



**Universidad
de Valparaíso
CHILE**

Teoría de Sistemas Operativos

Escuela de Ingeniería Civil en Informática
Universidad de Valparaíso, Chile

<http://informatica.uv.cl>

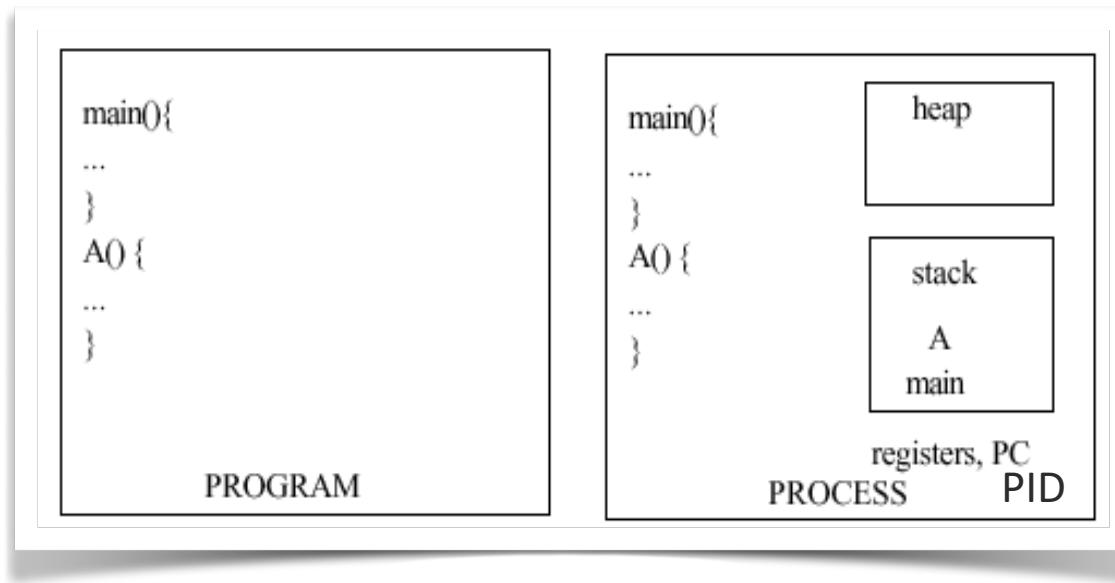
Procesos

Programas, procesos
Modelo de procesos
Cambios de contexto
Creación de procesos
Planificación de procesos

Definición

Programa en ejecución

En forma secuencial dentro el procesador asignado



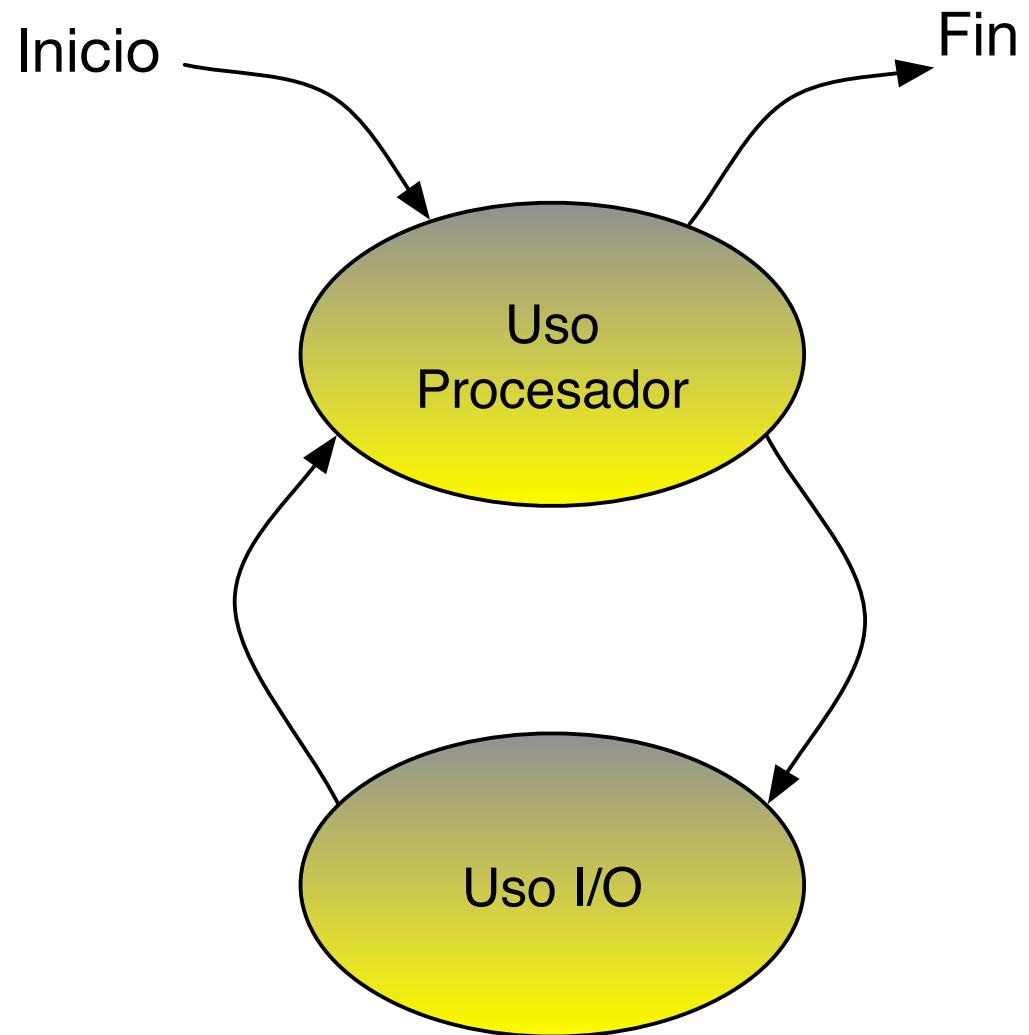
PID:
Process IDentifier

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

pid_t getpid(void);
```

El sistema operativo tiene una lista de procesos

Modelo de funcionamiento de un proceso



Bajo este esquema,
¿es posible tener
varios procesos en
funcionamiento?

¿Para qué?

Ejecución de procesos simultáneos

Concurrencia

El SO genera un paralelismo Virtual, mediante compartición de recursos.



Paralelismo

Los procesos se ejecutan en procesadores distintos (o cores)

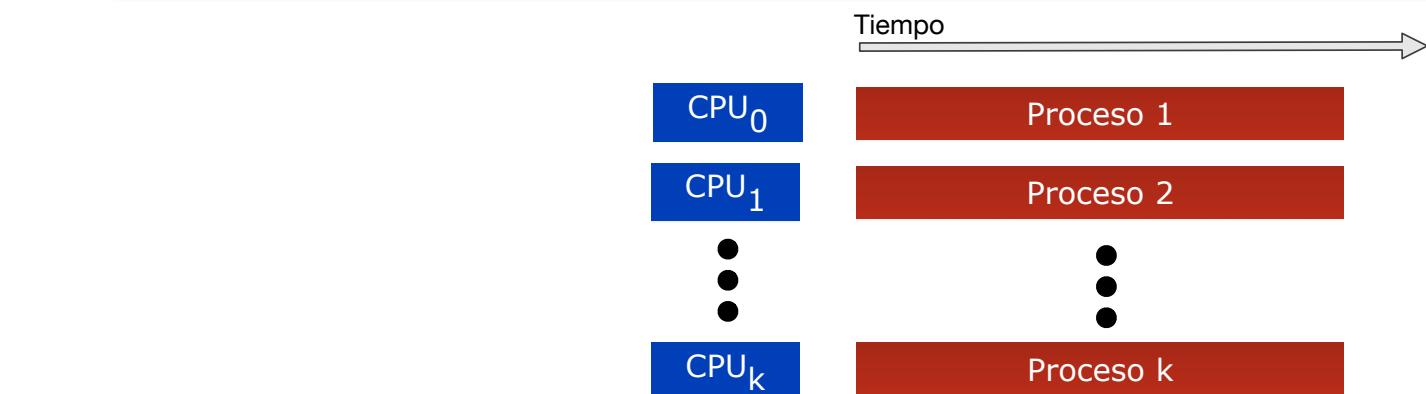


Tabla de procesos

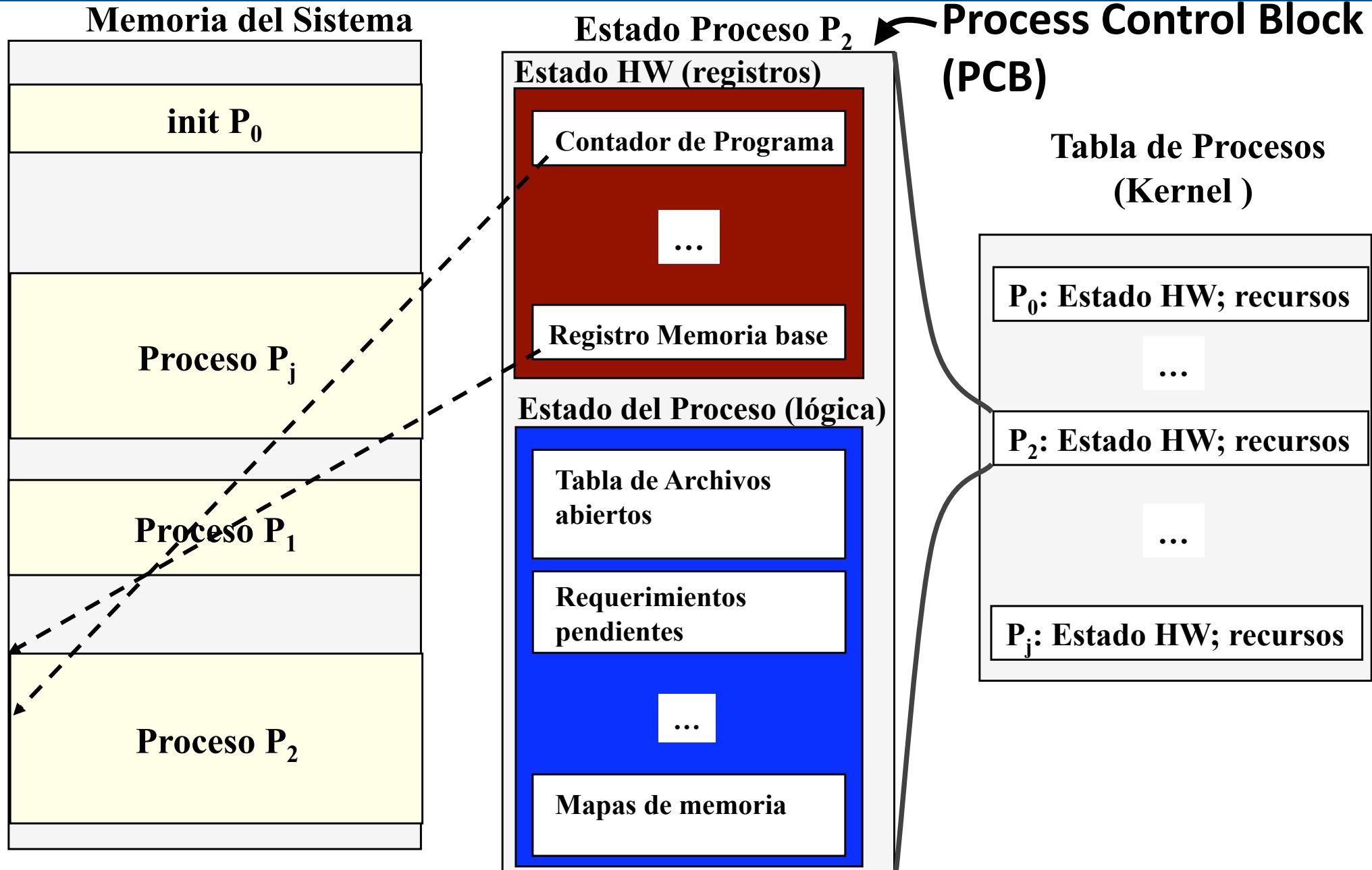
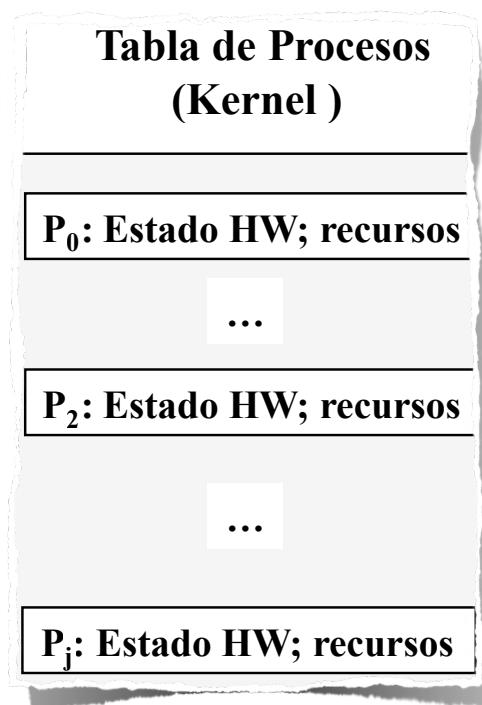


Tabla de procesos

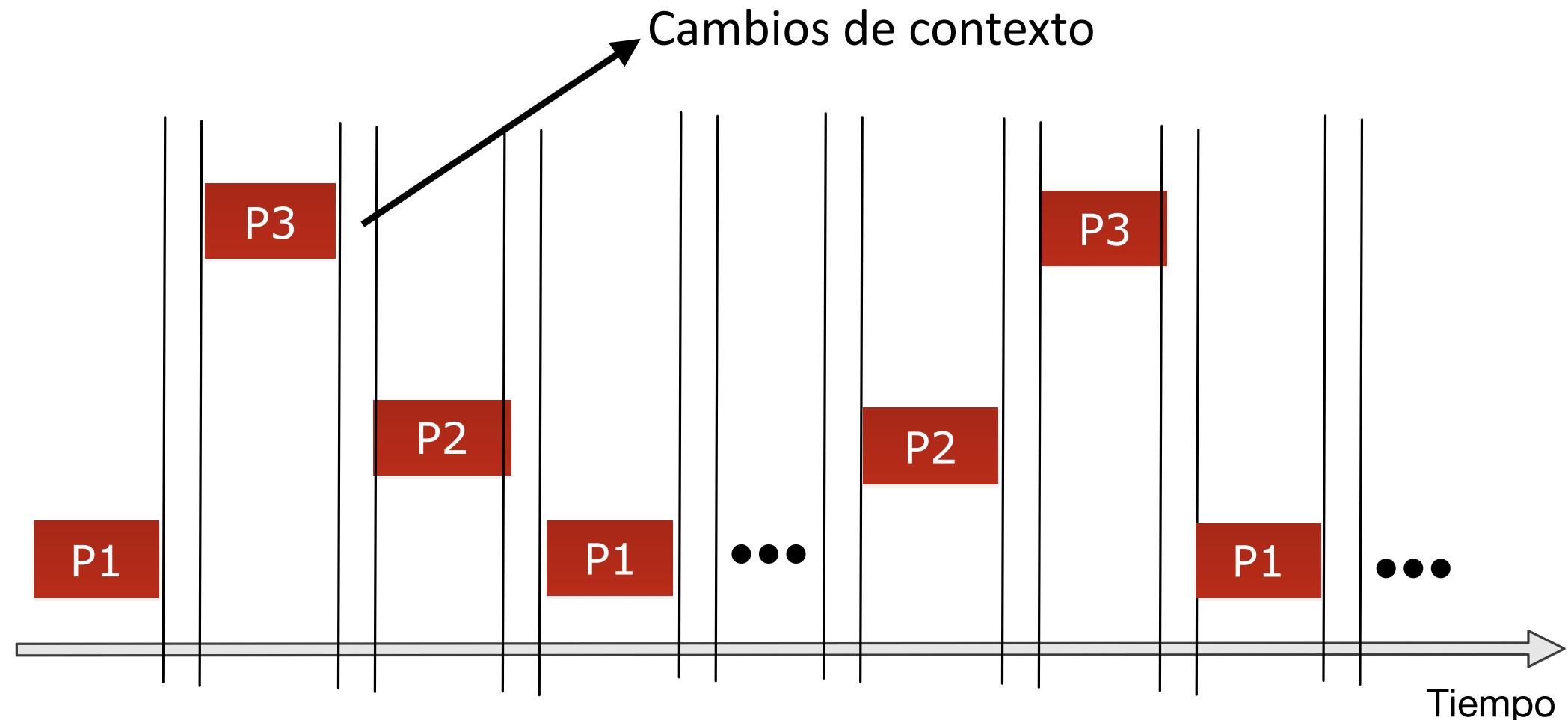
Es una estructura abstracta, que generalmente es accesible a través de comandos o funciones específicos.



PID	Process Name	User	CPU	# Threads	Real Memory	Virtual Memory
383	iTunes	gabriel	8.6	17	44.46 MB	1,017.90 MB
380	Firefox	gabriel	5.5	22	151.60 MB	1.12 GB
1359	Activity Monitor	gabriel	1.9	5	18.30 MB	1,003.46 MB
179	SystemUIServer	gabriel	0.6	7	25.78 MB	977.36 MB
1334	Microsoft PowerPoint	gabriel	0.3	3	98.57 MB	1.05 GB
191	Mac_SwapperDemon	gabriel	0.2	1	1.90 MB	596.42 MB
204	Adobe Reader	gabriel	0.2	5	29.53 MB	997.98 MB
192	Desktop Manager	gabriel	0.1	1	6.27 MB	917.55 MB
1335	Microsoft Database Daemon	gabriel	0.0	2	8.68 MB	918.29 MB
1324	Keynote	gabriel	0.0	11	133.16 MB	1.11 GB
415	Numbers	gabriel	0.0	11	82.68 MB	1.06 GB
189	HimmelBar	gabriel	0.0	3	21.94 MB	998.40 MB
181	Finder	gabriel	0.0	7	27.86 MB	954.77 MB
190	Google Notifier	gabriel	0.0	6	12.40 MB	926.56 MB
1336	Microsoft AU Daemon	gabriel	0.0	1	2.31 MB	864.66 MB
188	iTunes Helper	gabriel	0.0	2	2.49 MB	858.71 MB
172	AirPort Base Station Agent	gabriel	0.0	3	3.21 MB	908.83 MB
193	iCal	gabriel	0.0	3	14.67 MB	935.28 MB
176	UserEventAgent	gabriel	0.0	3	2.89 MB	856.39 MB
1316	mdworker	gabriel	0.0	4	7.44 MB	653.91 MB
1320	Quick Look Server	gabriel	0.0	8	34.71 MB	951.25 MB
389	Mail	gabriel	0.0	14	38.39 MB	993.31 MB

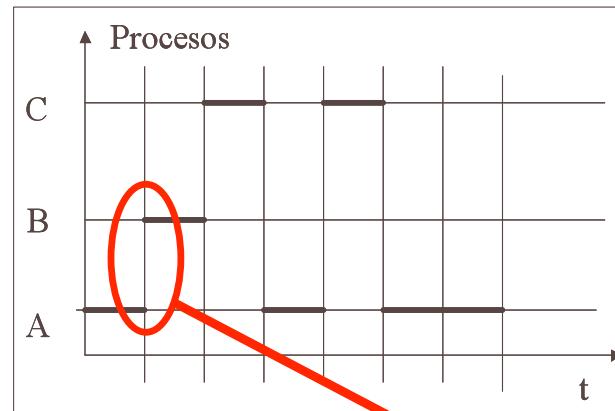
Muchos procesos concurrentes

Resultado

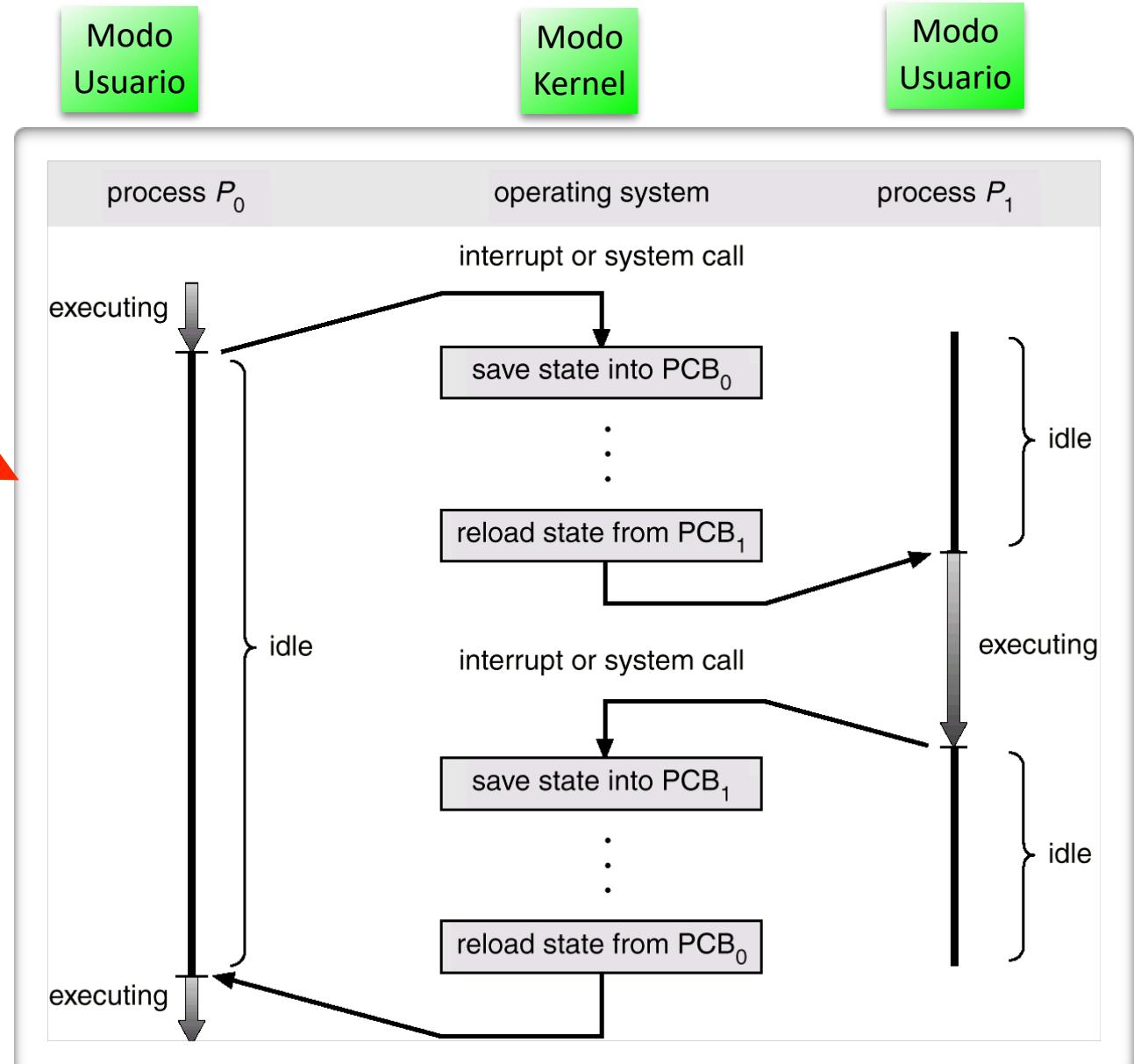


http://www.linfo.org/context_switch.html

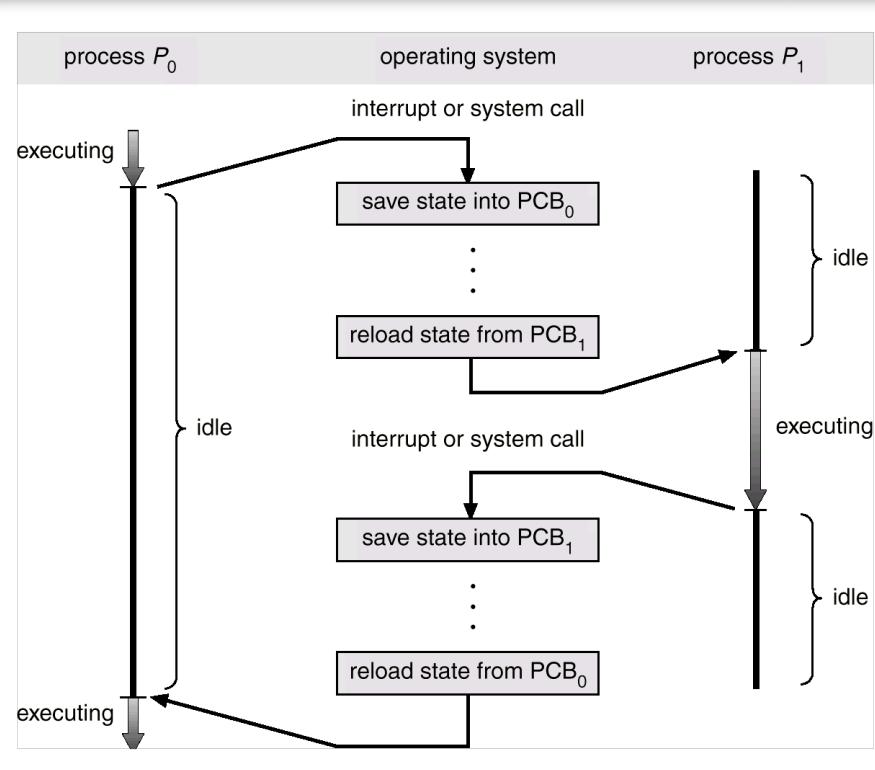
Cambios de contexto



PCB _j	Pointer	Proc State
Process number		
Process counter		
Registers		
Memory Limits		
List of open Files		
...		



Cambios de contexto



API: `getrusage(2)`

