Comunicação entre Processos (IPC)

Comunicação entre Processos

IPC- Interprocess communication

 designação de um conjunto de mecanismos através do qual dois ou mais processos comunicam entre si

A comunicação entre processos é suportada por todos os sistemas UNIX. Contudo, diferentes sistemas UNIX implementam diferentes métodos de *IPC*.

O UNIX suporta:

- Para processos correndo na mesma máquina
 - » Pipes e FIFOs
 - » Mensagens
 - » Semáforos
 - » Memória partilhada
- · Para processos correndo em máquinas diferentes
 - » Sockets
 - » TLI Transport Layer Interface
 - » XTI X/Open Transport Interface (baseado em TLI)

Os métodos de *IPC* definidos no standard Posix.1b são mensagens, semáforos e memória partilhada, mas a sua sintaxe é diferente da do System V.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

Comunicação entre Processos (IPC)

Métodos de IPC do UNIX System V

Pipes & FIFOs

 Permitem que processos a correr na mesma máquina troquem dados entre si, através de um pipeline.

Mensagens

 Permitem que processos a correr na mesma máquina troquem dados entre si, através de uma fila de mensagens.

Memória partilhada

 Permite que vários processos a correr na mesma máquina partilhem uma região de memória comum.

Semáforos & Mutexes

• Fornecem mecanismos para que os processos/threads sincronizem as suas acções.



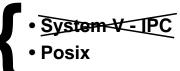
FEUP

MIEIC

Comunicação entre Processos (IPC)

A seguir ...

- Filas de mensagens
- Semáforos
- Memória partilhada





FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

Comunicação entre Processos (IPC)

Métodos POSIX para IPC

Os métodos IPC estão definidos na norma POSIX.1003.1b .

Os métodos POSIX para IPC são os mesmos do System V:

- filas de mensagens
- · semáforos
- memória partilhada

A sintaxe das primitivas POSIX que suportam estes métodos é totalmente diferente das do $System\ V$.

Para compilar programas que usem estas primitivas é necessário *link*ar com a biblioteca *realtime* (librt.a)

gcc prog1.c -o prog1 -lrt -Wall



FEUP

MIEIC

Comunicação entre Processos (IPC)

Filas de mensagens

Primitivas:

```
#include <mqueue.h>
mqd_t mq_open(char* name, int flags, mode_t mode, struct mq_attr* attrp);
     mq_send(mqd_t mqid, const char* msg, size_t len, unsigned priority);
int
     mq_receive(mqd_t mqid, char* buf, size_t len, unsigned* prio);
     mq_close(mqd_t mqid);
int
int
     mq_notify(mqd_t mqid, const struct mq_sigevent* sigvp);
int
     mq_getattr(mqd_t mqid, struct mq_attr* attrp);
     mq_setattr(mqd_t mqid, struct mq_attr* attrp, struct mq_attr* oattrp);
```

Sistemas Operativos

• Estas primitivas só estão disponíveis a partir da versão 2.6.6-rc1 do kernel do Linux, precisando ainda, para funcionamento correcto, de uma user-space library

• O comando uname -r permite saber qual a versão do kernel



FEUP

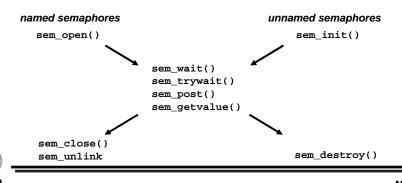
MIEIC

Faculdade de Engenharia da Universidade do Porto

Comunicação entre Processos (IPC)

Semáforos A norma POSIX suporta 2 tipos de semáforos:

- semáforos com nome (named semaphores)
- podem ser partilhados por vários processos
- semáforos sem nome (unnamed semaphores)
 - podem ser partilhados por vários processos que têm acesso a memória comum; para isso, o objecto semáforo tem de ser criado em memória partilhada



FEUP

Comunicação entre Processos (IPC)

Semáforos

Primitivas:

```
#include <semaphore.h>
sem_t* sem_open(char* name, int flags, mode_t mode, unsigned value);
      sem_unlink(char* name);
int
      sem_init(sem_t* sem, int pshared, unsigned value);
int
      sem_close(sem_t* sem);
      sem_destroy(sem_t* sem);
int
     sem_getvalue(sem_t* sem, int* sval);
      sem_wait(sem_t* sem);
int
int
       sem_trywait(sem_t* sem);
      sem_post(sem_t* sem);
int
```

NOTA:



name deve ser da forma /some_name com "/" no início

> MIEIC Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

Comunicação entre Processos (IPC)

