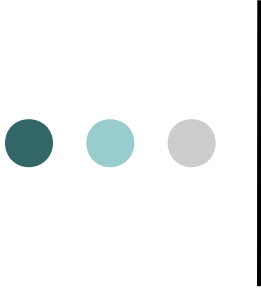


Sincronização de tarefas



Problemas

- Recursos partilhados mas que só podem ser usados por uma tarefa de cada vez.
 - Exemplos: acesso a uma variável global ou ficheiro para escrita; acesso a uma impressora.
- Dados cujo acesso não pode ser feito numa operação “atómica”, isto é, sem ser interrompida para execução de outras tarefas. (ver exemplos 1 e 2)
- Espera/Sinalização de eventos
 - Caso mais típico: Detectar que um recurso passou a estar disponível.
 - Objectivo: eliminar o “busy wait”.



Exemplo 1: tarefas concorrentes

- Uma possível sequência

Tarefa 1 - depósito

1 - Leitura do saldo

2 - Adição do valor depositado

5 - Actualização do saldo

Tarefa 2 - levantamento

3 - Leitura do saldo

4 - Subtracção do levantamento

6 - Actualização do saldo

- Outras possível sequência

Tarefa 1 - depósito

1 - Leitura do saldo

2 - Adição do valor depositado

6 - Actualização do saldo

Tarefa 2 - levantamento

3 - Leitura do saldo

4 - Subtracção do levantamento

5 - Actualização do saldo



Exemplo 2

```
struct {  
    char nome[80];  
    int idade; } pessoa;
```

Thread 1

```
ler nome  
ler idade  
gravar em ficheiro  
ler nome  
--- (zzzz)  
  
ler idade  
gravar em ficheiro  
...
```

Thread 2

```
--- (zzzz)  
  
imprimir nome  
imprimir idade      <==!!!  
--- (zzzz)
```



Sincronização

- Conceitos associados:
 - Zona crítica – conjunto de instruções que não deve ser executada por mais que uma tarefa em simultâneo.
 - Exclusão mútua – garantir que apenas uma tarefa acede a um dado recurso partilhado ou zona crítica de cada vez.



Exemplo

```
data_t ultimo_registro;

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, mythread, NULL);
    while(1) {

        ler_dados(); //Zona crítica

    }
}

void *mythread(void * arg) {
    while(1) {
        sleep(1);

        imprimir_dados(); //Zona crítica

    }
}
```



Mutex – (MUTual EXclusion)

- Mecanismo simples para exclusão mútua
 - Dois estados: trancado/destrancado.
 - Operações: *lock/unlock*
 - Exemplo: sala com uma única chave.
 - Acesso à sala exige pedido da chave (operação *lock*).
 - Se a chave estiver disponível fica com a sala e esta só é libertada quando devolve a chave (*unlock*)
 - Se a chave já foi requisitada, espera que a devolvam.



Mutex

- Permite implementar uma “disciplina”/protocolo de acesso.
 - Por si só não garante o acesso exclusivo: todas as tarefas têm que respeitar a disciplina de acesso.
- Resolve problema dos exemplos 1 e 2.



Mutex

- Principais funções da API:

- Operações:

- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
 - `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
 - `int pthread_mutex_unlock(pthread_mutex_t *mutex);`



Mutex

- Principais funções da API:

- Iniciação/destruição

- `pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;`
 - `int pthread_mutex_init(
pthread_mutex_t *mutex,
const pthread_mutexattr_t *attr);`
 - `int pthread_mutex_destroy(
pthread_mutex_t *mutex);`



Exemplo Mutex

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  
data_t ultimo_registro;
```

```
int main() {  
    pthread_t tid;  
    pthread_create(&tid, NULL, mythread, NULL);  
    while(1) {  
        pthread_mutex_lock(&mutex);  
        ler_dados();  
        pthread_mutex_unlock(&mutex);  
    }  
}
```

```
void *mythread(void * arg) {  
    while(1) {  
        sleep(1);  
        pthread_mutex_lock(&mutex);  
        imprimir_dados();  
        pthread_mutex_unlock(&mutex);  
    }  
}
```



Problemas de sincronização: exemplo 3

“Busy wait”

