

**SUPSI**

# Processes and threads

Operating Systems

Amos Brocco, Lecturer & Researcher

## 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

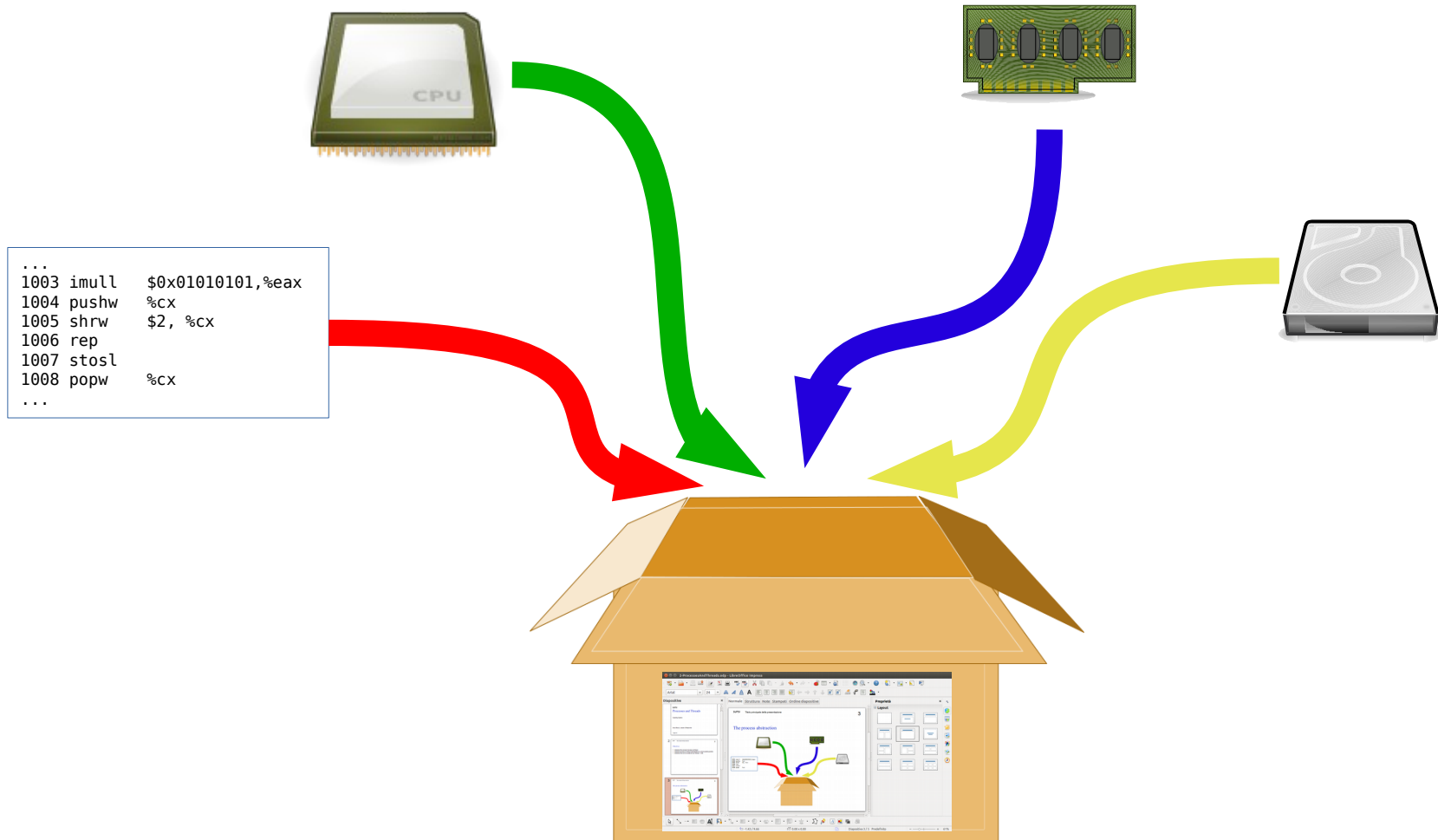
- Read it and try to understand the concepts.

### 📖 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!*

## The process abstraction

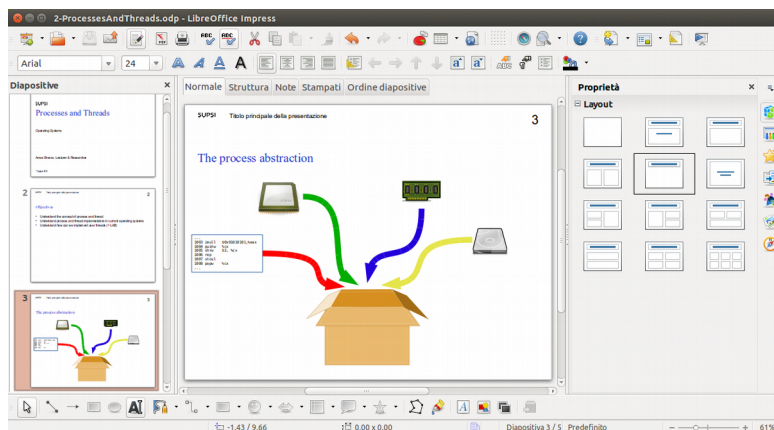




## The process

- 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

```
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;

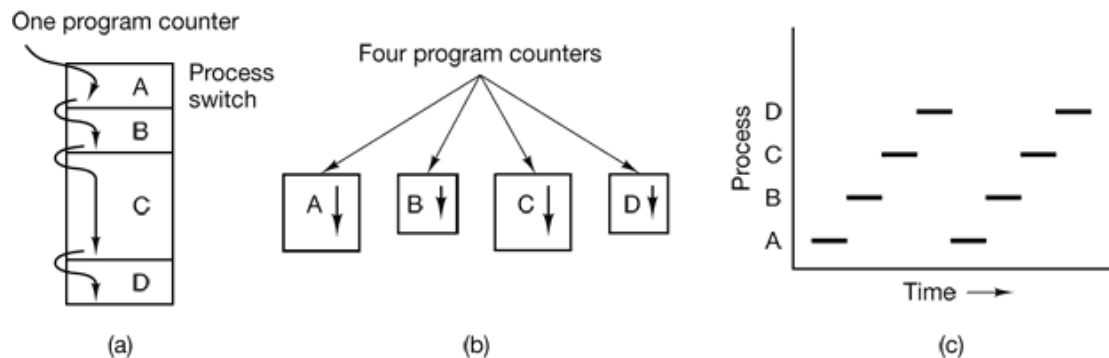
```

