1
t5	remainder section		1
t6		while (turn != i); (False)	1
t7		critical section	1
t8		turn = 0	0
t9		remainder section	0
t10		while (turn != i); (True)	0

- No instante t10, P1 é o único processo interessado em aceder à secção crítica, mas não consegue, porque passou a vez a P0 (em t8), o qual ainda se encontra na secção restante. Desta forma, o requisito do progresso é violado.

Solução de Peterson para 2 Processos (3/6): Algoritmo 2

- **Algoritmo 2** (uma segunda tentativa; ainda não é uma solução)
  - Variáveis partilhadas: `boolean flag[2]={FALSE, FALSE};`
  - Implicações:
    - (`flag[i] == TRUE`) $\Rightarrow$ (P_i pronto para entrar na sua secção crítica)
    - sendo n=2 então (`i==0` $\Leftrightarrow$ `j==1`) e (`i==1` $\Leftrightarrow$ `j==0`)
  - Processo P_i

```
while (TRUE) {
    flag[i] = TRUE;
    while (flag[j] == TRUE) {;}
        secção crítica
    flag[i] = FALSE;
        secção restante
}
```

  - Satisfaz a exclusão mútua e a espera limitada
  - Não satisfaz o progresso: se P_i e P_j se encontrarem na *secção de entrada*, podem ficar ambos impedidos de entrar na secção crítica

Solução de Peterson para 2 Processos (4/6): Algoritmo 2

Algoritmo 2 - demonstração da violação da propriedade do **progresso**:

```
while (1) {
        flag[i] = True;
        while (flag[j] == True);
        critical section
        flag[i] = False;
        remainder section
}
```

Instant	P0 (i=0, j=1)	P1 (i=1, j=0)	flag: | False | False |
t0	flag[0] = True		| True | False |
t1		flag[1] = True	| True | True |
t2		while (flag[0] == True); (True)	| True | True |
t3	while (flag[1] == True); (True)		| True | True |

- No instante t2, P1 fica retido na secção de entrada porque verifica que P0 está interessado em entrar na secção crítica. No instante t3, P0 também fica retido na secção de entrada, pois verifica que P1 está interessado em entrar na secção crítica. Sem que nenhum dos processos esteja na secção crítica, e querendo ambos aceder, nenhum deles o consegue. Desta forma, o requisito do progresso é violado.

Solução de Peterson para 2 Processos (5/6): Algoritmo 3

- **Algoritmo 3** (uma aparente solução, mas com problemas (*))
  - Combina as variáveis partilhadas dos Algoritmos 1 e 2
  - Processo P_i

```
while (TRUE) {
    flag[i] = TRUE; // A
    turn = j; // B
    while (flag[j] == TRUE && turn == j) {;}
        secção crítica
    flag[i] == FALSE;
        secção restante
}
```

  - Satisfaz os três requisitos de uma solução do PSC para 2 processos
  - (*) Mas não é uma solução fiável nas arquiteturas atuais:
    - os compiladores podem reordenar instruções independentes, tal como a própria CPU durante a execução (para manter as pipelines cheias); **exemplo**: se B passar a executar antes de A, qual é a consequência ?
    - escritas nas vars. partilhadas `flag` e `turn` não se propagam de forma imediata entre caches de diferentes núcleos: **race conditions** possíveis

Solução de Peterson para 2 Processos (6/6): Algoritmo 3

Algoritmo 3' - demonst. da violação da propriedade da **exclusão mútua**:

```
while (1) {
        turn = j; // B
        flag[i] = True; // A
        while (flag[j] == True && turn == j);
        critical section
        flag[i] = False;
        remainder section
}
```

Instant	P0 (i=0, j=1)	P1 (i=1, j=0)	flag: | False | False |	turn: 0
t0	turn = 1		| False | False |	1
t1		turn = 0	| False | False |	0
t2		flag[1] = True	| False | True |	0
t3		while (flag[0] == True **(F)** && turn == 0 **(T)**); **(F)**	| False | True |	0
t4		critical section	| False | True |	0
t5	flag[0] = True		| True | True |	0
t6	while (flag[1] == True **(T)** && turn == 1 **(F)**); **(F)**		| True | True |	0
t7	critical section		| True | True |	0

- A troca de ordem entre A e B, face ao algoritmo original, criou oportunidade para uma situação em que ambos os processos conseguem estar na secção crítica, violando assim o requisito da exclusão mútua.