Semáforos

PRODUTORcom buffer de



-CONSUMIDOR tamanho 1

#include <stdio.h> #include <stdlib.h> #include <pthread.h> #include <semaphore.h>
#define SHARED 0 /* sem. is shared between threads */ void *Producer(void *); /* the two threads */ /* the global semaphores */ sem_t empty, full; int data: /* shared buffer int numIters; int main(int argc, char *argv[]) { pthread_t pid, cid; printf("Main started.\n"); pthread_create(&pid, NULL, producer, NULL);
pthread_create(&cid, NULL, consumer, NULL); pthread join(pid, NULL); pthread_join(cid, NULL);
printf("Main done.\n"); return 0;

FEUP

Sistemas Operativos Comunicação entre Processos (IPC) /* Put items (1, ..., numIters) into the data buffer and sum them */ void *producer(void *arg) { int total=0, produced; printf("Producer running\n"); for (produced = 1; produced <= numIters; produced++) { sem_wait(&empty);</pre> data = produced; total = total+data; sem_post(&full); printf("Producer: total produced is %d\n",total); return NULL; /* Get values from the data buffer and sum them */ void *consumer(void *arg) { int total = 0, consumed; printf("Consumer running\n"); for (consumed = 1; consumed <= numIters; consumed++) { sem_wait(&full); > gcc pc.c -o pc -lpthread -lrt -Wall > ./pc 20 total = total+data; sem_post(&empty); Main started. Producer running printf("Consumer: total consumed is %d\n",total); Consumer running return NULL; Producer: total produced is 210 Consumer: total consumed is 210 Main done

Sistemas Operativos

FEUP

Comunicação entre Processos (IPC)

Faculdade de Engenharia da Universidade do Porto

Memória partilhada

Primitivas:

```
#include <sys/types.h>
#include <sys/mman.h>
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

NOTA:

- shm_open() não permite especificar o tamanho da região de memória partilhada
- o tamanho tem de ser especificado numa chamada ftruncate() subsequente
- para juntar e retirar a região de mem. partilhada do espaço de endereçamento de um processo é necessário usar as chamadas mmap() e munmap()



MIEIC

MIEIC

Comunicação entre Processos (IPC)

Memória partilhada

```
#include <unistd.h>
#include <sys/types.h>
int ftruncate(int fd, off_t length);

#include <sys/mman.h>
void* mmap(void *start, size_t length, int prot , int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
```



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

#define SHM_SIZE 10
//names should begin with '/'

int shmfd;
char *shm, *s;
sem_t *sem;
int i, n;
long int sum = 0;

char SEM_NAME[] = "/sem1";
char SHM_NAME[] = "/shm1";

Comunicação entre Processos (IPC)

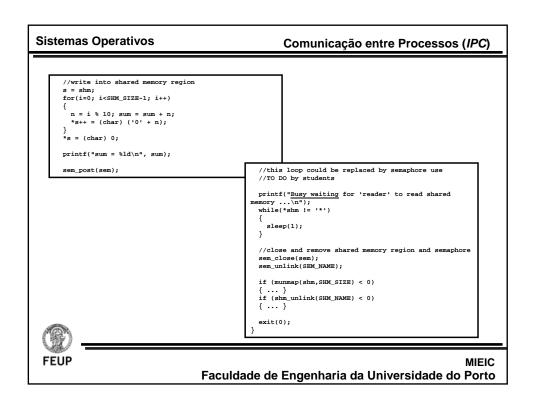
Exemplo

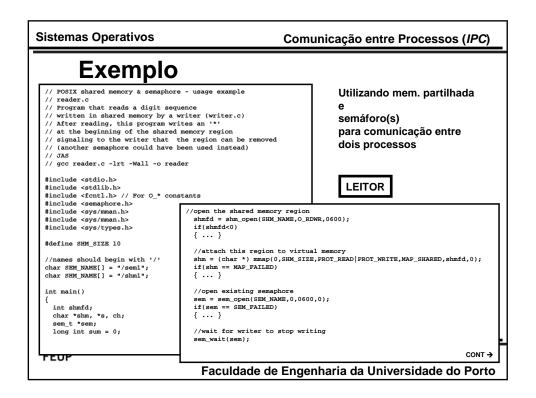
```
// POSIX shared memory & semaphore - usage example
// writer.c
// Program that writes a digit sequence in shared memory
// and waits for a reader (reader.c) to read it
// The reader must write an '*'
// at the beginning of the shared memory region
// for signaling the writer that the shared memory can be removed
// (another semaphore could have been used instead)
// JAS
// gcc writer.c -lrt -Wall -o writer
#include <stdio.h>
#include <stdio.h>
#include <fdntl.h> // For O_* constants
#include <sys/mman.h>
#include <sys/types.h>
if(snmfd<0)
if(snmfd<0)
if(snmfd<0)
if(snmfd<0)
```

Utilizando mem. partilhada e semáforo(s) para comunicação entre dois processos

ESCRITOR