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



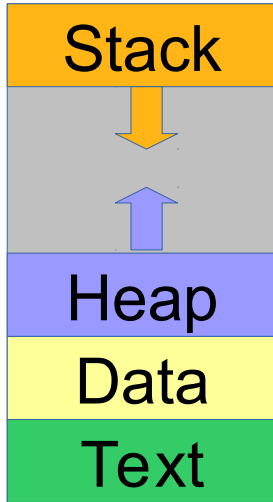
## Program vs process

- 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



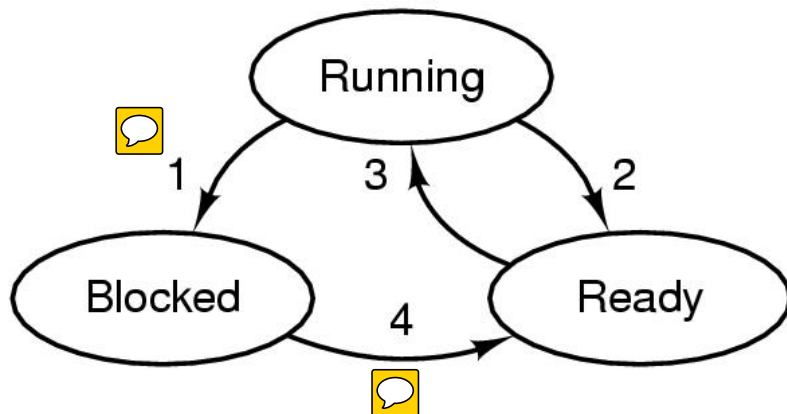


## Process attributes (overview)

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  They define the address space 	Root directory Working directory File descriptors User ID Group ID



## The process state (simplified)





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





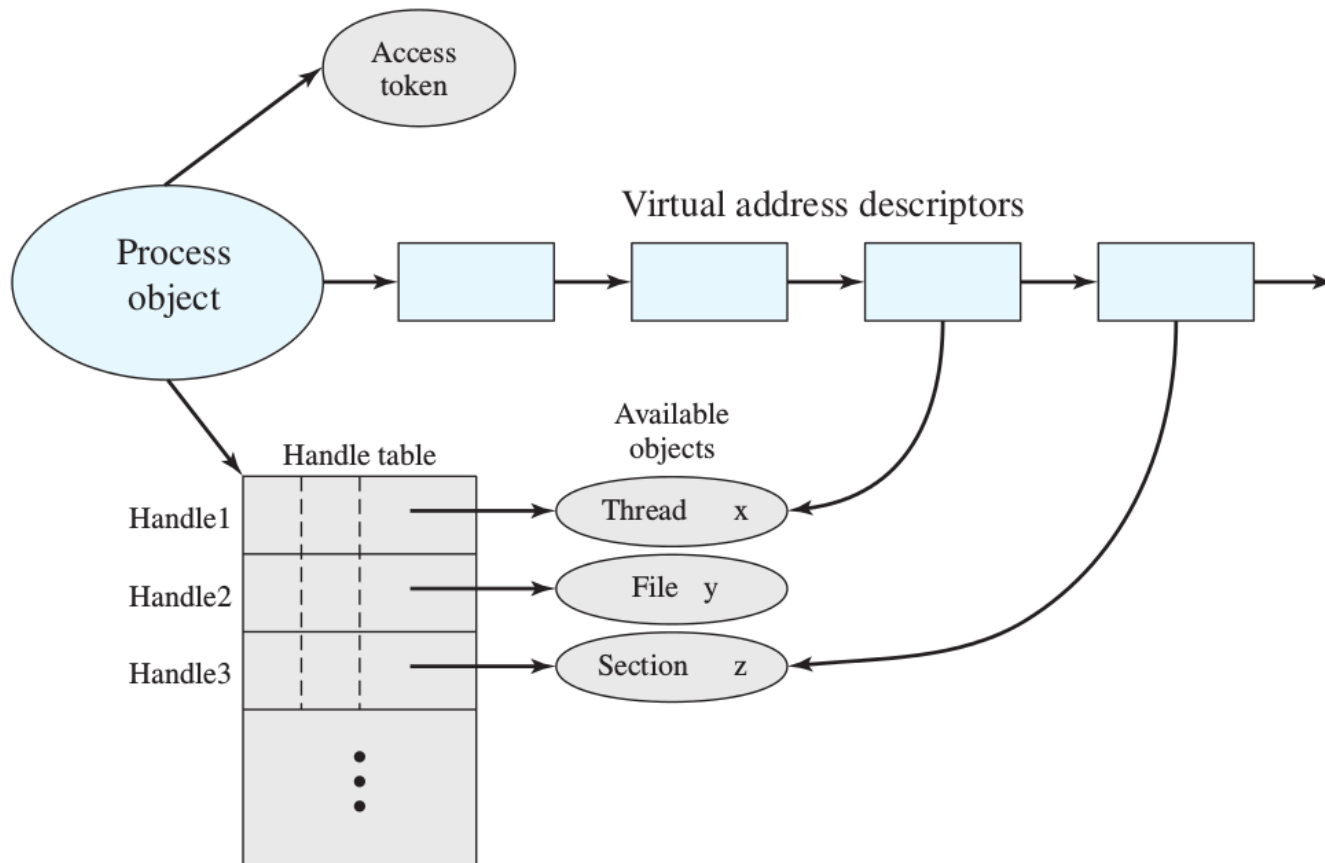


## Life of a process

- 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)



## Process implementation (Windows)



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

## Process creation (Windows example)

```
#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_ LPCTSTR 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
);
```