4.3 Hardware de Sincronização

Hardware de Sincronização (1/6)

- Soluções para o PSC baseadas em software deixaram de ser viáveis
- Em arquiteturas modernas, são necessárias soluções assistidas pelo Hardware

- Desligar temporariamente as interrupções (ou ignorá-las), equanto se executa o código de uma seccão crítica, permitiria evitar *race conditions*
  - mas esta opção não é escalável em sistemas multi-processadores/multi-núcleo
  - consome tempo pedir a todos os CPU/núcleos que inibam as interrupções, o que atrasa a entrada na secção crítica e diminui a eficiência do sistema
  - e desligar/ignorar as interrupções afeta a atualização do relógio do sistema

- Alternativa: utilizar facilidades/instruções da ISA para resolver o PSC
  - Barreiras/Cercas de Memória
  - Intruções atómicas (Test-and-Set, Compare-and-Swap, etc.)
    - atómicas = indivisíveis (não interrompíveis)

Hardware de Sincronização (2/6): Barreiras de Memória

- **Barreiras de Memória** (Memory Barriers/Fences): instruções que forçam a propagação de mudanças na memória, às caches de todos os núcleos / processadores, garantindo assim uma visão global consistente da memória
  - Asseguram que todos os `Loads` e `Stores` pendentes são finalizados
  - Previnem a reordenação das instruções imediatamente envolventes
  - Impactam negativamente o desempenho (usar com critério; operações de baixo nível, tipicamente usadas por desenvolvedores do kernel)
  - Solução de Peterson (Algoritmo 3) com Barreiras de Memória:

```
while (TRUE) {
    flag[i] = TRUE;
    memory_barrier();
    turn = j;
    while (flag[j] == TRUE && turn == j) {;}
        secção crítica
    flag[i] == FALSE;
        secção restante
}
```

Hardware de Sincronização (3/6): Test-and-Set

- **TestAndSet** (TAS): atribui TRUE ao parâmetro e retorna o valor original

```
boolean TestAndSet (boolean *target) {
    boolean ret = *target;
    *target = TRUE;
    return(ret);
}
```

- PSC com TAS:

```
while(TRUE) {
    while (TestAndSet(lock)) {;}
        secção crítica
    lock = FALSE
        secção restante
}
```

  - Variáveis partilhadas: `boolean lock = FALSE;`
  - Satisfaz exclusão mútua e progresso. Não satisfaz espera limitada.

Hardware de Sincronização (4/6): Compare-and-Swap

- **CompareAndSwap** (CAS): se o 1º parâmetro é igual ao 2º (compare), fica com o valor do 3º (swap); retorna sempre o valor original do 1º parâmetro

```
int CompareAndSwap (int *value, int expected, int new_value) {
    int temp = *value;
    if (*value==expected) *value=new_value;
    return(temp);
}
```

- PSC com CAS:

```
while(TRUE) {
    while (CompareAndSwap(&lock,0,1)) {;}
        secção crítica
    lock = 0
        secção restante
}
```

  - Variáveis partilhadas: `int lock = 0;`
  - Satisfaz exclusão mútua e progresso. Não satisfaz espera limitada.

Hardware de Sincronização (5/6): Compare-and-Swap

- PSC com CAS, satisfazendo os 3 requisitos:
  - Variáveis partilhadas: `boolean waiting[n]={FALSE, ..., FALSE};`
    `boolean lock = 0;`
  - Variáveis privadas: `int key, j; // j=0,1,...,n-1`

```
while (TRUE) {
    waiting[i] = TRUE;
    key = 1;
    while (waiting[i] && key)
            key = CompareAndSwap(&lock, 0, 1);
    waiting[i] = FALSE;
```

secção crítica

```
    j = (i+1) % n;
    while (j != i && !waiting[j])
            j = (j+1) % n;
    if (j == i) lock=0;
    else waiting[j]=FALSE;
```

secção restante

```
}
```

Hardware de Sincronização (6/6): Variáveis Atómicas

- Tipicamente, a instrução CompareAndSwap (CAS) não é usada diretamente, mas antes como bloco construtor de outras ferramentas que lidam c/ o PSC
- As **Variáveis Atómicas** são uma das ferramentas que usam a instrução CAS
  - podem ser usadas para garantir exclusão mútua quando uma só variável partilhada é atualizada (e.g., atualizar um contador partilhado)
  - sistemas que suportam variáveis atómicas oferecem tipos especiais de dados atómicos bem como funções específicas para lidar com eles
  - exemplo: `increment(&counter)` em que o código de `increment` é

```
void increment(atomic_int *v) {
        int temp;
        do {
           temp=*v;
      } while(temp != CompareAndSwap(v, temp, temp+1));
}
```

