SUPSI

Processes and threads

Operating Systems

Amos Brocco, Lecturer & Researcher

12 settembre 2022

Objectives

- Understand the concept of process and thread
- Understand process implementation in current operating systems
- Understand how to create processes

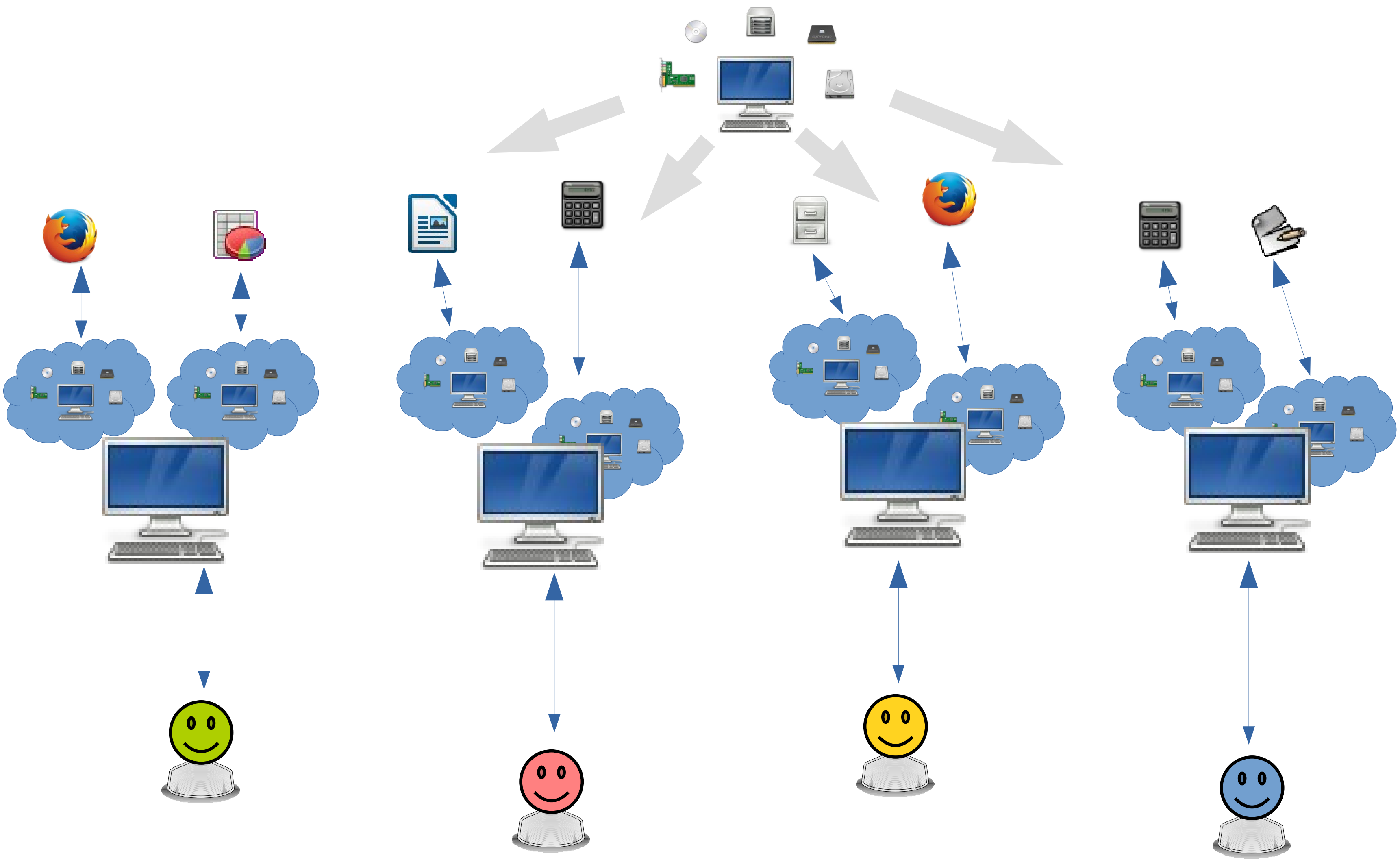
Browsing
- Get a rapid overview.

Reading
- Read it and try to understand the concepts.

Studying
- Read in depth, understand the concepts as well as the principles behind the concepts.

You are also encouraged to try out (compile and run) code examples!

