

Sistemas Operativos

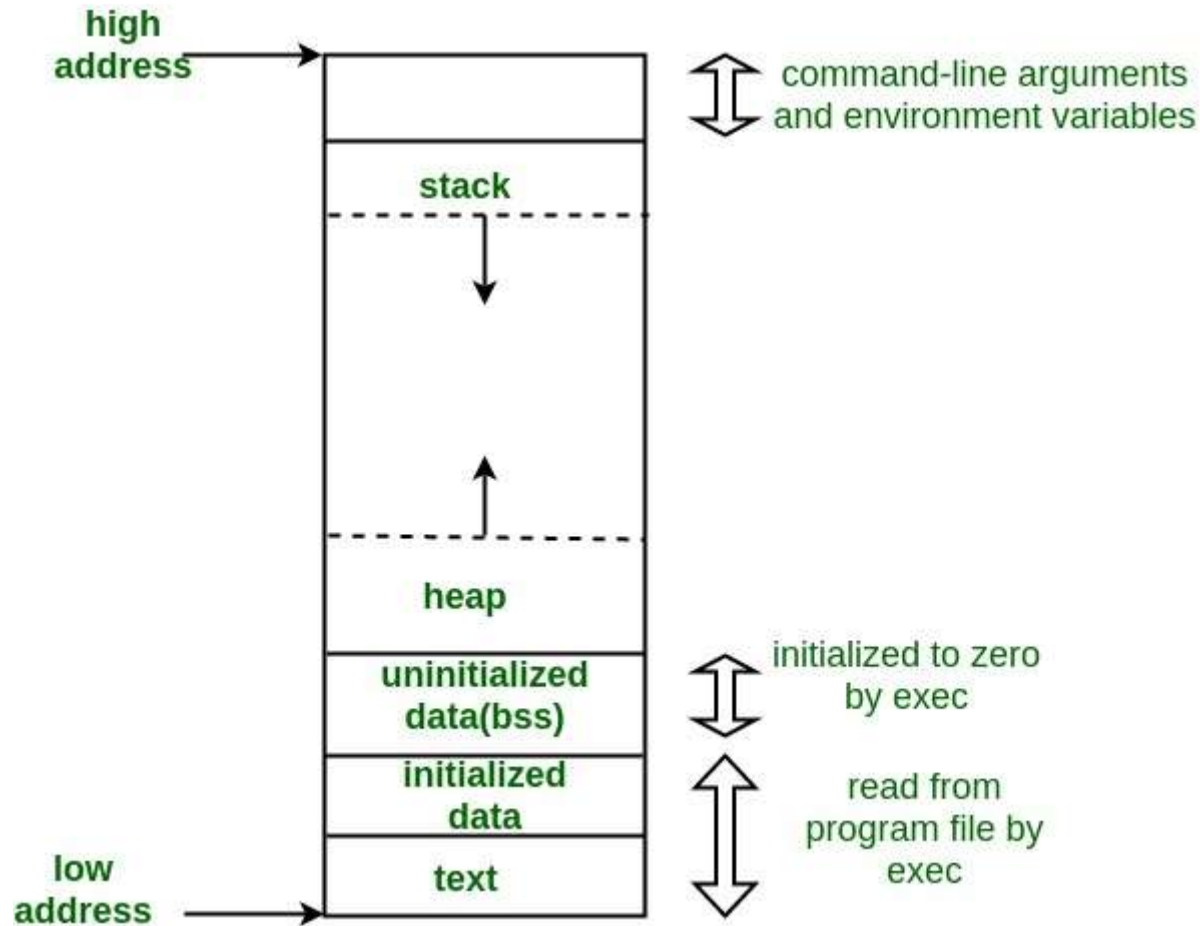
Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