  - as variáveis atómicas garantem modificações atómicas, mas não resolvem certos problemas: se vários processos testam ativamente (busy waiting) a mesma condição sobre um contador atómico, podem ser todos desbloqueados por outros processos que mudam esse contador

4.4 Trincos e Semáforos

Trincos Mutex (1/2) : Conceito, Operações e Aplicações

- As soluções anteriores, baseadas em instruções-máquina especializadas, são normalmente usadas pelos desenvolvedores de sistemas operativos
- Os programadores de aplicações usam mecanismos de mais alto nível
- **Trincos Mutex**: mecanismo de sincronização simples de alto nível
  - um trinco (*lock*) L assenta numa variável inteira que, após a sua inicialização (0), é apenas acessível através de duas operações atómicas (a atomicidade destas operações é garantida por variáveis atómicas)

```
void acquire (int *L) {
    while (*L == 1) {;}
    *L = 1;
}

void release (int *L) {
    *L = 0;
}
```

  - desta forma, um trinco L oscila apenas entre os valores 0 e 1

Trincos Mutex (2/2) : Conceito, Operações e Aplicações

- Abordagem ao PSC com `n` processos usando Trincos Mutex:
  - Trinco partilhado: `int L = 0;`
  - Processo P_i

```
while (TRUE) {
    acquire(&L);
        secção crítica
    release(&L);
        secção restante
}
```

- Esta abordagem é baseada numa *espera ativa* na operação `acquire`
  - trincos baseados em *espera ativa* são conhecidos por *spinlocks*
  - uma *espera ativa* consome CPU intensivamente; mas, se for de curta duração, pode ser mais eficiente que retirar o processo de execução
- A atomicidade de `acquire` e `release` é garantida com instruções atómicas
- Garante apenas Exclusão Mútua e Progresso; não garante Espera Limitada

Semáforos (1/6) : Conceito, Operações e Aplicações

- Semáforos são ferramentas que permitem formas mais sofisticadas de sincronização entre processos, em comparação com os Trincos Mutex
- Primeira versão de Semáforos, também baseada numa espera ativa:
  - um semáforo S assenta numa variável inteira que, após a sua inicialização, é acessível através de duas operações, atómicas

```
void wait (int *S) {
    while (*S <= 0) {;}
    *S = *S - 1;
}
void signal (int *S) {
    *S = *S + 1;
}
```

  - se S=0,1 (*Semáforo Binário*), comporta-se como um Trinco Mutex
  - se S=0,1,...,R (*Semáforo Contador*), pode controlar o acesso a R unidades de um recurso (consumindo cada processo 1 só unidade)

Semáforos (2/6) : Conceito, Operações e Aplicações

- Abordagem ao PSC com `n` processos usando semáforos c/ espera ativa:
  - Semáforo partilhado: `int S = 1;`
  - Processo P_i

```
while (TRUE) {
    wait(&S);
        secção crítica
    signal(&S);
        secção restante
}
```

  - Neste caso, S é um Semáforo Binário
  - Tal como na abordagem baseada nos Trincos Mutex:
    - hardware de sincronização garante que `wait` e `signal` são atómicos
    - é garantida Exclusão Mútua e Progresso, mas não Espera Limitada

Semáforos (3/6) : Implementação com Espera Passiva

- A operação `wait`, antes definida, tem uma séria desvantagem: assenta numa *espera ativa* (tal como os esquemas de exclusão mútua anteriores)
- Versão alternativa de semáforos baseada em *espera passiva*
  - Além de um contador inteiro (`value`), um semáforo inclui também uma lista (`list`) dos processos bloqueados nesse mesmo semáforo

```
typedef struct {
    int value;
    struct process *list;
} semaphore_t
```

  - Assumem–se também disponíveis duas operações auxiliares:
    - `block()` : suspende o processo invocador (passa ao estado bloqueado)
    - `wakeup(&P)` : move o processo `P` para a fila *ready* (ficando escalonável)

Semáforos (4/6) : Implementação com Espera Passiva

- As operações `wait` e `signal` sobre semáforos passam a ser:

```
void wait (semaphore_t *S) {
    S->value = S->value - 1;
    if (S->value < 0) {
        adicionar o processo invocador a S->list;
        block();
    }
}

void signal (semaphore_t *S) {
    S->value = S->value + 1;
    if (S->value <= 0) {
        remover o primeiro processo P de S->list ;
        wakeup(&P);
    }
}
```