```
#include <sys/time.h>
#include <sys/resource.h>

int getrusage( int who,
               struct rusage *u )
```

Información acerca de los cambios de contexto

`/proc/$PID/status`

```
Name: bash
State: S (sleeping)
Tgid: 3564
Pid: 3564
PPid: 3563
...
voluntary_ctxt_switches: 43
nonvoluntary_ctxt_switches: 26
```

Cambios de contexto

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/resource.h>
#include <sys/time.h>

int main(void) {
    u_int i = 0;
    struct rusage r_usage;

    for(i=0; i<2000000000;i++)
    ;

    getrusage(RUSAGE_SELF, &r_usage);

    printf("Cambios de contexto\n");
    printf("Ctxt Voluntarios      = %ld\n", r_usage.ru_nvcsw);
    printf("Ctxt No Voluntarios   = %ld\n", r_usage.ru_nivcsw);

    return(EXIT_SUCCESS);
}
```

Cambios de contexto

```
struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    long ru_maxrss; /* maximum resident set size */
long ru_ixrss; /* integral shared memory size */
long ru_idrss; /* integral unshared data size */
long ru_isrss; /* integral unshared stack size */
    long ru_minflt; /* page reclaims (soft page faults) */
    long ru_majflt; /* page faults (hard page faults) */
long ru_nswap; /* swaps */
    long ru_inblock; /* block input operations */
    long ru_oublock; /* block output operations */
long ru_msgsnd; /* IPC messages sent */
long ru_msgrcv; /* IPC messages received */
long ru_nssignals; /* signals received */
    long ru_nvcsw; /* voluntary context switches */
    long ru_nivcsw; /* involuntary context switches */
};
```

Cambios voluntarios: un proceso cede de forma explícita el procesos a otro

Multitarea no-apropiativa(*)

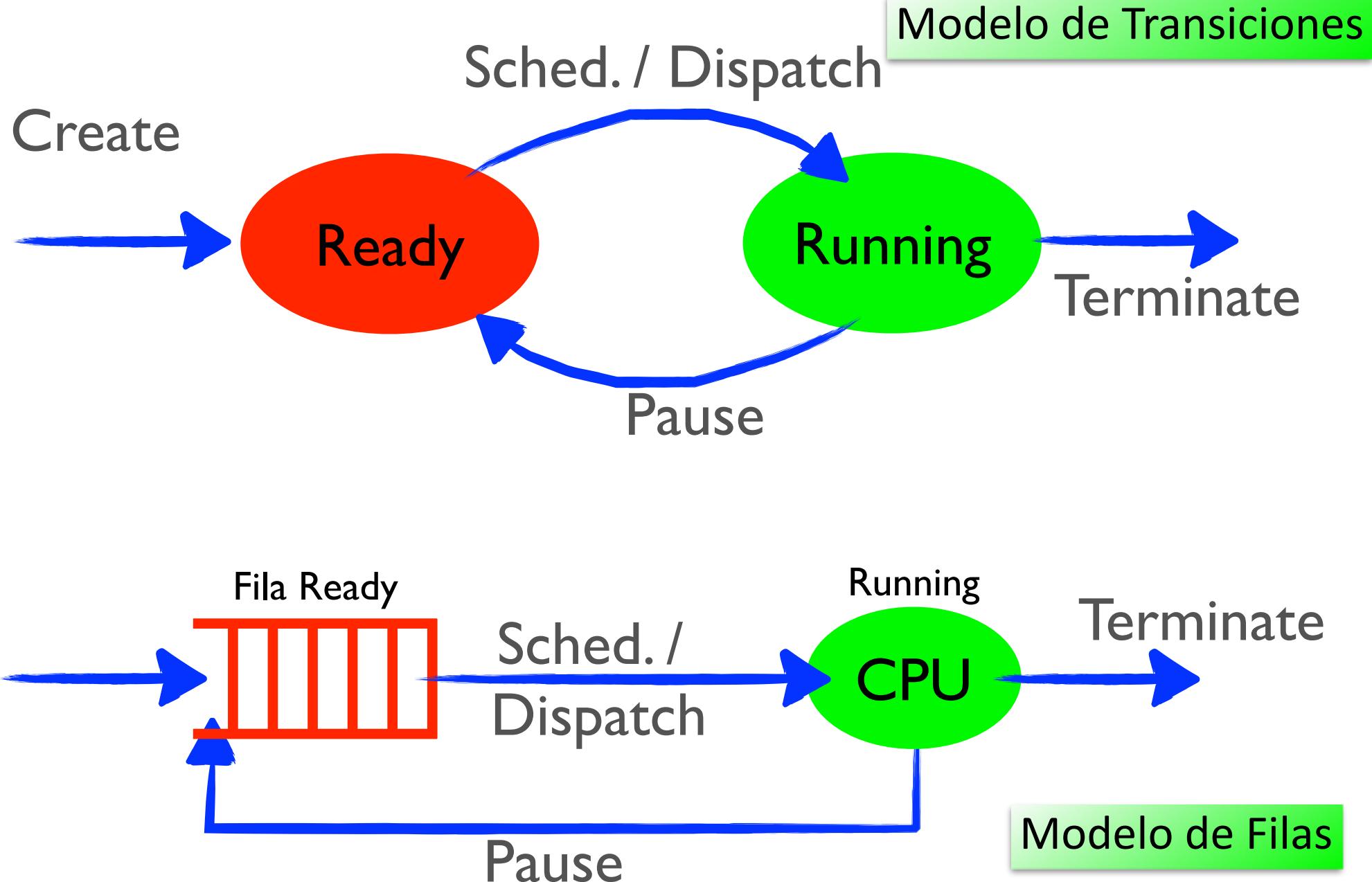
Cambios no-voluntarios: el planificador del sistema saca al proceso del procesador y transfiere su uso a otro proceso

Multitarea apropiativa (*)

