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. Díficil 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

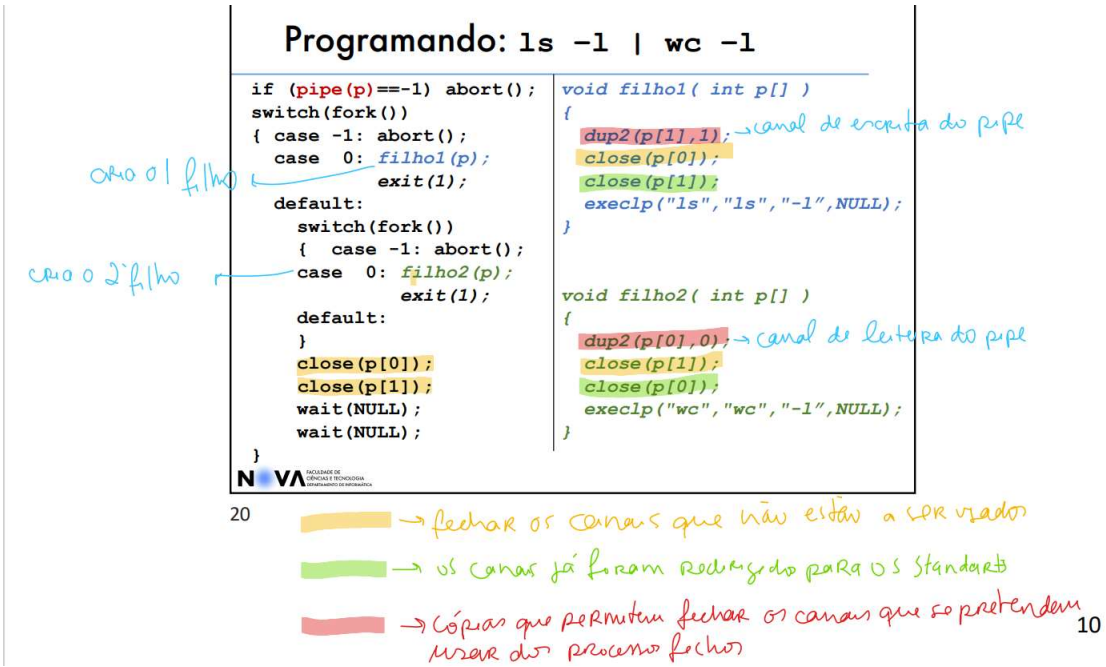
Funcionamento do pipe (2)

- Como um processo sabe que a comunicação terminou?
- ou como o SO determina que terminou a comunicação?
  - se todos os descritores de escrita forem fechados:
    - o canal de escrita é fechado
    - o read num pipe vazio sem canal para escrita devolve 0 (em vez de bloquear) como se "fim de ficheiro"
  - se todos os descritores de leitura forem fechados:
    - o canal de leitura é fechado
    - o write num pipe sem leitores dá erro (EPIPE)

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

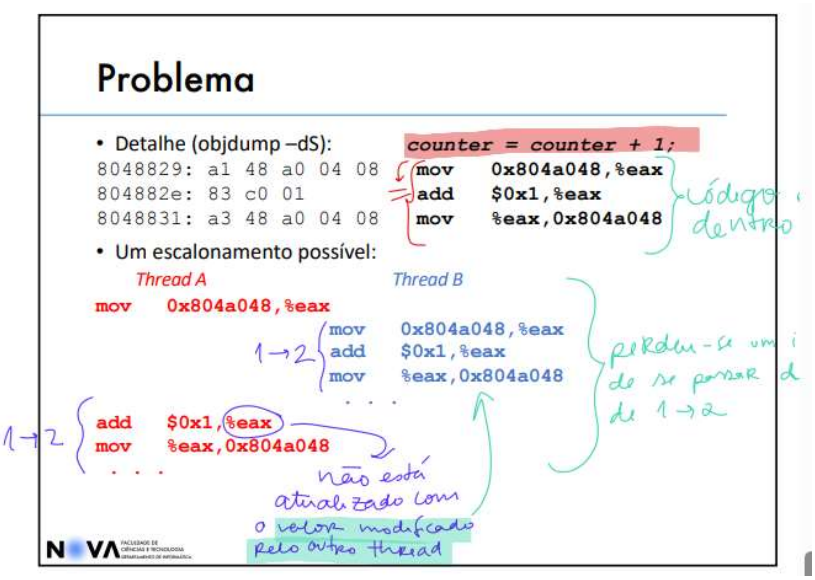
Why use threads?

- **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