Resumos

10 de outubro de 2023 12:14

OSTEP - 5 + PPT - 6

Fork()

- The **nons-determinism**, as it turns out, leads to some interesting problems, particulary in multi-threaded programs. It's we don't know wha write the respectible output first, if is the son or the father.

Adding a wait() call tothe code makes the output **deterministic**, because we know exactly who write first and who wait for that output.

Exec()

- Exec1(), execl(), execlp(), execle, execv(), execvpe();
- This systam call is useful when you want to run a program that is different from the calling program.

⇒ Execvp("nome do comando que se pretende executar", "comando + argumentos necessários ou pedidos pelo utilizador")

- Se quisermos correr outro executável sem ser um comando do Linux escreve-se da seguinte forma → execvp("wc","nome do programa em c (ou numa outra linguagem de programação??)")
- Ex.: execvp("ls", "ls -l") ou execvp("sleep"," sleep 2")

Redirect output:

Prompt> wc pe.c > newfile.txt

- The way the shell accomplishes this task is quite simple: when the child is created, before calling exec(), the shell closes **standard output** and opens the file newfile.txt. By doing so, any output from the soon-to-be-running program wc are sent to the file instead of the screen.

Pipe()

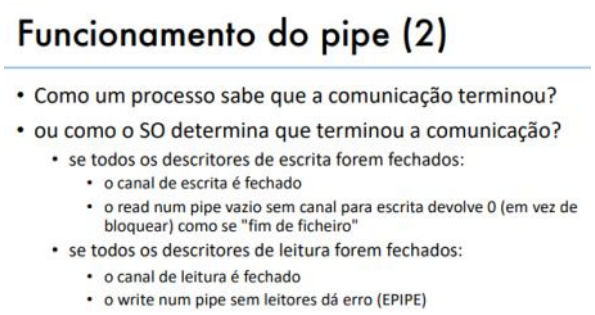
- The way to implement this system call is similar at the fork(), wait(), exec()
- The input of another process is connected to that same pipe; thus, the output of one process seamlessy is used as input to the next, and long useful chains of commandscan be strung together.

IPC - InterProcess Communication ->mecanismos que permitem aos processos 1) sincronizar as suas ações, 2) transferir informação
Existem 2 suportes 1) memória comum, 2) sem partilha de memória
2 modelos de comunicação 1) troca de mensagens, 2) streams de bytes / canais de IO
2) Dificil comunicção entre processos **concorrentes** - sendo pouco eficiente pois envolve escrita e leitura de discos;
Mas abstração de cnaal e ler/escrever sequências ordenadas de bytes é simples e conveniente
Pipes são anónimos, não têm um '*pathname*' associado e apenas são partilhados entre processos pai e filhos
Fifos ou named pipes são pipes com nomes são acessíveis com open()

- Criado com **mkfifo(name, mode)** (ou mknod())
- Acedido através de **open()**

Criação de pipes => return 2 canais fd[] - leitura e fd[1] - escreita; existe limite para a capacidade do pipe

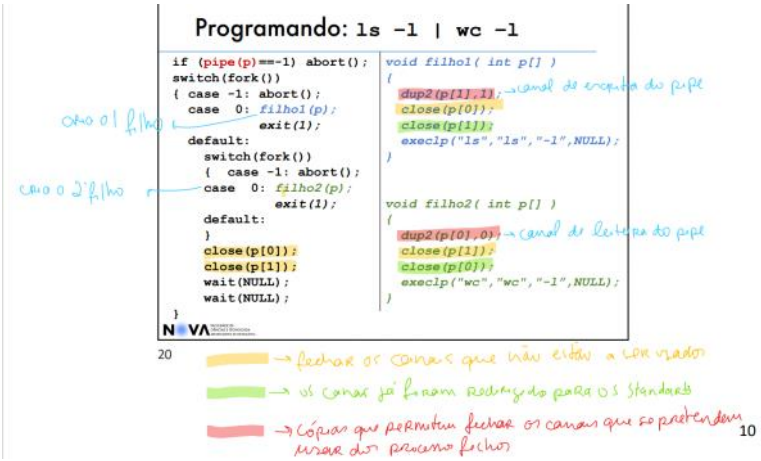
SO garante a comunicação sincronizada entre o pai e o filho



Quando um pipe envia faz close(fd[0]) e quando recebe faz close(fd[1])