Multiple resources shared between running programs

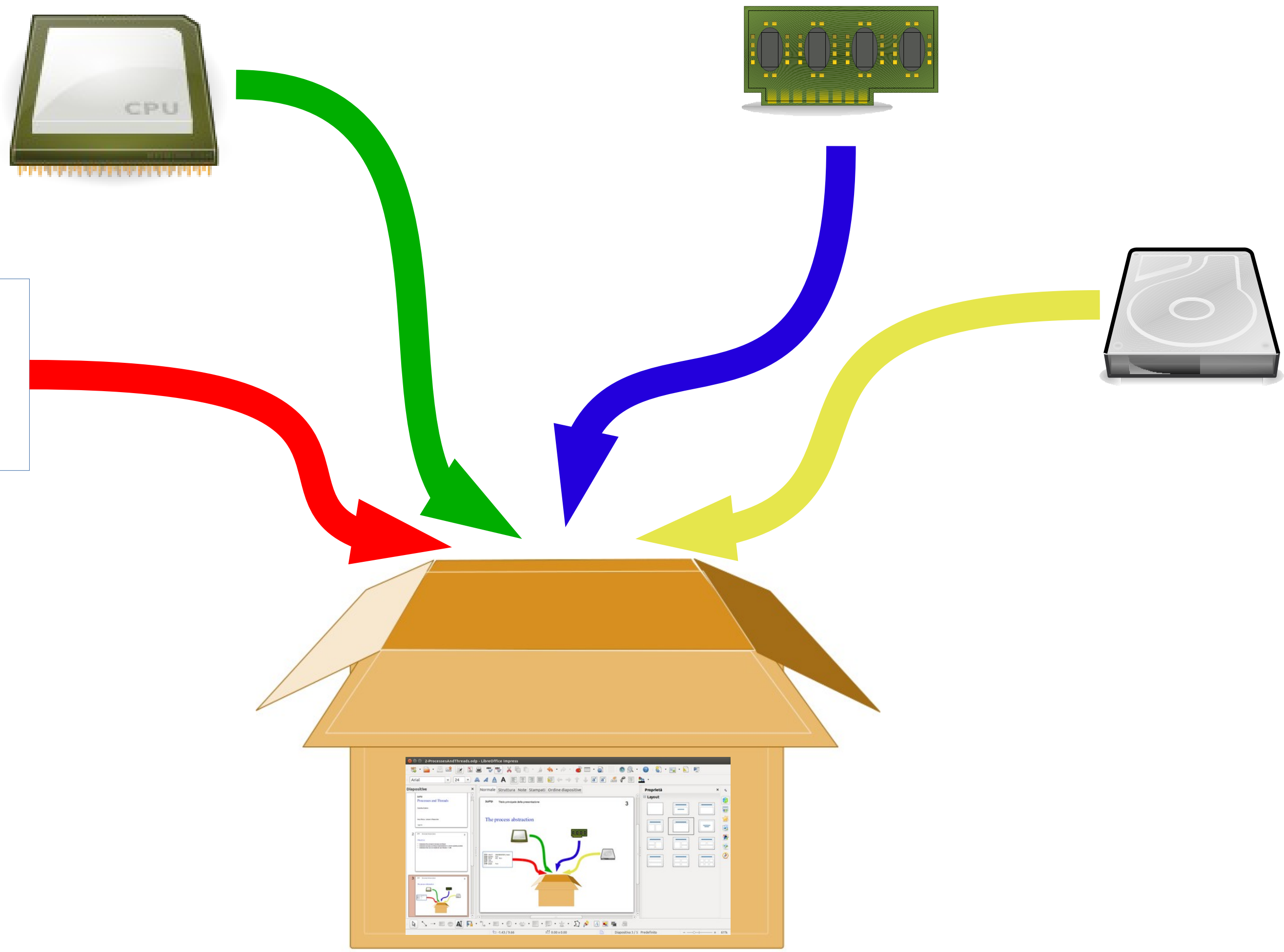


How to keep track of who uses what?

Question

The process abstraction

```
...
1003 imull   $0x01010101,%eax
1004 pushw   %cx
1005 shrw    $2, %cx
1006 rep
1007 stosl
1008 popw    %cx
...
```

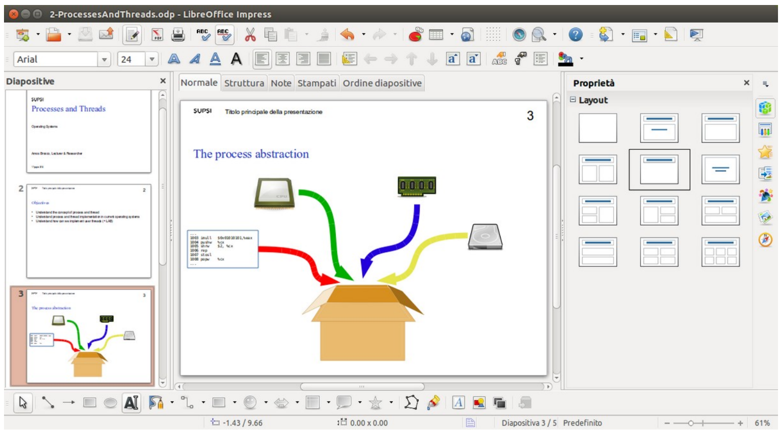


Answer

The process

- A process represents an instance of a **program that is being executed**
  - it comprises the execution **context** of the computation
    - Hardware context (program counter, stack pointer, processor status word, registers, address translation table)
    - Address space (regions of memory)
    - Control information
    - Credentials
- Processes require resources (CPU time, memory, access to the filesystem and to I/O devices) to accomplish their task.

The process




```
top - 14:15:25 up 38 min,  2 users,  load average: 0.24, 0.25, 0.28
Tasks: 237 total,   1 running, 236 sleeping,   0 stopped,   0 zombie
%Cpu(s):  4.2 us,  3.6 sy,  0.0 ni, 92.0 id,  0.1 wa,  0.2 hi,  0.0 si,  0.0 st
KiB Mem:   3662524 total,  3026708 used,   635816 free,    78524 buffers
KiB Swap:        0 total,        0 used,        0 free.  1792736 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1405 root      20   0  463528 127668 111568 S  11.3  3.5   1:32.65 Xorg
 2163 attila    20   0 1672000 136156  36164 S   9.9  3.7   1:31.67 compiz
 4523 attila    20   0  338416  11972   8792 S   5.0  0.3   0:00.48 gnome-scre+
 2224 attila    20   0 3107180 217652  20356 S   2.0  5.9   1:18.26 java
 3745 attila    20   0 1656936 149852  64928 S   1.0  4.1   1:18.32 soffice.bin
 4521 attila    20   0   29156   1724   1172 R   0.7  0.0   0:00.09 top
    7 root      20   0       0      0      0 S   0.3  0.0   0:01.09 rcu_sched
    8 root      20   0       0      0      0 S   0.3  0.0   0:00.45 rcuos/0
```

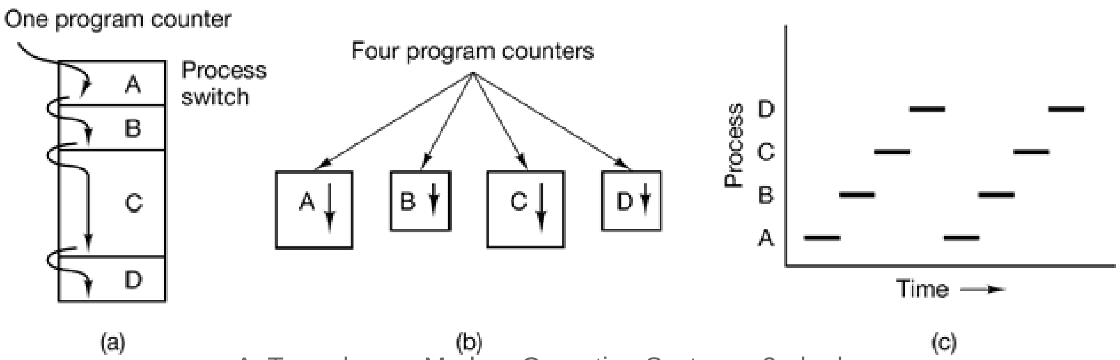
```c
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;

#ifdef CONFIG_SMP
    struct llist_node wake_entry;
    int on_cpu;
    struct task_struct *last_wakee;
    unsigned long wakee_flips;
    unsigned long wakee_flip_decay_ts;

    int wake_cpu;
#endif
    int on_rq;

    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
#ifdef CONFIG_CGROUP_SCHED
    struct task_group *sched_task_group;
#endif

#ifdef CONFIG_PREEMPT_NOTIFIERS
    /* list of struct preempt_notifier: */
    struct hlist_head preempt_notifiers;
```