## 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 creation (example \*)

```
#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 (Portable Operating System Interface for uniX) way to create a process soon (→ fork, exec)

## Process creation (example)

**Table 4.5** Linux clone () flags

<b>CLONE_CLEARID</b>	Clear the task ID.
<b>CLONE_DETACHED</b>	The parent does not want a SIGCHLD signal sent on exit.
<b>CLONE_FILES</b>	Shares the table that identifies the open files.
<b>CLONE_FS</b>	Shares the table that identifies the root directory and the current working directory, as well as the value of the bit mask used to mask the initial file permissions of a new file.
<b>CLONE_IDLETASK</b>	Set PID to zero, which refers to an idle task. The idle task is employed when all available tasks are blocked waiting for resources.
<b>CLONE_NEWNS</b>	Create a new namespace for the child.
<b>CLONE_PARENT</b>	Caller and new task share the same parent process.
<b>CLONE_PTRACE</b>	If the parent process is being traced, the child process will also be traced.
<b>CLONE_SETTID</b>	Write the TID back to user space.
<b>CLONE_SETTLS</b>	Create a new TLS for the child.
<b>CLONE_SIGHAND</b>	Shares the table that identifies the signal handlers.
<b>CLONE_SYSVSEM</b>	Shares System V SEM_UNDO semantics.
<b>CLONE_THREAD</b>	Inserts this process into the same thread group of the parent. If this flag is true, it implicitly enforces CLONE_PARENT.
<b>CLONE_VFORK</b>	If set, the parent does not get scheduled for execution until the child invokes the <i>execve()</i> system call.
<b>CLONE_VM</b>	Shares the address space (memory descriptor and all page tables):



## Working with processes

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

```

attila@dusty: ~
File Modifica Visualizza Cerca Terminale Aiuto
attila@dusty:~$ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  24320  1788 ?        Ss   Feb03   0:01 /sbin/init
root         2  0.0  0.0      0     0 ?        S    Feb03   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    Feb03   0:17 [ksoftirqd/0]
root         6  0.0  0.0      0     0 ?        S    Feb03   0:00 [migration/0]
root         7  0.0  0.0      0     0 ?        S    Feb03   0:00 [migration/1]
root         9  0.0  0.0      0     0 ?        S    Feb03   0:06 [ksoftirqd/1]
root        11  0.0  0.0      0     0 ?        S    Feb03   0:00 [migration/2]
root        13  0.0  0.0      0     0 ?        S    Feb03   0:05 [ksoftirqd/2]
root        14  0.0  0.0      0     0 ?        S    Feb03   0:00 [migration/3]
root        16  0.0  0.0      0     0 ?        S    Feb03   0:06 [ksoftirqd/3]
root        17  0.0  0.0      0     0 ?        S<   Feb03   0:00 [cpuset]
root        18  0.0  0.0      0     0 ?        S<   Feb03   0:00 [khelper]
root        19  0.0  0.0      0     0 ?        S<   Feb03   0:00 [netns]
root        21  0.0  0.0      0     0 ?        S    Feb03   0:00 [sync_supers]
root        22  0.0  0.0      0     0 ?        S    Feb03   0:00 [bdi-default]
root        23  0.0  0.0      0     0 ?        S<   Feb03   0:00 [kintegrityd]
root        24  0.0  0.0      0     0 ?        S<   Feb03   0:00 [kblockd]
root        25  0.0  0.0      0     0 ?        S<   Feb03   0:00 [ata_sff]
root        26  0.0  0.0      0     0 ?        S    Feb03   0:00 [khubd]
root        27  0.0  0.0      0     0 ?        S<   Feb03   0:00 [nd]
root        30  0.0  0.0      0     0 ?        S    Feb03   0:00 [khungtaskd]
root        31  0.0  0.0      0     0 ?        S    Feb03   0:13 [kswapd0]
root        32  0.0  0.0      0     0 ?        SN   Feb03   0:00 [ksnd]
root        33  0.0  0.0      0     0 ?        SN   Feb03   0:00 [khugepaged]
root        34  0.0  0.0      0     0 ?        S    Feb03   0:00 [fsnotify_mark]
root        35  0.0  0.0      0     0 ?        S    Feb03   0:00 [ecryptfs-kthrea]
  
```

Process Viewer						
File View Process Tools Help						
Name	ID	Priority	CPU	Mem Usage	User Name	Full Path
AdobeARM.exe	2256	Normal	0	7,844 K	utente1	C:\Programmi\File comuni\Adobe\ARM\1.0\Ado...
alg.exe	2260	Normal	0	912 K	SERVIZIO LOCALE	C:\WINDOWS\System32\alg.exe
aspnet_regiis...	2916	Normal	0	7,960 K	SYSTEM	c:\WINDOWS\Microsoft.NET\Framework\v2.0....
csrss.exe	596	Normal	0	2,612 K	SYSTEM	C:\WINDOWS\system32\csrss.exe
ctfmon.exe	1888	Normal	0	856 K	utente1	C:\WINDOWS\system32\ctfmon.exe
Explorer.EXE	1532	Normal	0	2,084 K	utente1	C:\WINDOWS\Explorer.EXE
HotFixInstall...	3140	Normal	1	5,432 K	SYSTEM	c:\9d8a426282d4c389dd7136ebf2b3\HotFixIn...
ieexplore.exe	3528	Normal	0	1,524 K	utente1	C:\Programmi\Internet Explorer\ieexplore.exe
ieexplore.exe	3720	Normal	0	53,672 K	utente1	C:\Programmi\Internet Explorer\ieexplore.exe
lsass.exe	676	Normal	0	3,252 K	SYSTEM	C:\WINDOWS\system32\lsass.exe
msiexec.exe	2200	Normal	3	17,736 K	SYSTEM	C:\WINDOWS\system32\msiexec.exe
MsiExec.exe	2776	Normal	1	5,444 K	SYSTEM	c:\WINDOWS\system32\MsiExec.exe
MsmgEng.exe	1060	Normal	16	145,284 K	SYSTEM	c:\Programmi\Microsoft Security Client\Antimal...
msseces.exe	1880	Normal	0	3,556 K	utente1	C:\Programmi\Microsoft Security Client\mssece...
NDP20SP2-K...	1032	Normal	0	4,956 K	SYSTEM	C:\WINDOWS\SoftwareDistribution\Download\...
powershell.exe	3552	Normal	0	316 K	utente1	C:\WINDOWS\system32\windowspowershell\v...
PrcView.exe	2920	Normal	0	5,200 K	utente1	C:\DOCUME~1\utente1\IMPOST~1\Temp\Dire...
services.exe	664	Normal	1	2,744 K	SYSTEM	C:\WINDOWS\system32\services.exe
smss.exe	496	Normal	0	136 K	SYSTEM	C:\WINDOWS\System32\smss.exe
spoolsv.exe	1700	Normal	0	4,640 K	SYSTEM	C:\WINDOWS\system32\spoolsv.exe
svchost.exe	424	Normal	0	376 K	SYSTEM	C:\WINDOWS\system32\svchost.exe -k Local5...
svchost.exe	888	Normal	0	1,608 K	SYSTEM	C:\WINDOWS\system32\svchost -k DcomLaunch
svchost.exe	964	Normal	0	1,752 K	SYSTEM	C:\WINDOWS\system32\svchost -k rpcss
svchost.exe	1128	Normal	2	27,952 K	SYSTEM	C:\WINDOWS\system32\svchost.exe -k netsvcs

Client: Server Runtime Process 5.1.2600.5512. © Microsoft Corporation. All rights reserved.



## 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
```





