System Programming

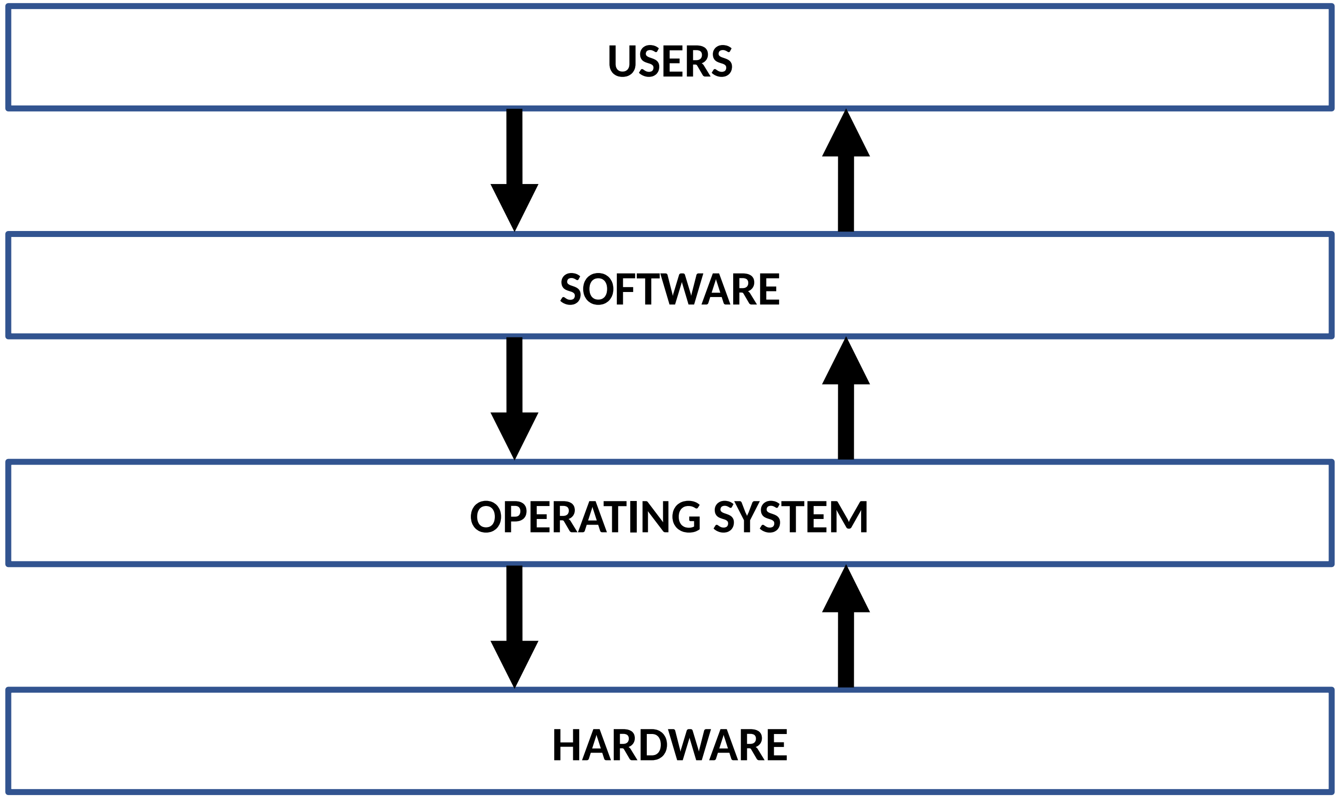
https://opale.iut-clermont.uca.fr/info/wiki/doku.php?id=progsys:progsys

Class Syllabus

1. Introduction
2. Process
3. File Management System
4. Pipelines
5. Signals
6. Memory Management
7. Ressources Management
8. Socket API using C Programming