(*) Conceptos revisados en la parte introductoria

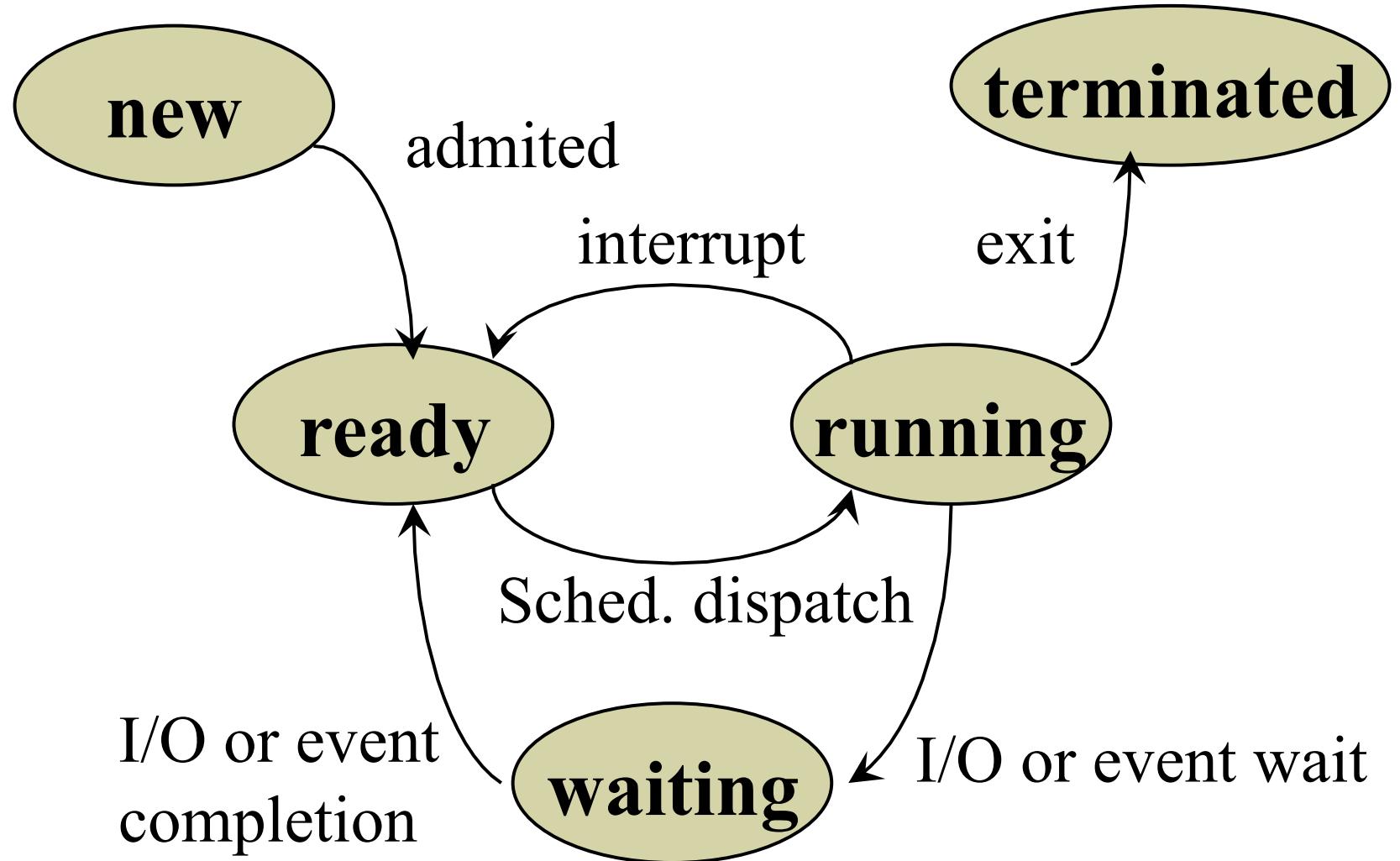
Modelos de procesos

Modelos



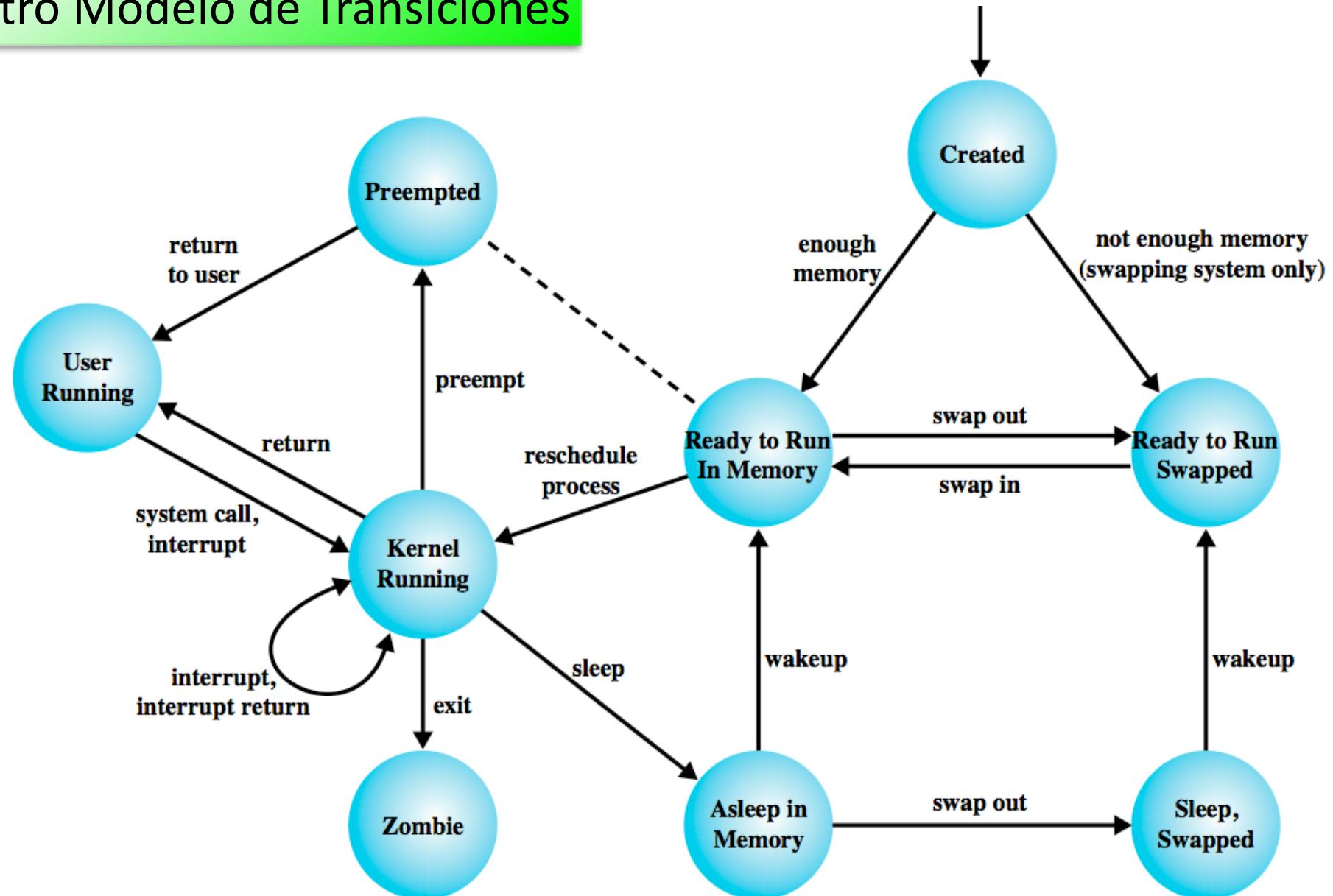
Modelos

Otro Modelo de Transiciones



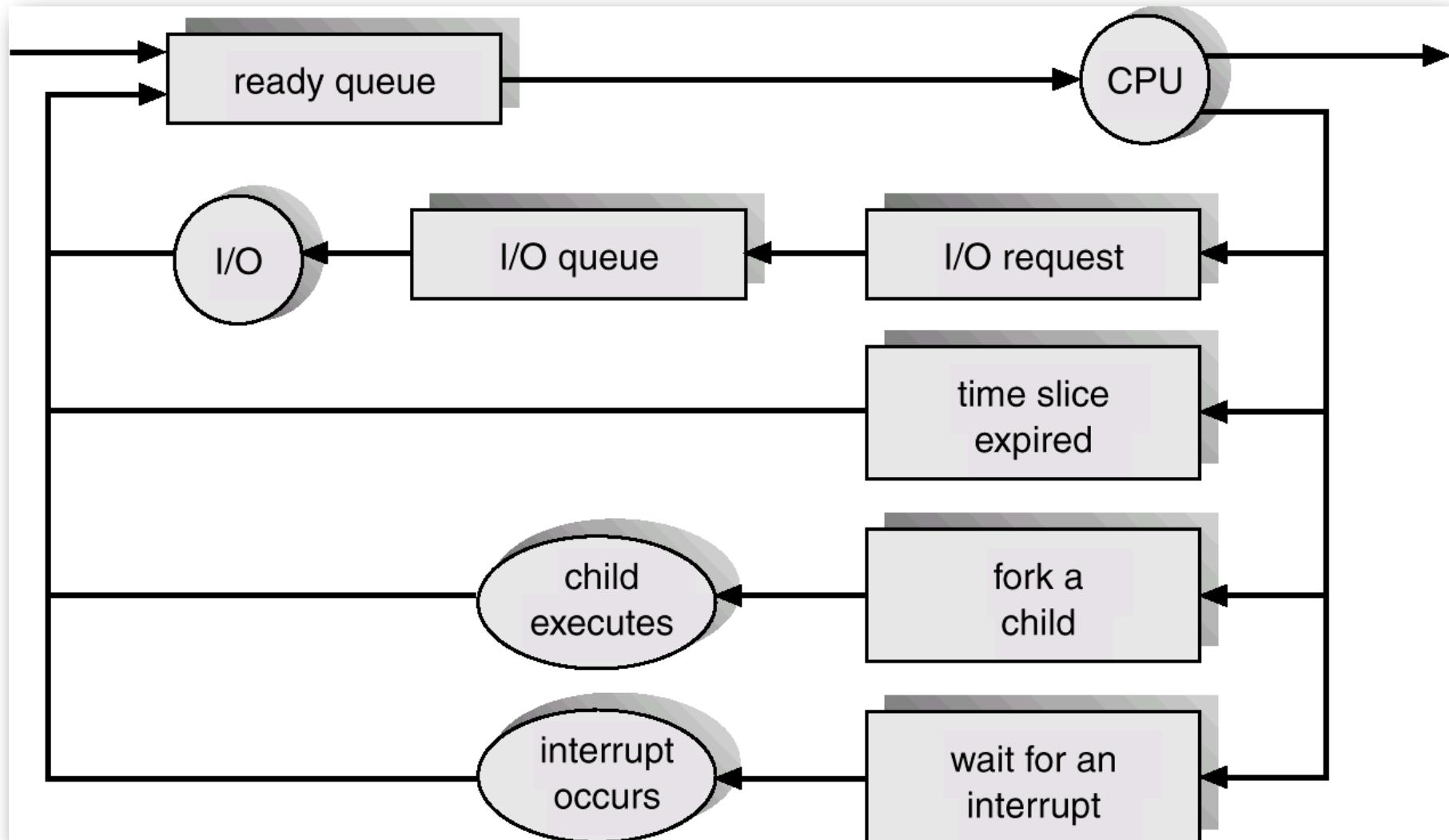
Modelos

Otro Modelo de Transiciones

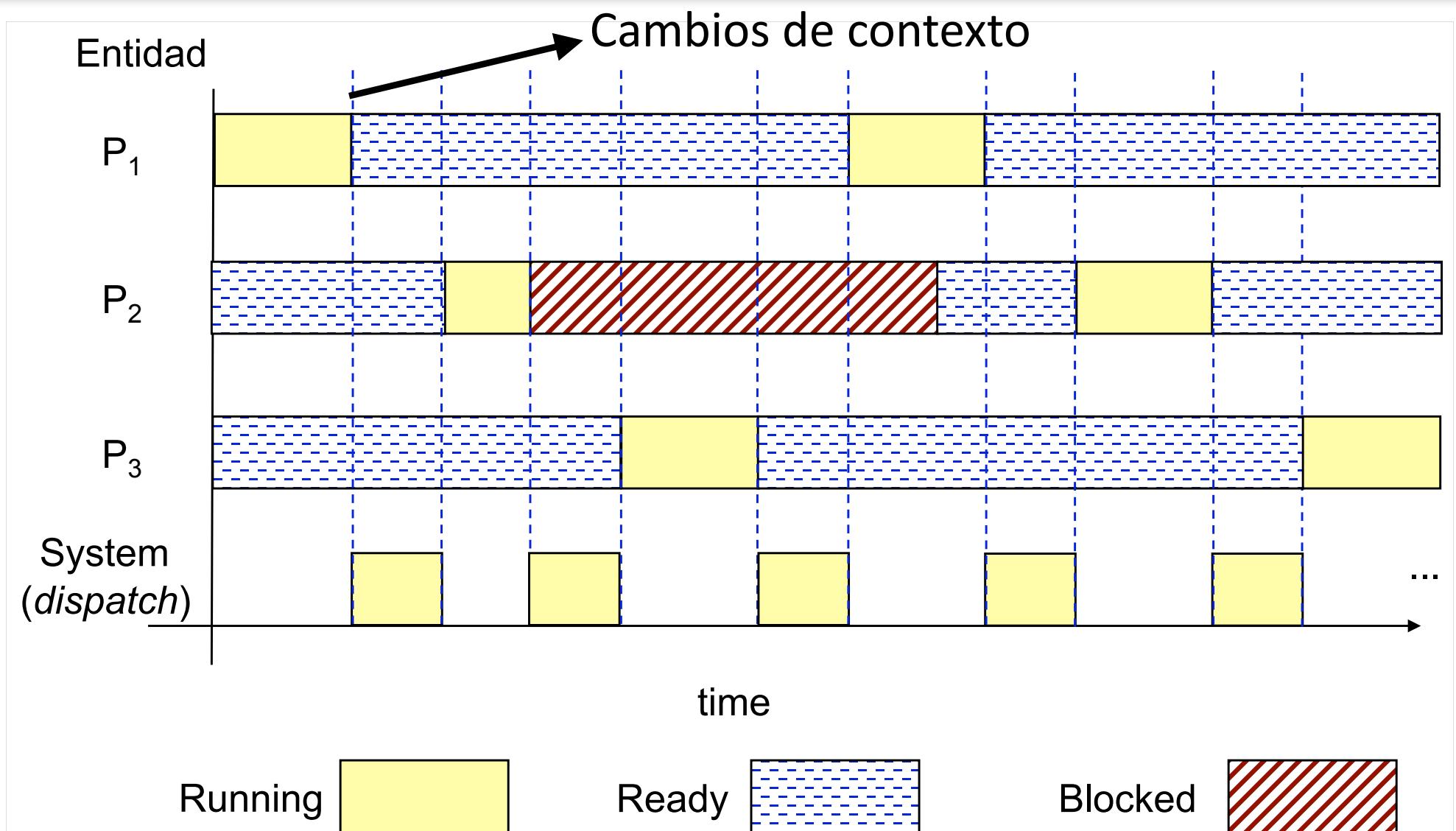


Modelos

Otro Modelo de Filas

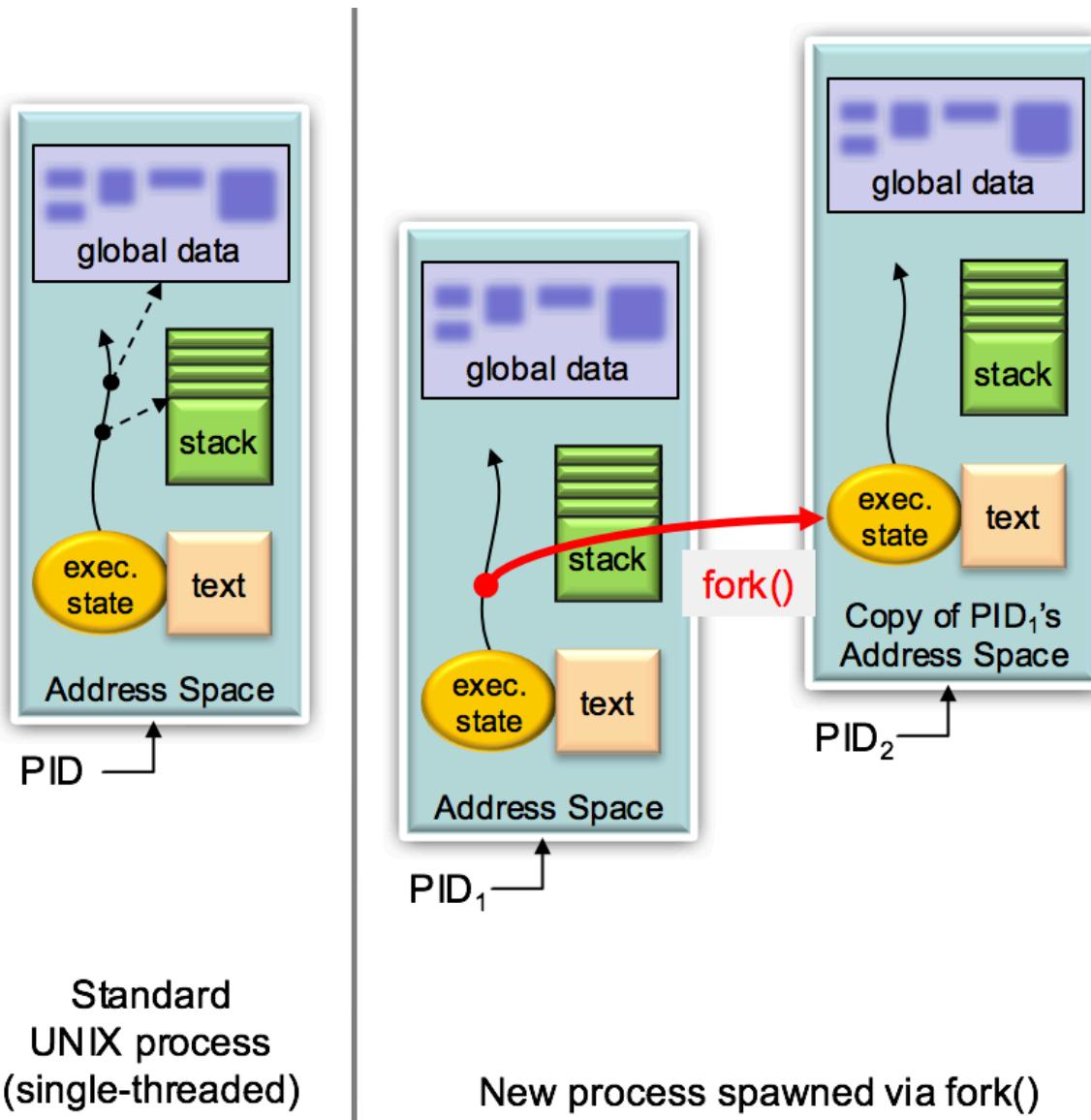


Resultado



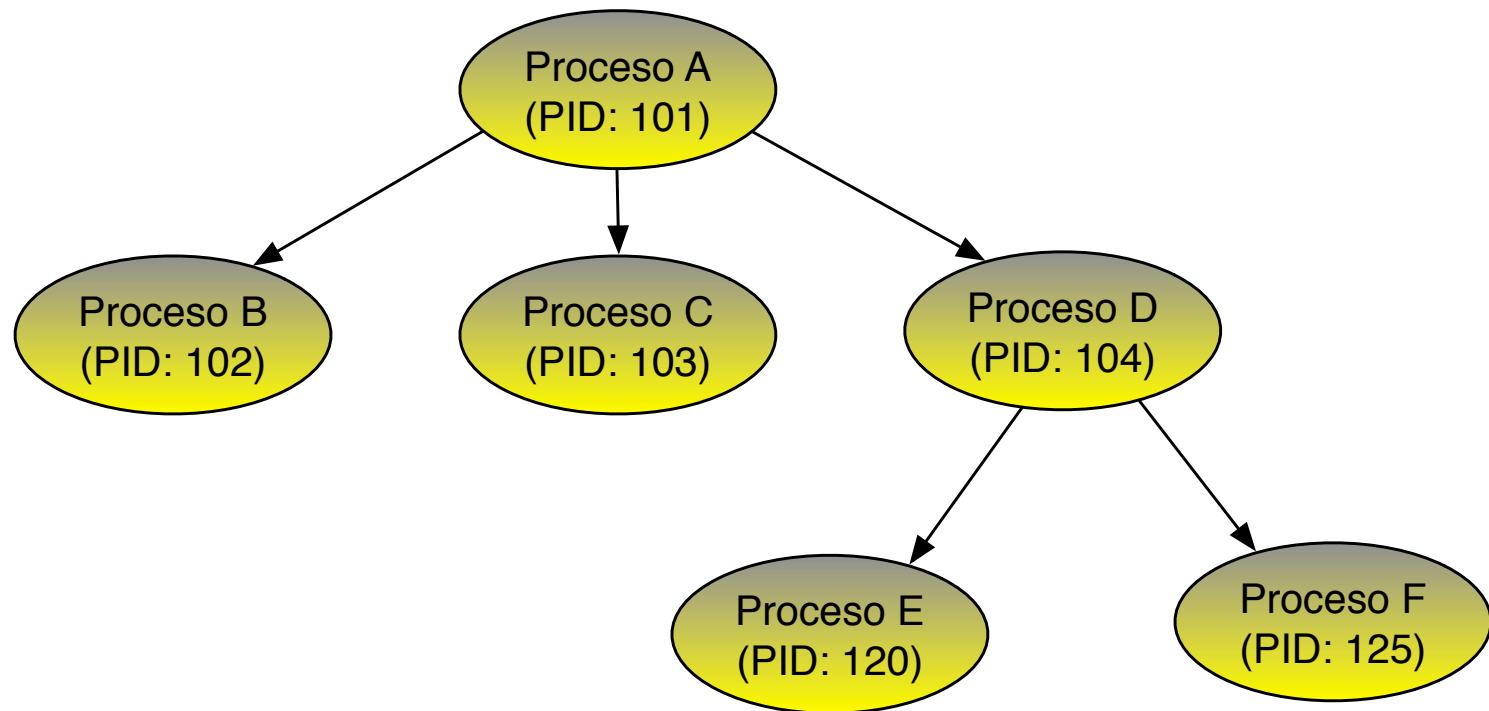
Creación de Procesos

Procesos que crean procesos



Procesos que crean procesos

Se establece una relación jerárquica (**PADRE-HIJO**)