```
int flag=0;

void produtor() {
    calcular();
    flag = 1;
}

void consumidor() {
    while(flag==0);

    usar();
}
```

Com semáforo

```
pseudo_semaforo sem = ZERO;

void produtor() {
    calcular();
    post(sem);
}

void consumidor() {
    wait(sem);

    usar();
}
```



Semáforos (*semaphore*)

- Funcionam como um contador (0, 1, 2, ...), sobre o qual podem ser realizadas duas operações:
 - (tentar) decrementar
(wait/acquire/pend/consume)
 - Se estiver a 0, aguarda.
 - incrementar
(post/release/signal/produce)



Semáforos - operações

- Decrementar
 - Corresponde a um pedido de autorização para “consumir”/bloquear/aceder a um dado recurso.
 - Operação potencialmente bloqueante
 - Caso o semáforo tenha um valor maior que 0, decrementa o valor do semáforo.
 - Caso o semáforo esteja a zero, a tarefa fica bloqueada na operação, até que o semáforo possa ser decrementado.
- Incrementar
 - Corresponde a devolver/produzir 1 unidade de recurso.
 - Operação não bloqueante



Semáforos vs mutex

- O semáforo é um mecanismo mais flexível do que o mutex.
 - É possível replicar o funcionamento do mutex com um semáforo, mas não o contrário.
- Em vez das operações trancar/destrancar (*lock/unlock*) temos as operações decrementar/incrementar
 - No caso do mutex, o mutex só pode ser “destrancado” (operação *unlock*) pela própria tarefa que fez o *lock*.
 - No caso do semáforo, qualquer tarefa pode, a qualquer altura, incrementar o valor do semáforo (exemplo 3)



Semáforos POSIX

- Semáforo POSIX: variável do tipo `sem_t`
 - Necessário indicar valor inicial (a ver nos slides seguintes)
- Operações
 - `int sem_wait(sem_t * sem)` – operação “decrementar”.
 - `int sem_post(sem_t * sem)` – operação “incrementar”
 - `int sem_trywait(sem_t * sem)` – se o semáforo estiver a 0, retorna erro em vez de bloquear.
 - `int sem_getvalue(sem_t * sem, int * sval)` – obtém o valor actual de um semáforo
- Todas as funções retornam -1 em caso de erro



Semáforos POSIX

- Podemos criar semáforos com e sem nome.
 - Semáforos com nome: através do nome, diferentes programas podem utilizar o semáforo.
 - Semáforos sem nome:
 - Programas multi-thread (semáforo usado apenas num único processo).
 - Para utilização entre processos é necessário usar memória partilhada.
- Linux: `man sem_overview`



Semáforos POSIX (sem nome)

- `int sem_init(sem_t *sem, int pshared, unsigned int value)`
 - Inicia o semáforo.
 - Parâmetro `pshared` indica ao sistema que se pretende partilhar o semáforo entre processos.
 - Para isso, semáforo deverá ser criado em memória partilhada.
- `int sem_destroy(sem_t * sem)`
 - Liberta os recursos ocupados pelo semáforo.
 - Chamada automaticamente quando processo termina



Semáforos POSIX (com nome)

- `sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value)`
 - Abre um semáforo existente (neste caso não são usados os dois últimos parâmetros) ou cria um semáforo (`oflag=O_CREAT`).
- `int sem_close(sem_t *sem)`
 - Fecha a ligação ao semáforo.
- `int sem_unlink(const char *name)`
 - Marca o semáforo para eliminação, logo que não esteja a ser usado por nenhum processo.
 - Os semáforos são automaticamente eliminados quando o sistema é desligado.

Exemplo 1: semáforo sem nome entre threads do mesmo processo

```
sem_t sem; data_t ultimo_registro;

int main() {
    pthread_t tid;
    sem_init(&sem, 0, ?);
    pthread_create(&tid, NULL, mythread, NULL);
    while(1) {
        sem_??;
        ler_dados();
        sem_??;
    }
}

void *mythread(void * arg) {
    while(1) {
        sleep(1);
        sem_??;
        imprimir_dados();
        sem_??;
    }
}
```



Exemplo 1: semáforo sem nome entre threads do mesmo processo