Operating System (OS) : Definition

- Main program connecting the hardware and the software



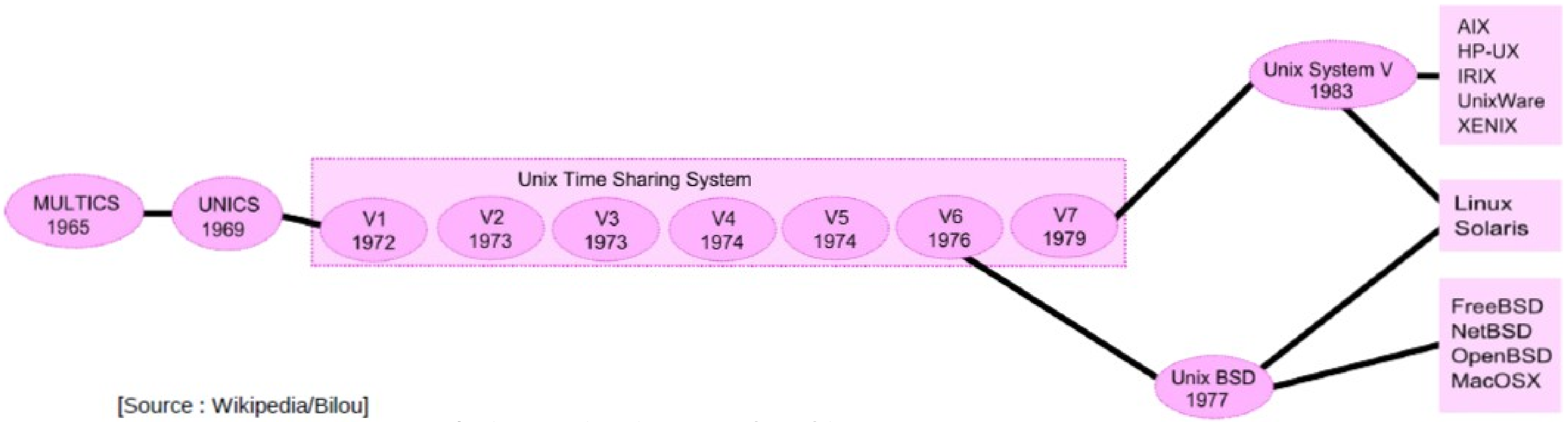
Operating System (OS) : Objectives

- Provide users with an interface masking the hardware

- Managing and sharing the ressources of the machine
  - Process (order, communication)
  - Memory
  - File System
  - Network

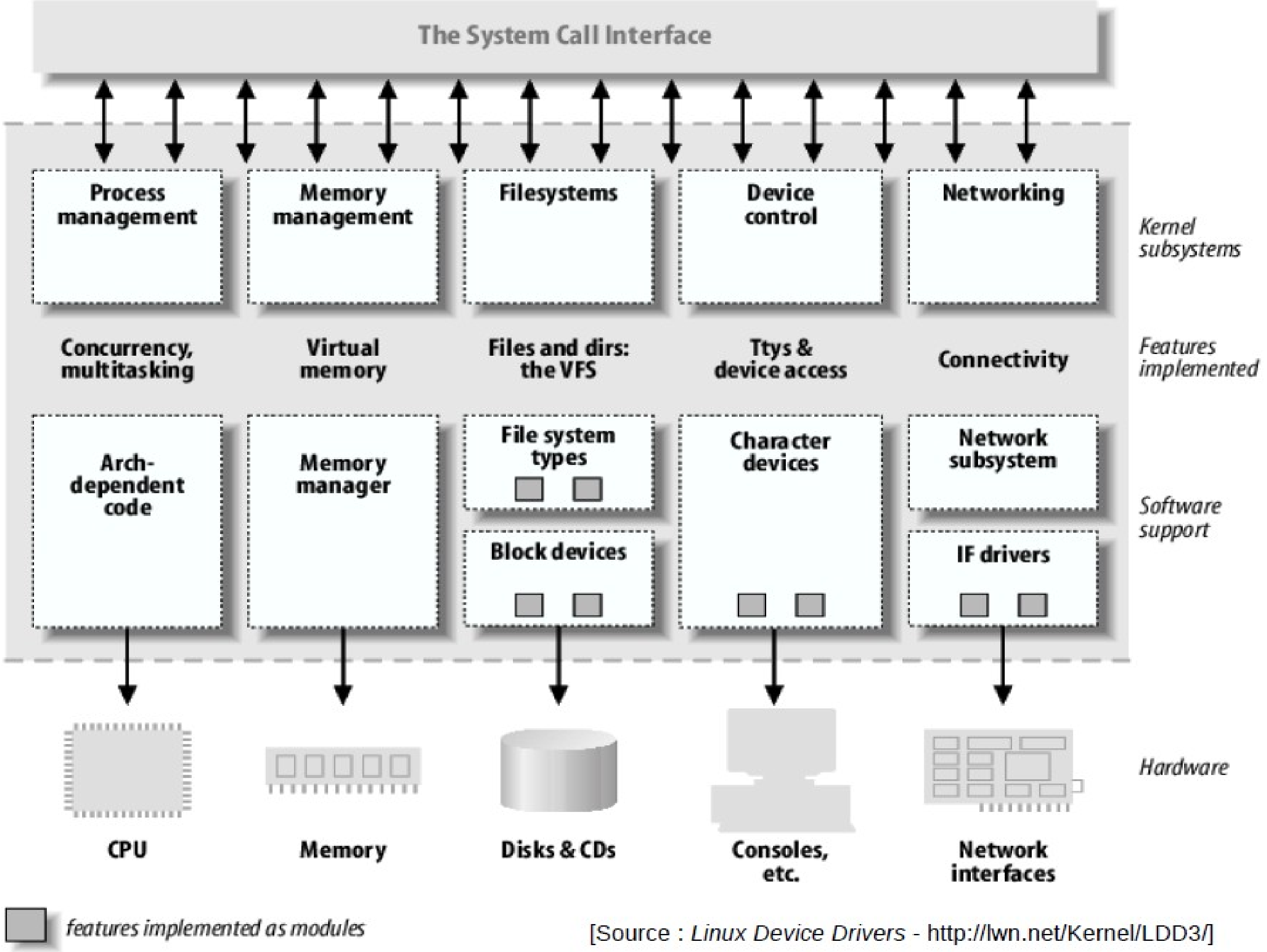
Operating System (OS) : Families

- DOS
- Windows
- UNIX
  - AIX (IBM)
  - Solaris, Open Solaris
  - LynxOS (RTOS)
  - QNX (RTOS)
  - Linux
  - OpenBSD, FreeBSD
  - NetBSD
  - MacOS X
- AS400
- *etc.*