Code excerpt from /usr/src/linux-headers-3.13.0-29/include/linux/sched.h

Explaination

Program vs process

- A process is an **active entity**, whereas a program is a **passive entity** ("machine code resting on some storage media")
- In **multiprogrammed** systems more than one process at a time can reside in memory
  - Depending on the number of execution units (cores) these processes might be executed sequentially, in parallel or with pseudo-parallelism

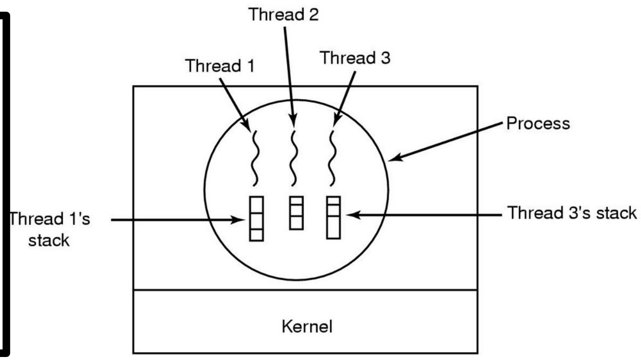


A. Tanenbaum, Modern Operating Systems, 2nd ed

The active part of a process: threads

- A process can have one or more **threads** (or **paths**) **of execution** *
  - Threads in a process share some resources (→ concurrency problems)

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	



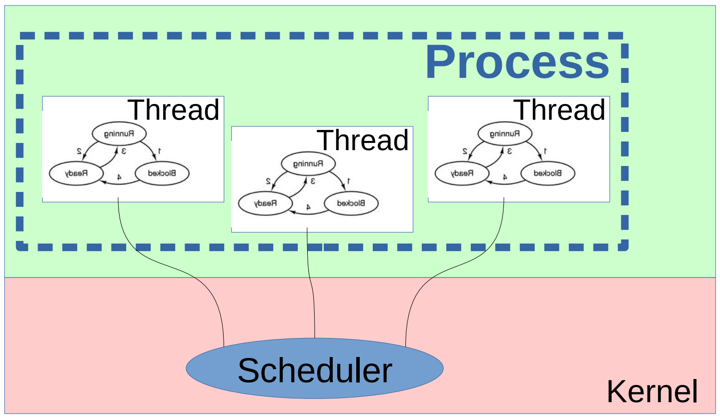
A. Tanenbaum, Modern Operating Systems, 2nd ed

  - When a process has multiple threads of execution we call it a **multi-threaded process**, otherwise it is called a **single-threaded process**

* typically simply referred to as **threads**

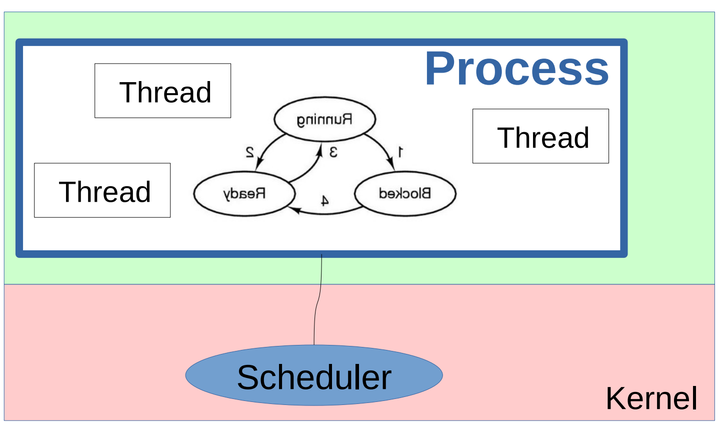
Answer

Threads implementation



- **Kernel level threads**
  - “the kernel knows what threads are”
  - Thread scheduling is done by the kernel
  - If a thread blocks, other threads within the same process can continue executing
  - Note: kernel level threads still run in unprivileged (user) mode!

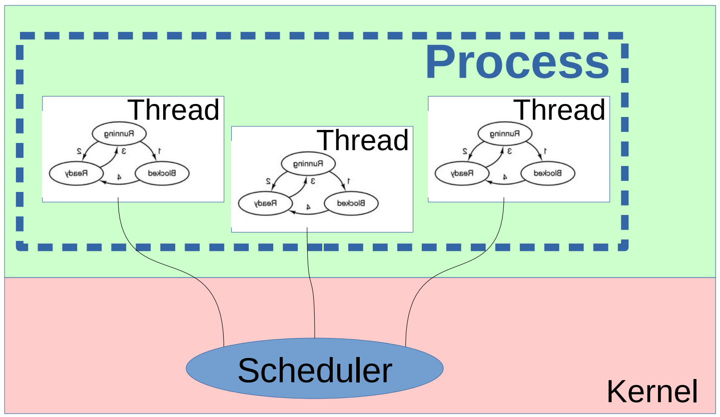
What you're used to



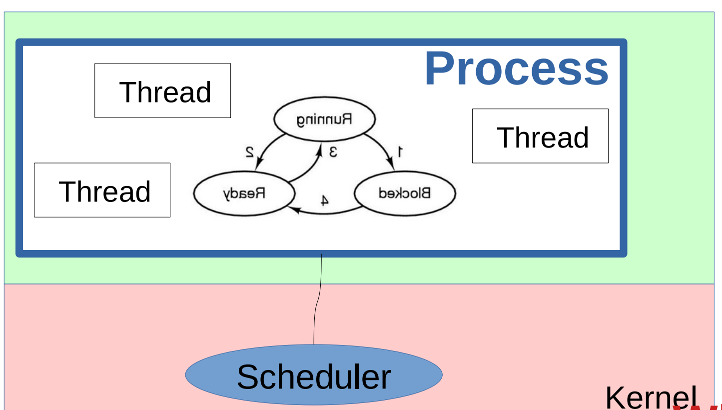
- **User level threads**
  - “the kernel doesn't know anything about threads”
  - Thread scheduling is done by the process
    - When the kernel schedules the process its threads are given a chance to run
  - If a thread blocks, the whole process (including other user threads) is blocked

Explaination

Threads implementation



- **Kernel level threads**
  - “the kernel knows what threads are”
  - Thread scheduling is done by the kernel
  - If a thread blocks, other threads within the same process can continue executing



- **User level threads**
  - “the kernel doesn't know anything about threads”
  - Thread scheduling is done by the process
    - When the kernel schedules the process its threads are given a chance to run
  - If a thread blocks, the whole process (including other user threads) is blocked