Ano letivo 2024/2025

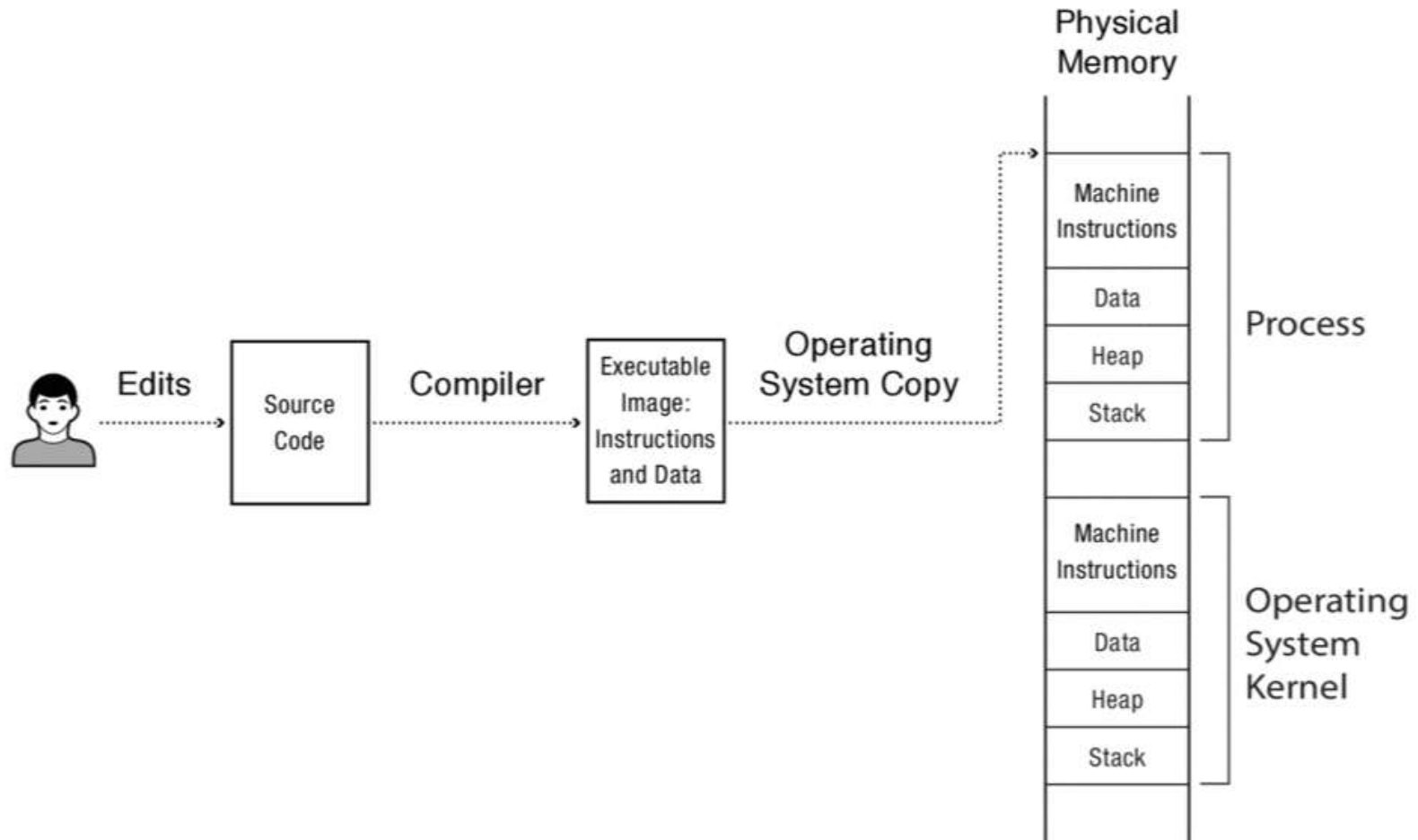
Nuno Lau (nunolau@ua.pt)

- Programa em execução
- Criar um processo
 - Inicialização do Sistema
 - Execução de chamada ao sistema por processo em execução
 - Pedido do utilizador para criar novo processo
 - Início de um *batch script*
- Processos podem correr em:
 - *foreground*: interage com utilizador
 - *background*: executa sem interação, *daemon*