Read() devolve os bytes disponíveis, até ao número pedido
Write() escreve apenas os bytes que couberem até alguém ler para poder continuar -> só escretias de menos bytes do que a capacidade do pipe são **atómicas** ou **indivisíveis**

Na redirecionação dos canais para pipes após alterar-se tem que se fechar todos os canais que não são usados
Para redirecionar para canais já abertos tem que se realizar uma cópia desse canal através do dup(int oldfile) ou do dup2(int oldfile, int newfile)



Modern systems include a strong conception of the notion of a **user**. The user, after entering a password to establish credentials, logs in to gain access to system resources. The user may then launch one or many pro esses, and exercise full control over them (pause them, kill them, etc.). Users generally can only control their own processes; it is the job of the operating system to parcel out resources (such as CPU, memory, and disk) to each user (and their processes) to meet overall system goals up.

Kill()

- Can be used to send arvitrary signals to processes, as can the slightly more user friendly killal. Be sure o use these carefully; if you accidently kill your window manager, the computer you are sitting in front of may become quite difficult to use.

SUMMARY

- Each process has a name; in most systems, that name is a number known as a **process ID** (**PID**)
- The **fork()** system call is used in UNIX systems to create a new process. The creator is called the parent; the newly created process is called the child. As sometimes occurs in real life, the child process is nearly identical copy of the parent.
- The **wait()** system call allows a parent to wait for its child to complete execution.
- The **exec()** family of system calls allows a child to break free from its similarity to its parent and execute na entirely new program.
- A UNIX **shell** commonly uses fork(), wait(), and exec() to launch user commands; the separation of fork and exec enables features like **input/output redirection**, **pipes**, and other cool features, all without changin anything about the programs being run.
- Process control is available in the form of **signals**, whcih can cause jobs to stop, continue, or even terminate.
- Which processes can be controlled by a particularperson is encapsulated int the notion of a **user**; operating systems allows multiple users onto the system, and ensures users can only control their own processes.
- A **superuser** can control all processes (and indeed do many other things); this role should be assumed infrequently and with caution for security reasons.

The tool top is also quite helpful, as it displays the processes of the system and how much CPU and other resources they are eating up.

OSTEP - 26 + PPT 7, 8

- **Thread** - abstration for single runnign process (parecido a um processo)
  - PC - contador do programa - rastreia onde o programa está buscando as instruções
  - Conjunto privado de registos
  - TCBs - bloco de controle de thread -> armazena o estado de cada thread
    - Thread-local (armazenamento local) - parâmetros, valores de retorno ...colocados no stack/pilha

    Difference between threads and processes concerns the stack. Each thread has the own stack, but all processes share the same stack

- **Multi-thread** - has more than one point of execution

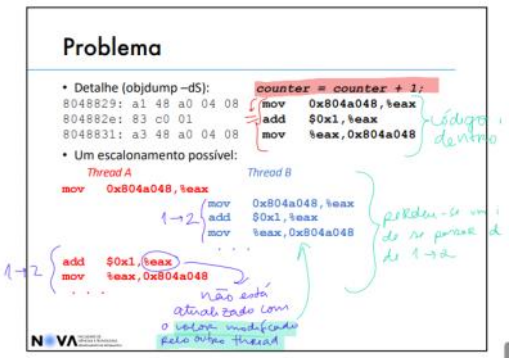
Why use threads?

- **Paralelismo** - tarefa de separar o programa e dirigir cada parte para um CPU (gestão e comando das coisas - multiplos CPUs -> **paralelização**), permitindo, assim, por norma, os programas rodarem mais rápido
- **Evitar o bloqueio do processo do programa devido aos I/O lentos** - evita os travamentos quando se pretende utilizar a CPU para realizar outra(s) operação(ões), tais como: I/O, cálculos..., enquanto se espera de entrada ou saída de dados (I/O) -> **Threading** permite a sobreposição de I/O com outras atividades dentro de um único programa (tal como multiprogramação faz com os processos entre programas)

Os processos são uma escolha mais acertada para tarefas logicamente separadas, onde é necessário pouco compartilhamento de estruturas de dados na memória.