[Source : Wikipedia/Bilou]

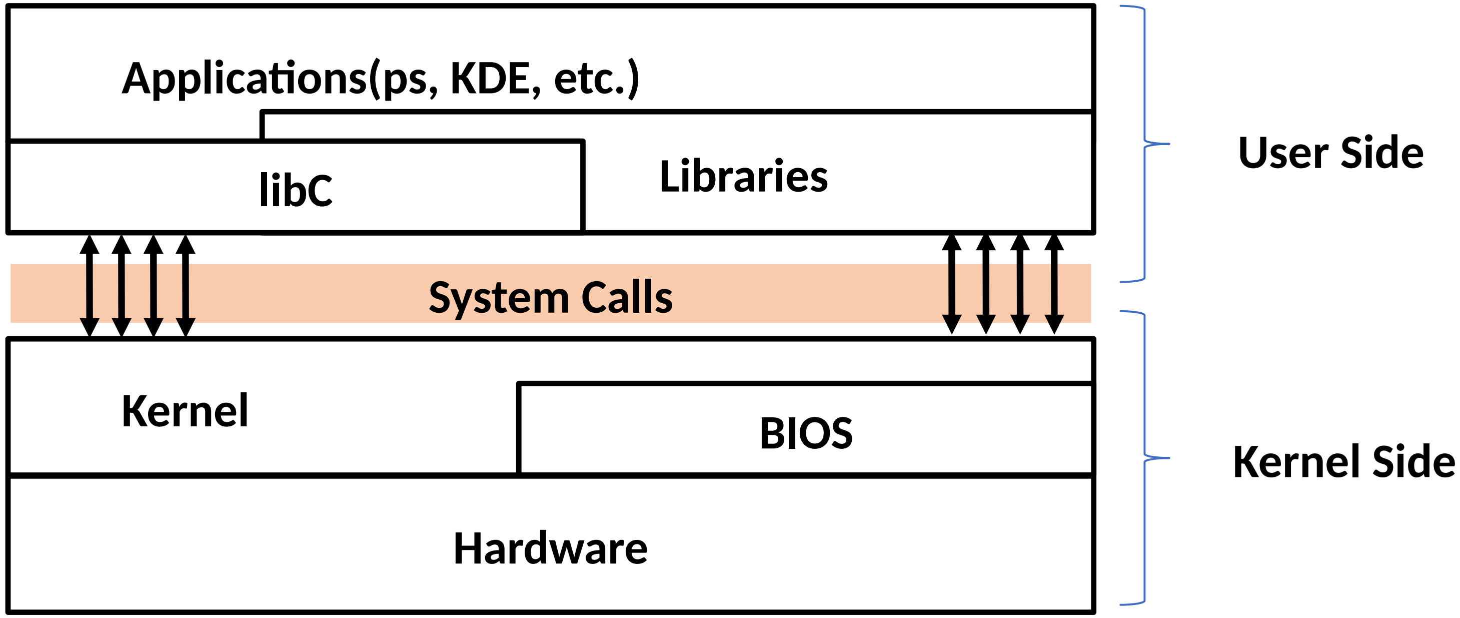
Linux Kernel Functions



[Source : *Linux Device Drivers* - http://lwn.net/Kernel/LDD3/]

System Call (Syscall)

- Communication from the **Application layer** to the **System layer** is handled by the **System Call API**



Did I just read API?

- API = Application Programming Interface

Program	API	Library or System

Program:

```
int a, b, c;
a = 2;
b = 3;
c = mult(a, b);
printf("%d", c);
```

API:

```
int mult(int a, int b);
```

mult is a function taking 2 integers as arguments and returning their product

Library or System:

First implementation

```
int mult(int a, int b)
{
    return a * b;
}
```

Second implementation

```
int mult(int a, int b)
{
    int i, ret=0;
    for(i=1; i<=b; ++i)
      ret += a;
    return ret;
}
```

"POSIX" Systems

- **POSIX = Portable Operating System Interface (Unix based)**

- Family of standards specified by the **IEEE** (**Institute of Electrical and Electronics Engineers**)

- **POSIX** defines
  - Shell Command-Lines (ksh, ls, man, etc.)
  - System call API
  - Real-time extensions
  - Thread API (light process)

Class Purpose

- The System Programming Class aims to present the various POSIX System Calls and how you can use them

- Those System Calls are regrouped in 6 main categories
  1. Process
  2. Hard Drive storage
  3. Pipes
  4. Signals
  5. Memory management
  6. Resources management
  7. Socket API

Chapter 2
Process

Class Syllabus

1.1) General Instructions

1.2) Cycle of life

- 1.2.1) Creation
- 1.2.2) Errors
- 1.2.3) Waiting
- 1.2.4) Replacement
- 1.2.5) Termination

1.3) Communication between processes

1.4) Light Processes (threads)

Process : definition (1/2)