Memória de um processo

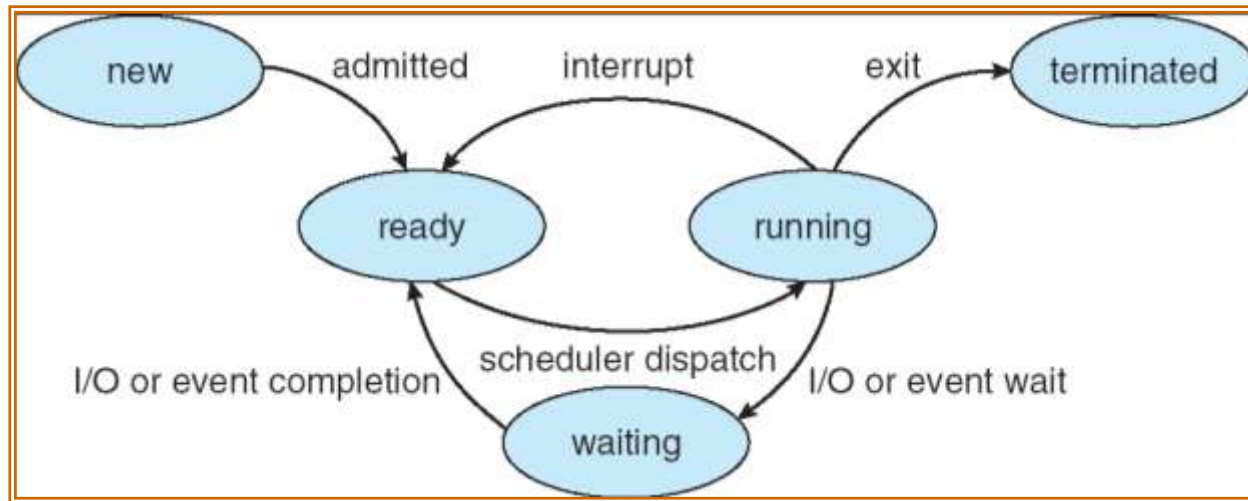


Memória processo/OS

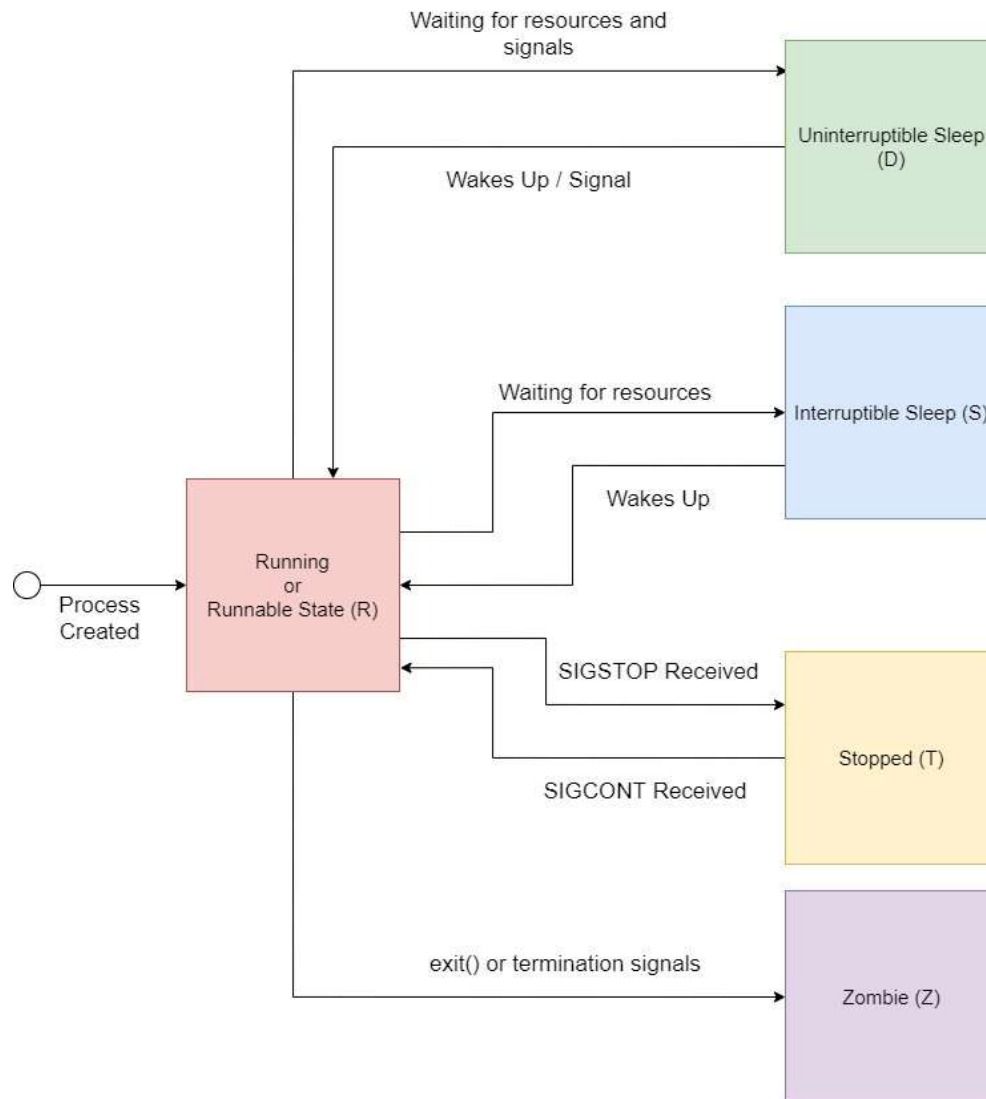


- Um processo em execução pode estar nos seguintes estados
 - *New*
 - *Running*
 - *Waiting*
 - *Ready*
 - *Terminated*

Estados de um processo



Estados de um processo



- O *Process Control Block* (PCB) é a estrutura que, no SO, armazena a informação sobre um processo
- Inclui os seguintes dados:
 - Estado do processo
 - *Program Counter*
 - Registos do CPU
 - Tipo de escalonamento
 - Informação sobre a memória do processo
 - Informação sobre I/O
 - *Accounting*