Semáforos (5/6) : Implementação com Espera Passiva

- Nesta implementação, o contador (`value`) pode ser < 0 ; nesse caso, o módulo é o nº de processos em `list` (processos bloqueados no semáforo)
- As operações `wait` e `signal` continuam a ter de ser atómicas
  - sistema mono-processador: basta inibir as interrupções durante essas operações (também aplicável nos mutex e semáforos com espera ativa)
  - sistema multi-processador: inibir interrupções em todas as CPUs é demorado e prejudica o desempenho; usar HW de sincronização ou spinlocks para proteger `wait` e `signal` (vistas como secções críticas)
    - não se eliminam completamente as *esperas ativas* ...
    - mas *esperas ativas* até 10 instruções são rápidas
- Assumindo que a gestão da fila dos semáforos é FIFO, esta abordagem garante Espera Limitada (além de garantir Exclusão Mútua e Progresso)

Semáforos (3/6) : Outras Formas de Sincronização

- A protecção do acesso a secções críticas (garantindo acesso de um só processo - semáforos binários - ou de vários em um número limitado - semáforos contadores-), é apenas um dos usos possíveis dos semáforos.
- Os semáforos são também ferramentas genéricas de sincronização.
  Por exemplo: garantir que P_j executa B só depois de P_i executar A
  - Semáforo partilhado: `int S = 0;`
  - Estrutura dos processos:

P_i :	P_j :
⋮	⋮
A	`wait(&S)`
`signal(&S)`	B

  - Neste caso, qualquer uma das duas versões de semáforos (com/sem espera ativa) pode ser usada para forçar a sincronização pretendida.

Semáforos (6/6) : Deadlocks e Starvation

- Notar que uma implementação de semáforos baseada numa fila de processos bloqueados pode resultar numa situação de *Deadlock* (*i.e.*, num impasse)
  - genericamente: todos os processos de um conjunto aguardam por um evento que só pode ser provocado por outro processo desse conjunto
  - com semáforos: todos os processos bloqueados em semáforos só podem ser desbloqueados por processos também já bloqueados em semáforos
  - exemplo, sendo S e Q dois semáforos inicializados a 1:

P_0	P_1
(1) $wait(S)$;	(2) $wait(Q)$;
(3) $wait(Q)$;	(4) $wait(S)$;
$\vdots$	$\vdots$
$signal(S)$;	$signal(Q)$;
$signal(Q)$;	$signal(S)$;

- *Starvation* (bloqueamento indefinido): ocorre quando um processo nunca é removido da fila de espera do semáforo no qual está suspenso (exemplo: usar LIFO em vez de FIFO como política de remoção de processos da fila)

4.5 Problemas Clássicos de Sincronização

Problemas Clássicos de Sincronização (1/7) : Buffers Limitados

- Acesso concorrente, por dois processos, a um array com `N` buffers
- Um processo *produtor* tenta encher o array
- Um processo *consumidor* tenta esvaziar o array

- Constantes, Tipos e Variáveis partilhadas:

```
#define N ...
typedef ...  item_t ;
item_t buffer[N];
semaphore_t full, empty, mutex;
```

- Inicialização:

```
full->value = 0;
empty->value = N;
mutex->value= 1;
```

Problemas Clássicos de Sincronização (2/7) : Buffers Limitados

- Processo produtor:

```
item_t nextProduced;
int in = 0;

while (TRUE) {
    produceItem(&nextProduced);
    wait(&empty);
    wait(&mutex);
    // seccao critica
    buffer[in] = nextProduced;
    in = (in + 1) % N;
    signal(&mutex);
    signal(&full);
}
```

Problemas Clássicos de Sincronização (3/7) : Buffers Limitados

- Processo consumidor:

```
item_t nextConsumed;
int out = 0;

while (TRUE) {
    wait(&full);
    wait(&mutex);
    // seccao critica
    nextConsumed = buffer[out];
    out = (out + 1) % N;
    signal(&mutex);
    signal(&empty);
    consumeItem(&nextConsumed);
}
```

Prob. Clássicos de Sincronização (4/7) : Múltiplos Leitores e Escritores

- Acesso concorrente, por vários processos, ao mesmo conjunto de dados
- Alguns processos são *leitores* (acessos apenas de leitura)
- Outros processos são *escritores* (acessos de leitura e escrita)
- Com apenas leitores, a coerência é certa; com pelo menos um escritor e outro processo (escritor ou leitor), serão necessários cuidados especiais ...