```
//create the shared memory region
shmfd = shm_open(SHM_NAME,O_CREAT|O_RDWR,0600);
if(shmfd<0)
{ ... }
if (ftruncate(shmfd,SHM_SIZE) < 0)
{ ... }
//attach this region to virtual memory
shm = (char *) mmap(0,SHM_SIZE,PROT_READ|PROT_WRITE,MAP_SHARED,shmfd,0);
if(shm == MAP_FAILED)
{ ... }
//create & initialize semaphore
sem = sem_open(SEM_NAME,O_CREAT,0600,0);
if(sem == SEM_FAILED)
{ ... }
```





```
Sistemas Operativos

Comunicação entre Processos (IPC)

//read the message
s = shm;
for (s=shm; *s!=0; s++)
{
ch = *s;
putchar(ch);
sum = sud \n", sum);
//once done, sigmal exiting of reader
//could be replaced by semaphore use (TO DO by students)
*shm = '*';
//close semaphore and unmap shared memory region
sem_close(sem))
if (munmap(shm,SRM_SIZE) < 0)
{ ... }
exit(0);

MIEIC

Faculdade de Engenharia da Universidade do Porto
```

Sincronização de Threads

Sincronização de threads

A sincronização de threads pode ser feita recorrendo a:

- semáforos (ver cap. anterior)
- mutexes
 - podem ser vistos como semáforos inicializados em 1 . servindo fundamentalmente p/ garantir a exclusão mútua de secções críticas
- condition variables (variáveis de condição)
 - permitem que um thread aceda a uma secção crítica apenas quando se verificar uma determinada condição sem necessidade de ficar a ocupar o processador para testar essa condição; enquanto ela não se verificar o thread fica bloqueado

Estes 2 últimos mecanismos de sincronização foram introduzidos pela norma POSIX que definiu a API de utilização de threads.



FEUP

Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

Sincronização de Threads

Mutexes

Sequência típica de utilização de um mutex:

- · Criar e inicializar a variável do mutex.
- · Vários threads tentam trancar (lock) o mutex.
- · Só um deles consegue. Esse passa a ser o dono do mutex.
- O dono do mutex executa as instruções da secção crítica.
- O dono do mutex destranca (unlock) o mutex.
- · Outro thread adquire o mutex e repete o processo.
- Finalmente, o mutex é destruído



Sincronização de Threads

Mutexes - Inicialização

Um mutex é uma variável de tipo pthread_mutex_t .

Antes de poder ser usado, um *mutex* tem de ser inicializado. Há 2 formas alternativas de fazer a inicialização.

Inicialização estática, quando a variável é declarada:

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

Inicialização dinâmica , invocando pthread_mutex_init():

mutx

· apontador p/ a variável que representa o mutex

attr

- permite especificar os atributos do mutex a criar; ver pthread_mutexattr_init()
- se igual a NULL é equivalente a inicialização estática (por omissão)



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

Sincronização de Threads

Mutexes - Lock e Unlock

```
int pthread_mutex_lock (pthread_mutex_t *mutx);
int pthread_mutex_trylock (pthread_mutex_t *mutx);
int pthread_mutex_unlock (pthread_mutex_t *mutx);
int pthread_mutex_destroy (pthread_mutex_t *mutx);
```

pthread_mutex_lock

 tenta adquirir o mutex; se ele já estiver locked, bloqueia o thread que executou a chamada até que o mutex esteja unlocked

pthread_mutex_trylock

- se o mutex ainda não estiver locked, faz o lock
- se o mutex estiver locked, não bloqueia o thread e retorna EBUSY

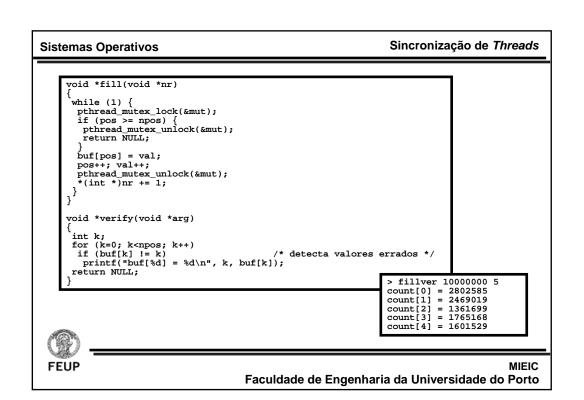
pthread_mutex_unlock

- faz o unlock do mutex
- retorna erro se o mutex já estiver unlocked ou estiver na posse de outro thread (NOTA: lock e unlock de um dado mutex têm de ser feitos pelo mesmo thread)

pthread_mutex_destroy
 destrói o mutex