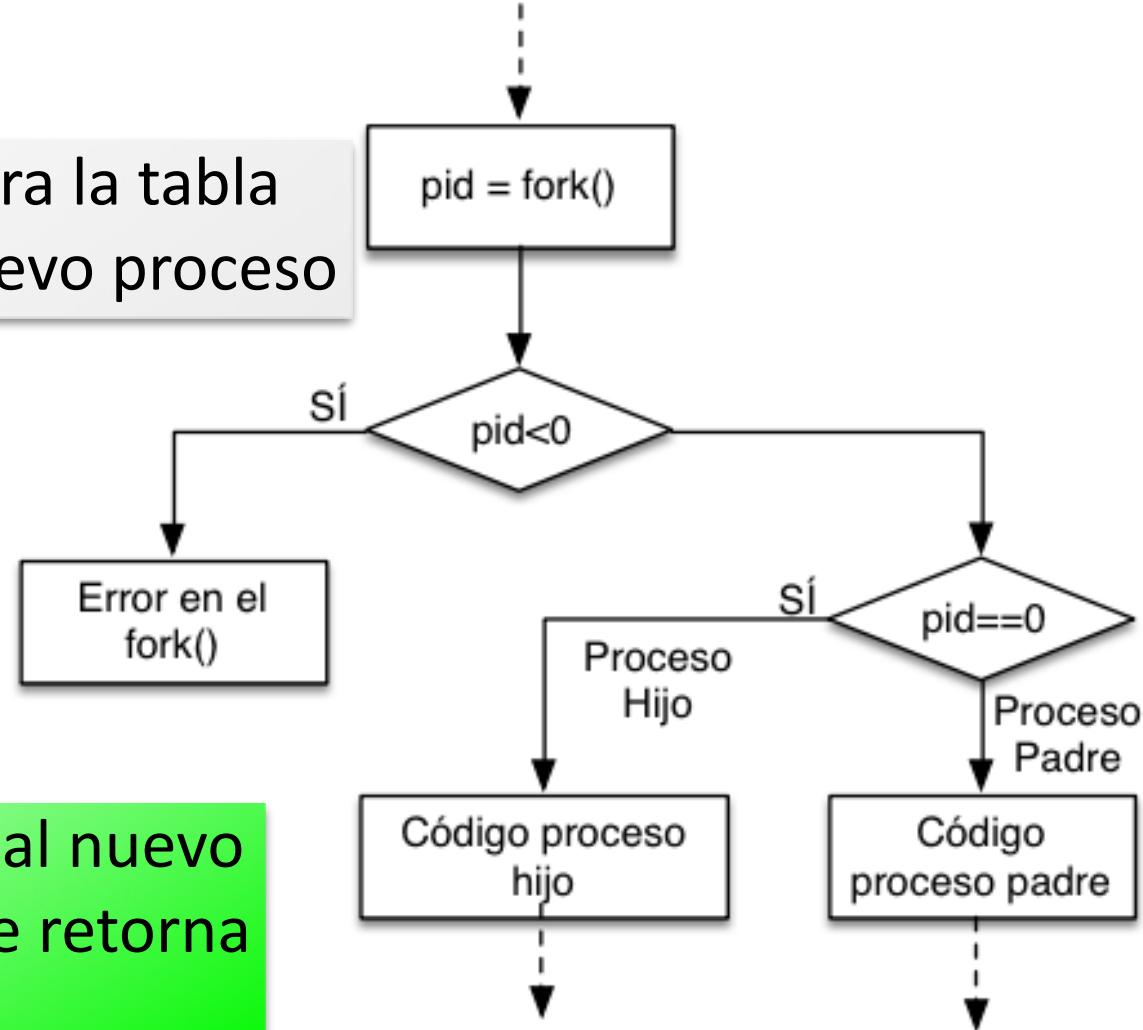


La programación con procesos Padre-Hijo normalmente se conoce como “programación concurrente”

Creación

A través del syscall fork()

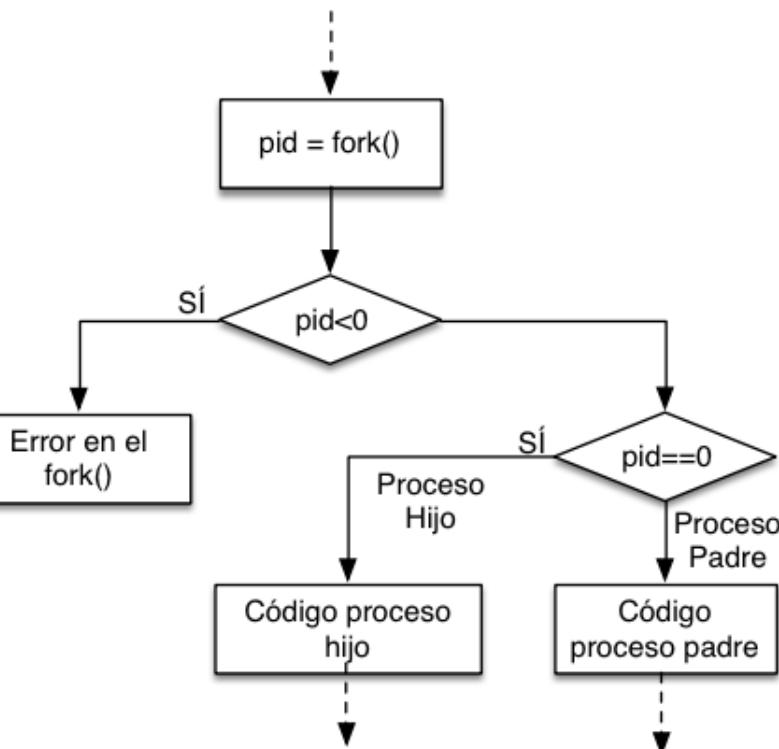
Reserva espacio para la tabla de procesos del nuevo proceso



El SSOO asigna un PID al nuevo proceso. Dicho valor se retorna al proceso padre.

Creación

A través del syscall fork()

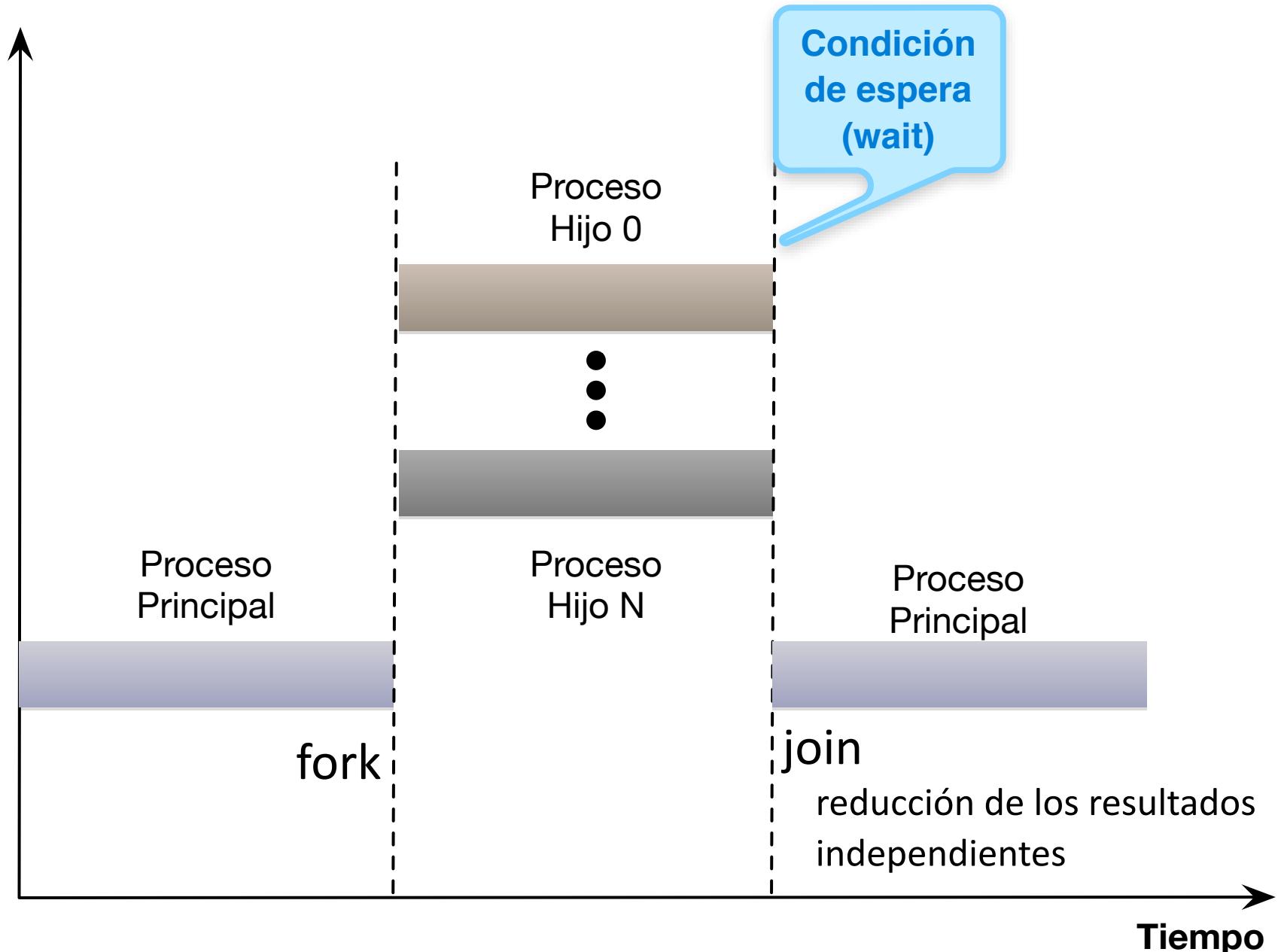


```
#include <unistd.h>
pid_t fork();
```

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    pid_t pid;
    pid = fork();

    if( pid < 0) {
        perror("Error en fork()");
        exit(EXIT_FAILURE);
    }
    else if( pid == 0 ) {
        //Código del hijo
    }
    else if(pid>0){ //Padre
        //Código del parent
    }
}
```

Y la idea es ...



Sincronización básica

Funciones de sincronización

`wait()`

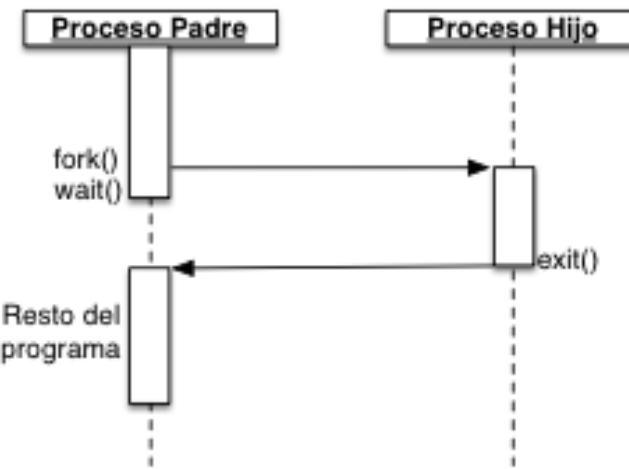
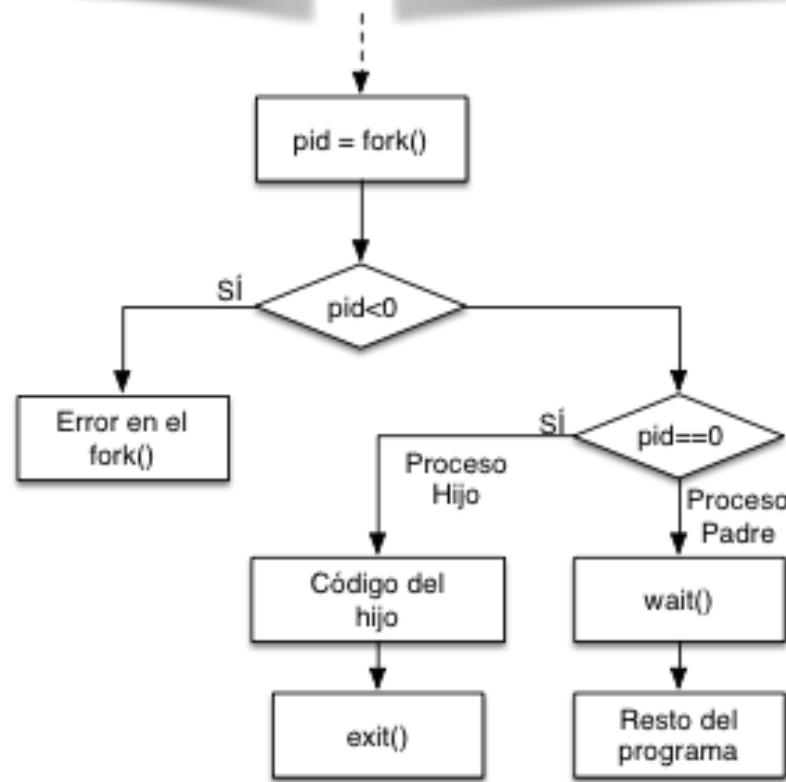
La función `wait()` forzará a un proceso padre para que espere a un proceso hijo que se detenga o termine. La función regresa el PID del hijo o -1 en caso de error. El estado de la salida hijo es regresado en la variable `&statloc`.

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

pid_t wait (int *statloc);
```