- Variante *first*: nenhum leitor espera, salvo se um escritor já conseguiu aceder
- Variáveis partilhadas:
```
semaphore_t mutex, wrt;
int readcount;
```
- Inicialização:
```
mutex->value = wrt->value = 1;
readcount=0;
```
- Processo escritor:

```
wait(&wrt);
    /* acesso (escrita / leitura) */
signal(&wrt);
```

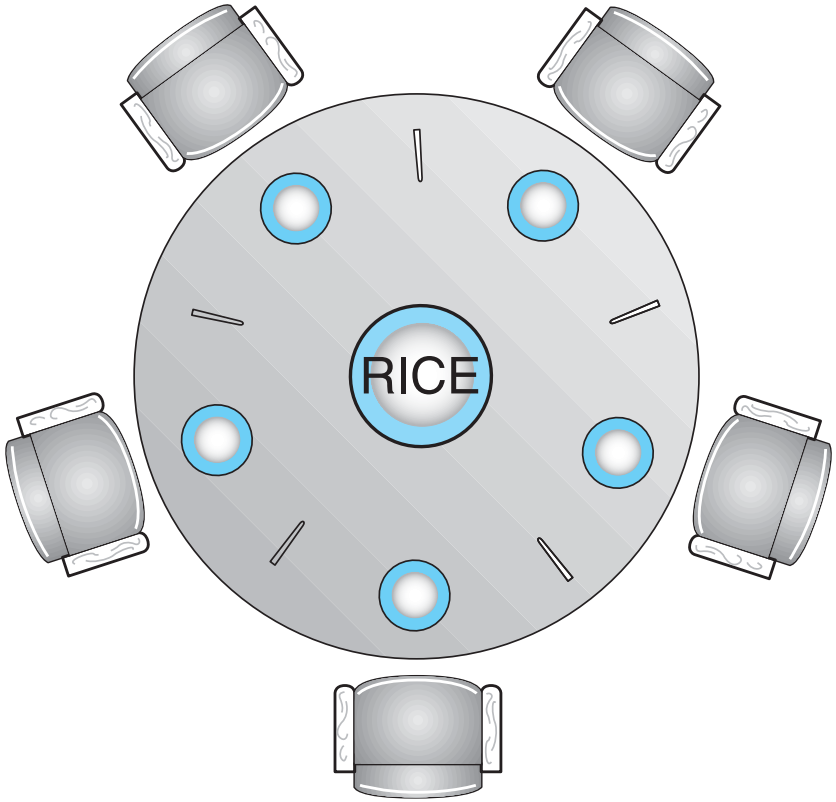
Prob. Clássicos de Sincronização (5/7) : Múltiplos Leitores e Escritores

- Processo leitor :

```
wait(&mutex);
readcount = readcount + 1;
if (readcount == 1) wait(&wrt);
signal(&mutex);
   /* acesso (leitura) */
wait(&mutex);
readcount = readcount - 1;
if (readcount == 0) signal(&wrt);
signal(&mutex);
```

- nenhum escritor consegue aceder enquanto existirem leitores a aceder
- quando um escritor está na secção crítica e há n leitores tentando aceder, um leitor está bloqueado em *wrt* e n-1 leitores estão bloqueados em *mutex*
- quando um escritor sai da secção crítica, pode ser escalonado o leitor bloqueado em *wrt* ou outro escritor também bloqueado em *wrt*

Problemas Clássicos de Sincronização (6/7) : "Filósofos-à-Mesa"



- Variáveis partilhadas: `semaphore_t garfos[5];`
- Inicialização: `for (int i=0; i<5; i++) garfos[i]->value=1;`

Problemas Clássicos de Sincronização (7/7) : "Filósofos-à-Mesa"

- Filósofo *i*:

```
while(TRUE) {
    wait(&garfo[i]);
    wait(&garfo[(i+1)%5]);
        /* comer */
    signal(&garfo[i]);
    signal(&garfo[(i+1)%5]);
        /* pensar */
}
```

- Solução que garante exclusão mútua mas pode gerar *deadlocks* ...
  - exemplo: todos pegam no seu garfo esquerdo ao mesmo tempo
- Alternativas: **a)** permitir, no máximo, 4 filósofos à mesa; **b)** permitir usar os garfos apenas se ambos estiverem livres; **c)** um filósofo par e um ímpar pegam nos garfos pela ordem inversa; e evitar *starvation* ?