## Working with processes (Unix, C)

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

```
#include <unistd.h>

pid_t getpid(void);
```



## Creating new processes (Unix, C): fork

```
#include <unistd.h>

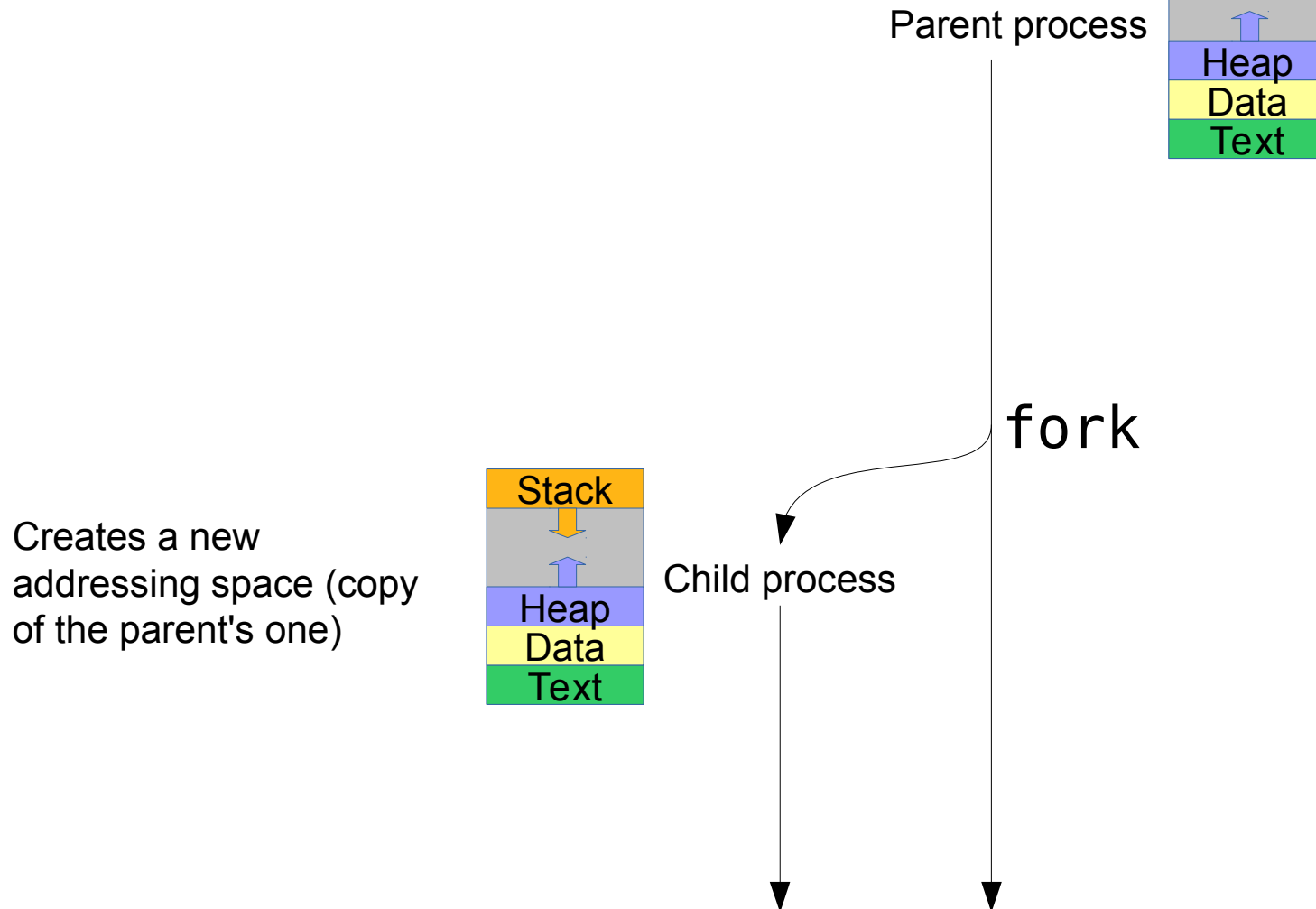
pid_t fork(void);
```

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



## Creating new processes (Unix, C): fork





## 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);
```



## fork 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 %d, parent pid is %d\n",
            getpid(), getppid());
    } else {
        printf("I'm the parent %d children pid is %d\n",
            getpid(), cpid);
    }
    return 0;
}
```