```
Sistemas Operativos
                                                                                       Sincronização de Threads
              #include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
   M
              #define MAXPOS 10000000 /* nr. max de posições */
#define MAXTHRS 100 /* nr. max de threads */
#define min(a, b) (a)<(b)?(a):(b)</pre>
              int npos;

pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER; /* mutex para a sec.crít. */
int buf[MAXPOS], pos=0, val=0; /* variáveis partilhadas */
    t
              void *fill(void *);
void *verify(void *);
    e
              /* array para contagens */
/* tid's dos threads */
   X
    e
               } npos = min(atoi(argv[1]), MAXPOS); /* nr. efectivo de posições */
nthr = min(atoi(argv[2]), MAXTHRS); /* nr. efectivo de threads */
for (k=0; k<nthr; k++) {
    count[k] = 0; /* criação dos threads fill()</pre>
    S
                                                                  /* criação dos threads fill() */
                 pthread_create(&tidf[k], NULL, fill, &count[k]);
               /* espera pelos threads fill() */
               /* thread-verificador */
 FEUP
                                                                                                                continua
```



Sincronização de Threads

Condition variables

- Mutexes
 - permitem a sincronização no acesso aos dados
 - são usados para trancar (lock) o acesso
- Condition variables
 - permitem a sincronização com base no valor dos dados
 - são usadas para esperar

Sem condition variables,

um programa que quisesse esperar que uma certa condição se verificasse teria de estar continuamente a testar (possivelmente numa secção crítica) o valor da condição (polling), consumindo, assim, tempo de processador.

As condition variables

permitem fazer este teste sem busy-waiting.

Uma condition variable é sempre usada conjuntamente com um mutex.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

Sincronização de Threads

Espera pela condição (x == y) em <u>busy-waiting</u>:

```
while (1) {
  pthread_mutex_lock(&mut);
   if (x == y)
     break;
   pthread_mutex_unlock(&mut);
   SECÇÃO CRÍTICA
pthread_mutex_unlock(&mut);
```

Sincronização de Threads

Espera pela condição (x == y) usando variáveis de condição:

Thread A ... pthread_mutex_lock(&mut); while (x != y) pthread_cond_wait(&var,&mut); /* SECÇÃO CRÍTICA */ pthread_mutex_unlock(&mut); ...

```
Thread B
...
pthread_mutex_lock(&mut);

/* MODIFICA O VALOR DE x E/OU y */
pthread_cond_signal(&var);
pthread_mutex_unlock(&mut);
...
```

Se (x !=y)pthread_cond_wait bloqueia o thread A e simultaneamente (de forma indivisível) liberta o mutex mut.

Quando a thread B sinalizar a variável de condição var,

o thread A é desbloqueado;

pthread_cond_wait() só retorna depois de A ter readquirido o mutex mut, tendo, para isso, de "competir" com outras threads que necessitem do mutex.

Quando a *thread* A readquirir o *mutex* isso não significa que a condição (x==y, neste caso) <u>ainda</u> seja verdadeira. Daí a necessidade do ciclo while, na *thread* A.



FEUP

MIEIC

Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

Sincronização de Threads

Condition variables - Inicialização

Uma condition variable é uma variável de tipo pthread_cond_t .

Antes de poder ser usada, uma *condition variable* tem de ser inicializada e <u>tem de ser criado um *mutex* associado</u>.

Inicialização estática, quando a variável é declarada:

```
pthread_cond_t mycondvar = PTHREAD_COND_INITIALIZER;
```

Inicialização dinâmica , invocando pthread_cond_init():

cvar

apontador p/ a condition variable

attı

- permite especificar os atributos do condition variable a criar;
 ver pthread_condattr_xxx()
 se igual a NULL é equivalente a inicialização estática (por defeito)

FEUP

MIEIC

Sincronização de Threads

Condition variables - wait e signal