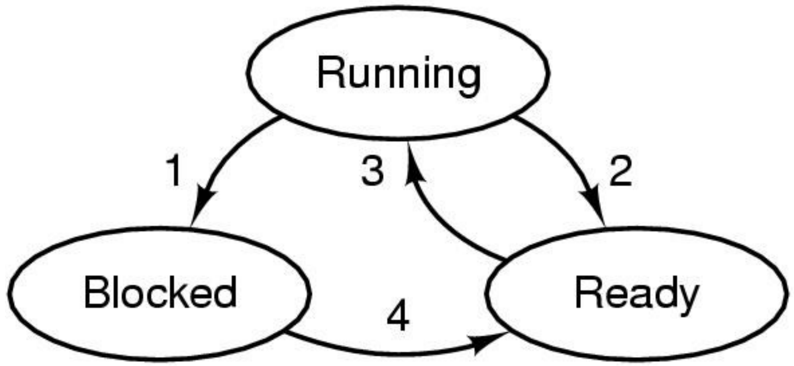
What you are going to implement in the LAB

Explaination

The process/thread state (simplified)

Since (typically) not all processes can be running at the same time (unless we have a sufficient number of CPUs), some need to wait.

In order to remember which processes are running and which aren't the OS maintains a state for each process.



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Image taken from “Modern Operating Systems”, 2/E, Andrew S. Tanenbaum, Prentice Hall, 2001

Explaination

Process implementation (Windows)

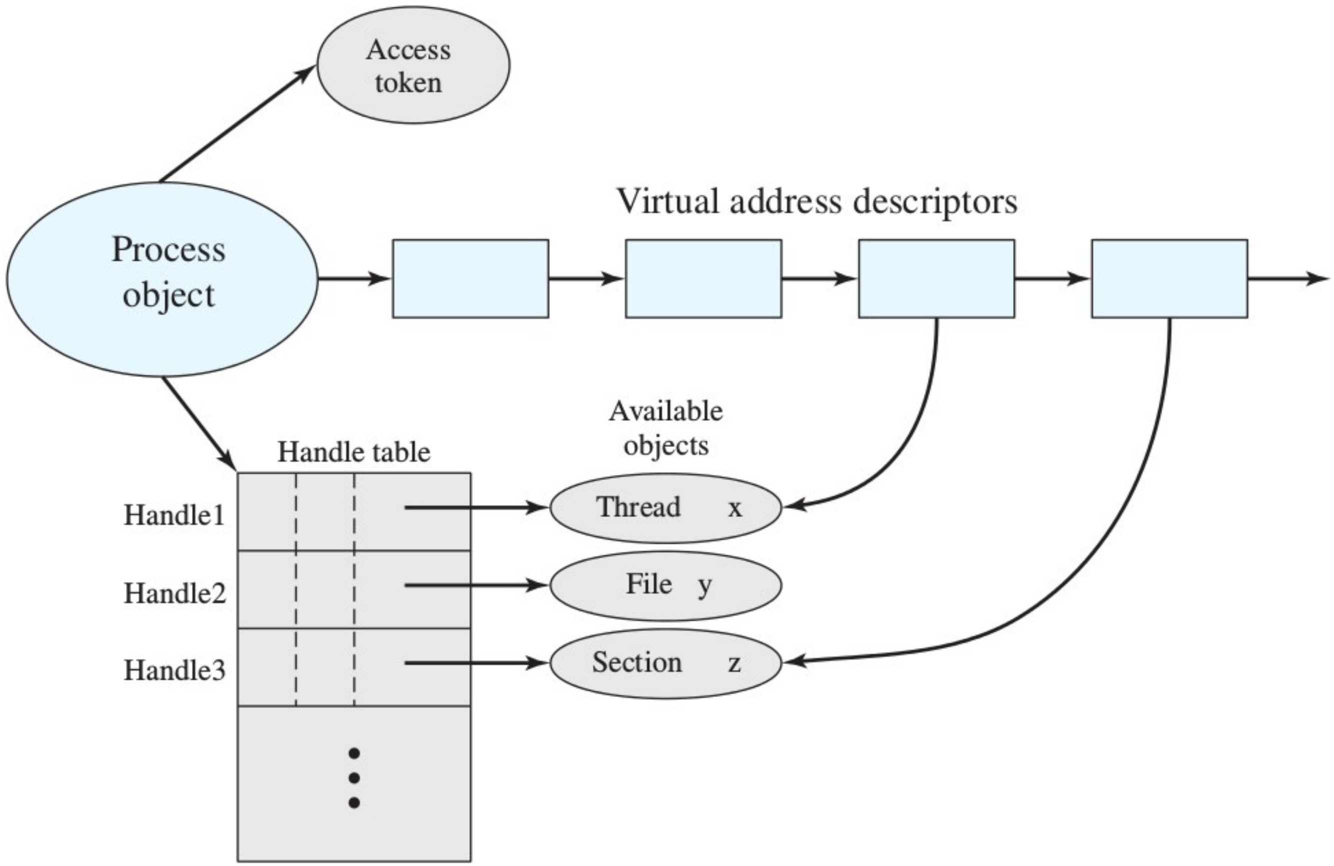


Figure 4.12 **A Windows Process and Its Resources**

Image from Operating Systems: Internals and Design Principles, Sixth Edition, William Stallings, Prentice Hall 2008

Process implementation (Linux)

```
struct task_struct {
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags; /* per process flags, defined below */
    unsigned int ptrace;

#ifdef CONFIG_SMP
    struct llist_node wake_entry;
    int on_cpu;
    struct task_struct *last_wakee;
    unsigned long wakee_flips;
    unsigned long wakee_flip_decay_ts;

    int wake_cpu;
#endif
    int on_rq;

    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    struct sched_entity se;
    struct sched_rt_entity rt;
#ifdef CONFIG_CGROUP_SCHED
    struct task_group *sched_task_group;
#endif

#ifdef CONFIG_PREEMPT_NOTIFIERS
    /* list of struct preempt_notifier: */
    struct hlist_head preempt_notifiers;
```

Code excerpt from /usr/src/linux-headers-3.13.0-29/include/linux/sched.h

Process attributes (summary)

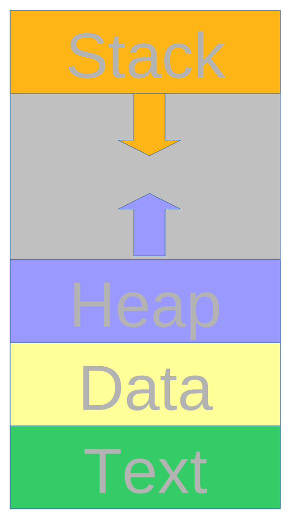
Process management	**Memory management**	**File management**
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state	They define the address space	Group ID
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		

Image taken from “Modern Operating Systems”, 2/E, Andrew S. Tanenbaum, Prentice Hall, 2001

How are processes created and destroyed?