## Run 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)

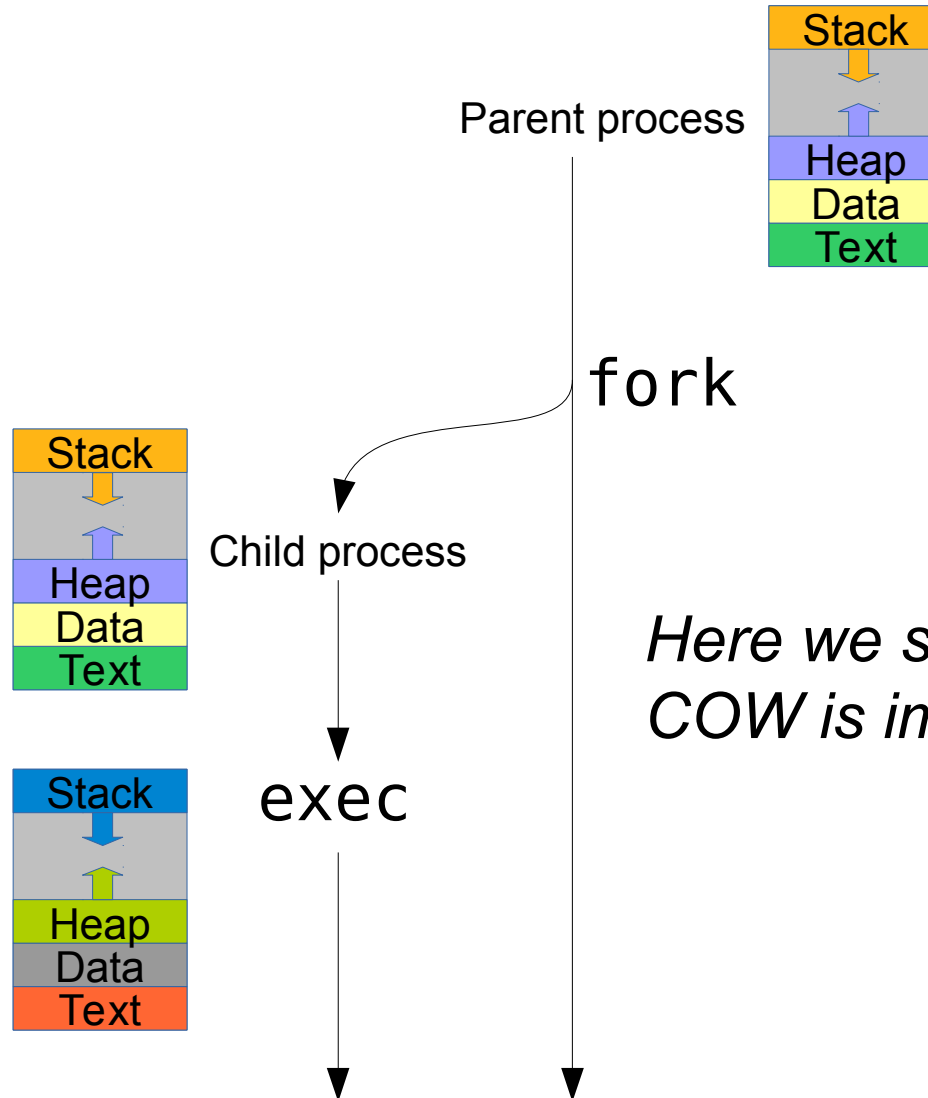
\* Block Started by Symbol, uninitialized data



exec

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

Segments are replaced



*Here we see why  
COW is important!*



## 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;
}
```





## 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);
```



## 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>

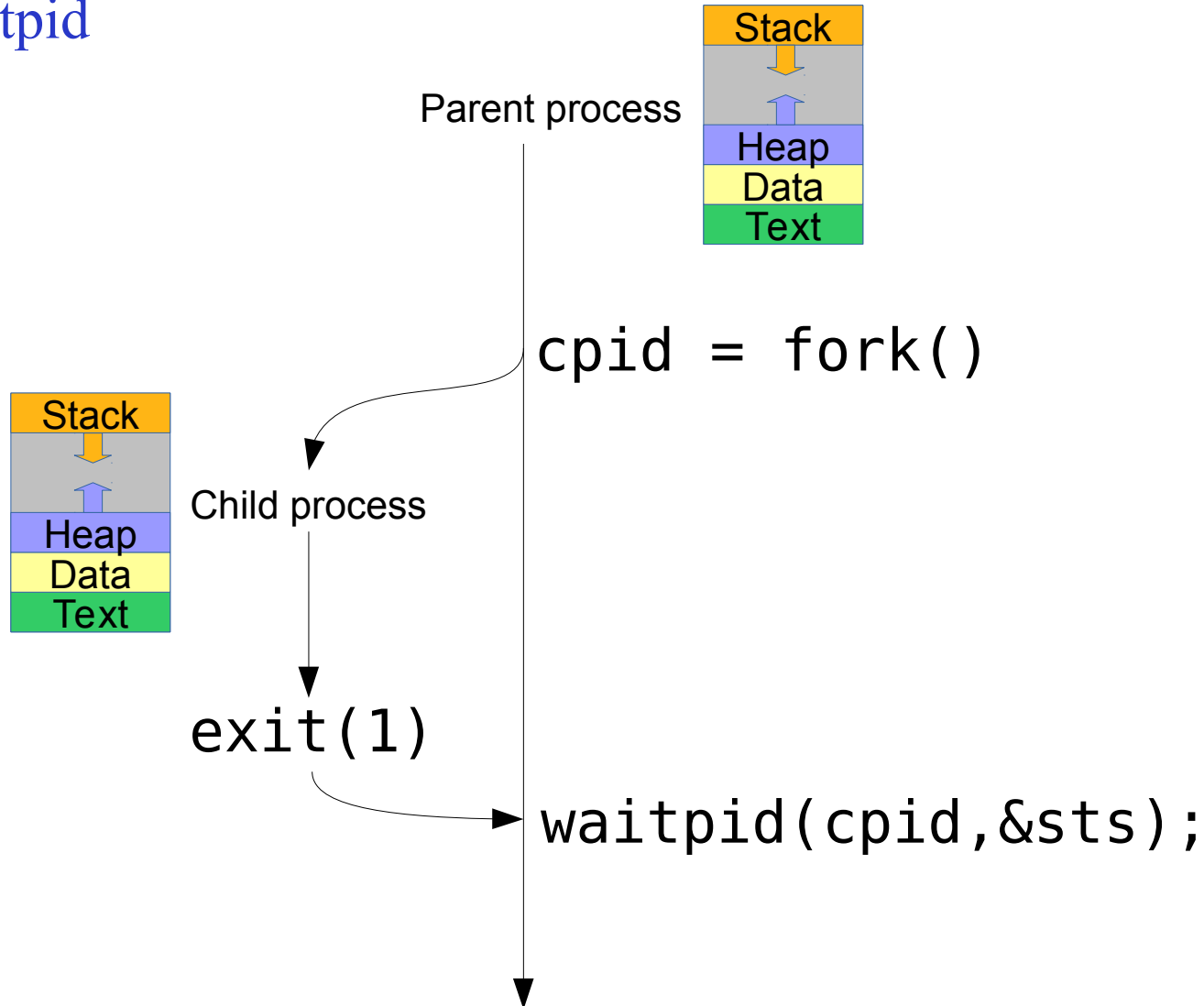
pid_t wait(int *status);