Process Control Block

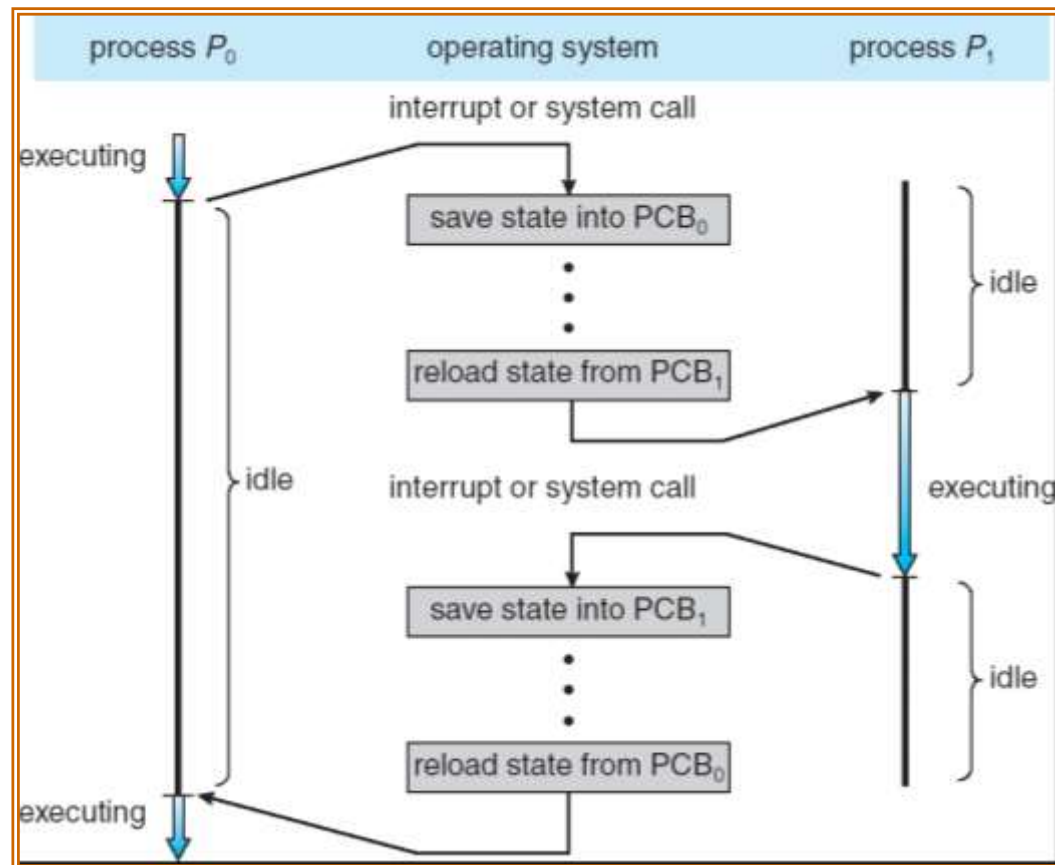


Process Control Block

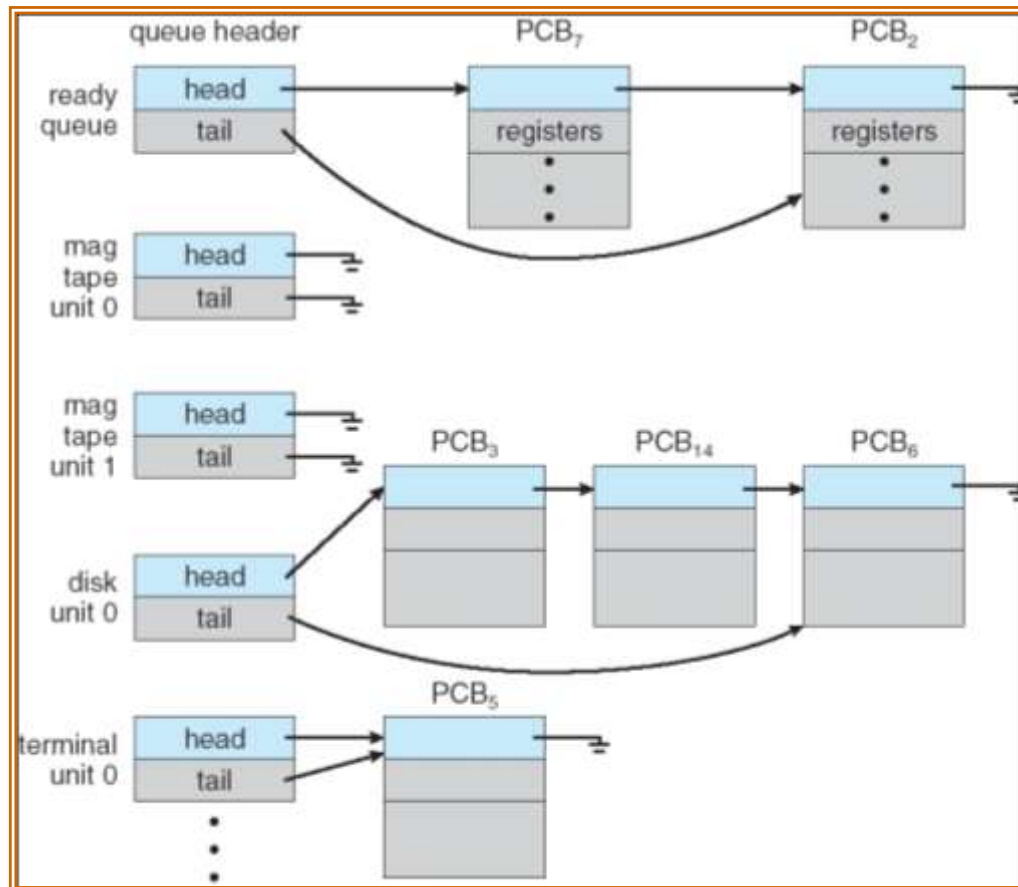
Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Mudança de contexto