```
sem_t sem; data_t ultimo_registro;

int main() {
    pthread_t tid;
    sem_init(&sem, 0, 1);
    pthread_create(&tid, NULL, mythread, NULL);
    while(1) {
        sem_wait(&sem);
        ler_dados();
        sem_post(&sem);
    }
}

void *mythread(void * arg) {
    while(1) {
        sleep(1);
        sem_wait(&sem);
        imprimir_dados();
        sem_post(&sem);
    }
}
```

21



Exemplo: semáforos com nome

```
int main()
{
    sem_t *my_lock;
    ...
    //iniciado com uma unidade
    my_lock = sem_open ("/mysem", O_CREAT | O_RDWR,
                        0600, 1);
    ...
    sem_wait (my_lock);

    /*Critical Section*/

    sem_post (my_lock);
    ...
}
```



Exemplo: semáforos sem nome partilhados por múltiplos processos

```
...
int main()
{
    sem_t *my_lock; //a variável sem_t deverá estar num
                    //segmento de memória partilhada

    ...
    //permitida partilha entre processos
    //iniciado com uma unidade
    sem_init(my_lock, 1, 1) ;

    ...
    fork() ;

    ...
    sem_wait (my_lock);

    /*Critical Section*/

    sem_post (my_lock);

    ...
}
```



Semáforos System V

- *System V semaphores (semget(2), semop(2), etc.) are an older semaphore API. POSIX semaphores provide a simpler, and better designed interface than System V semaphores; on the other hand POSIX semaphores are less widely available (especially on older systems) than System V semaphores.*
 - semget(), semop(), semctl()
 - As funções operam sobre um conjunto de semáforos.



Sincronização

Tópicos adicionais



Funções não reentrantes

- Funções que não podem ser executadas por várias tarefas em simultâneo.
 - Chamar a função enquanto ela já está a ser executada: “reentrada” na função.
- Exemplo: funções que usam variáveis estáticas/globais.
- Exemplos de funções potencialmente não-reentrantes: `ctime()`, `gethostbyname()`, `rand()`, `strerror()`, `strsignal()`, `strtok()`, `system()`,
- A “Single Unix Specification” indica as funções que deverão ser obrigatoriamente reentrantes.
- Linux: `man 7 pthreads` (Thread-safe functions)



Funções não reentrantes

- Possíveis abordagens:
 - Evitar usar funções não reentrantes em programas com múltiplas threads e nos handlers dos sinais...
 - Ou garantir que as funções não reentrantes usadas não podem ser executadas “simultaneamente” em mais do que um ponto do programa (por exemplo handler de um sinal e programa principal, ou em duas threads):
 - Sinais: bloquear os sinais quando existe risco de interferência entre o handler respectivo e o programa principal.
 - Threads: usar mutexes ou semáforos.



Problemas na sincronização de tarefas

- Aplicações complexas, com mais do que um semáforo, ou com protocolos de troca de mensagens entre várias tarefas.
 - deadlock – dois (ou mais) processos ficam bloqueados, cada um à espera de um recurso que o outro possui.
 - Analogia (versão simplificado do “jantar dos filósofos”, 1965): duas pessoas querem almoçar mas apenas estão disponíveis um garfo e uma faca. Um pega na faca e o outro pega no garfo e ambos aguardam que o outro termine e devolva o talher em falta...



(cont.)

- livelock – dois processos ficam em execução cíclica sem conseguir passar para um estado seguinte.
 - Analogia: duas pessoas face a face a tentar desviar-se uma da outra, sem obedecer a convenções de sentido ...
- Starvation – uma dada tarefa fica à espera de acesso a um recurso e nunca é atendida.



Leitores/escritores

- `pthread_rwlock_init(3)`
 - initialize read-write lock object
- `pthread_rwlock_rdlock(3)`
 - lock read-write lock object for reading
- `pthread_rwlock_wrlock(3)`
 - lock read-write lock object for writing
- `pthread_rwlock_unlock(3)`
 - unlock read-write lock object