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.



## waitpid





## waitpid example

```
#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);
    }
}
```

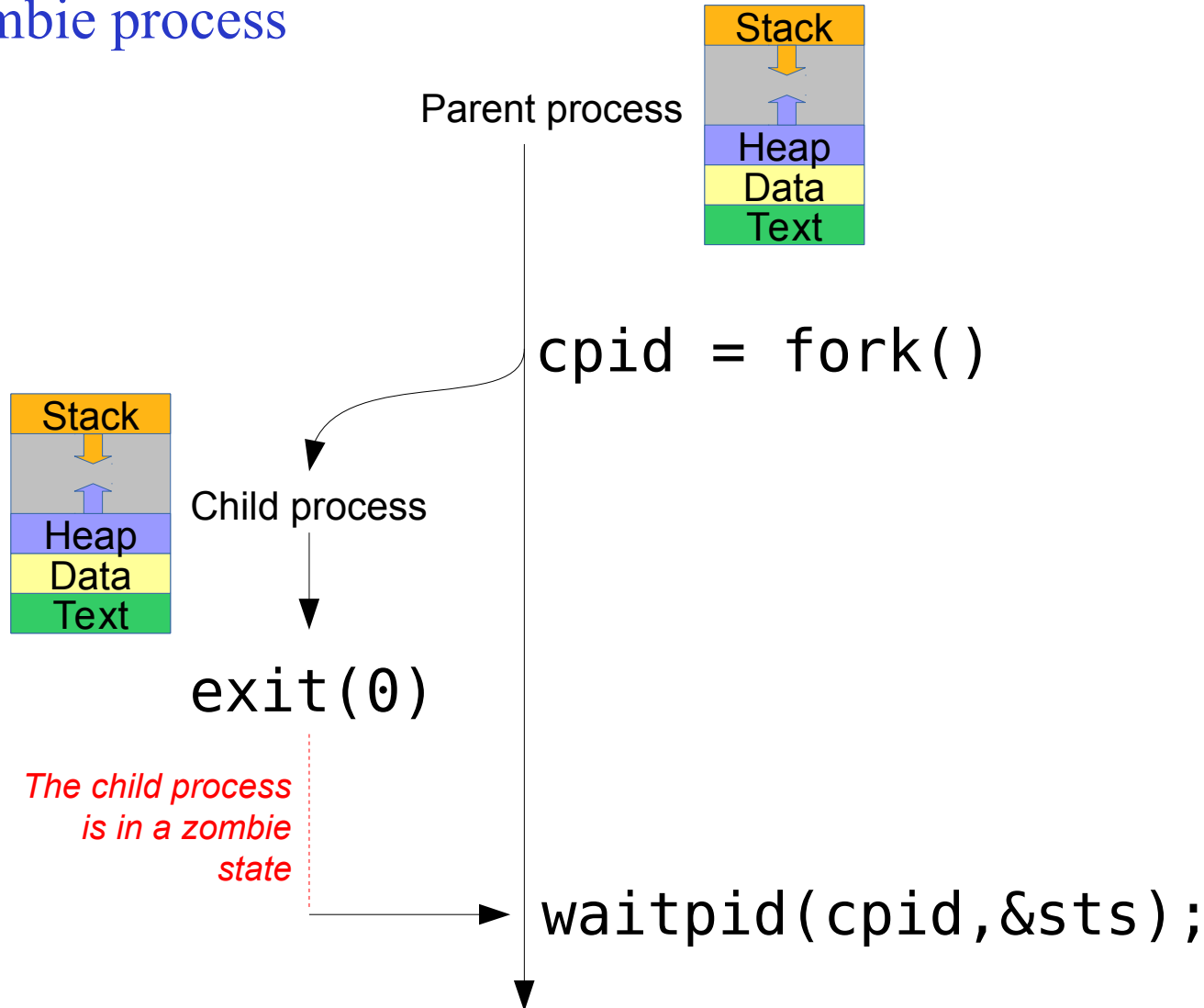


## Zombie process

- When a child process terminates its parent must 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 (double fork)
  - 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)