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

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



Funciones de sincronización

```

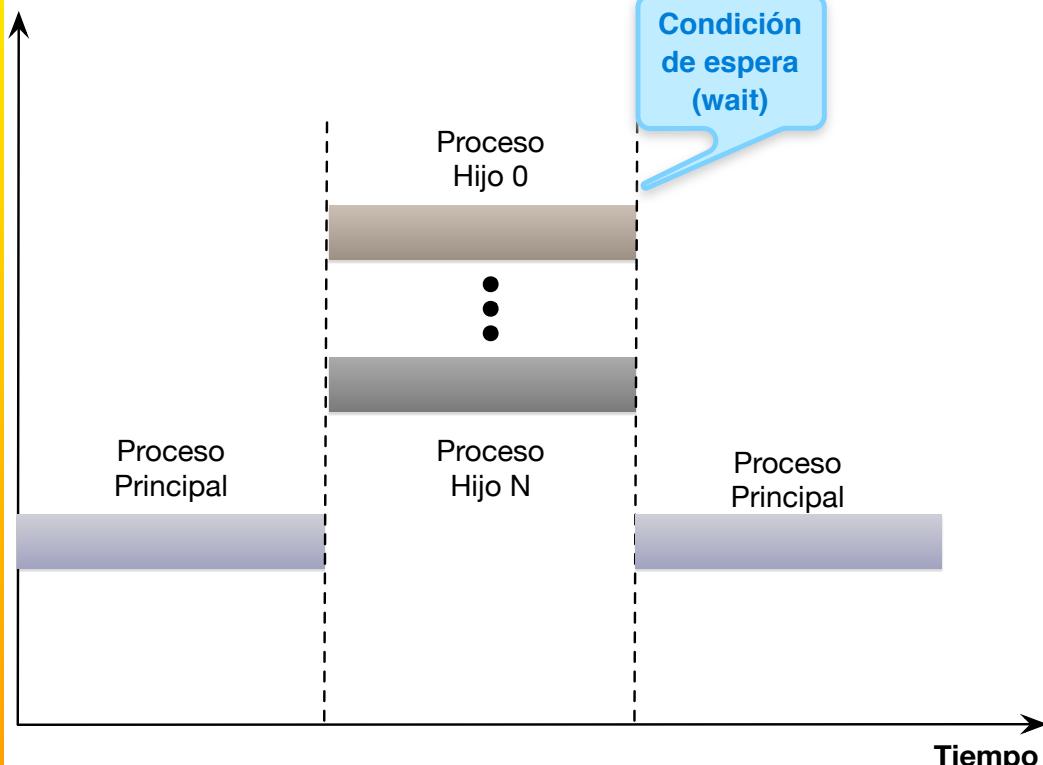
int main(int argc, char* argv[]){
    pid_t ret;
    int i;

    for (i=0; i<NUM_HIJOS; i++) {
        if(( ret = fork()) == 0) {
            switch(i){
                case 0: //Hijo #0
                    codigo_hijo_0();
                    break;
                case N: //Hijo #N
                    codigo_hijo_N();
                    break;
            }
            exit(EXIT_SUCCESS);
        }
        else if (ret == -1) {
            perror("falló en fork");
            exit(EXIT_FAILURE);
        }
    }

    for(i=0; i<NUM_HIJOS; i++){
        wait(NULL);
    }

    exit(EXIT_SUCCESS);
}

```



Funciones de sincronización

```

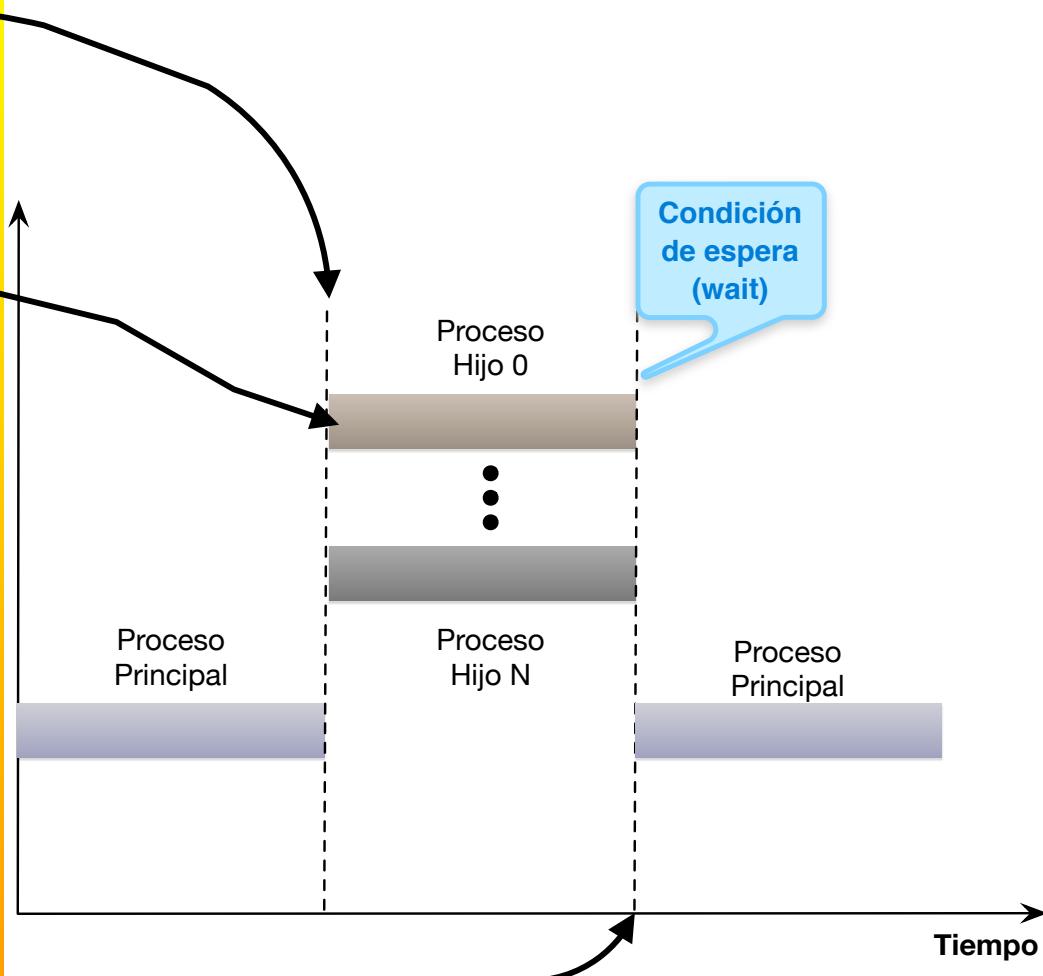
int main(int argc, char* argv[]){
    pid_t ret;
    int i;

    for (i=0; i<NUM_HIJOS; i++) {
        if(( ret = fork()) == 0) {
            switch(i){
                case 0: //Hijo #0
                    codigo_hijo_0();
                    break;
                case N: //Hijo #N
                    codigo_hijo_N();
                    break;
            }
            exit(EXIT_SUCCESS);
        }
        else if (ret == -1) {
            perror("falló en fork");
            exit(EXIT_FAILURE);
        }
    }

    for(i=0; i<NUM_HIJOS; i++){
        wait(NULL);
    }

    exit(EXIT_SUCCESS);
}

```



Ejemplo de ejecución no determinista

```
void hola(void) {
    char *msg = "Este ejercicio";
    int i;
    for (i = 0; i < strlen(msg); i++) {
        printf("%c", msg[i]);
        fflush(stdout);
        usleep(SLEEP_TIME);
    }
}

void mundo(void) {
    char *msg = "es un rotundo fracaso... ";
    int i;
    for (i = 0; i < strlen(msg); i++) {
        printf("%c", msg[i]);
        fflush(stdout);
        usleep(SLEEP_TIME);
    }
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>

#define SLEEP_TIME 1000000 //useg
```

```
int main() {
    pid_t pid;

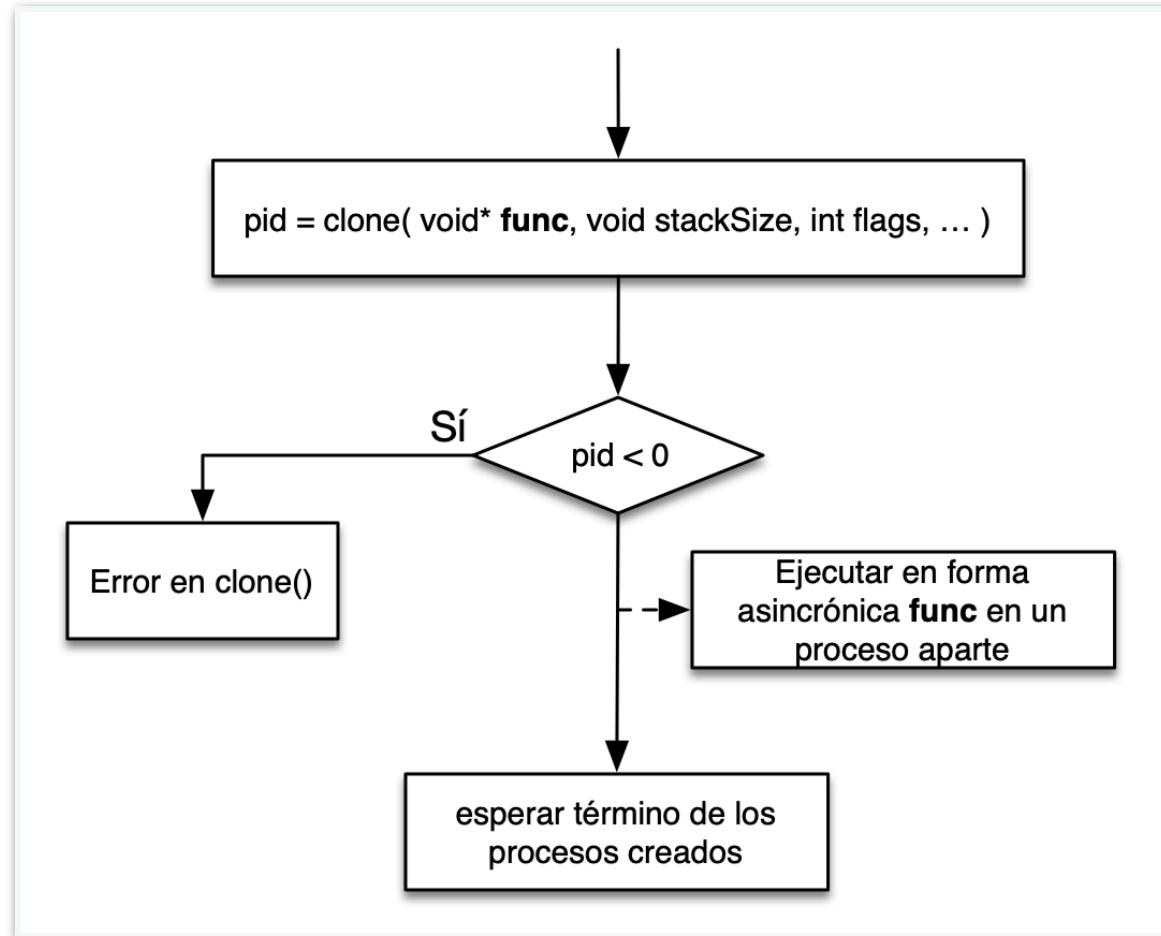
    if((pid=fork()) == 0) { //Hijo
        hola();
    } else if(pid > 0) { //Padre
        mundo();
    }

    return(EXIT_SUCCESS);
}
```

Ejemplo de ejecución no determinista

```
[gabriel@ssoo:~/code_test_ssoo/process/fork_example$ for i in {1..10}; do ./main; echo ""; done
E esst eu ne jreortciucnidoo fracaso...
E esst eu ne jreortcuincdioo fracaso...
E esst e uenj errotcuincdioo fracaso...
E esste uenj erortcuincdioo fracaso...
E esst eu ne jerroctiucndioo fracaso...
E estse uenj reortcuincido fracaso...
E esst eu ne jreortcuincdioo fracaso...
E esst eu ne ej erroctiuncdioo fracaso...
E esst eu enj erroctuincdioo fracaso...
E esst e uenj erroctuincdioo fracaso...
gabriel@ssoo:~/code_test_ssoo/process/fork_example$ ]
```

Creación mediante clone()



flags

SIGCHLD

Obliga al kernel a enviar la señal SIGCHLD desde el proceso hijo al proceso padre cuando el primero termina.

Esto permite que el proceso padre espere el término del proceso hijo a través de la función `wait()`.

CLONE_VM

Permite que el proceso hijo COMPARTA la memoria del proceso padre.

Otras formas de crear procesos

Creación

A través del
syscall exec()

```
#include <unistd.h>

int execl( const char *path, const char *arg, ...);
int execlp( const char *file, const char *arg, ...);
int execle( const char * path, const char *arg, ...,
            char * const envp[]);
int execv( const char * path, char *const argv[]);
int execvp( const char *file, char *const argv[]);
int execve(const char *filename, char *const argv [],
           char *const envp[])
```

Cuando se utiliza el syscall exec*(), el proceso llamador es **reemplazado** por el programa invocado.

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

int main(int argc, char *argv[]){
    printf("HOLA MUNDO, PID:%d\n", getpid() );
    execl("./test", "test",0);
    printf("HOLA MUNDO2, PID:%d\n", getpid() );
    return(EXIT_SUCCESS);
}
```

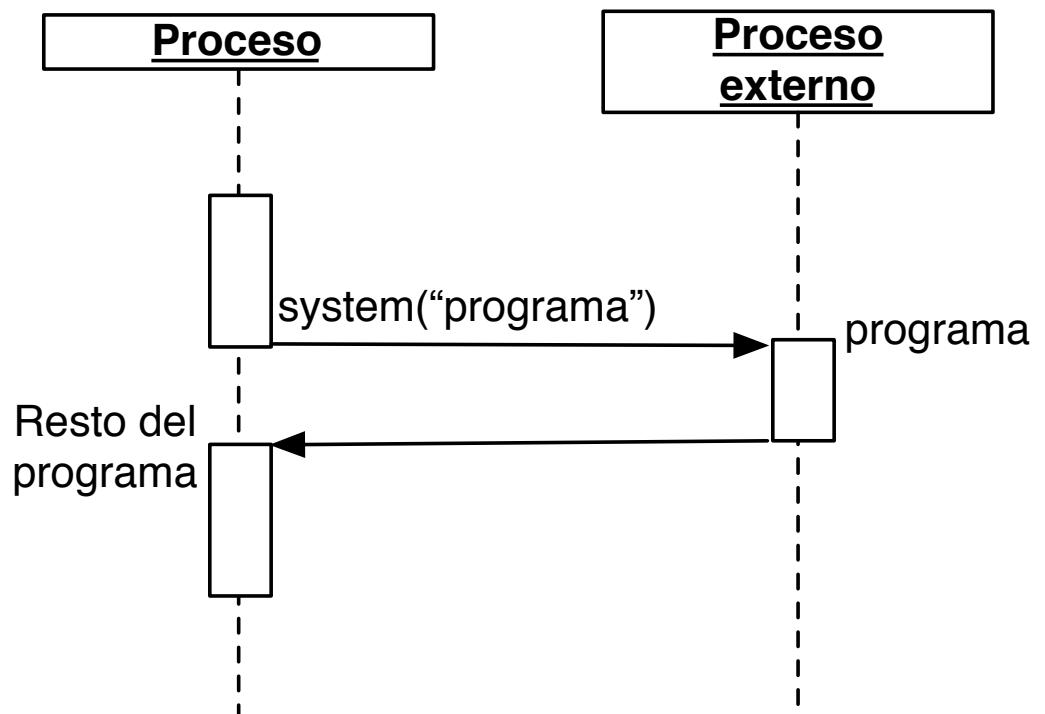
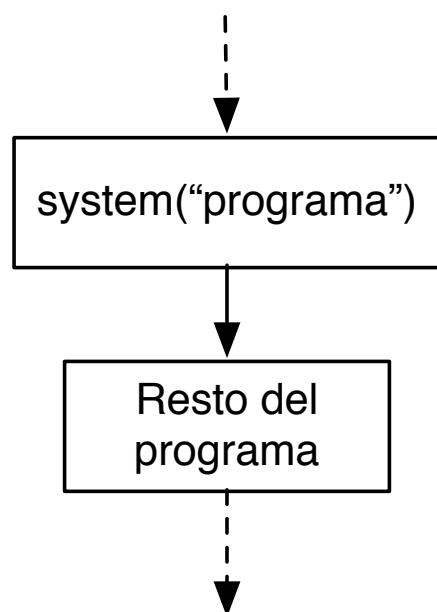
ejemplo.c

El programa test imprime en pantalla su PID.

¿Cuál es la salida del programa ejemplo.c?

Ejecutar otro programa sincrónico

```
#include <stdlib.h>
int system(const char * prgstring);
```



Término del proceso

Cuando se termina la función **main()**.

Llama a la función registradas por `atexit()` y retorna al kernel.
`status` puede ser: `EXIT_SUCCESS` o `EXIT_FAILURE` (0 ó 1)

Otra función de salida:
_exit(int status).

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

void limpiar();

int main() {

    if(atexit(limpiar) != 0) {
        perror("Error en atexit:");
        exit(EXIT_FAILURE);
    }

    printf("Ultima instruccion...\n");
    exit(EXIT_SUCCESS);
}

void limpiar(void){
    printf("Aca va el codigo final final\n");
}
```

