

Fundamentos de Sistemas de Operação

Unix Windows NT Netware Macos DOS/VIS Vax/VMS
Linux Solaris HP/UX AIX Mach Chorus

*Processos, Threads e o SO:
Sincronização*

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...

Comunicação e Sincronização

□ Na comunicação

- *Entre processos¹, há transferência de informação (cópia de bytes) entre os espaços de endereçamento dos processos: há emissor(es) e receptor(es);*
- *Entre threads, há cópia de bytes do emissor para uma área (buffer) de recepção partilhado pelas threads.*

□ “Erros” na comunicação

- *No “interior” de um sistema computacional², (por premissa) não há perda ou corrupção de informação, pelo que se algo “se perder ou corromper” à (nossa) má programação se deve ☺*

1: Não estamos a considerar o caso da memória partilhada entre processos (não é tratada em FSO)

2: Não estamos a considerar o caso das redes de computadores

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; char buf[MAX]; // globais

T1, consumidor
...
while (1) {
    ...
    while (!haDados);
    ...Dados "retirados" e processados
    haDados= 0;
}

T2, produtor
...
while (1) {
    ...
    while (haDados);
    ...Dados "inseridos"
    haDados= 1;
}
```

□ Comportamento:

- Não há lugar a “concorrência”: Alternância, “ora eu, ora tu” ☺,
- Espera activa!!! (consumo de CPU e desperdício de energia)
- Funciona? Porquê? ☺

Sincronização (3)

- Imagine várias threads “consumidoras” (T_1, T_2, \dots)

```
int haDados= 0; char buf[MAX]; // globais

T1, T2,... consumidoras                                TP, produtor
...
while (1) {                                              ...
    ...
    while (!haDados);                                 while (haDados);
    ...Dados "retirados" e processados                ...Dados "inseridos"
    haDados= 0;                                         haDados= 1;
}
}
```

- Comportamento:
 - Funciona?
 - Porquê? ☺

Comunicação vs. Sincronização

- Os mecanismos apresentados anteriormente (*wait/pthread_join, pthread_mutex_lock/unlock*) não são suficientes para resolver muitos dos problemas que se colocam
 - 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 tem uma escolha: usar uma primitiva que permite 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

Demo: Sincronização com VC (1)

```
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);  
}
```

Demo: Sincronização com VC (2)

```
void *child(void *arg) {
    printf("child begins and sleeps 5s\n"); sleep(5);
    myThrSign();
    printf("child ends\n");
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;

    printf("parent begins\n");
    Pthread_create(&p, NULL, child, NULL);
    myThrWait();

    printf("parent ends\n");
    return 0;
}
```