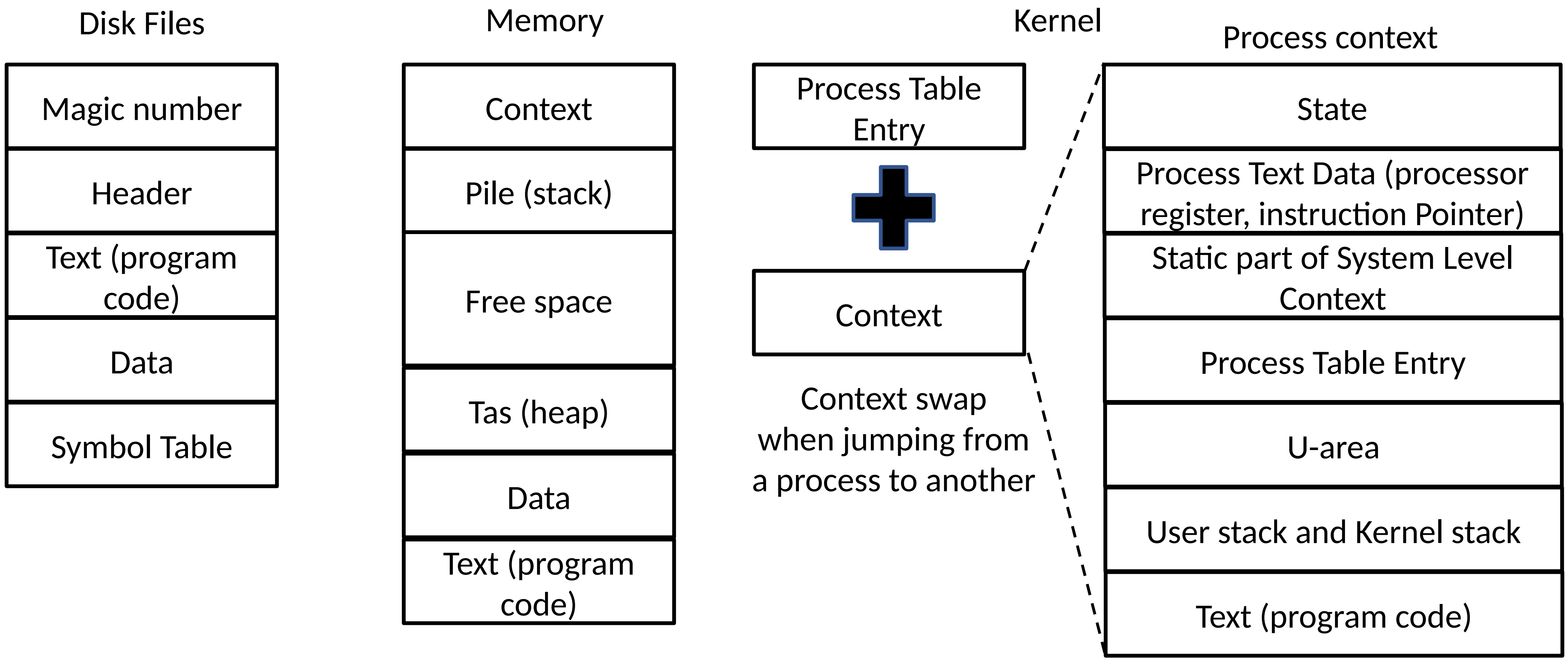
- A **process** is the running instance of a **program** (also know as **task**)

- A **program** is a **file** containing either programming code or data

- There is way more data needed to describe a process running into the system

- All this data allow the system to run multiple process at the same time and to jump from one to another

Process : definition (2/2)



Some of those data (Process Table, U-area) will be explained later

U-Area (User Area)

- Specific data of a process stored in a stack segment
    - Real and effective **UID** (**User ID**)
    - Timer field (user and system)
    - Reactions to signals (future classes)
    - Error field recording errors encountered during a system call
    - Return value field containing the result of system calls
    - Current directory (**pwd**) and current root (**chroot**)
    - User file descriptor table
    - Limits (man *ulimit*)
    - *umask* : masks mode settings on files the process creates