Threads

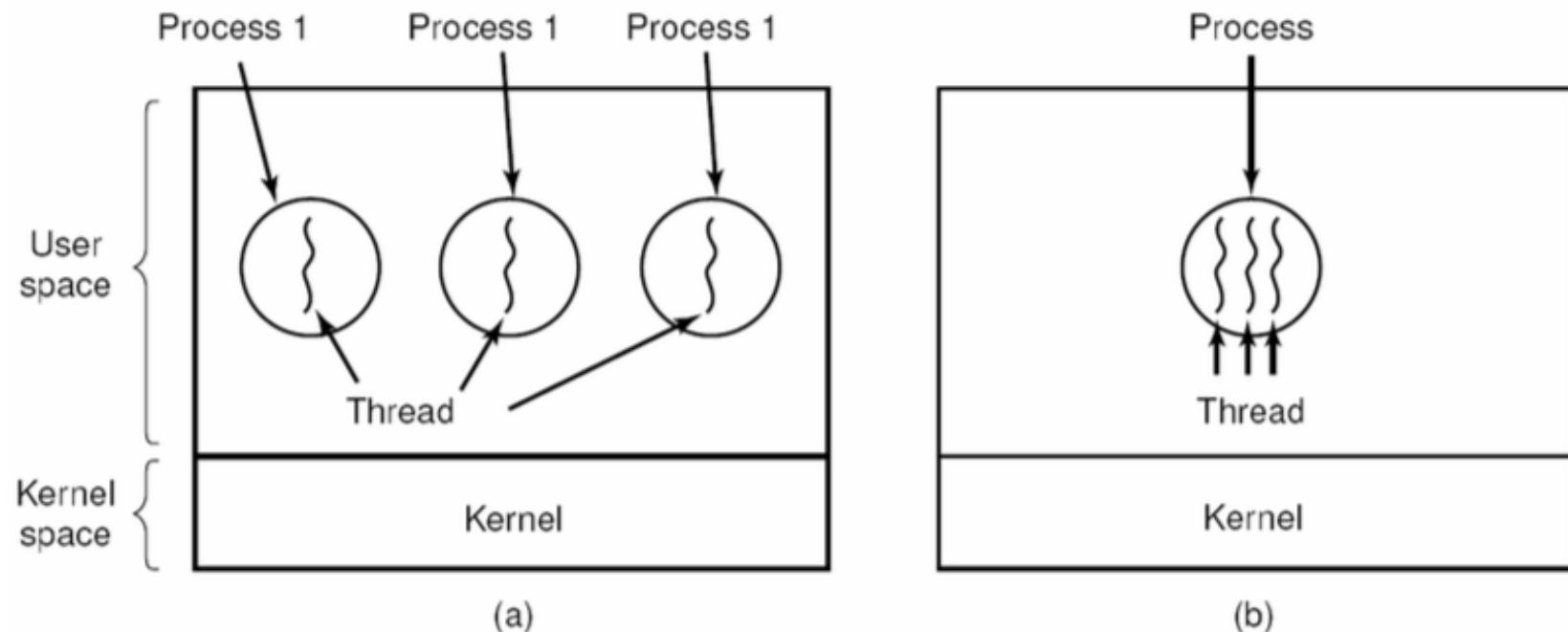
Introducción

Hasta ahora, un proceso es, a la vez:

- **Unidad de asignación de recursos**
- **Unidad de planificación y ejecución**

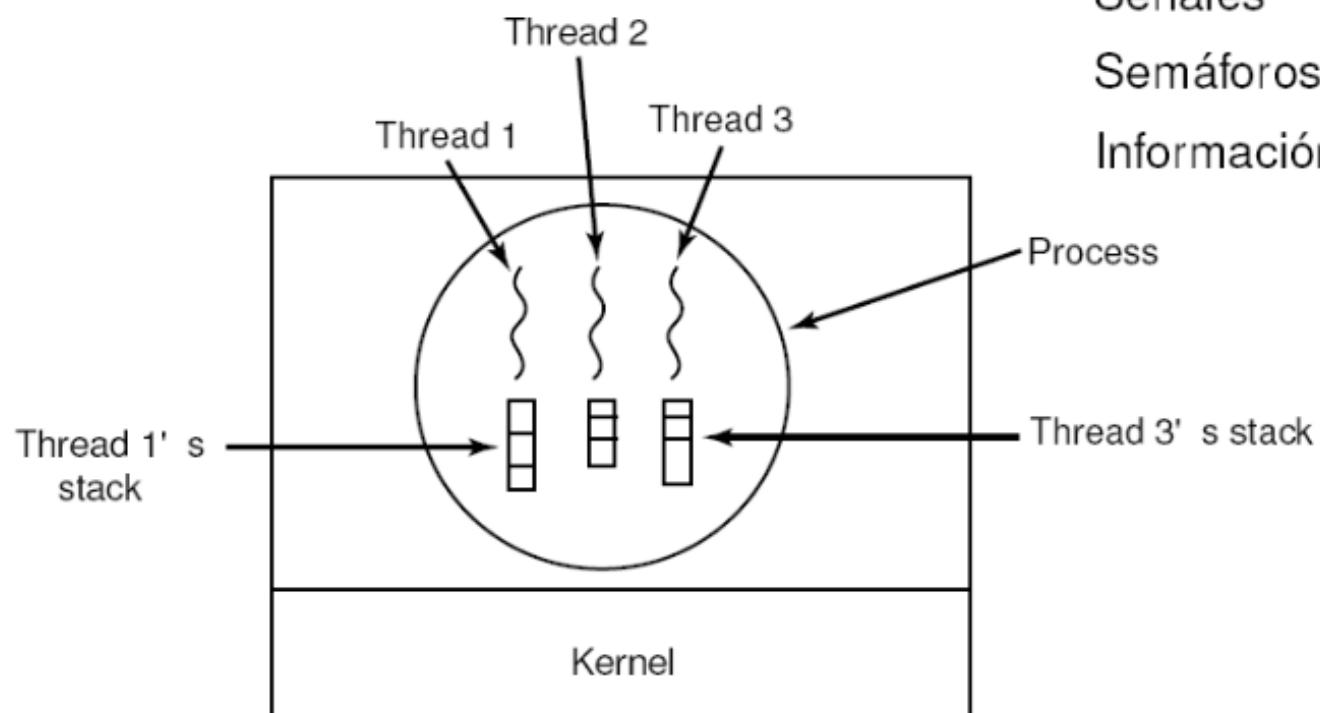
Aunque no tiene porqué: **hilo** es la unidad de planificación y **proceso** es la unidad de asignación de recursos

Un proceso puede tener **varios hilos**



Hilos / Procesos

Elementos por hilo	Elementos por proceso
Contador de programa	Espacio de direcciones
Pila	Variables globales
Estado + contexto	Ficheros abiertos
Memoria privada (var. locales)	Procesos hijos
	Cronómetros
	Señales
	Semáforos
	Información contable



Alcances

Por qué implementar threads

- Descomponer la aplicación en múltiples subprocessos secuenciales paralelos
- Permiten solapar E/S y cómputo dentro de un mismo proceso
- Con múltiples CPUs se puede conseguir verdadero paralelismo
- Son más rápidos de crear y destruir

Sin thread

Procesador de textos: guardar automáticamente, operaciones periódicas...

Hoja de cálculo: Recálculo de las celdas.

Servidor web: Hilo despachador que asigna consultas a hilos trabajadores.

Aplicación que procesa gran cantidad de datos: solapamos E/S y cálculo.

Sería imposible

Características

No existe protección (ni se necesita) entre los hilos de un mismo proceso

Comparten variables globales → **Sincronización**

Mejoran el rendimiento (menos sobrecarga del *kernel*)

- Creación más rápida
- Comunicación más eficiente

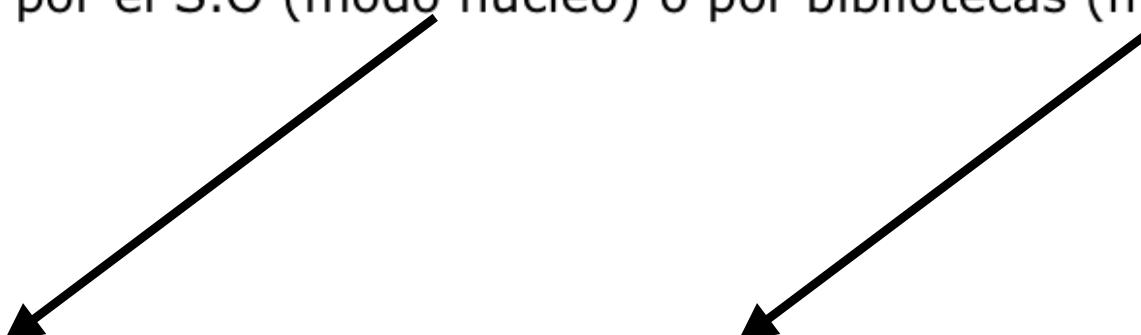
Soportados por el S.O (modo núcleo) o por bibliotecas (modo usuario)

Kernel thread

Native threads

User thread

Green threads



User Threads

Ventajas

- El núcleo no sabe que existen
- Tabla de hilos privada en el proceso para cambios de contexto
- Cambio de contexto mucho más rápido entre hilos (no se pasa al kernel)
- Cada proceso puede tener su algoritmo de planificación

Desventajas

- Llamadas bloqueantes al sistema bloquean el proceso.
Solución: funciones no bloqueantes
- Tienen que ceder la CPU entre ellos, lo que los limita a commutar dentro del mismo proceso
- Precisamente queremos hilos en procesos que tienen mucha E/S para obtener paralelismo, es decir, que se están bloqueando muy frecuentemente.

Kernel Threads

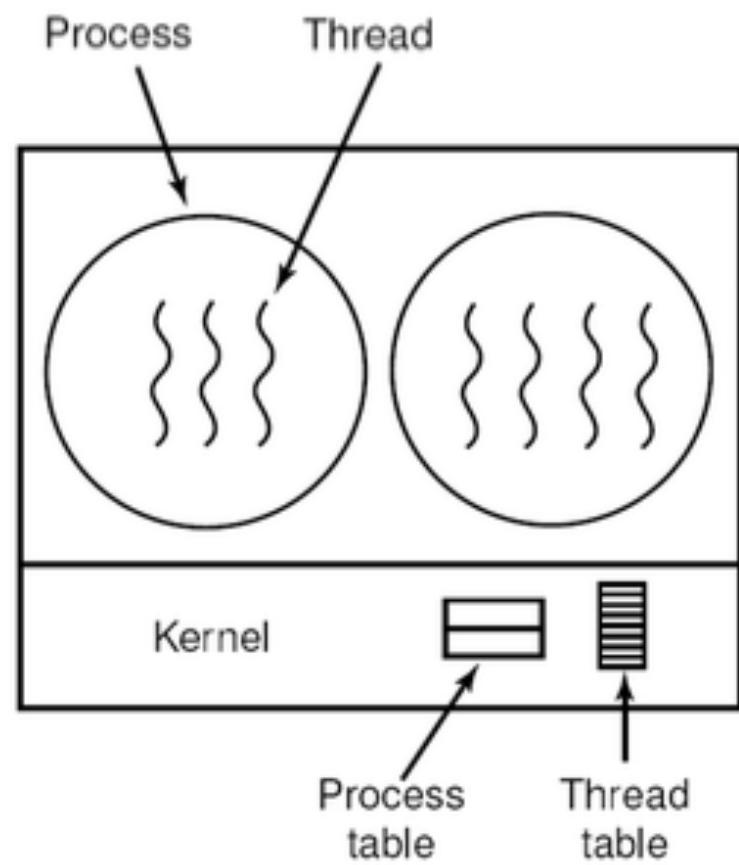
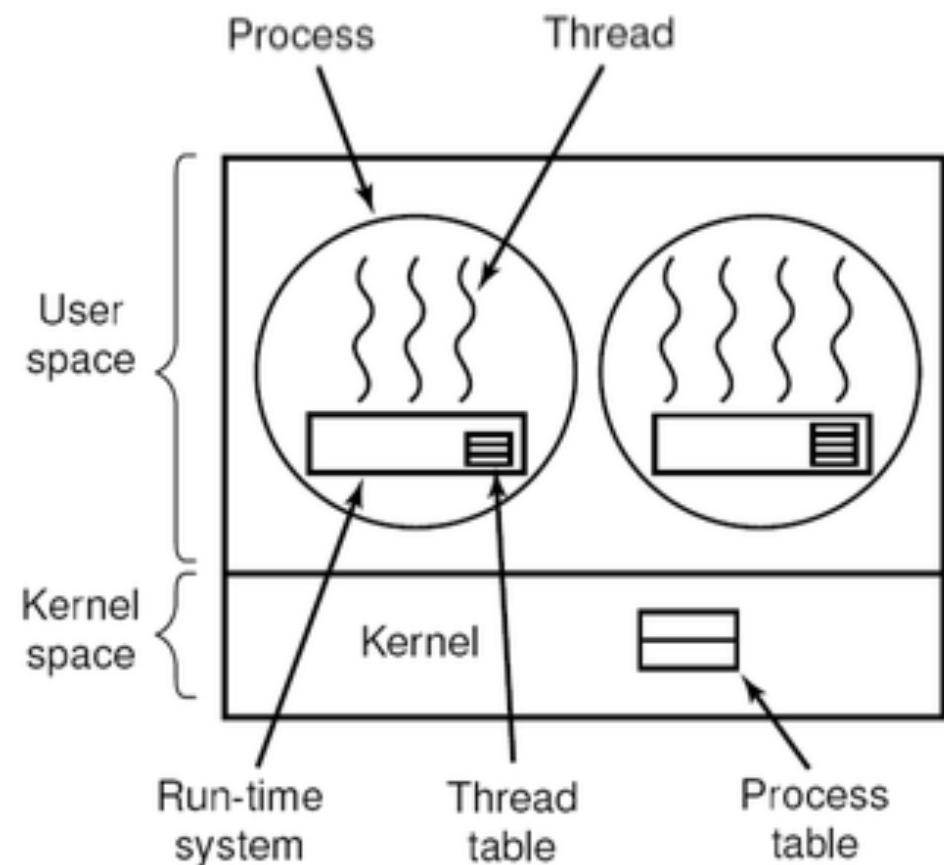
Ventajas

- El núcleo mantiene la tabla de hilos
- Las llamadas bloqueantes no necesitan funciones especiales
- Los fallos de página no suponen un problema
- Al bloquearse un hilo, el núcleo puede conmutar a otro hilo de otro proceso

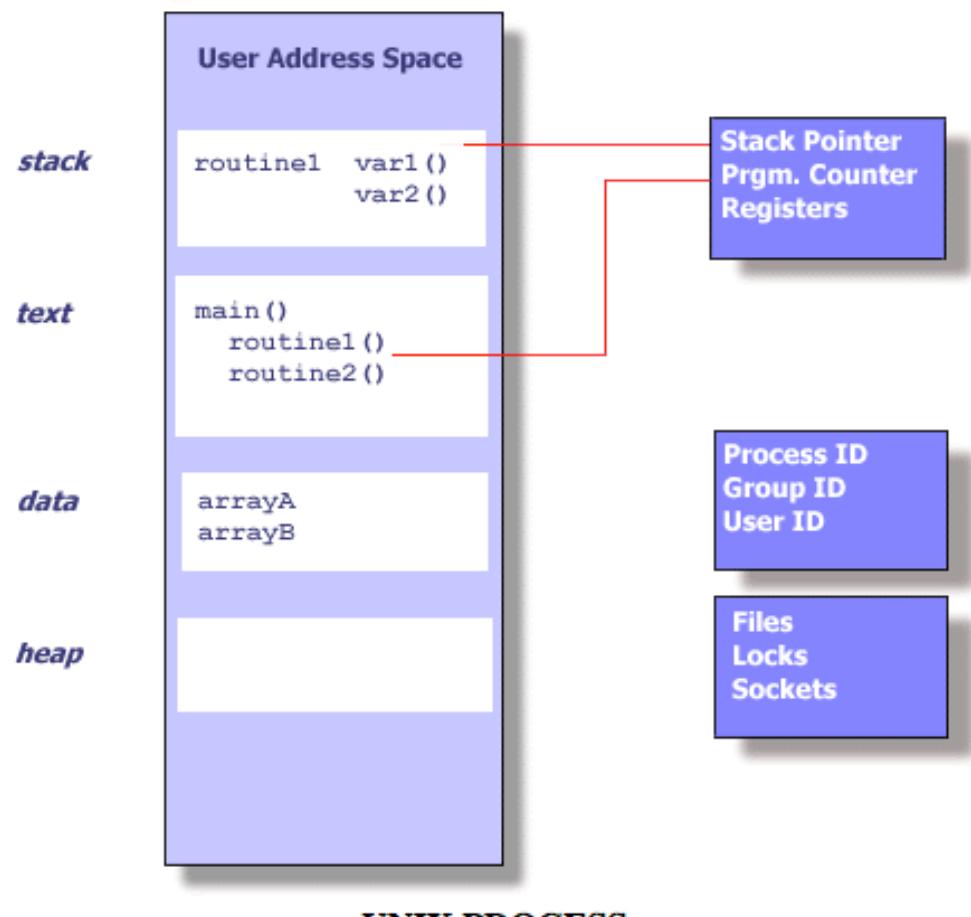
Desventajas

- Las llamadas bloqueantes son llamadas al sistema, e.d. más costosas
- La creación y destrucción de hilos es más costoso ⇒ Reutilización de hilos