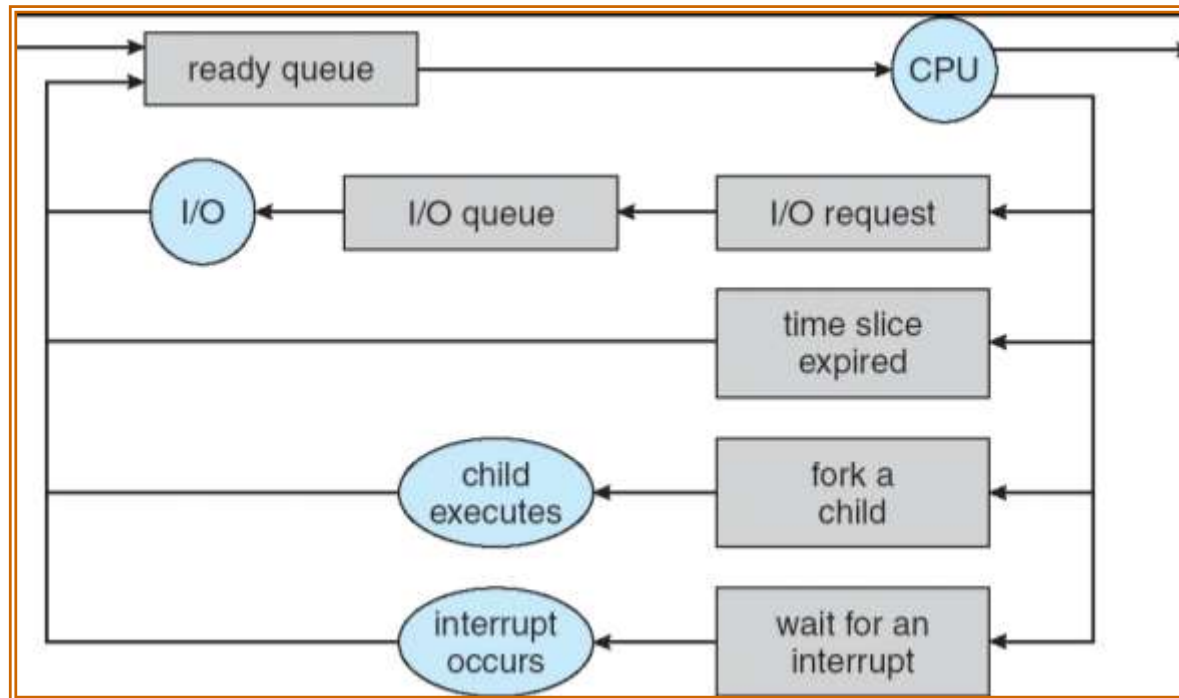
- Quando o SO troca o processo que está no estado *Running*