## Zombie process





## 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;
}
```



## Inherited (zombie) process

```
#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;
}
```



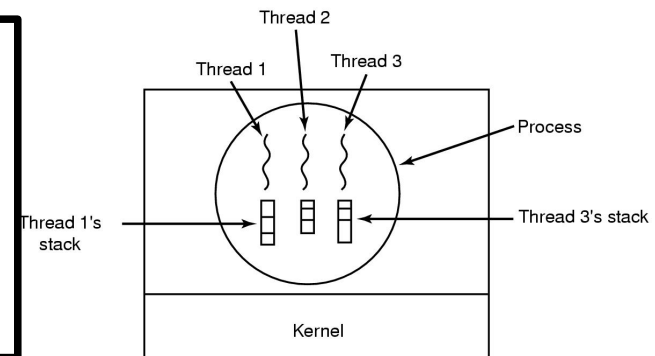




## The active part of a process: threads

- 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	



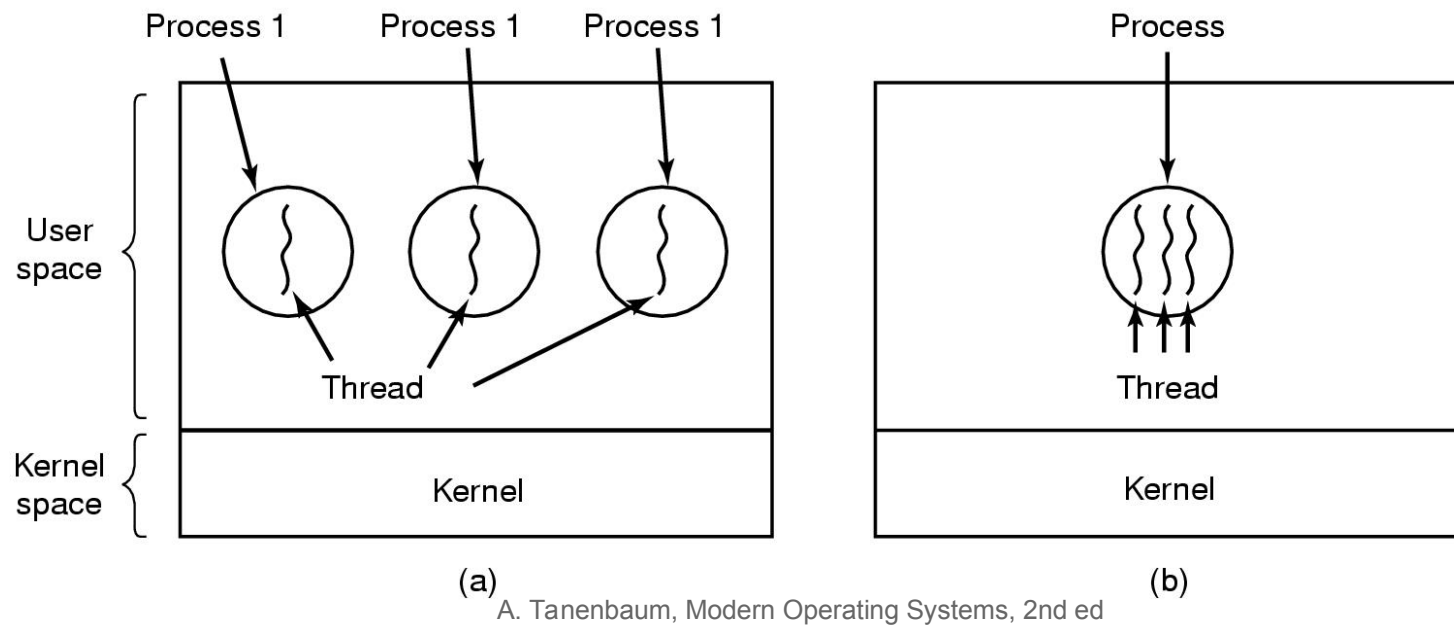
A. Tanenbaum, Modern Operating Systems, 2nd ed

- 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**



## The active part of a process: threads



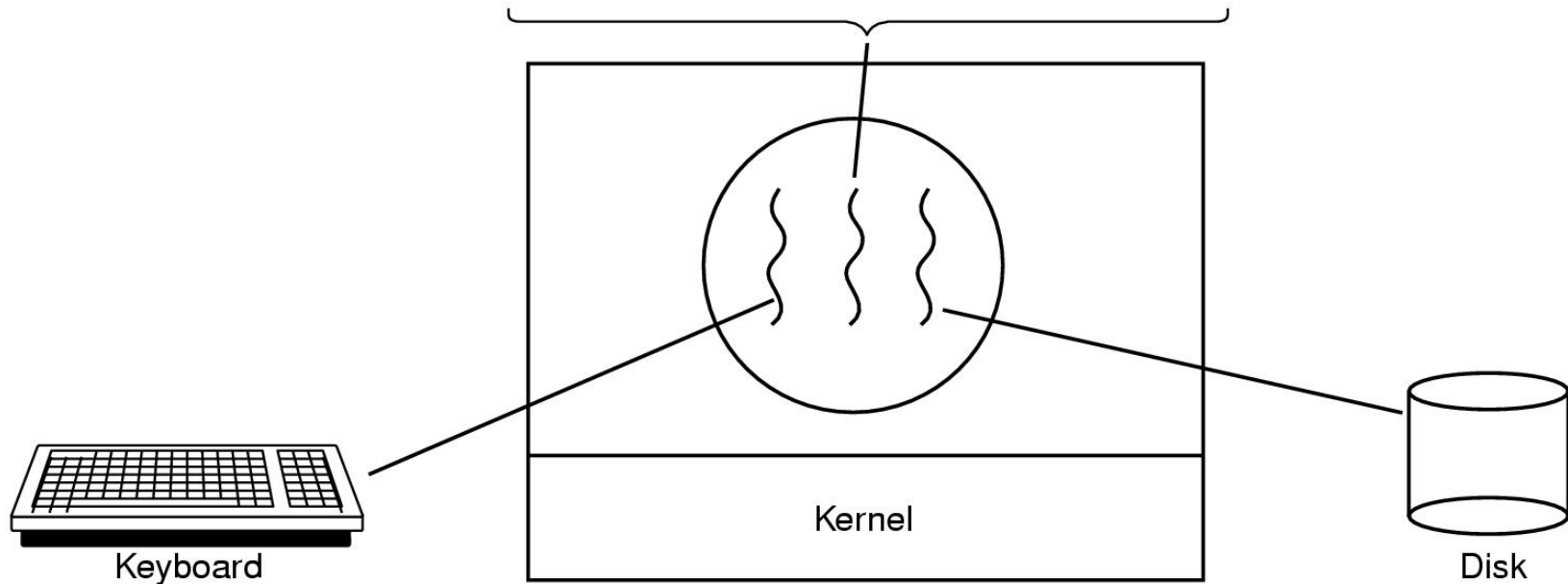
Multiple single-threaded  
processes

One multi-threaded  
process



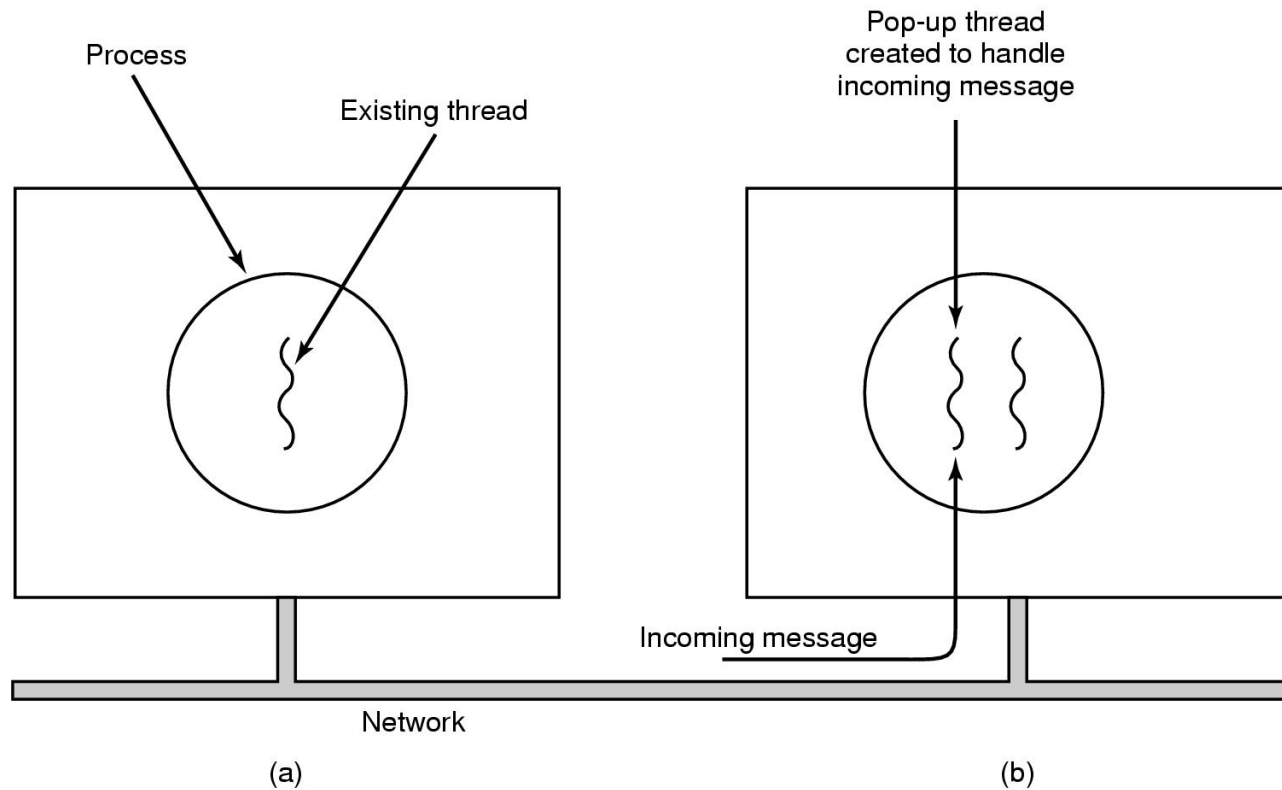
## Why multi-threading?

Four score and seven years ago, our fathers brought forth upon this continent a new nation: conceived in liberty, and dedicated to the proposition that all men are created equal.	Now we are engaged in a great civil war testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.	