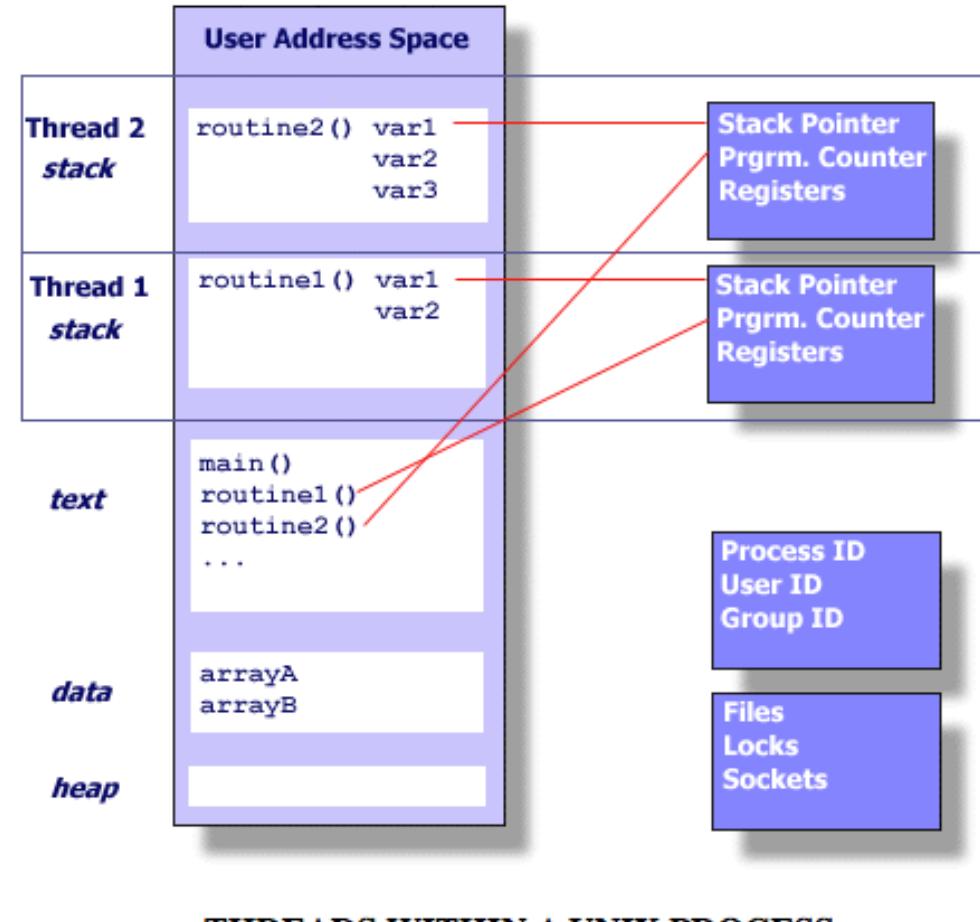
Kernel/User Threads



Procesos v/s Threads



UNIX PROCESS



THREADS WITHIN A UNIX PROCESS

Ejemplo User Threads

```
void hola(void *arg ) {
    char *msg = " Hola";
    int i;
    for ( i = 0 ; i < strlen ( msg ) ; i++ ) {
        printf (" %c", msg [i]);
        fflush ( stdout );
        usleep (1000000) ;
    }
}
```

```
void mundo ( void *arg ) {
    char *msg = " mundo ";
    int i;
    for ( i = 0 ; i < strlen ( msg ) ; i++ ) {
        printf (" %c", msg [i]);
        fflush ( stdout );
        usleep (1000000) ;
    }
}
```

```
int main(int argc , char *argv []) {
    pthread_t h1;
    pthread_t h2;
    pthread_create (&h1 , NULL , (void*)hola , NULL );
    pthread_create (&h2 , NULL , (void*)mundo , NULL);
    printf ("Fin create\n");

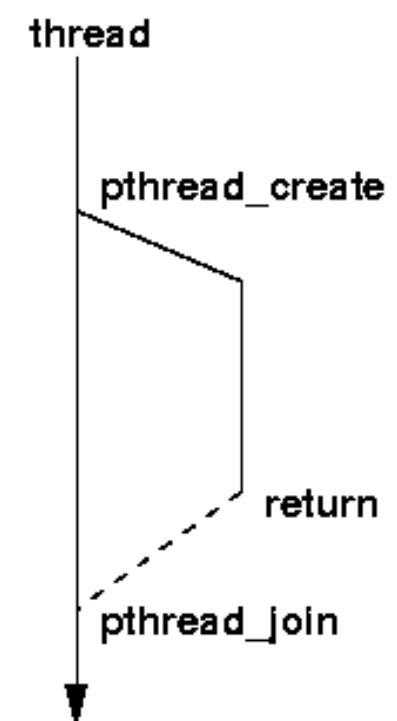
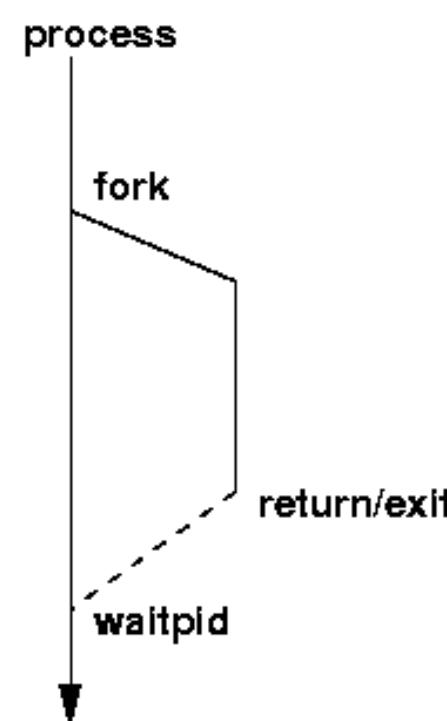
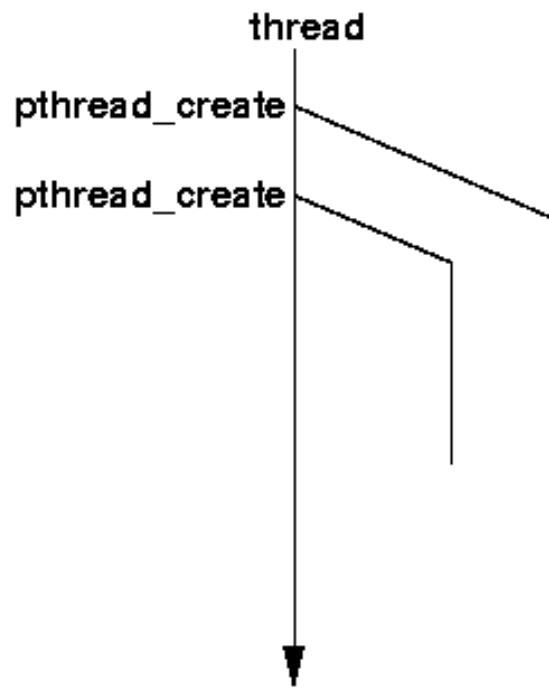
    pthread_join(h1, NULL);
    pthread_join(h2, NULL);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
```

Ejemplo User Threads

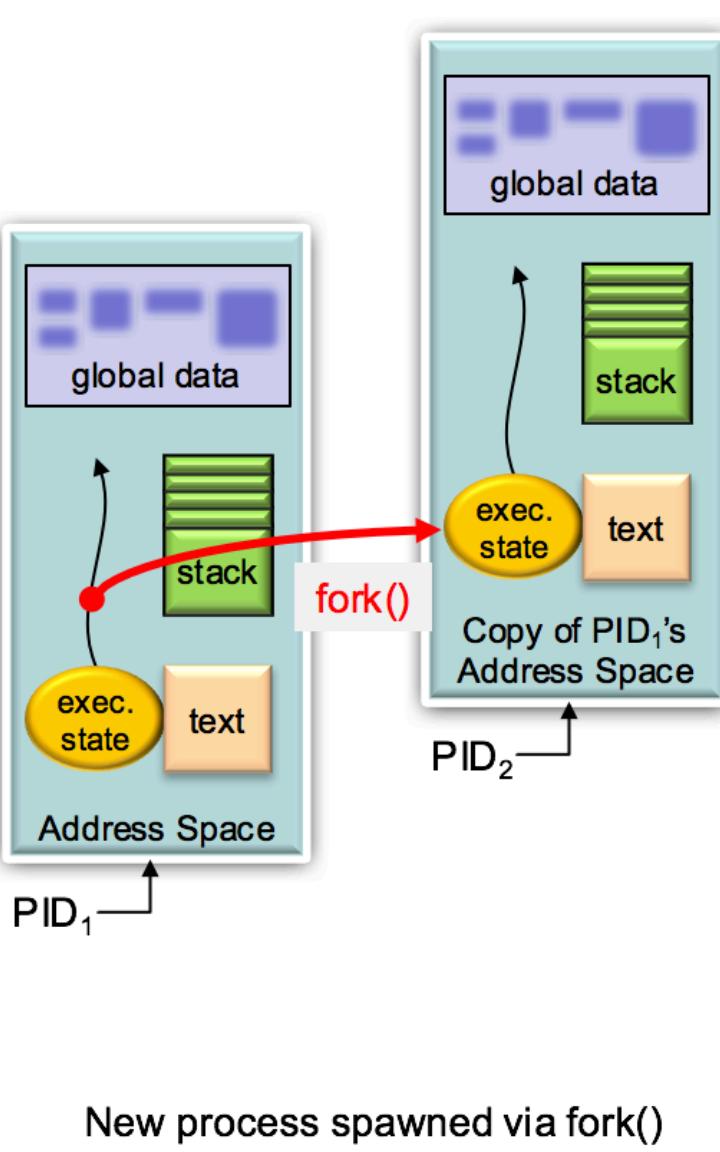
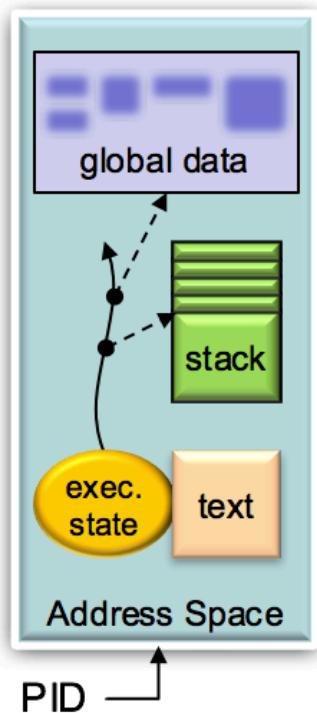
```
int main(int argc , char *argv []) {
    pthread_t h1;
    pthread_t h2;
    pthread_create (&h1 , NULL , (void*)hola , NULL );
    pthread_create (&h2 , NULL , (void*)mundo , NULL);
    printf ("Fin create\n");

    pthread_join(h1, NULL);
    pthread_join(h2, NULL);
}
```



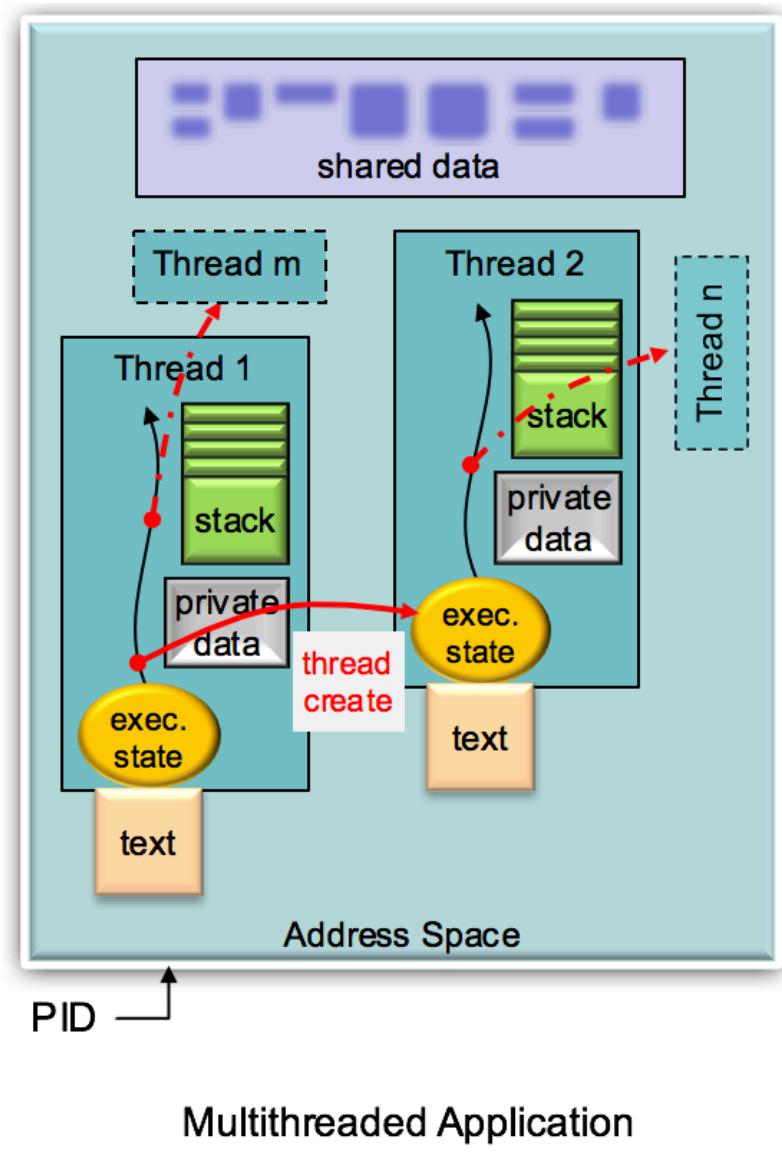
Procesos/Threads

Procesos / Threads



Standard
UNIX process
(single-threaded)

New process spawned via `fork()`



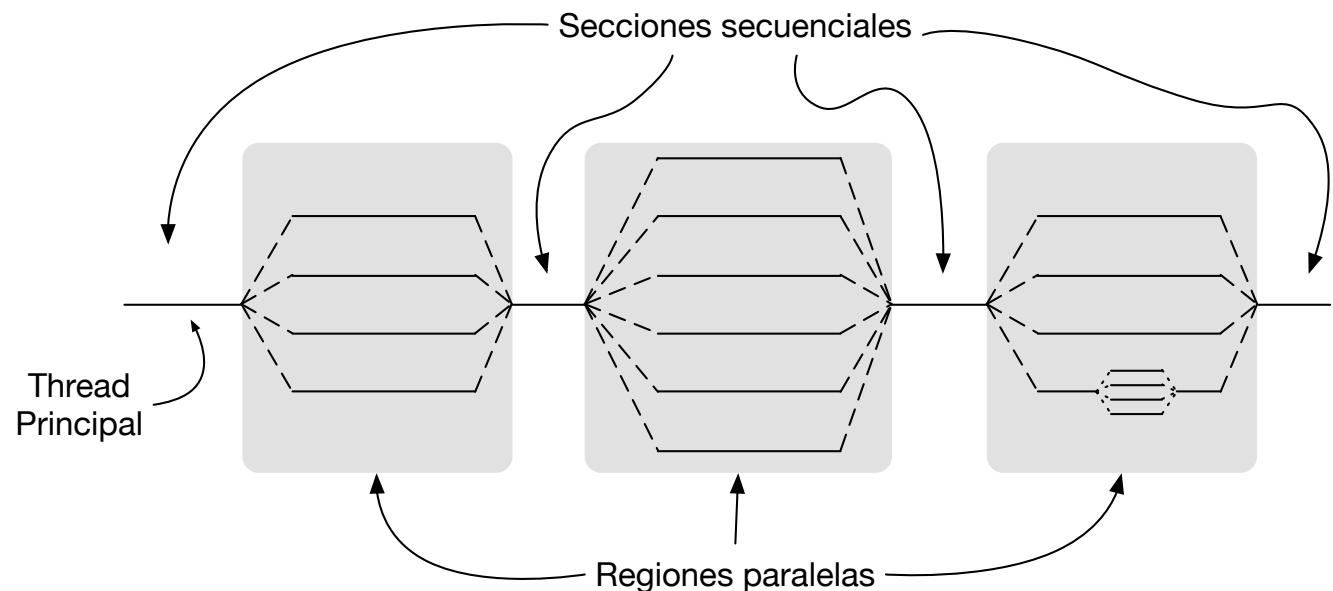
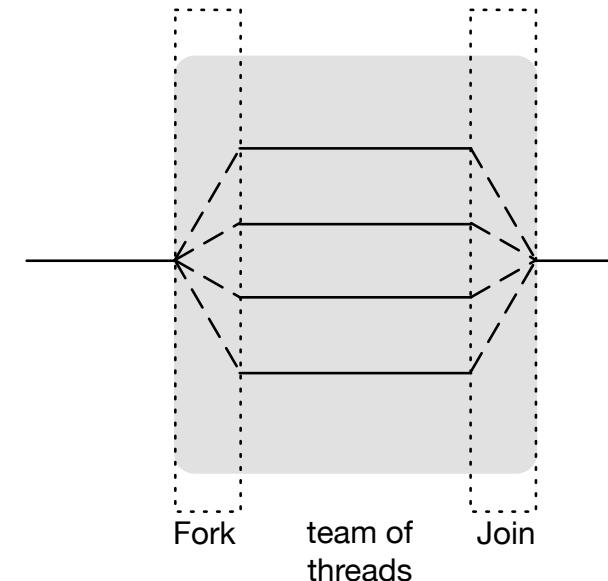
Multithreaded Application

Procesos / Threads

Idea base

El thread principal genera un grupo (team) de threads
(modelo FORK-JOIN)

