

Fundamentos de Sistemas de Operação

Unix Windows NT Netware Mac OS DOS/VMS Vax/VMS
Linux Solaris HP/UX AIX Mach Chorus

*Processos, Threads e o SO:
Sincronização*

Comunicar usando um ficheiro

(revisão 1)

```
int main()
{
    ...
    p=fork();
    ...
    if (p) {                                // pai escreve
        fd=open("ficheiro", O_WRONLY...);
        write(fd, "ola", 3);
    } else {                                 // filho lê
        fd=open("ficheiro", O_RDONLY...);
        read(fd, buf, 3);
    }
    ...
}
```

- Que acontece se o filho ler e o pai ainda não escreveu?
- Como garantir que o filho só lê depois do pai escrever?

Comunicar usando um ficheiro (revisão 2)

```
int main()
{
    ...
    p=fork();
    if (!p) {                                // filho escreve... e termina!
        fd=open("ficheiro", O_WRONLY...);
        write(fd, "ola", 3);
    } else {
        wait(NULL);                          // pai espera...
        fd=open("ficheiro", O_RDONLY...);
        read(fd, buf, 3);                  // ... depois lê
    }
}
```

- Assim garantimos que o pai só lê depois do filho escrever...
- Mas só corre um de cada vez ☹ ... má utilização de recursos... CPUs desaproveitados... programa demora mais...

Comunicação vs. Sincronização (revisão)

- Há comunicação entre processos quando
 - Há transferência de informação (bytes) entre os espaços de endereçamento dos processos: há emissor(es) e receptor(es).
- Há sincronização entre processos quando
 - Um processo assinala a outro(s) a ocorrência de um dado evento; por ex., um processo espera que outro termine. O evento acontece quando termina!
- Note-se que há uma certa dualidade entre as duas
 - quando um processo espera por uma mensagem de outro, há simultaneamente sincronização e comunicação...

Sincronização (1)

- Se pretendemos sincronizar processos ou threads sem recorrer à “morte” de um processo, ou de uma thread, as primitivas `wait()`/`waitpid()` ou `pthread_join()` não servem...
- A **espera activa** pode resolver...

```
int inSync= 0;  
  
Thread 1  
...  
<esperar por T2>  
while (!inSync) ;  
...Sincronização conseguida  
...  
  
Thread 2  
...  
...  
...  
inSync= 1;  
...
```



Sincronização (2)

□ E agora?

```
int haDados= 0;  
  
T1, consumidor  
...  
while (1) {  
    ...  
    while (!haDados) ;  
    ...Dados "esvaziados" e processados  
    haDados= 0;  
}  
  
T2, produtor  
...  
while (1) {  
    ...  
    haDados= 1;  
    ...  
}
```

□ A coisa complica-se... (tente prosseguir nesta via)

- Nada impede T_2 de continuar a produzir dados antes de T_1 ter acabado de os processar... receita para um desastre ☺

Comunicação vs. Sincronização

- Os mecanismos apresentados anteriormente (*join*, *lock*) não são suficientes para resolver muitos dos problemas que se colocam
 - São geralmente *ineficientes* (e.g., *espera activa* = *consumo de CPU*)
 - São *inadequados* pois as soluções podem ser *muito complexas* ou *aplicáveis apenas em casos específicos* (conduzindo frequentemente a *erros de programação!*)
- Podemos dizer que são mecanismos “de baixo nível”
 - São “primitivos” e equiparáveis a uma “*linguagem assembly*” quando precisamos de “*linguagens de alto-nível*” como C ou Java ☺ para minimizar os erros que cometemos ao programar...

Enriquecendo a API

□ Variável condicional

- É (*pode ser vista como*) uma fila de espera na qual “se penduram” as *threads* que não podem avançar por não estar satisfeita uma dada condição (*digamos que a condição é falsa*)
- Uma outra *thread* (*não suspensa, naturalmente*) altera a condição
- As *threads* “adormecidas” são acordadas e podem prosseguir. O programador pode em seguida implementar um mecanismo que permita que todas avancem, ou só uma...

