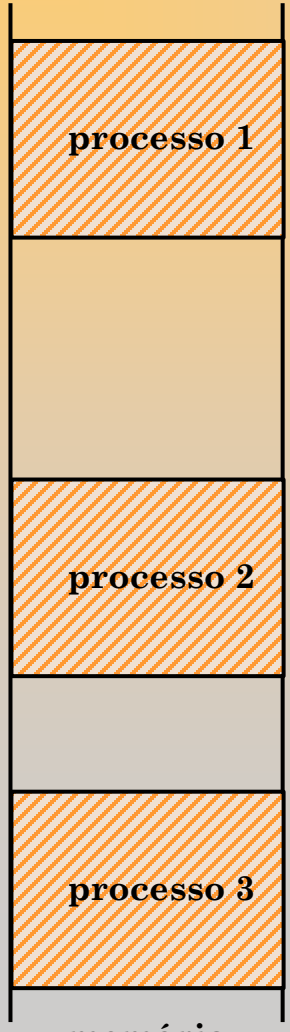
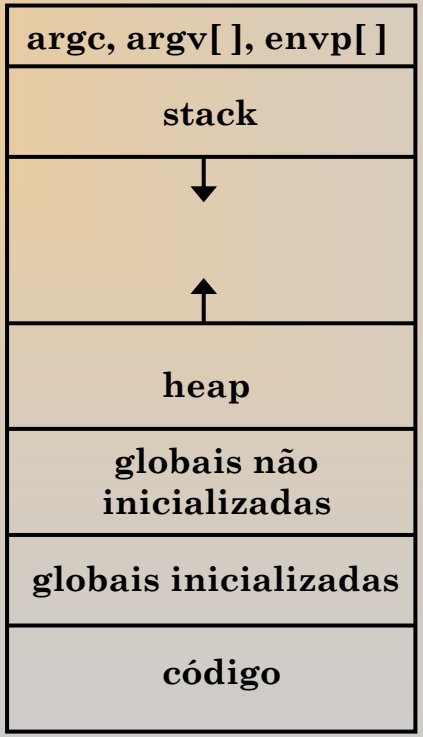
Programação de sistema

Criação de processos

Processos

Processo – programa em execução ocupando um certo espaço na memória com o seu código, variáveis globais, variáveis locais, variáveis de ambiente e um stack e um heap.





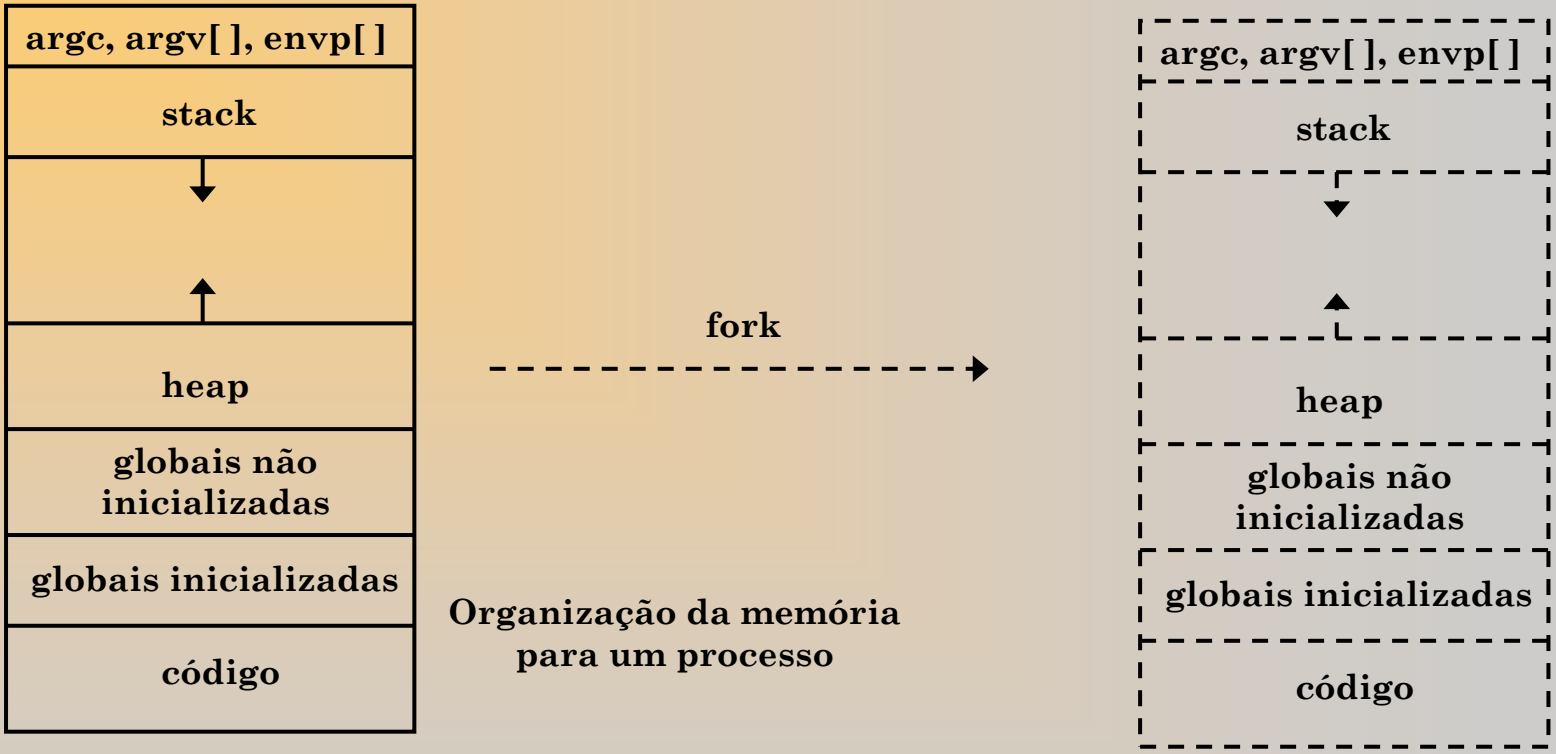
Os processos podem ser várias instâncias do mesmo ou de diferentes programas em execução