Question

Life of a process

- A process can be created...
  - During the boot sequence
  - Upon request of another process (using a system call)
  - When the executing process ends (in batch systems)
- A process ends...
  - When it's done with its tasks (voluntarily)
  - When it encounters an error and cannot continue (voluntarily)
  - When it encounters a fatal error (involuntarily)
  - When it gets killed by another process (involuntarily)

Answer

Example: Process creation (Windows)

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
void _tmain( int argc, TCHAR *argv[] ) {
    TCHAR* cmd_line = "notepad.exe";
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );
    if( !CreateProcess( NULL,   // No module name (use command line)
        cmd_line,       // 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; }
    WaitForSingleObject( pi.hProcess, INFINITE ); // Wait until child process exits.
    CloseHandle( pi.hProcess );  // Close process handle
    CloseHandle( pi.hThread );  // Close thread handle
}
```

http://msdn.microsoft.com/en-us/library/windows/desktop/ms682425%28v=vs.85%29.aspx

```
BOOL WINAPI CreateProcess(
  _In_opt_     LPCTSTR lpApplicationName,
  _Inout_opt_  LPTSTR lpCommandLine,
  _In_opt_     LPSECURITY_ATTRIBUTES
lpProcessAttributes,
  _In_opt_     LPSECURITY_ATTRIBUTES
lpThreadAttributes,
  _In_         BOOL bInheritHandles,
  _In_         DWORD dwCreationFlags,
  _In_opt_     LPVOID lpEnvironment,
  _In_opt_     LPCTSTR lpCurrentDirectory,
  _In_         LPSTARTUPINFO lpStartupInfo,
  _Out_        LPPROCESS_INFORMATION
lpProcessInformation
);
```

Explaination

Example: Process creation (Linux *)

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/syscall.h>

int main(int argc, char *argv[]) {
   long pid;
   pid = syscall(SYS_clone, SIGCHLD,
                 NULL, NULL, NULL );
   sleep(3);
   printf("Hello world, pid=%ld\n", pid);
   return 0;
}
```

man clone

```
NAME
     clone, __clone2 - create a child process

SYNOPSIS
     /* Prototype for the glibc wrapper function */

     #include <sched.h>

     int clone(int (*fn)(void *), void *child_stack,
            int flags, void *arg, ...
            /* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );

     /* Prototype for the raw system call */

     long clone(unsigned long flags, void *child_stack,
            void *ptid, void *ctid,
            struct pt_regs *regs);
```

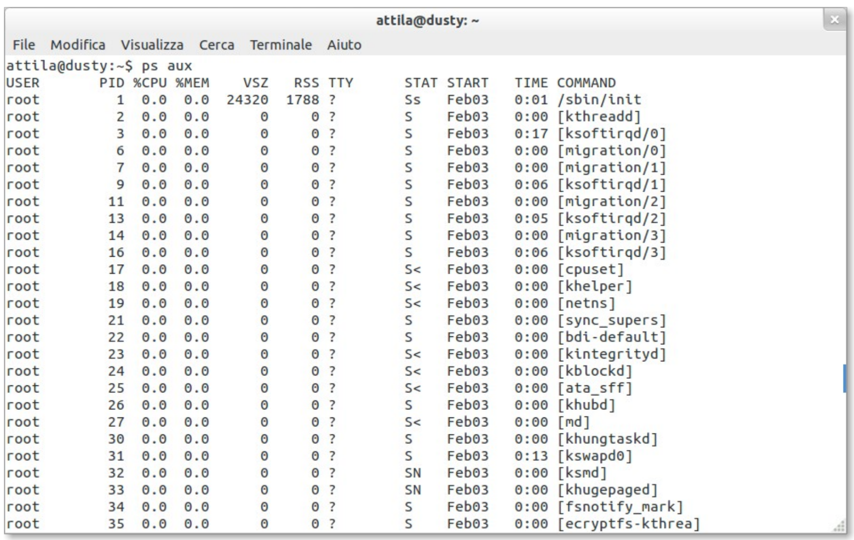
* we'll see a POSIX (**P**ortable **O**perating **S**ystem **I**nterface for uni**X**) way to create a process soon (→ fork, exec)

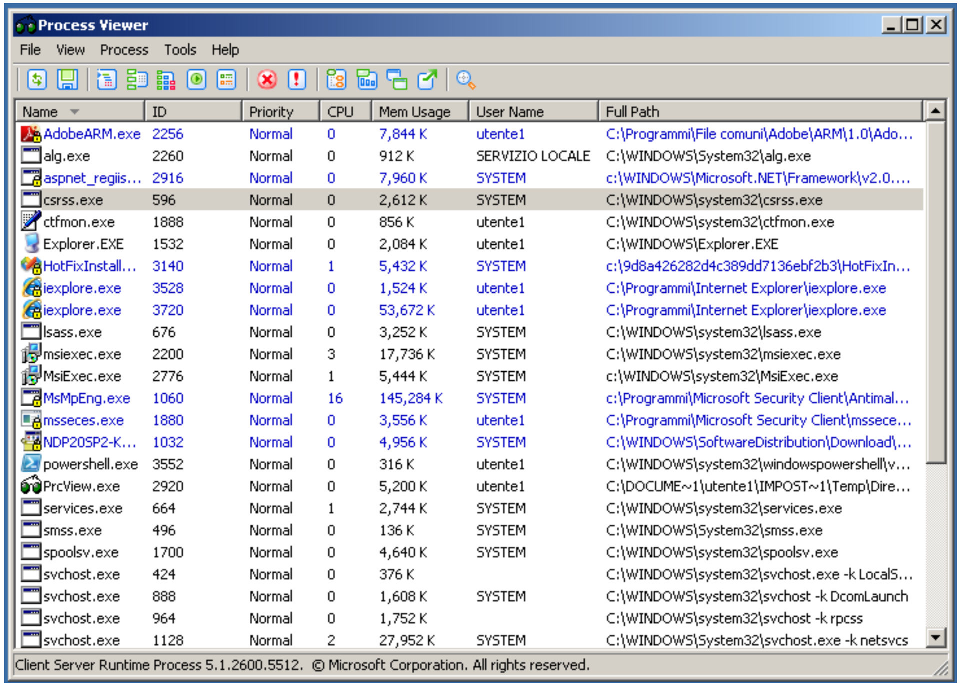
Explaination

How does the system identify processes?