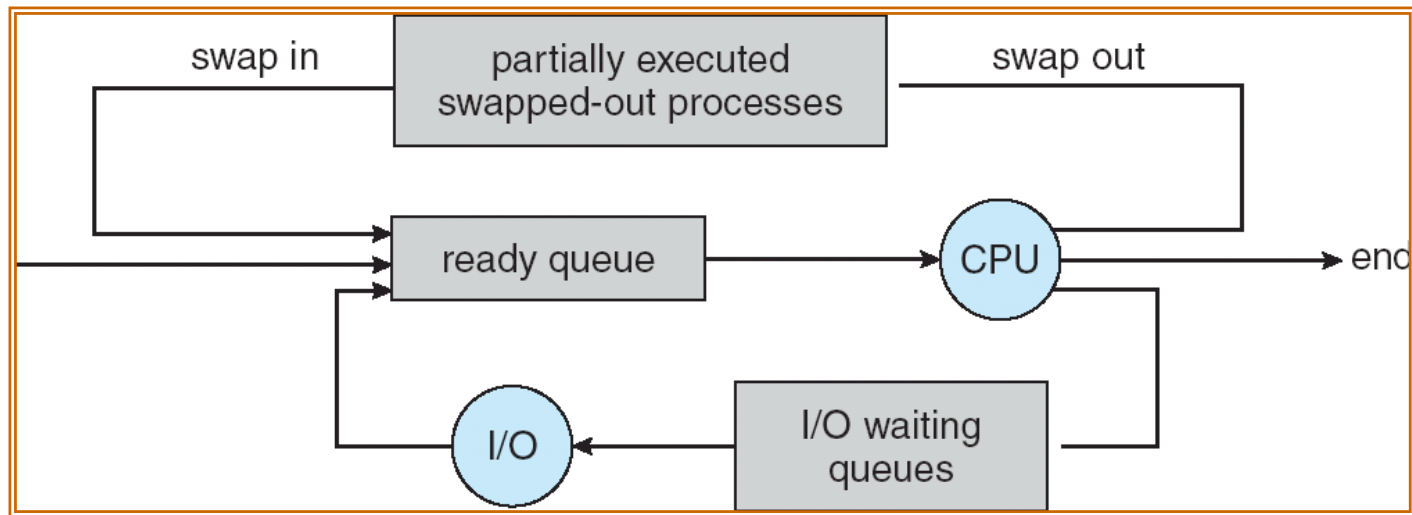
Estruturas de suporte ao escalonamento



Representação do escalonamento



Representação do escalonamento



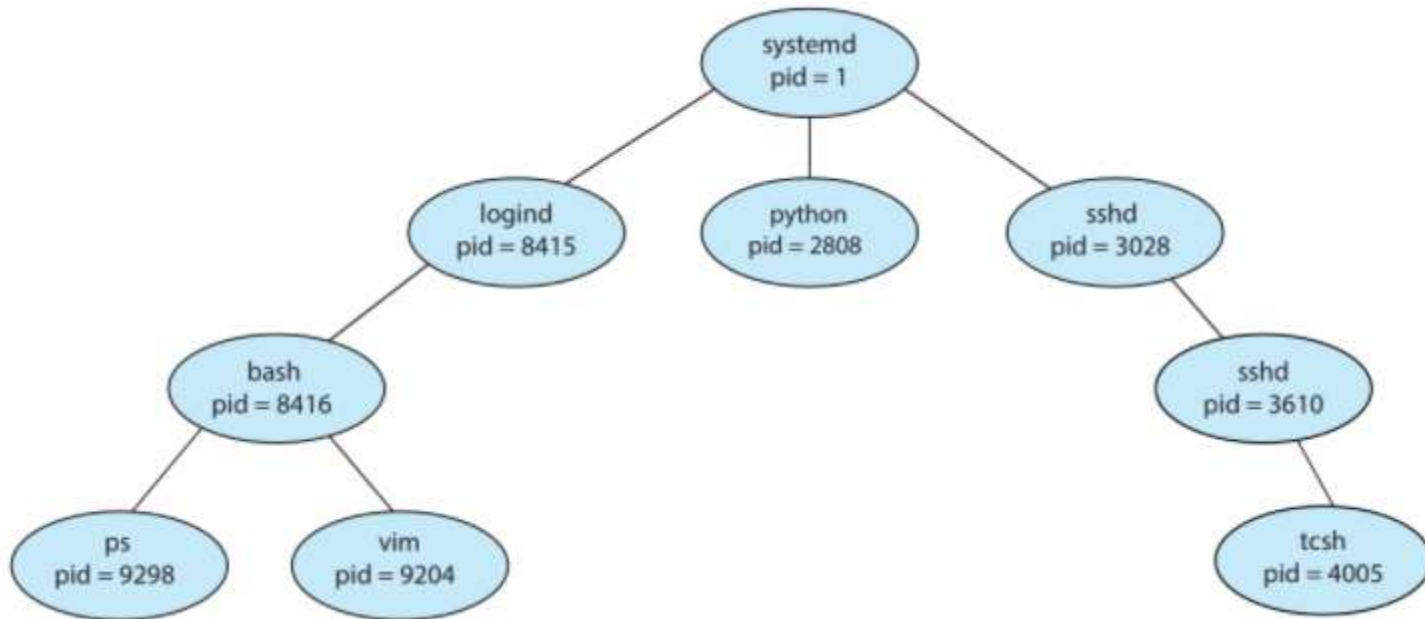
Directoria /proc

- directoria virtual (ficheiros só existem em memória)
- podem ser consultadas várias informações sobre o Sistema
 - Permite alterar parâmetros do kernel
- Contém informações gerais:
 - Ex: `cpuinfo`; `meminfo`; etc
- Uma directoria para cada processo
 - Ex: `root`; `cwd`; ficheiros abertos; mapa de memória; etc
- Mais informações em:

<http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html>

- Quando um processo cria um novo processo
 - Processo criador é designado de **processo pai**
 - Novo processo é designado de **processo filho**
- Pode ser formada uma hierarquia de processos
- Hierarquia de processos
 - Processo pode saber **pid** do pai
 - Quando filho morre é enviado o sinal **SIGCHLD** ao pai
 - Pai recolhe *exit code* dos filhos
 - Quando pai morre, filho é herdado pelo processo 1 (**init**) ou **systemd**
 - A partir do kernel 3.4, um processo pode nomear-se como pai dos processos orfãos seus descendentes (ex: **systemd**, **upstart**)