programa num ficheiro executável

header (entry point, tamanhos)
dados globais inicializados
código

O serviço fork( )

Processo – Instância de um programa em execução no sistema



Organização da memória para um processo

O serviço fork( ) cria um novo processo filho duplicando toda a memória do pai

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

retorna > 0 no processo pai

retorna 0 no processo filho

-1 erro

Pid's e códigos de terminação

Cada processo no sistema tem um identificador único (> 0) designado por *pid*.

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);          /* obtém o seu próprio pid */
pid_t getppid(void);         /* obtém o pid do pai */
```

Pode obter-se a lista de processos do sistema (ou apenas do utilizador), juntamente com mais informação acerca de cada um, utilizando no terminal a chamada ps. Consultar através do man as inúmeras opções de ps.

Existe uma hierarquia e uma relação pai-filho entre todos os processos. Para manter essa relação, quando o pai de um processo termina, os seus filhos são adotados por um processo do sistema (init).
Quando um processo filho termina, para que este possa abandonar o sistema, o pai tem que recolher o seu código de terminação. Enquanto isso não acontecer o filho permanece no estado de zombie.

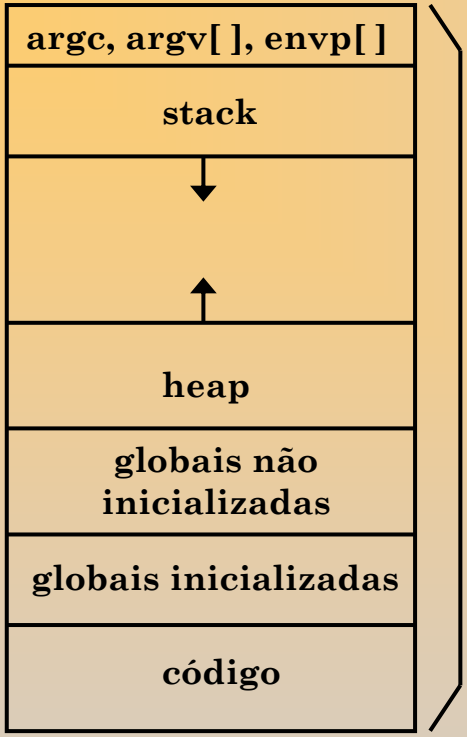
```
# include <sys/types.h>
# include <wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

0, WNOHANG

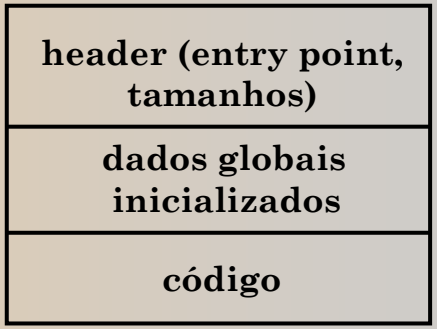
WIFEXITED(status) – valor *true* se o processo terminou normalmente
se a terminação foi normal WEXITSTATUS(status) dá o código de terminação

Execução de código de um ficheiro



serviços exec( )

ficheiro executável



prog

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* NULL */ );
int execv(const char *pathname, const char *argv[ ]);
int execle(const char *pathname, const char *arg0, ... /* NULL, char * const envp[ ] */ );
int execve(const char *pathname, const char *argv[ ], char * const envp[ ]);
int execlp(const char *filename, const char *arg0, ... /* NULL */ );
int execvp(const char *filename, const char *argv[ ]);
```

fork( ) + exec( ) + waitpid( )

A função system( ) da biblioteca standard do C, executa num novo processo um ficheiro executável, espera que esse processo termine e continua depois o código do invocador.

```
#include <stdlib.h>

int system(const char* cmdstring);
```

Exemplo:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   printf("Antes de system\n\n");
   system("ps –l");
   printf("\nDepois de system\n");
   return 0;
}
```

```
Antes de system

STATE      UID     PID    PPID NI     SZ TTY      TIME CMD
S          apm     387     451  2    572 n00   0:00.03 sh
W          apm     451    3905  2    384 n00   0:00.00 prog
S          apm     707     387  2    456 n00   0:00.05 ps
S          apm    3905       1  2   1700 n00   0:00.19 csh

Depois de system
```

Para manter um processo parado ou inativo durante um certo número de segundos pode usar-se:

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```