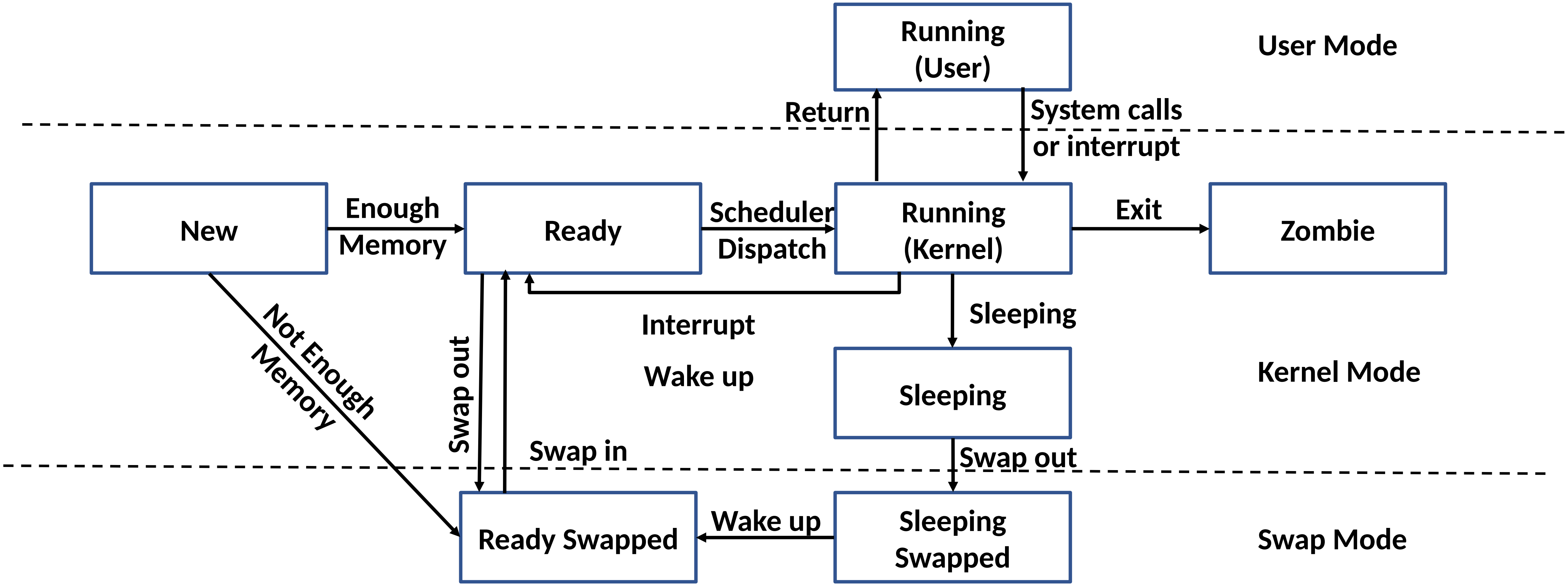
Process execution (1/2)

- The processor can only run one process at any time (expect for a multiple cores processor)

- The System switch between processes, giving the illusion of multitasking

- Processes might be in different states depending on their current activity : new, ready, running, sleeping, terminated, etc.

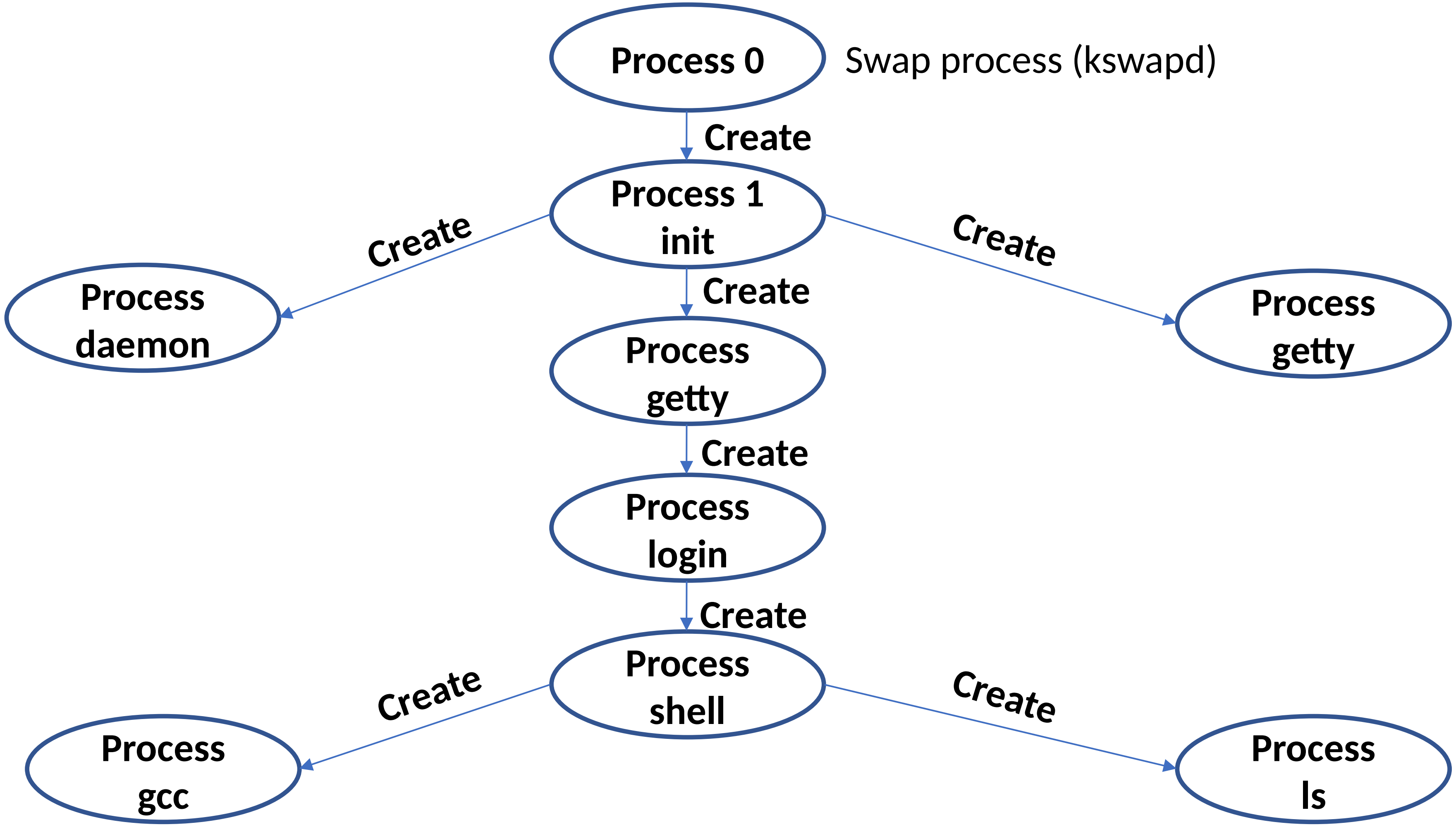
Process execution (2/2)