Working with processes

- The kernel assigns each process with a process identifier (**PID**)





Answer

Linux /proc

- The Linux kernel exports many details about processes through the **`/proc`** virtual filesystem
  - Each process has its own sub-directory, named after the PID value

```
root@host:/proc/4361# ls
attr             cpuset   latency    mountstats     personality  stat
autogroup        cwd      limits     net            projid_map   statm
auxv             environ  loginuid   ns             root         status
cgroup           exe      map_files  numa_maps      sched        syscall
clear_refs       fd       maps       oom_adj        schedstat    task
cmdline          fdinfo   mem        oom_score      sessionid    timers
comm             gid_map  mountinfo  oom_score_adj  smaps        uid_map
coredump_filter  io       mounts     pagemap        stack        wchan
```

Explaination

Working with processes (Unix, C)

- The **getpid** function enables a process to know its assigned identifier

```
#include <unistd.h>

pid_t getpid(void);
```

Explaination

Creating new processes (Unix, C): fork

```
#include <unistd.h>

pid_t fork(void);
```

1. Creates a process data structure for the child process
2. Creates a new descriptor for the child process (→ **new PID**)
3. Copies the addressing space of the parent into the child's one*
4. Return values are:

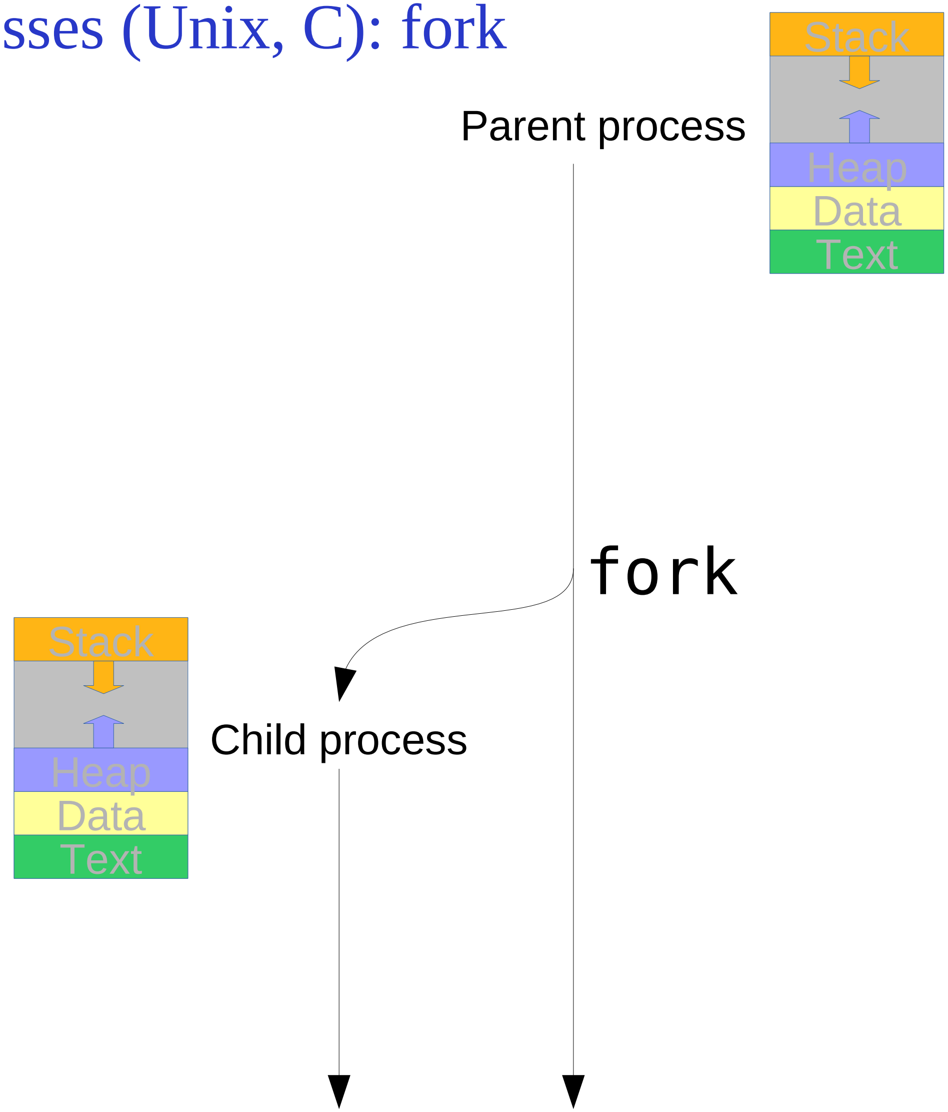
   To parent: the child's PID
   To child: 0 (zero)

   If fork fails: -1

* typically implemented as Copy-On-Write (**COW**) for efficiency reasons

Explaination

Creating new processes (Unix, C): fork



Creates a new addressing space (copy of the parent's one)

Explaination

Process hierarchy in Unix

- Fork creates a process hierarchy
  - The root is the **init** process(PID=1, created during the boot sequence)
- A process can obtain the identifier of its parent process using **getppid** (get parent pid)

```
#include <unistd.h>

pid_t getppid(void);
```

Explaination

fork example

```c
#include <unistd.h>
#include <stdio.h>

int main() {
	pid_t cpid;

	cpid = fork();

	if (cpid == (pid_t) -1) {
		printf("Error!\n");
	} else if (cpid == 0) {
		printf("I'm the child %d, parent pid is %d\n",
			getpid(), getppid());
	} else {
		printf("I'm the parent %d children pid is %d\n",
			getpid(), cpid);
	}
	return 0;
}
```

Running another executable

```
#include <unistd.h>

int execl(const char *path, const char*arg0, ...);
int execlp(const char *file, const char *arg0, ...);
...
```