O problema do agendamento não controlado

Data race / race condition acontece quando os resultados dependem do timing de execução do código



A parte do código que resulta numa partilha de variável(share resource) - portanto esta secção do código não deve ser executada por mais que um thread ao mesmo tempo - chama-se critical section (região crítica)

Para resolver esta região críttica queremos fazer uma **mutual exclusion**, esta propriedade garante que se um thread está a executar tal porção de código o(s) outro(s) threads têm que aguardar

- Solução **mutex** (ou **lock**)

SUMMARY

- **Critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- **Race condition** (or **data race**) arises aif multiple threads of execution enter the critical section at roughly the same time; both attempt to update shared data structure, leading to a surprising (and perhaps endesirable) outcome.
- **Indeterminate** program consists of one or more race conditions; the output of th program varies from runt o run, depending on which threads ran when. The outcome is thus nt **determinisc**, something we usually expect from computer system.
- The avoid these problems, threas should use soome kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

OSTEP - 27 + PPT 7, 8

```
#include <pthread.h>
int
pthread_create(pthread_t       *thread,
         const pthread_attr_t *attr,
               void           *(*start_routine)(void*),
               void           *arg);
```

- Thread Creation
  - Arguments:
    1) Pointer to a structure of type *pthread_t* -> use this structure to interact with this threads
    2) Used to specify any attributes this thread might have ??? (por norma os atributos já estão todos certos portanto costuma-se colocar NULL neste parâmetro)
    3) Which function should this thread start running (in C, we call this function **function pointer**); a parte (void *) afirma que a função retorna um valor do tipo void *

If this routine instead required an integer argument, instead of a void pointer, the declaration would look like this:

```
int pthread_create(..., // first two args are the same
                   void *(*start_routine)(int),
                   int  arg);
```

If instead the routine took a void pointer as an argument, but returned an integer, it would look like this:

```
int pthread_create(..., // first two args are the same
                   int (*start_routine)(void *),
                   void *arg);
```

    4) Args to be passed to the funciton where the thread begins execution

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- Thread Completion (Pthread_join())
  - Arguments:
    1) Used to specify which thread to wait for
    2) A pointer to the return value expect to get back
  - Locks:

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

Doing so sets the lock to the default values and thus makes the lock usable. The dynamic way to do it (i.e., at run time) is to make a call to `pthread_mutex_init()`, as follows:

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

    - Arguments:
      1) The addres of the lock itself
      2) (is optional) attributes - NULL in simply uses the defaults
  - Condition Variables
    - Permitem controlar o teste concorrente de condições e aguarda - usadas junto com um mutex para exclusão mútua

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

    - Wait() -> puts the calling thread to sleep, and thus waits for some other thread to signal it

```
pthread_cond_signal(&cond)
pthread_cond_broadcast(&cond)
```

- Signal() - desbloqueia um, eventualmente poderá desbloquear os restantes
- Broadcast() - desbloqueia todos os threads

For example, to compile a simple multi-threaded program, all you have to do is the following:

```
prompt> gcc -o main main.c -Wall -pthread
```

As long as `main.c` includes the pthreads header, you have now successfully compiled a concurrent program. Whether it works or not, as usual, is a different matter entirely.

SUMMARY

- **Keep it simple.** Above all else, any code to lock or signal between threads should be as possible. Tricky thread interactions lead to bugs.
- **Minimize thread interactions.** Try to keep the number of ways in which threads interact to a minimum. Each interaction should be carefully thought out and constructed with tried and true approaches (many of which we will learn about in the coming chapters).
- **Initialize locks and condition variables.** Failure to do so will lead to code that sometimes works and sometimes fails in very strange ways.
- **Check your return codes.** Of course, in any C and Unix programming you do, you should be checking each and every return code and it's ttrue here as well. Failure to do so will lead to bizarre and hard to understand behavior, making you ikely to (a) scream, (b) pull some of your hair out, or © both.
- **Be careful with you pass arguments to, and return values from, threads.** In particular, any time you are passing a reference to a variable allocated on the stack, you are probably doing something wrong.
- **Each thread has its own stack.** As related to the point above, please remember that each thread has its own stack. Thus, if you have a locally-allocated variable inside of some function a thread is executing, it is essentially private to that thread; no other thread can (easily) access it. To share data between threads, the values must be in the **heap** or otherwise some locale that is globally accessible.
- **Always use condition variables to signal between threads.** While it is often tempting to use flag, don't do it.
- **Use the manual pages.** On Linux, in particular, the pthread man pages are highly informative and discuss much of the nuances presented here, often in even ore detail. Read them carefully!