El paralelismo se agrega incrementalmente a la solución secuencial



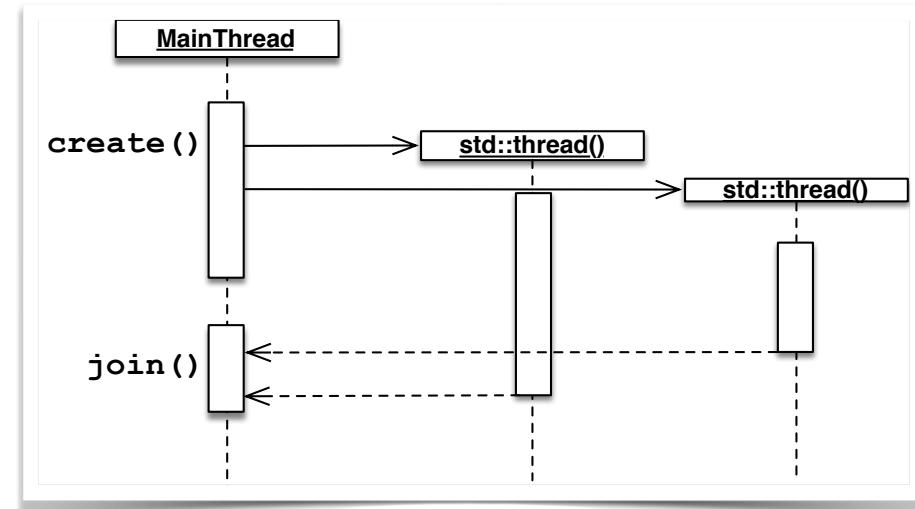
Threads C++ (standard \geq 2011)

```
#include <thread>

void threadMain(){
    std::cout << "thread staff" << std::endl;
}

int main(int argc, char** argv){
    std::thread t01(threadMain);
    std::cout << "main thread" << std::endl;
    t01.join();

    return(EXIT_SUCCESS);
}
```



```
$ ./example
main thread
thread staff
```

```
$ ./example
main thread
thread staff
```

Threads C++ (standard ≥ 2011)

```
int main(int argc, char** argv){  
  
    std::cout << "Total threads = " <<  
        std::thread::hardware_concurrency() <<  
        std::endl;  
  
    std::string msg = "hello!!!";  
    std::thread t01(threadMain, msg);  
  
    std::cout << "main thread" <<  
        std::endl;  
    std::cout << "main thread: id = " <<  
        std::this_thread::get_id() <<  
        std::endl;  
    std::cout << "main thread: child id = " <<  
        t01.get_id() <<  
        std::endl;  
  
    t01.join();  
  
    return(EXIT_SUCCESS);  
}
```

```
void threadMain(const std::string& msg){  
    std::cout << "thread child: start" <<  
        std::endl;  
  
    std::cout << "thread child: id = " <<  
        std::this_thread::get_id() <<  
        std::endl;  
  
    std::cout << "thread child: msg=" <<  
        msg <<  
        std::endl;  
}
```

`std::thread::hardware_concurrency()`

Sugerencia para la cantidad máxima de threads

`std::this_thread`

Referencia al thread actual

Threads C++ (standard \geq 2011)

```
void sumaParcial(const std::vector<uint32_t> &v,
                  uint32_t &suma,
                  size_t beginIndex,
                  size_t endIndex) {
    suma = 0;
    for (size_t i = beginIndex; i < endIndex; ++i){
        suma += v[i];
    }
}
```

Creación de threads (1)

Con punteros a funciones

```
=====THREADS=====
//(1) Separación del trabajo
std::thread t1(sumaParcial, std::ref(v), std::ref(suma1), 0, v.size() / 2);
std::thread t2(sumaParcial, std::ref(v), std::ref(suma2), v.size() / 2, v.size());

t1.join();
t2.join();

//(2) Reducción (Consolidación de resultados parciales)
sumaThreads = suma1 + suma2;
```

Threads C++ (standard ≥ 2011)

```
void sumaParcial(const std::vector<uint32_t> &v,
                  uint32_t &suma,
                  size_t beginIndex,
                  size_t endIndex) {
    suma = 0;
    for (size_t i = beginIndex; i < endIndex; ++i){
        suma += v[i];
    }
}
```

```
t main(int argc, char** argv){

    uint32_t totalElementos = 100000000;

    uint32_t sumaSerial = 0;
    uint32_t suma1 = 0;
    uint32_t suma2 = 0;
    uint32_t sumaThreads = 0;

    std::vector<uint32_t> v;

    for(size_t i=0; i < totalElementos; i++){
        v.push_back(i);
    }

    //=====SERIAL=====
    sumaParcial( std::ref(v), std::ref(sumaSerial), 0, v.size() );

    //=====THREADS=====
    //((1) Separación del trabajo
    std::thread t1(sumaParcial, std::ref(v), std::ref(suma1), 0, v.size() / 2);
    std::thread t2(sumaParcial, std::ref(v), std::ref(suma2), v.size() / 2, v.size());

    t1.join();
    t2.join();

    //((2) Reducción (Consolidación de resultados parciales)
    sumaThreads = suma1 + suma2;

    std::cout << "=====Serial====" << std::endl;
    std::cout << "sumaSerial: " << sumaSerial << std::endl;

    std::cout << "=====Threads====" << std::endl;
    std::cout << "suma1: " << suma1 << std::endl;
    std::cout << "suma2: " << suma2 << std::endl;
    std::cout << "suma1 + suma2: " << sumaThreads << std::endl;

    return(EXIT_SUCCESS);
}
```

Threads C++ (standard \geq 2011)

```
class CsumaParcial{
public:
    void operator()(const std::vector<uint32_t> &v,
                     uint32_t& sum,
                     size_t beginIndex, size_t endIndex){

        sum = 0;
        for (size_t i = beginIndex; i < endIndex; ++i) {
            sum += v[i];
        }
    }
};
```

Creación de threads (2)

Con functors

```
=====THREADS=====
//(1) Separación del trabajo

CsumaParcial sumador1 = CsumaParcial();
CsumaParcial sumador2 = CsumaParcial();

std::thread t1(std::ref(sumador1), std::ref(v), std::ref(sumaParcial1), 0, v.size() / 2);
std::thread t2(std::ref(sumador2), std::ref(v), std::ref(sumaParcial2), v.size() / 2, v.size());
t1.join();
t2.join();

//(2) Reducción (Consolidación de resultados parciales)
//sumaThreads = suma1.suma() + suma2.suma();
sumaThreads = sumaParcial1 + sumaParcial2;
```

Threads C++ (standard \geq 2011)

```
int main(int argc, char** argv){  
  
    uint32_t totalElementos = 100000000;  
  
    uint32_t sumaSerial = 0;  
    uint32_t sumaParcial1 = 0;  
    uint32_t sumaParcial2 = 0;  
    uint32_t sumaThreads = 0;  
  
    std::vector<uint32_t> v;  
  
    for(size_t i=0; i < totalElementos; i++){  
        v.push_back(i);  
    }  
  
    //=====SERIAL=====  
    CsumaParcial CsumaSerial = CsumaParcial();  
    CsumaSerial( std::ref(v), std::ref(sumaSerial), 0, v.size() );  
  
    //=====THREADS=====  
    // (1) Separación del trabajo  
  
    CsumaParcial sumador1 = CsumaParcial();  
    CsumaParcial sumador2 = CsumaParcial();  
  
    std::thread t1(std::ref(sumador1), std::ref(v), std::ref(sumaParcial1), 0, v.size() / 2);  
    std::thread t2(std::ref(sumador2), std::ref(v), std::ref(sumaParcial2), v.size() / 2, v.size());  
    t1.join();  
    t2.join();  
  
    // (2) Reducción (Consolidación de resultados parciales)  
    // sumaThreads = suma1.suma() + suma2.suma();  
    sumaThreads = sumaParcial1 + sumaParcial2;  
  
    std::cout << "====Serial====" << std::endl;  
    std::cout << "sumaSerial: " << sumaSerial << std::endl;  
  
    std::cout << "====Threads====" << std::endl;  
    std::cout << "suma1: " << sumaParcial1 << std::endl;  
    std::cout << "suma2: " << sumaParcial2 << std::endl;  
    std::cout << "suma1 + suma2: " << sumaThreads << std::endl;  
  
    return(EXIT_SUCCESS);  
}
```

```
class CsumaParcial{  
public:  
    void operator()(const std::vector<uint32_t> &v,  
                    uint32_t& sum,  
                    size_t beginIndex, size_t endIndex){  
  
        sum = 0;  
        for (size_t i = beginIndex; i < endIndex; ++i) {  
            sum += v[i];  
        }  
    }  
};
```

```
int main(int argc, char** argv){  
    uint32_t totalElementos = 100000000;  
  
    uint32_t sumaSerial = 0;  
    uint32_t sumaParcial1 = 0;  
    uint32_t sumaParcial2 = 0;  
    uint32_t sumaThreads = 0;  
  
    std::vector<uint32_t> v;  
  
    for(size_t i=0; i < totalElementos; i++){  
        v.push_back(i);  
    }  
}
```

Threads C++ (standard \geq 2011)

Creación de threads (3)

Con Lambda Functions

```
//=====SERIAL=====  
for(auto& num : v){  
    sumaSerial += num;  
}
```

```
=====THREADS=====  
//(1) Separación del trabajo  
auto sumaThread = [](std::vector<uint32_t> &v, uint32_t& suma , size_t left, size_t right) {  
    suma = 0;  
    for (size_t i = left; i < right; ++i){  
        suma += v[i];  
    }  
};  
  
std::thread t1( sumaThread, std::ref(v), std::ref(sumaParcial1), 0, v.size()/2 );  
std::thread t2( sumaThread, std::ref(v), std::ref(sumaParcial2), v.size()/2, v.size() );  
  
t1.join();  
t2.join();  
  
//(2) Reducción (Consolidación de resultados parciales)  
sumaThreads = sumaParcial1 + sumaParcial2;
```

Tasks C++ (standard \geq 2011)

```
#include <future>

auto sumaParcial = [](std::vector<uint32_t> &v, size_t left, size_t right) {
    uint32_t suma = 0;
    for (size_t i = left; i < right; ++i){
        suma += v[i];
    }

    return suma;
};

//=====THREADS=====
//(1) Separación del trabajo
auto t1 = std::async(std::launch::async, sumaParcial, std::ref(v), 0, v.size() / 2);
auto t2 = std::async(std::launch::async, sumaParcial, std::ref(v), v.size() / 2, v.size());

sumaParcial1 = t1.get();
sumaParcial2 = t2.get();

//(2) Reducción (Consolidación de resultados parciales)
//sumaThreads = suma1.suma() + suma2.suma();
sumaThreads = sumaParcial1 + sumaParcial2;
```

Threads C++ (standard ≥ 2011)

Dos ejecuciones distintas...

```
./example
Total threads = 4
main thread
main thread: id = 0x7fff8cee2380
main thread: child id = 0x70000e0c0000
thread child: start
thread child: id = 0x70000e0c0000
thread child: msg=hello!!

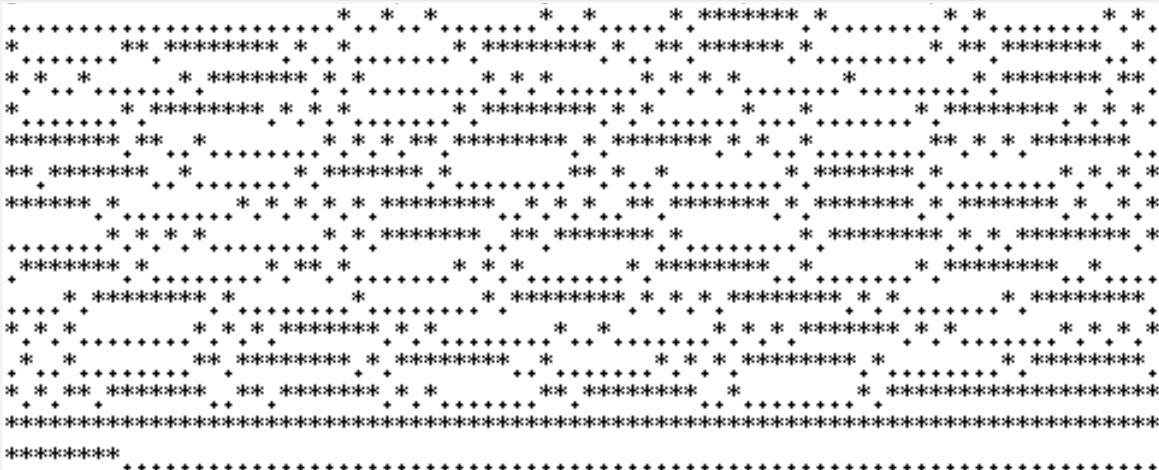
./example
Total threads = 4
main thread
main thread: id = 0x7fff8cee2380
thread child: start

thread child: id = 0x700008507000
thread child: msg=main thread: child id = hello!!0x700008507000
```

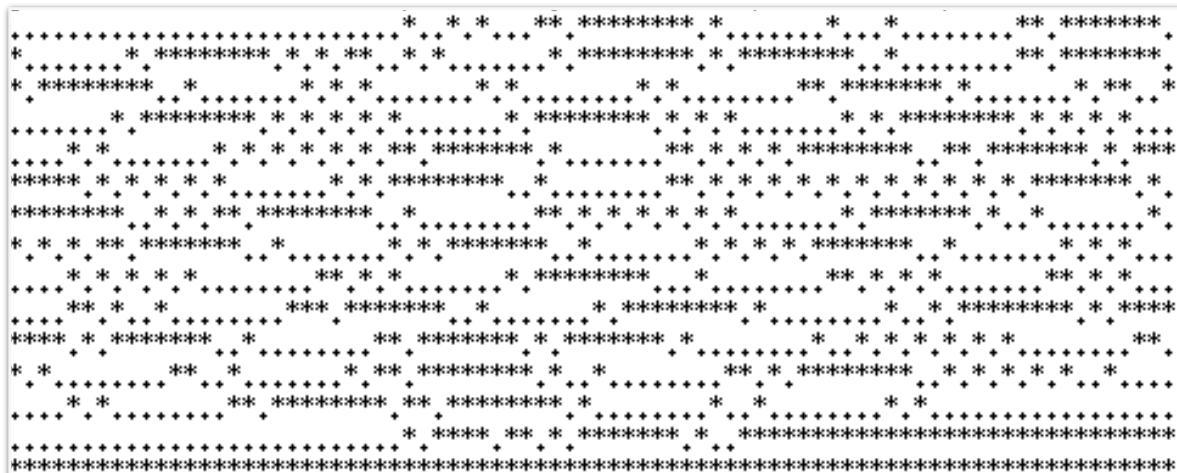
Threads C++ (standard ≥ 2011)

Dos ejecuciones distintas...

```
void threadMain(const std::string& sym, const uint32_t cMax){  
    for(size_t i = 0; i < cMax; ++i) {  
        std::cout << sym;  
    }  
}  
  
int main(int argc, char** argv){  
    std::thread t01(threadMain, std::string("."), 600);  
    std::thread t02(threadMain, std::string("*"), 600);  
  
    t01.join();  
    t02.join();  
  
    return(EXIT_SUCCESS);  
}
```



A terminal window displaying the output of the program. The screen is filled with the character '.' repeated in a grid pattern, indicating that the first thread (t01) has printed 600 lines of '.'.



A terminal window displaying the output of the program. The screen is filled with the character '*' repeated in a grid pattern, indicating that the second thread (t02) has printed 600 lines of '*'.

Race Condition

Situación donde el resultado depende del orden relativo de ejecución de dos o más procesos o hilos.

```
int main(int argc, char** argv){  
  
    std::cout << "MAIN: balance=" << balance << std::endl;  
  
    std::thread t00(depositar, 0, 1000000);  
    std::thread t01(depositar, 1, 1000000);  
  
    t00.join();  
    t01.join();  
  
    std::cout << "MAIN: balance=" << balance << std::endl;  
  
    return(EXIT_SUCCESS);  
}
```

```
static volatile int balance = 0;  
  
void depositar(const uint32_t thID, const uint32_t totalDinero){  
  
    for(size_t i = 0; i < totalDinero; i++) {  
        balance = balance + 1;  
    }  
  
    std::cout << "Thread:" << thID << " FIN." << std::endl;  
}
```

```
gabriel@homeserver:~/UV/02-pthreads/Ejercicios/example05$ ./example  
MAIN: balance=0  
Thread:0 FIN.  
Thread:1 FIN.  
MAIN: balance=1044228  
gabriel@homeserver:~/UV/02-pthreads/Ejercicios/example05$ ./example  
MAIN: balance=0  
Thread:0 FIN.  
Thread:1 FIN.  
MAIN: balance=1029762
```

Código:
example05

Race Condition

```
#include <mutex>
```

```
std::mutex mux; //Debe ser global

thread_loop{
    mux.lock();
    //
    // Sección crítica
    //
    mux.unlock();
}
```

```
std::mutex mux;

void depositar_safe(const uint32_t thID, const uint32_t totalDinero){

    for(size_t i = 0; i < totalDinero; i++) {
        mux.lock();
        balance = balance + 1;
        mux.unlock();
    }

    std::cout << "Thread:" << thID << " FIN." << std::endl;
}
```

```
std::mutex g_DMutex;

void threadMain(const std::string& msg){

    g_DMutex.lock();
    std::cout << "thread child: start" <<
        std::endl;
    g_DMutex.unlock();

    g_DMutex.lock();
    std::cout << "thread child: id = " <<
        std::this_thread::get_id() <<
        std::endl;
    g_DMutex.unlock();

    g_DMutex.lock();
    std::cout << "thread child: msg=" <<
        msg <<
        std::endl;
    g_DMutex.unlock();
}
```