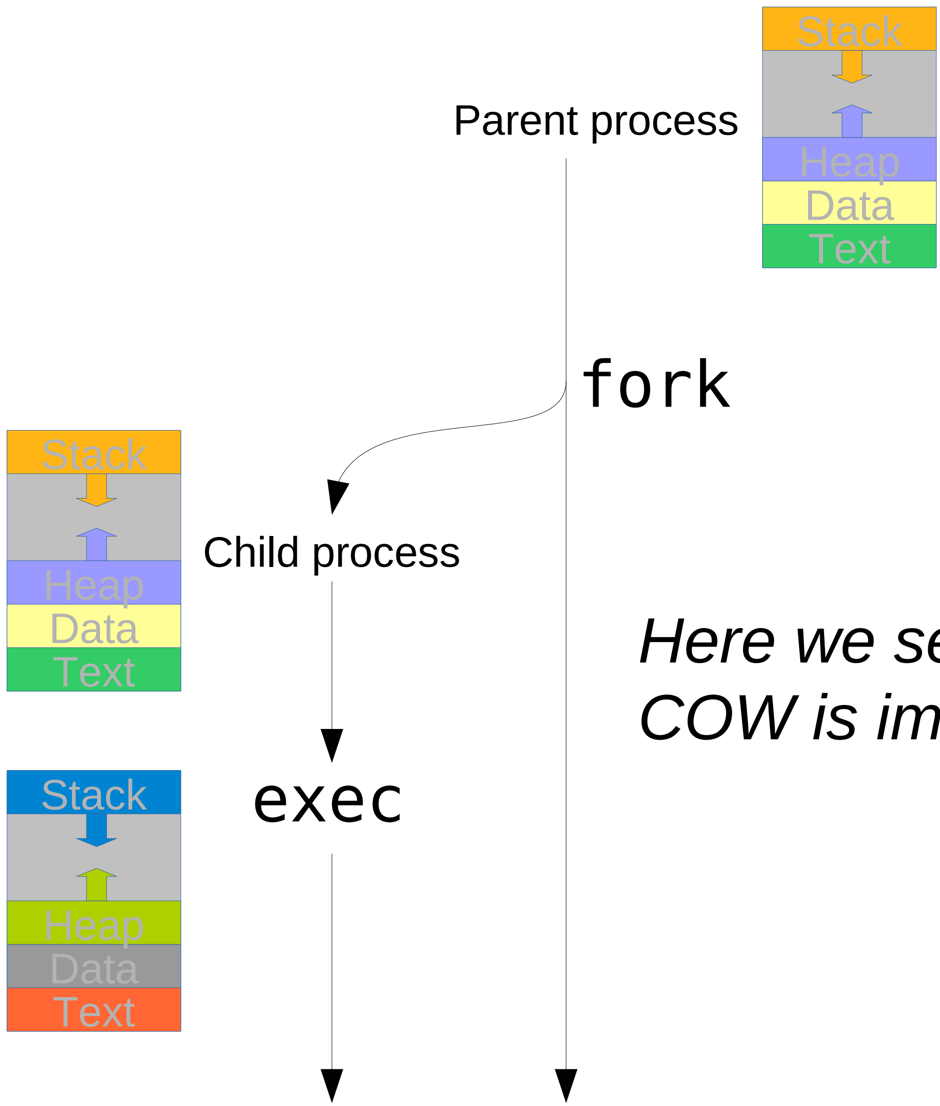
These functions (usually called right after a fork) **replace the Text, Data, BSS * segments of the process** with those loaded from a file, and set up the **Stack** and **Heap segments** accordingly

There exist many different variants of exec (see `man exec`)

* Block Started by Symbol, uninitialized data

Explaination

exec



Creates a new addressing space (copy of the parent's one)

Segments are replaced

Here we see why COW is important!

Explaination

exec example

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

int main() {
	pid_t cpid;

	cpid = fork();

	if (cpid == (pid_t) -1) {
		printf("Error!\n");
	} else if (cpid == 0) {
		execl("/bin/ls", "ls", 0, NULL);
		printf("This should never be printed, unless exec fails\n");
	} else {
		printf("I'm the parent\n");
	}
	return 0;
}
```

Explaination

Terminating a process

- A process terminates when
  - **`return`** is invoked from the main procedure
  - the **`exit(int)`** procedure is called
- The return value / exit value can be read by the parent process

```
#include <stdlib.h>

void exit(int status);
```

Explaination

Wait for the child process exit value

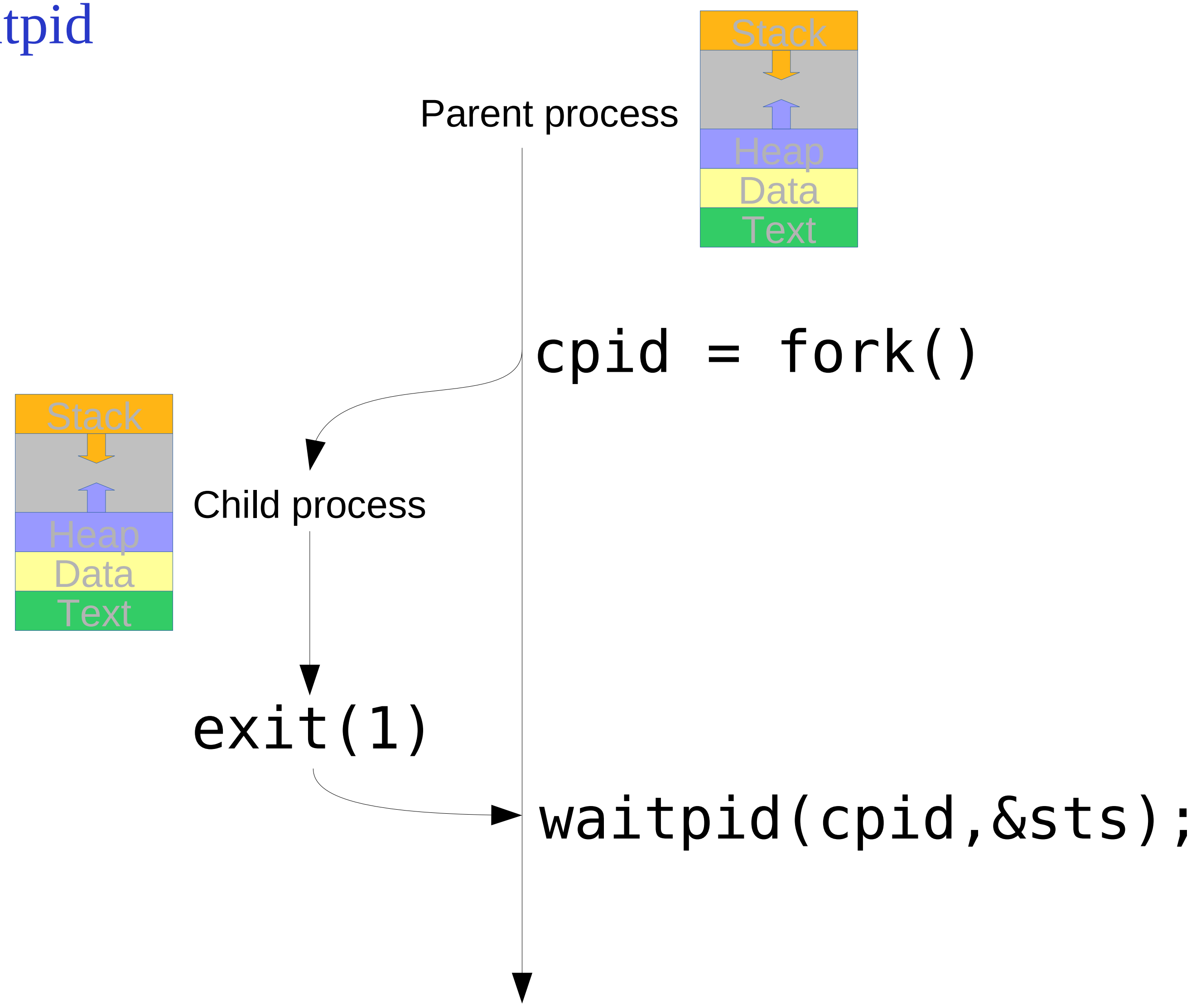
- The parent process can wait for the child process to terminate (and get its exit value) with **wait** e **waitpid**

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

pid_t wait(int *status);

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

The SIGCHLD signal is sent to the parent of a child process when it exits, is interrupted, or resumes after being interrupted. By default the signal is simply ignored.

Explaination

waitpid



Explaination

waitpid example

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
pid_t cpid;
int sts;
int main(void) {
  if (cpid = fork()) {
    waitpid(cpid, &sts, 0);
    printf("Child process exited with status = %d\n", WEXITSTATUS(sts));
  } else if (cpid == 0) {
    printf("Child process\n");
    exit(42);
  }
}
```