Process creation (1/4)

- All the processes share a **father/child** relationship : **process tree**
- Initial process is **0** (<*swapper*>). He gives birth to **1** (<*init*>)
- Each new process is the child of another, and any process can gave birth to a child and become a father
- A father process **should** wait the death of all of his child before ending
- If a process die before his child, they will get adopted by <*init*> => no orphans

Process creation (2/4)



Process creation (3/4)

- Each process has his own PID (Process ID) and PPID (Parent Process ID)
- **PPID** is also a **PID**, both of **pid_t** type (**integer**)
- Using the "**ps**" command show the process tree

```
$> ps axo stat,ppid,pid,tty,user,comm
STAT PPID    PID     TT      USER        COMMAND
S    0       1       ?       root        init
S    1       2       ?       root        keventd
Ss   1       816     ?       root        inetd
Ss+  1       1119    tty1    root        getty
S    23094   6851    ?       www-data    apache2
S+   14970   14974   pts/2   toto        gnuplot
R+   14956   17569   pts/1   toto        ps
```

Process creation (4/4)

- To create a new process, the **father** will duplicate himself and the new instance will load a new running code

- System primitives
  - **fork() :** create a new process by duplicating the caller
  - **exec() :** replace the programming code (in memory) of the clone by the one of the process to execute (read on hard disk) => replacement primitive
  - **wait() :** notify the father of the death of one of his child and enable the recovering of termination data (blocking call)

- **SIGCHLD** signal can also by a child to inform his father

How to : fork() (1/3)

- **fork()** creates to duplicate of a process
- All the data of the process (U-area) are duplicated, with the exception of PID and PPID
- Opened file descriptors of the father are duplicated too => the child has the same open files
- Information about the child (running time, etc.) are reset

How to : fork() (2/3)

- Headers :

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

- **pid_t** is of integer type
- **pid_t fork(void);**
- Other useful primitives :
  - **pid_t getpid(void);** /* My PID */
  - **pid_t getppid(void);** /* PID of my father*/
  - **pid_t getcwd(char* buf, size_t size);** /* What's my ...
  - **pid_t getwd(char* buf);** ... working directory */

How to : fork() (3/3)

```
int main(int argc, char* argv[]) {
     pid_t child = fork();      /* From here, father and child run the same programming code */
    switch(child) {/* Differences starts here */
        case -1 :
            perror("fork"):
            exit(errno);
        case 0:    /* Child programming code */
            printf("Child : my PID is %d\n, getpid()"):
            printf("Child : my father's PID is %d\n", getppid()):
            break;
        default:   /* Father programming code */
            printf("Father : my child's PID is %d\", child):
    }
    printf("%d : This sentence will appear in both processes\n", getpid());
    return 0;
}
```

Handling system errors (1/2)

- Each system function returns a return code
  - **fork()** return -1 if an error occurs
  - **getenv()** return NULL if an error occurs
- Return code doesn’t tell the type of error encountered
- How to know the type of this error
  - Global variable **errno**
  - Defined in **<errno.h>**
  - **errno** value isn't significant if there's no error encountered
  - **errno** value is specific for each system call (mentioned in *man*)

Handling system errors (2/2)

- Display the error message
  - **void perror (const char *msg)**

- **perror()** display a message on the standard error output with the last know error encountered by a system call (see *man*)

- In System Programming (like every other type of programming), error handling is **MANDATORY!**

Using strace

- **Strace** shell command enable you to follow every system call (and signal)
  Your very best friend in SysProg ⏪

- **Usage exemple :**

```
$> strace cat /dev/null
[…]
open("/dev/null", O_RDONLY|O_LARGEFILE) = 3 […]
$> strace cat /dev/lapinblanc
[…] open("/dev/lapinblanc", O_RDONLY|O_LARGEFILE)=-1 ENOENT (No such file or directory)
[…]
```

How to : wait (1/3)

- After a **fork()**, the father can use the **wait()** function
- **wait()** is a blocking function, waiting for any child of the father
- **waitpid()** is a blocking function, waiting for any or a specified child

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

/* Waiting for a child death, informations recovered in status */
pid_t wait(int* status);

/* Waiting for a specified child death (or any) */
pid_t waitpid(pid_t wpid, int* status, int options);
```

How to : wait (2/3)

- Interpretartion of **status** is done using macros
  - **WEXITSTATUS, WCOREDUMP, etc. (man 2 wait)**