```
int pthread_cond_wait (pthread_cond_t *cvar, pthread_mutex_t *mutx);
int pthread_cond_signal (pthread_cond_t *cvar);
int pthread_cond_broadcast (pthread_cond_t *cvar);
int pthread_cond_destroy (pthread_cond_t *cvar);
```

pthread_cond_wait

- bloqueia o thread que fez a chamada até que a condição especificada seja "assinalada"
- deve ser chamada após pthread_mutex_lock()
- durante o bloqueio o mutex é libertado

pthread_cond_signal

 usada para "assinalar" ou "acordar" outro thread (MANUAL: "desbloqueia pelo menos 1 thread") que está à espera da condição

pthread_cond_broadcast

• <u>desbloqueia todos os threads</u> que nesse momento estiverem bloqueados na variável cvar (os threads desbloqueados "lutarão" pela aquisição do mutex de acordo com a política de escalonamento, como se cada um tivesse executado pthread_mutex_lock(); prosseguirá o que obtiver o mutex)

pthread_cond_destroy

· destrói a condition variable



NOTA: pthread_cond_signal e pthread_cond_broadcast não têm qualquer efeito se não houver processos bloqueados em cvar

FEUP

MIEIC Faculdade de Engenharia da Universidade do Porto

Sistemas Operativos

Sincronização de Threads

Condition variables

```
int x = 0, y = 10;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cvar = PTHREAD_COND_INITIALIZER;
```

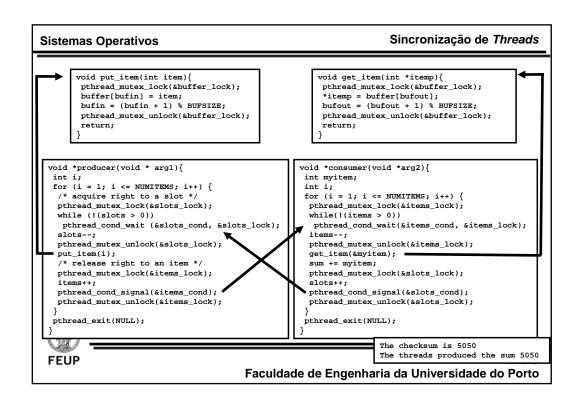
```
void *test(void *a)
{
  while (1) {
    pthread_mutex_lock(&mut);
    while (x != y)
        pthread_cond_wait(&cvar, &mut);
    printf("x = y = %d\n", x);
    x = 0;
    y = y + 10;
    pthread_mutex_unlock(&mut);
}
```

```
void *incr(void *a)
{
  while (1) {
    pthread_mutex_lock(&mut);
    x = x + 1;
    if (x == y)
        pthread_cond_signal(&cvar);
    pthread_mutex_unlock(&mut);
  }
}
```



MIEIC

Sistemas Operativos Sincronização de Threads int main(void){ **Condition** Problema do pthread_t prodtid, constid; int i, total; slots = BUFSIZE; total = 0; PRODUTORvariables -CONSUMIDOR for (i = 1; i <=NUMITEMS; i++) total += i; printf("The checksum is %d\n", total); #include <stdio.h> if (pthread_create(&constid, NULL, consumer, NULL)){ #include <stdlib.h> perror("Could not create consumer"); #include <unistd.h> #include <pthread.h> if (pthread_create(&prodtid, NULL, producer, NULL)){ #define BUFSIZE 8 perror("Could not create producer"); #define NUMITEMS 100 exit(EXIT_FAILURE); int buffer[BUFSIZE]; pthread_join(prodtid, NULL); pthread_join(constid, NULL); printf("The threads produced the sum %d\n", sum); int bufout = 0; int items = 0; exit(EXIT_SUCCESS);//EXIT_SUCCESS e EXIT_FAILURE <- stdlib.h int slots = 0; buffer lock = PTHREAD MUTEX INITIALIZER: pthread cond t slots cond = PTHREAD COND INITIALIZER: pthread_cond_t items_cond = PTHREAD_COND_INITIALIZER; pthread mutex t slots lock = PTHREAD MUTEX INITIALIZER; continua pthread_mutex_t items_lock = PTHREAD_MUTEX_INITIALIZER; **FEUP MIEIC** Faculdade de Engenharia da Universidade do Porto



Sincronização de Threads

Notas finais

- Os mutexes e as condition variables poderão ser partilhados entre processos se forem criados em memória partilhada e inicializados com um atributo em que se inclua a propriedade PTHREAD_PROCESS_SHARED
- Em Linux os threads são implementados através da chamada clone() a qual cria um processo-filho do processo que a invocou partilhando parte do contexto do processo-pai.
- · As funções invocadas num thread têm de ser thread-safe
 - as funções thread-unsafe são, tipicamente, funções não reentrantes que guardam resultados em variáveis partilhadas
 em Unix, algumas chamadas de sistema que são thread-unsafe têm versões thread-safe (têm o mesmo nome acrescido de _r)



FEUP

MIEIC