4.6 Exercícios

Exercício 4.1: Requisitos do Problema da Secção Crítica

Considere o seguinte algoritmo, executado concorrentemente, numa máquina com um só núcleo de execução, pelos processos P0 e P1, sendo `lock` uma variável inteira (de valor inicial 0) partilhada por P0 e P1. Assuma como não-atómicas (divisíveis) as secções de código de entrada e saída.

```
while (1) {
        while (lock == 1) {;}   // A      // secção de entrada
        lock = 1;               // B
        ...                     // C      // secção crítica
        lock = 0;               // D      // secção de saída
        ...                     // E      // secção restante
}
```

a) Com base na tabela abaixo, apresente uma situação que demonstre que este algoritmo não respeita o requisito da **Exclusão Mútua** de uma solução para o Problema da Secção Crítica com 2 processos. Para cada instante **t+n** (n=0,1,...), preencha cada célula das colunas **P0** e **P1** com uma letra correspondente a uma ação do programa principal (A, B, C ou D), ou deixe-a vazia caso o processo em causa não realize nenhuma ação naquele instante (por cada linha, só pode haver uma ação realizada). Preencha ainda a coluna **lock** com o valor que esta variável assume logo após a execução da ação realizada em cada linha. Nota: a linha do instante **t+0** já está preenchida, e a tabela tem o número mínimo de linhas necessárias para a demonstração; no entanto, caso encontre conveniente, pode acrescentar as linhas que entender necessárias para suportar a sua solução.

Instante	P0	P1	lock
t+0	**A**		**0**
t+1			
t+2			
t+3			
t+4			
t+5			

b) Usando a mesma metodologia, demonstre que a **Espera Limitada** também não é respeitada (assuma que a 1ª linha da tabela está preenchida da mesma forma que em a), e que poderá acrescentar mais linhas para suportar a sua solução).

Exercício 4.2: Semáforos e Deadlocks

Dois processos, A e B, executam concorrentemente uma sequência de 6 ações. Algumas ações já estão definidas, bem como a sua ordem: no processo A, a 2ª e 3ª ações são a invocação das funções f1() e f2(); no processo B, a 3ª e a 4ª ações são a invocação das funções g1() e g2(). As outras ações estão por definir, mas têm de ser operações Wait e Signal sobre semáforos com espera passiva, sendo que para um semáforo S a operação Wait(S) deve sempre preceder a operação Signal(S), podendo haver lugar a outras ações entre essas duas operações.

Processo A	Processo B
f1();	
f2();	g1();
	g2();

a) Assumindo que os processos A e B partilham 2 semáforos X e Y, de valor inicial 1, e que cada processo tem de realizar operações Wait e Signal sobre ambos os semáforos, complete a tabela abaixo com as ações de cada processo ao longo do tempo, de forma a garantir que todas as ações previstas são realizadas (ou seja, não há *deadlock*). Assuma ainda a execução numa máquina com um só núcleo de execução, donde em cada instante só pode ser executada uma ação (ou em A, ou em B). Preencha ainda as duas últimas colunas com o valor do contador dos semáforos logo após a execução de cada ação. Nota: a linha do instante **t+0** já está preenchida, e a tabela tem o número exato de linhas necessárias para a demonstração solicitada.

Instante	Processo A	Processo B	Contador de X	Contador de Y
t+0	Wait(X)		0	1
t+1				
t+2				
t+3				
t+4				
t+5				
t+6				
t+7				
t+8				
t+9				
t+10				
t+11				

b) Usando a mesma metodologia, preencha a tabela abaixo de forma a gerar um cenário de *deadlock* (assuma que a 1ª linha da tabela está preenchida da mesma forma que em a) e que, desta vez, pode não necessitar de preencher todas as linhas da tabela).

REFERÊNCIAS

- "Operating System Concepts, 10th Ed.", Silberschatz & Galvin, Addison-Wesley, 2018: Capítulos 6 e 7
- "Fundamentos de Sistemas Operacionais, 6a Ed.", Silberschatz, Galvin & Gagne, LTC, 2004: Capítulo 7
- "The Weirdest Bug in Programming - Race Conditions" (`https://www.youtube.com/watch?v=bhpzTWtee2A`)
- "The 80's Algorithm to Avoid Race Conditions (and Why It Failed)" (`https://www.youtube.com/watch?v=QAzuAn3nFGo`)