OSTEP - 28.1 28.2 + PPT 8

lock é apenas uma variável, portanto deve ser declarada para ser usada.

```
1  lock_t mutex; // some globally-allocated lock 'mutex'
2  ...
3  lock(&mutex);
4  balance = balance + 1;
5  unlock(&mutex);
```

- Os bloqueios apenas permitem que um thread execute de cada vez

Lock in thread -> mutex (library pthread)

- Pode se usar bloqueios diferentes para proteger variáveis diferentes => aumenta a simultaniedade, muitas vezes protegeremos dados e estruturas de dados diferentes com bloqueios diferentes, permitindo assim que mais threads estejam em código bloqueio de uma só vez (uma abordagem mais **refinada**)

OSTEP - 39.1 39.5 + PPT 3

Armazenamento persistente ou dispositivo de armazenamento de estado sólido mais moderno (**SSD**) armazena informação permanentemente ou por um longo período. Os dados permanecem intactos, não são perdidos, quando há falta de energia.

Duas abstrações importantes:

- **File / arquivo**
  - é uma matriz linear de bytes, cada um dos quais você pode ler ou escrever
  - Cada tem um nome de baixo nível (geralmente um número) => **inode number**
- **Directory / diretorias**
  - Nome de baixo nível => numero inode
  - Conteúdo bastante específico
  - Contem uma lista de pares (nome legível pelo usuário, nome de baixo nível)
  - Cada diretório remete a um arquivo ou a um outro diretório
  - É possivel criar uma árvore mostrando a hierarquia dos diretórios => **directory tree** (**directory hierarchy**)
    - Diretório raiz - /
    - Segunda parte do nome de um ficheiro, geralmente serve para identificar o tipo de ficheiro que este é (ex.: .txt -> ficheiro de escrita/bloco de notas) => convenção, não é obrigatório

Creating files

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
              S_IRUSR|S_IWUSR);
```

- Open():
  - 2nd argument:
    - O_CREAT -> cria o ficheiro caso ele não exista
    - O_WRONLY -> apenas pode escrever
    - O_TRUNC -> caso o ficheiro já existe, coloca o size of zero bytes (counter de onde a página está a zero) para eliminar tudo o que existia lá
  - 3rd argument => especifica as permissões, neste caso torna o arquivo possível de ler e escrever pelo proprietário
  - Return => **file descriptor** (identificador opaco que lhe dá o poder de realizar determinadas operções /ou/ um ponteiro para um objeto do tpo arquivo) -> é um inteiro, privado por processo, e é usado em sistemas UNIX para acessar arquivos; se tiver permissões para ler e escrever pode aceder ao arquivo desde que este esteja criado

```
// option: add second flag to set permissions
int fd = creat("foo");
```

- Creat():
  - Is the same that open() with the next flags: O_CREAT | O_WRONLY | O_TRUNC

Reading and Writing Files

- Ferramenta para rastrear as chamadas ao sistema feitas por um programa -> (LINUX) **strace**
  - Rastreia cada chamada ao sistema feita por um programa enquanto ele é executado e despejar o rastreamento na tela

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)       = 3
read(3, "hello\n", 4096)                = 6
write(1, "hello\n", 6)                  = 6
hello
read(3, "", 4096)                       = 0
close(3)                                = 0
...
prompt>
```

    - File descriptor in this case is 3
    - File descriptor of the screen program (terminal) is 1 (standart output)
    - O cat tenta ler mais do arquivo, mas como já não existe nada no buffer o rea() retorna 0 e o programa sabe que isso significa que leu o arquivo inteiro

Reading and Writing, but not Sequentially

- Lseek() -> permite alterar ou saber a posição corrente
  uma chamada de sistema que é usada para alterar a localização do ponteiro de leitura/gravação de um descritor de arquivo. A localização pode ser definida em termos absolutos ou relativos.

```
off_t lseek(int fildes, off_t offset, int whence);
```

  - Arguments:
    - 1st -> file descriptor
    - 2nd -> offset, o deslocamento do ponteiro (medido em bytes)
    - 3rd -> whence, determina exatamente como a procura é realizada