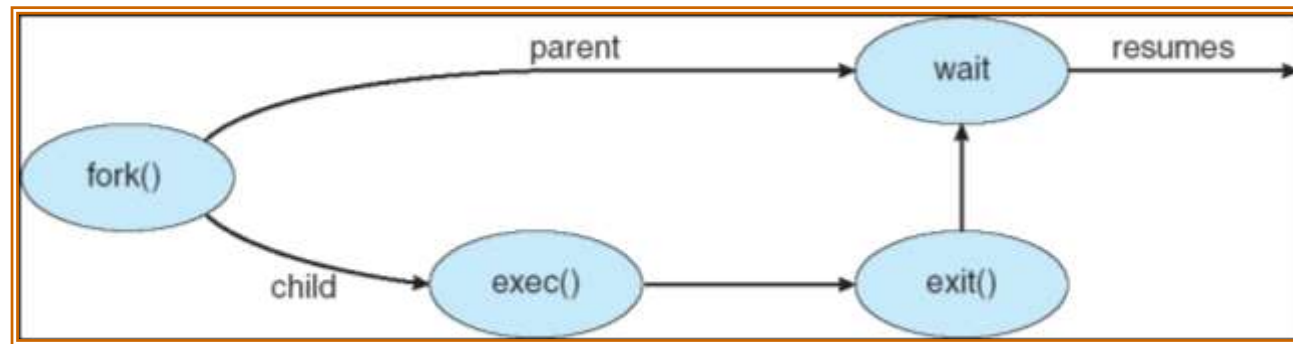
Árvore de processos



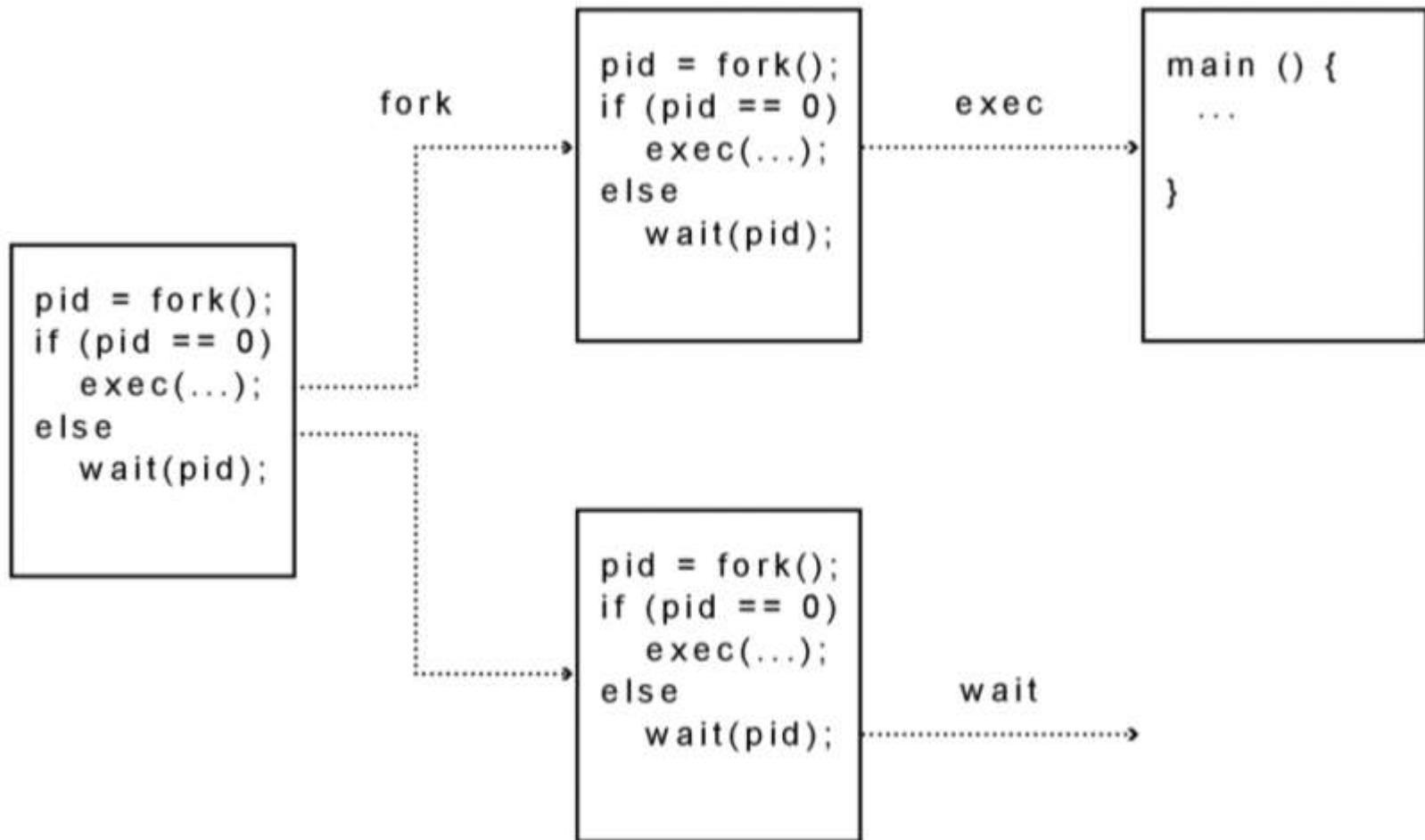
- I/O intensivos
 - Fazem muitas chamadas ao sistema relacionadas com I/O
 - Muitos pequenos períodos de utilização do CPU
- CPU intensivos
 - Fazem poucas chamadas I/O
 - Poucos e longos períodos de utilização do CPU
- Num sistema com *timesharing* e de modo a otimizar a utilização do CPU é positivo que a lista de processos em execução seja equilibrada entre os 2 tipos