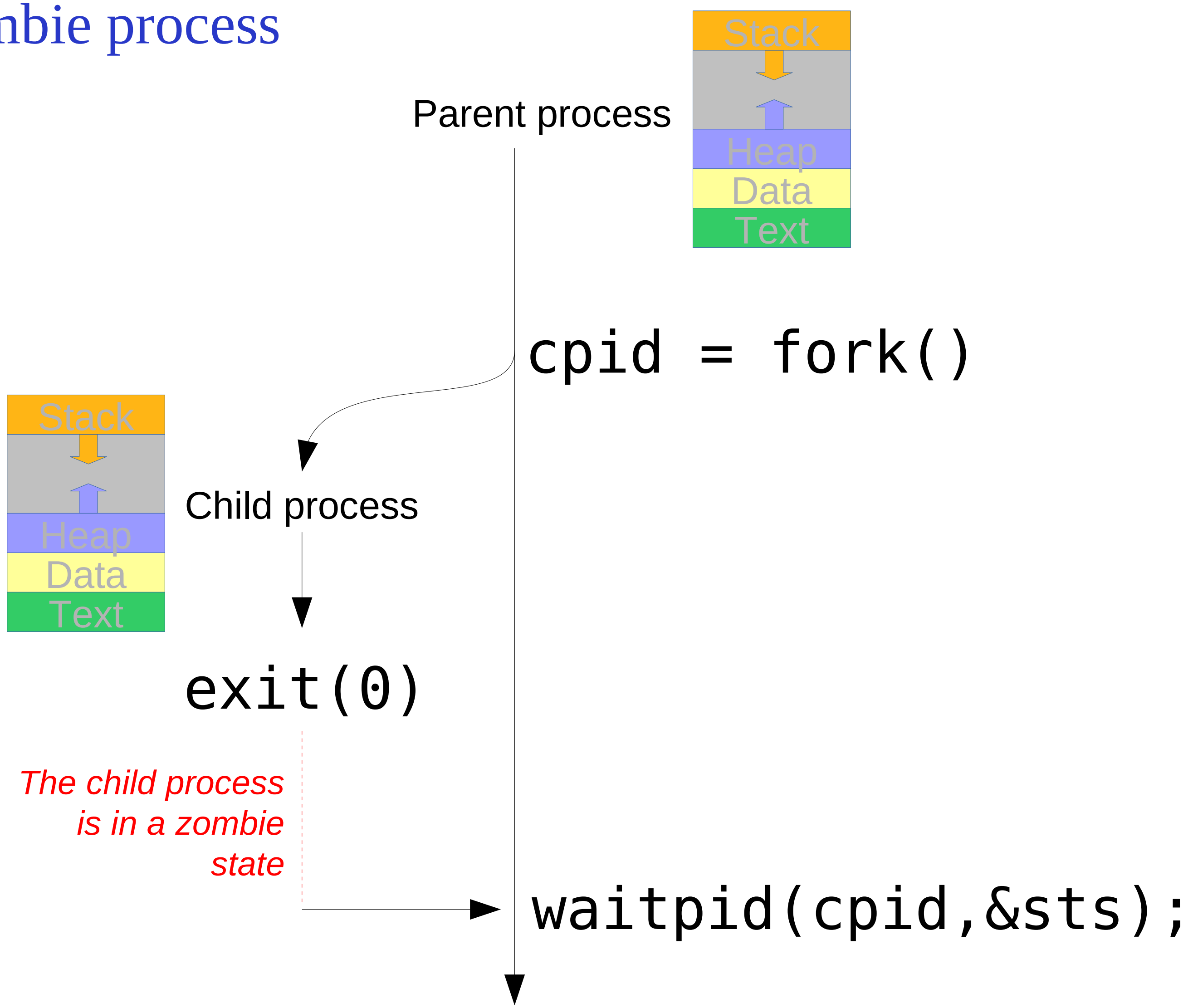
Explaination

Zombie process

- When a child process terminates its parent <u>must</u> wait for it with **`wait`** or **`waitpid`**
- As long as the parent process does not wait for a child, the latter remains in a **zombie** (o defunct) state, and cannot be removed from the process list
- If the parent terminates, the child is inherited by the *init* process (PID 1): this behavior is called *re-parenting*, and is used to create Unix daemons (*<u>double fork</u>*)
  - The SIGCHLD signal is sent to the parent when the child exits
  - The child process is called *orphaned process*
  - If the child was in a zombie state, the init process will take care of it (i.e. reaping it)

Explaination

Zombie process



Explaination

Zombie example

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

int main() {
	pid_t cpid;

	cpid = fork();

	if (cpid == (pid_t) -1) {
		printf("Error!\n");
	} else if (cpid == 0) {
		printf("I'm the child\n");
	} else {
		printf("I'm the parent\n");
		sleep(30);
	}
	return 0;
}
```

Explaination

Inherited (zombie) process

```c
#include <unistd.h>
#include <stdio.h>

int main() {
	pid_t cpid;

	cpid = fork();

	if (cpid == (pid_t) -1) {
		printf("Error!\n");
	} else if (cpid == 0) {
		printf("I'm the child\n");
		sleep(30);
	} else {
		printf("I'm the parent\n");
	}
	return 0;
}
```

Explaination

Wrap Up