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
  location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
  the file plus offset bytes.
```

  - Esta função apenas altera uma variável na memória do sistema operacional que rastreia
  - Esta função é utilizada para escrever ou ler dados num sitio especifico do ficheiro, através dessa alteração da variável
  - **Open fle table** -> ???

Let's make this a bit clearer with a few examples. First, let's track a process that opens a file (of size 300 bytes) and reads it by calling the `read()` system call repeatedly, each time reading 100 bytes. Here is a trace of the relevant system calls, along with the values returned by each system call, and the value of the current offset in the open file table for this file access:

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

Second, let's trace a process that opens the *same* file twice and issues a read to each of them.

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

In this example, two file descriptors are allocated (`3` and `4`), and each refers to a *different* entry in the open file table (in this example, entries `10` and `11`, as shown in the table heading; OFT stands for Open File Table). If you trace through what happens, you can see how each current offset is updated independently.

In one final example, a process uses `lseek()` to reposition the current offset before reading; in this case, only a single open file table entry is needed (as with the first example).

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

Here, the `lseek()` call first sets the current offset to `200`. The subsequent `read()` then reads the next 50 bytes, and updates the current offset accordingly.

OSTEP - 30.1 30.2 + PPT 9

Condition Variables

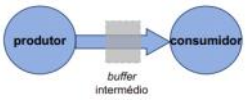
- É uma fila explícita na qual os threads podem se colocar quando algum estado de execução não é desejado (esperando pela condição); algum thread, quando muda o referido estado, pode então despertar um (ou mais) desses threads em espera e, assim, permitir que eles continuem
- Declarar a condição: pthread_cond_t c;
- 2 operações associadas:
  - Wait() -> quando o thread deseja adormecer
    - Necessita de um mutex como parâmetro (assumir que está bloqueado quando wait() é chamado)
    - Responsabilidade é libertar o bloqueio e colocar o thread de chamada em suspensão (atomicamente); quando o thread é ativado, ele deve readquirir o bloqueio antes de retornar ao chamador
  - Signal() -> quando o thread mudou algo no programa e, portanto, deseja despertar um thread adormecido que espera nesta condição

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

- The sleeping, waking and locking all are built around it.
- Embora não seja necessário em todos os casos, é provavelmente mais simles e melhor manter o bloqueio enquanto sinaliza ao usar variáveis de condição. Segure o bloqueio ao chamar o sinal.

Problema do buffer limitado / produtor/consumidor

- Modelo comunicação por uma fila FIFO (limitada).
  - Caso de comunicação entre threads ou processos
- Consideremos Thread Produtor: produz dados que põe na fila
- Thread Consumidor: retira do buffer os dados anteriores



- Levou à invenção do semáforo generalizado

Exemplo:

A bounded buffer is also used when you pipe the output of one program into another, e.g., `grep foo file.txt | wc -l`. This example runs two processes concurrently; `grep` writes lines from `file.txt` with the string `foo` in them to what it thinks is standard output; the UNIX shell redirects the output to what is called a UNIX pipe (created by the **pipe** system call). The other end of this pipe is connected to the standard input of the process `wc`, which simply counts the number of lines in the input stream and prints out the result. Thus, the `grep` process is the producer; the `wc` process is the consumer; between them is an in-kernel bounded buffer; you, in this example, are just the happy user.

- Problem the critical sections

```
1  void *producer(void *arg) {
2      int i;
3      int loops = (int) arg;
4      for (i = 0; i < loops; i++) {
5          put(i);
6      }
7  }
8
9  void *consumer(void *arg) {
10     while (1) {
11         int tmp = get();
12         printf("%d\n", tmp);
13     }
14 }
```

Figure 30.7: **Producer/Consumer Threads (v1)**

- 1st try -> is broken

```
por( T e) {
  while (bufffull())
      ;
  lock(m);
  bufput( e );
  unlock(m);
}
```

```
T tirar() {
  while (bufempty())
      ;
  lock(m);
  T e = bufget();
  unlock(m);
  return e;
}
```

Desperdício de CPU

Só funciona para 1 produtor e 1 consumidor

O while/for não está protegido da concorrência entre threads diferentes