We have come to dedicate a portion of that field as a final resting place for those who here gave their	lives that this nation might live. It is altogether fitting and proper that we should do this.	But, in a larger sense, we cannot dedicate, we cannot consecrate we cannot hallow this ground. The brave men, living and dead,	who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here.	It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that from these honored dead we take increased devotion to that cause for which	they gave the last full measure of devotion, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom and that government of the people by the people
--	---	---	--	--	--	--	---



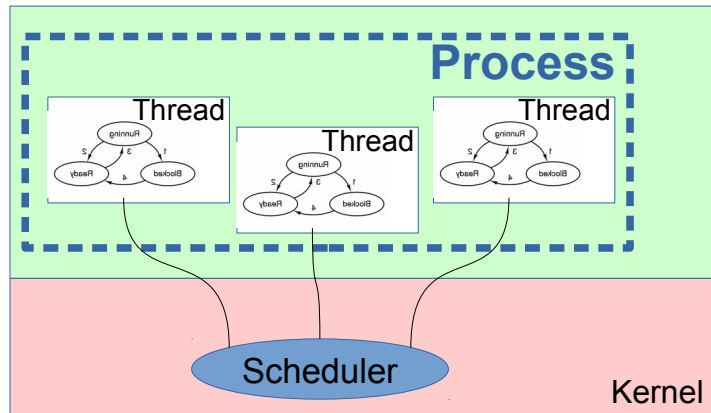


## Why multi-threading?



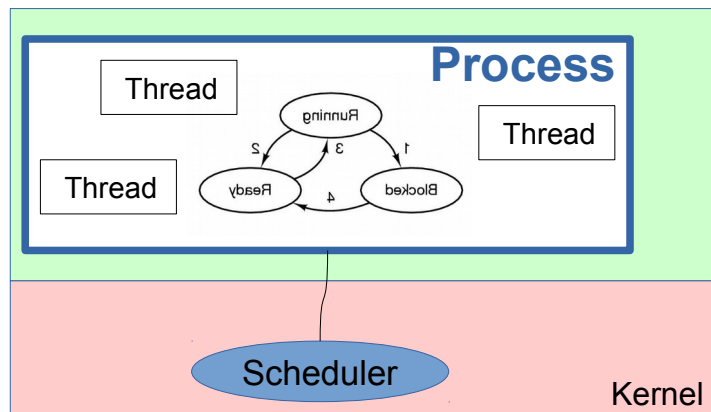


## Threads implementation



- 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!



- 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