- Um processo pode criar novos processos
 - O processo criador designa-se de processo pai e os criados de processos filhos
 - Os filhos podem, por sua vez, criar novos processos
- Partilha de recursos
 - Pai e filhos partilham recursos
 - Filhos partilham um subconjunto dos recursos do pai
 - Pai e filhos não partilham recursos
- Execução
 - Pai e filhos executam em paralelo
 - Pai espera que filho(s) terminem

Criação de processos



Criação de processos



Criação de processos - POSIX

```
#include <errno.h>
#include <stdio.h>
...
pid_t childpid;
...
childpid=fork();
switch(childpid)
{
    case -1:
        fprintf(stderr,"ERROR: %s\n", sys_errlist[errno]);
        exit(1);
        break;
    case 0:
        /* Child's code goes here */
        execlp("/bin/ls","ls",NULL);
        break;
    default:
        /* Parent's code goes here */
        wait(NULL);
        printf("Child completed");
        break;
}
```

Criação de processos – Win32

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _tmain( int argc, TCHAR *argv[] ) {
    STARTUPINFO si; PROCESS_INFORMATION pi;
    ZeroMemory( &si, sizeof(si) ); si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( argc != 2 ) {
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }

    // Start the child process.
    if( !CreateProcess( NULL, // No module name (use command line)
        argv[1], // Command line
        NULL, // Process handle not inheritable
        NULL, // Thread handle not inheritable
        FALSE, // Set handle inheritance to FALSE
        0, // No creation flags
        NULL, // Use parent's environment block
        NULL, // Use parent's starting directory
        &si, // Pointer to STARTUPINFO structure
        &pi ) // Pointer to PROCESS_INFORMATION structure
    ) {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess ); CloseHandle( pi.hThread );
}
```

Criação de processos – Java

```
import java.io.*;
public class OSProcess {

    public static void main(String args[]) {
        try {

            // Create process
            ProcessBuilder pb = new ProcessBuilder(args[0]);
            Process p = pb.start();

            // Get BufferedReader linked to process input stream
            BufferedReader input =
                new BufferedReader
                    (new InputStreamReader(p.getInputStream()));

            // read output from process and print it
            String line;
            while ((line = input.readLine()) != null) {
                System.out.println(line);
            }
            input.close();
        }

        catch (Exception err) {
            err.printStackTrace();
        }
    }
}
```

Criação de processos – Python

```
from multiprocessing import Process

def f(name):
    print('hello', name)

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Processos em *scripts*

- **exec**
- **wait**
- Processos em *background*
- *Process group*
- **jobs**
- **parent.sh parent2.sh child.sh**

Adaptado de:

<https://gist.github.com/CMCDragonkai/f58afb7e39fcc422097849b853caa140>

POSIX *Input / Output*

- Abrir e fechar ficheiros

```
int open(const char *path, int oflag, ... /*, mode_t mode */);  
int close(int filedes);
```

- Ler / Escrever

```
ssize_t read(int fd, void *buf, size_t count);  
ssize_t write(int fd, const void *buf, size_t nbytes);
```

- Duplicar *file descriptors*

```
int dup (int oldfd);  
int dup2 (int oldfd, int newfd);
```

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

#include <stdio.h>

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
    int fd, nr;
    char buf[BUFSIZE];

    nr = read(0, buf, BUFSIZE);

    printf("read bytes=%d buf=%.*s\n", nr, nr, buf);

    return 0;
}
```

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

#include <stdio.h>

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

    fd = open("writel.txt", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);

    write(fd, "message1\n", 9);

    close(fd);

    return 0;
}
```

Redirecionamento *output*

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

#include <stdio.h>

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

    fd = open("rdex1.txt", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);

    dup2(fd, 1); // close 1, then make 1 refer to same file as fd
    close(fd);  // close fd

    execlp("ls", "ls", NULL);

    return 0;
}
```

Redir *output* no processo filho

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

#include <stdio.h>

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

    switch( fork() ) {
        case 0: // child
            fd = open("redirforkexecl.txt", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);

            dup2(fd, 1); // close 1, then make 1 refer to same file as fd
            close(fd); // close fd

            execlp("ls", "ls", NULL); // exec ls

            break;
        default: //parent
            printf("pid=%d\n", getpid());
            break;
    }

    printf("END.\n");

    return 0;
}
```