```
1  int loops; // must initialize somewhere...
2  cond_t  cond;
3  mutex_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          Pthread_mutex_lock(&mutex);            // p1
9          if (count == 1)                        // p2
10             Pthread_cond_wait(&cond, &mutex);  // p3
11         put(i);                                // p4
12         Pthread_cond_signal(&cond);            // p5
13         Pthread_mutex_unlock(&mutex);          // p6
14     }
15 }
16
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex);            // c1
21         if (count == 0)                        // c2
22             Pthread_cond_wait(&cond, &mutex);  // c3
23         int tmp = get();                       // c4
24         Pthread_cond_signal(&cond);            // c5
25         Pthread_mutex_unlock(&mutex);          // c6
26         printf("%d\n", tmp);
27     }
28 }
```

Figure 30.8: **Producer/Consumer: Single CV And If Statement**

- 2ndd try ->

```
por( T e) {
  lock(m);
  while (bufffull())
      ;
  bufput( e );
  unlock(m);
}
```

```
T tirar() {
  lock(m);
  while (bufempty())
      ;
  T e = bufget();
  unlock(m);
  return e;
}
```

Desperdício de CPU

Não funciona. Deadlock

Fica eternamente à espera -> exclusão mútua => deadlock

```c
int loops;
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        while (count == 1)                     // p2
            Pthread_cond_wait(&cond, &mutex);  // p3
        put(i);                                // p4
        Pthread_cond_signal(&cond);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            Pthread_cond_wait(&cond, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&cond);            // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

Figure 30.10: **Producer/Consumer: Single CV And While**

- 3rd try

```
por( T e) {
  lock(m);
  while (bufffull())
    wait(cond, m);
  bufput( e );
  signal(cond);
  unlock(m);
}
```

```
T tirar() {
  lock(m);
  while (bufempty())
    wait(cond, m);
  T e = bufget();
  signal(cond);
  unlock(m);
  return e;
}
```

Não funciona. Há duas condições distintas.

Put() espera por get(), e vice-versa, mas get() não espera por get(), nem put() por put()

```c
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Figure 30.12: **Producer/Consumer: Two CVs And While**

- 4th try -> THE CORRECT ONE

```
por( T e) {
  lock(m);
  while (bufffull())
    wait(space, m);
  bufput( e );
  signal(elem);
  unlock(m);
}
```

```
T tirar() {
  lock(m);
  while (bufempty())
    wait(elem, m);
  T e = bufget();
  signal(space);
  unlock(m);
  return e;
}
```

O uso do signal() avia a outra função que pode executar o unlock() avisa-se a si mesma para esperar por si mesma
Permite mais simultaneidade e eficiência

- Com 1 produtor e 1 consumidor, esta abordagem é mais eficiente, pois reduz as mudanças de contexto; com múltiplos produtores ou consumidores (ou ambos), permite até que ocorra produção ou consumo simultâneo, aumentando assim a simultaneidade.

```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

Figure 30.13: **The Correct Put And Get Routines**

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        while (count == MAX)                   // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        put(i);                                // p4
        Pthread_cond_signal(&fill);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (count == 0)                     // c2
            Pthread_cond_wait(&fill, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&empty);           // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
}
```

Figure 30.14: **The Correct Producer/Consumer Synchronization**

- Para verificar uma condição num programa multi-thread o mais correto é usar um while()

Em Java para usar threads tem que se verificar se o que estamos a usar suporta as threads ou é necessário fazer alguma configuração

- Métodos:
  - Run() - interface runnable
  - Start()
  - Join()

```
class MyThread extends Thread {
  static long counter = 0;  //shared
  private long max;

  MyThread( long l ) {
    max = l;
    start();
  }

  public void run() {
    for (long i=0; i<max; i++) {
      counter = counter + 1;
    }
  }
```

Região crítica

- Métodos oferecidos:
  - Wait() - bloquear o thread aguardando por uma notificação (eq. Cond_wait)
  - Notify() - desbloquear um thread que aguarda neste objeto (eq. Cond_signal)
  - Notifyall() - desbloquear todos os threads aguardando neste objeto (eq. Cond_broadcast)

Exclusão mútua

algo que é partilhado por todos os threads

```
class MyThread extends Thread {
  static long counter = 0;
  static Object mtx = new Object(); //or  MyThread.class
  private long max;
  MyThread( long l ) {
    max = l;
    start();
  }
  public void run() {
    for (long i=0; i<max; i++) {
      synchronized ( mtx ) {
        counter = counter + 1;     // Critical Section
      }
    }
  }
}
```

outra opção na troca de mtx, diretamente no synchronized(...)

→ mutex em Java ⇓ sincronização