```
/* [...] in the father, after using a fork() */
int status = 0;
pid_t pid;

if ( ( pid = wait(&status) )  == -1 ) {
	perror("wait");
	exit(errno);
}

printf("My son %d ended with the exit code %d\n", pid, WEXITSTATUS(status));
/* [...]*/
```

How to : wait (3/3)

```
int main(int argc, char* argv[]) {
	int status = 0;
	pid_t returnCode;
	pid_t child = fork();
	switch(child) {
		case -1 :
			perror("fork"): exit(errno);
		case 0:	/* Child programming code */
			printf("Child : my PID is %d\n", getpid()):
			break;
		default:	/* Father programming code */
			printf("Father : my child's PID is %d\n", child):
			returnCode = wait(&status);
			if ( returnCode = -1 )
			printf("My child %d ended with the code %d\n", returnCode, WEXITSTATUS(status));
	}
	return 0;
}
```

How to : exec (1/2)

- **exec()** is replacement primitive, replacing the running code of a process by another one
- **exec()** launch a process with arguments, like while using the command line
- **exec()** family enables to specify the arguments of the child in different ways

```
#include <unistd.h>
/* exec() family*/
int execl(const char *path, const char *arg, ..., NULL);
int execlp(const char *file, const char *arg, ..., NULL);
int execle(const char *path, const char *arg, ..., NULL, char *const *envp[]);
int execv(const char *path, const char *argv[]);
int execvp(const char *file, const char *argv[]);
int execvP(const char *file, const char *search_path , const char *argv[]);
```

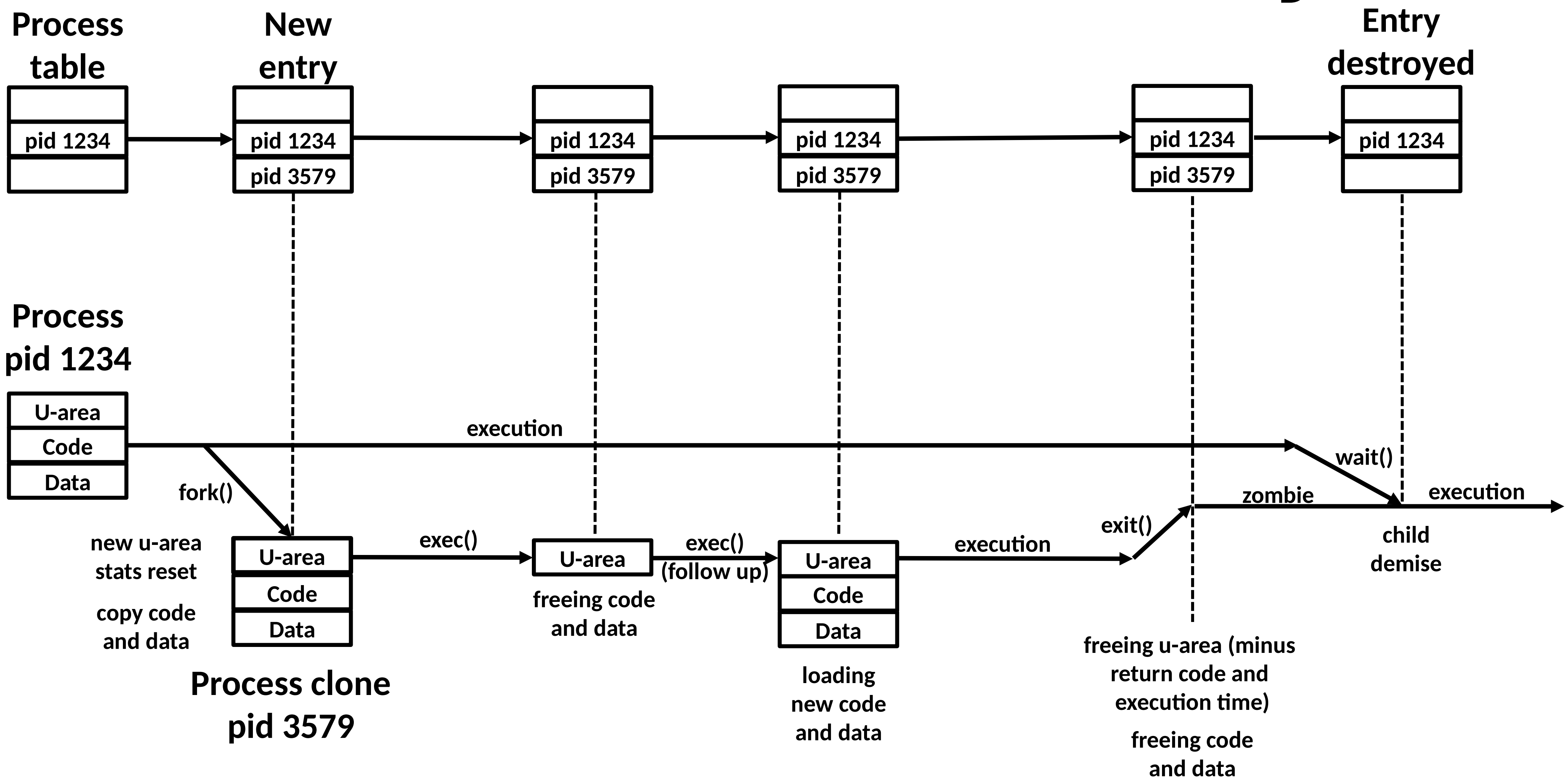
How to : exec (2/2)