Sincronização com VC (1)

- Retomemos o exemplo anterior, re-escrito agora com as nossas funções `myThrWait()` e `myThrSign()` ...

```
int inSync= 0;  
  
Thread 1  
...  
myThrWait(); // esperar por T2  
...Sincronização conseguida  
...  
  
Thread 2  
...  
...  
myThrSign();  
...
```



- No qual as funções `myThrWait()` e `myThrSign()` recorrem a uma variável condicional...

Sincronização com VC (2)

```
int inSync= 0;  
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;  
pthread_cond_t cnd = PTHREAD_COND_INITIALIZER;  
  
void myThrSign() {  
    pthread_mutex_lock(&mtx);  
    inSync= 1;  
    pthread_cond_signal(&cnd);  
    pthread_mutex_unlock(&mtx);  
}  
  
void myThrWait() {  
    pthread_mutex_lock(&mtx);  
    while (!inSync) pthread_cond_wait(&cnd, &mtx);  
    pthread_mutex_unlock(&mtx);  
}
```

Sincronização com VC (3)

□ Como funciona myThrWait() ?

```
void myThrWait() {  
    pthread_mutex_lock(&mtx);  
    while (!inSync) pthread_cond_wait(&cnd, &mtx);  
    pthread_mutex_unlock(&mtx);  
}
```

1. O *lock* tranca o *mutex*
2. Se a condição é falsa, o *wait* “mete” a *thread* na “fila” *cnd* e adormece-a, e **atomicamente** destranca o *mutex* (veremos mais tarde o porquê do teste ser com um *while* em vez dum *if*)
3. Quando a condição for verdadeira, uma ou mais *threads* da fila são acordadas, e é-lhes entregue o controle, sendo que vão competir para conseguir trancar o *mutex*.

Sincronização com VC (4)

□ Como funciona myThrSign() ?

```
void myThrSign() {  
    pthread_mutex_lock(&mtx);  
    inSync= 1;  
    pthread_cond_signal(&cnd);  
    pthread_mutex_unlock(&mtx);  
}
```

1. O lock tranca o mutex
2. A condição **inSync** é tornada verdadeira
3. O signal age sobre a “fila” **cnd** (e sabemos que isso vai acordar -- pelo menos -- uma thread)
4. O mutex tem de ser destrancado

Produtor/Consumidor de 1 item (1)

□ E agora?

```
T1, consumidor
char v;
while (1) {
    consome(&v);
    printf("%c", v);
}
```

```
int haDados= 0;
char dado;
```

```
T2, produtor
...
while (1) {
    c='A';
    produz(c)
}
```

□ Notas:

- *haDados* indica que **dado** tem um valor válido, e portanto o *consumidor* pode consumir esse valor, colocando depois **haDados** a zero
- O *produtor* só pode meter um valor em *dados* se **haDados** está a zero; senão, tem de esperar...

Produtor/Consumidor de 1 item (2)

□ Tentativa 1

```
void consome(char *v) {  
    pthread_mutex_lock(&mtx);  
    while (!haDados) pthread_cond_wait(&cnd, &mtx);  
    *v= dado; haDados= 0;  
    pthread_cond_signal(&cnd);  
    pthread_mutex_unlock(&mtx);  
}  
}
```

□ Notas:

- Se **não haDados** o consumidor bloqueia
- Se há, mete **haDados** a zero e sinaliza o produtor (verdade??)

Produtor/Consumidor de 1 item (3)

□ Tentativa 1

```
void produz(char v) {  
    pthread_mutex_lock(&mtx);  
    while (haDados) pthread_cond_wait(&cnd, &mtx);  
    dado= v; haDados= 1;  
    pthread_cond_signal(&cnd);  
    pthread_mutex_unlock(&mtx);  
}  
}
```

□ Notas:

- Se **haDados** o produtor bloqueia
- Se não há, copia o caracter para **dado**, mete **haDados** a um e sinaliza o consumidor (verdade??)

Demo: Produtor/Consumidor de 1 item

□ Tentativa 1

- *Funcionou?*
 - Então porquê as observações “verdade??” nos slides anteriores?
- *Altere o programa para 2 consumidores (além do produtor).*
- *E agora? Funcionou? (TPC: porquê?)*