```
int main(int argc, char* argv[]) {
        int status = 0, error = 0;
        pid_t returnCode;
        pid_t child = fork();
        switch(child) {
                case -1 :
                        perror("fork"): exit(errno);
                case 0:         /* Child programming code */
                        printf("Replacement of the code, here we will do a <ls –l>\n");
                        error = execl("/bin/ls", "ls", "-l", NULL);
                        printf("This will never appear, except if the execlp call has failedn");
                        fprintf(stderr, "execlp() failed with the error code %d\n", error);
                        exit(errno); break;
                default:        /* Father programming code */
                        printf("Father : my child's PID is %d\n", child):
                        if ( wait(&status) = -1 ) { perror("wait"); exit(errno); }
        }
        return 0;
}
```

Processes Death

- A process doesn't immediately disappear at the end of his running time
- In fact, when a process is terminated (call to **exit()**), he becomes a zombie
- He will disappear only after the father hears about his termination (using **wait()** or a signal)
- *<init>* is always in **wait()**, which enables him to adopt orphans processes so they can properly disappear

Process life: Summary



Sharing data between father and child

- **Data** (variables) are duplicated (exception : open file descriptors)

- **Memory** isn't shared between father and child => modifications in a process aren't visible or reverberated in the other one

- Duplication is done using *copy-on-write* (optimisation non négligeable)

- To use shared memory, see the system API (**shm_open(), shm_unlink(), etc.)**

Exec() arguments and line command : argc and argv (1/2)

- **int main(int argc, char* argv[])** : why?
- **argc** : number of line command arguments
- **argv** : the arguments, in an array of char
- **argv[0]** : name of the program (**argc** is always >= 1)

```c
int main(int argc, char* argv[])
{
  int i = 0;
  for (i = 0; i < argc; ++i)
    printf("argument %d : %s\n", i, argv[i]);
  return 0;
}
```

```
/home/prof>./toto -alF ab 12
argument 0 : ./toto
argument 1 : -alF
argument 2 : ab
argument 3 : 12
/home/prof>
```

Exec() arguments and line command : argc and argv (2/2)

- **exec()** family uses arguments : number of line command arguments

```
pid_t child = fork();
switch(child) {
	case -1 :
		perror("fork"): exit(errno);
	case 0:		/* Child programming code */
		printf("Replacement of the code, here we will do a <ls –l>\n");
		error = execl("/bin/ls", "ls", "-l", NULL);
		printf("This will never appear, except if the execlp call has failedn");
		fprintf(stderr, "execlp() failed with the error code %d\n", error);
		exit(errno);	break;
	default:	/* Father programming code */
		printf("Father : my child's PID is %d\n", child):
		if ( wait(&status) = -1 ) { perror("wait"); exit(errno); }
}
```

Using the environment (1/2)

- The environment is built on key/values associations
  - *Environment variable* name, **char** value
- Examples : **PS1, USER, SHELL, etc.**
- Father's environment is inherited by any child

```
#include <stdlib.h>
char* getenv(const char *name);                    /* return the value of a key */

char* setenv(const char *name, const char *value, int overwrite);  /* define a key/value pair */

/* take a string "key=value" and execute setenv("key", "value", "1") */
int putenv(const char *string);

void unsetenv(const char *name);                   /* delete a key (and also the value) */
```

Using the environment (2/2)

```
int main(int argc, char* argv[]) {
	int status = 0;
	pid_t returnCode;
	setenv("toto", "titi", 1);				/* changing the environment before the fork */
	pid_t child = fork();
	switch(child) {
		case -1 :
			perror("fork"): exit(errno);
		case 0:	/* Child programming code */
			printf("toto = %s\n", getenv("toto"));	/* display "titi" */
			break;
		default:	/* Father programming code */
			if ( wait(&status) = -1 ) { perror("wait"); exit(errno); }
	}
	return 0;
}
```

Sharing data between father and child

- **fork()** creates a new concurrent process
- A process can also split himself in several **threads**, contained inside the same process
- **Threads** runs in **concurrent computing** but share their resources => allow the developer to make a implement a bit of parallelism **(see you in 3A)**
- **Threads** use API like **POSIX Threads**

POSIX Threads

- **<pthread.h>**

- **pthread_create** create a new thread to run a specific function

- A thread ends with the execution of **pthread_exit()** OR when it reaches the end of the running function

- Synchronisation (waiting) : **pthread_join()**

```
void* thread_function(void* data) {
        int* x = (int* ) data;
        printf("Value %d\n", *x);
        *x = 4;     /* changing value (no possible conflict here). thread stops here, same as pthread_exit() */
        return 0;
}
int main(int argc, char* argv []) {
        int value = 0, error = pthread_create(&t, NULL, &fonction_du_thread, &valeur);
        pthread_t t;
        if (!error) {
                printf("thread creates, waiting for the end\n"); error = pthread_join(t, NULL);
                if (error) {
                fprintf(stderr, "pthread_join : %s\n", strerror(error)); exit(error);
        }
        printf("value should be equal to 4 : %d\n", valeur); return 0;
}
```