

# *Introducción a los Sistemas Operativos*

## Procesos - IV



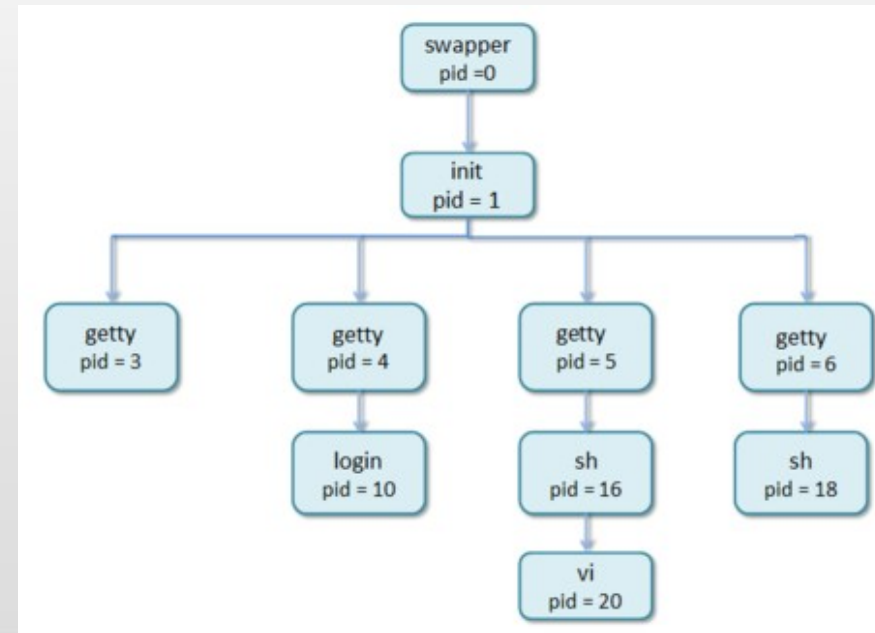
- ✓ Versión: Agosto 2024
- ✓ Palabras Claves: Procesos, Linux, Windows, Creación, Terminación, Fork, Execve, Relación entre procesos

Los temas vistos en estas diapositivas han sido mayormente extraídos del libro de Andrew S. Tanenbaum (Sistemas Operativos Modernos)



# Creación de procesos

- ✓ Un proceso es creado por otro proceso
- ✓ Un proceso padre tiene uno o más procesos hijos.
- ✓ Se forma un árbol de procesos



# Actividades en la creación

- ✓ Crear la PCB
- ✓ Asignar PID (Process IDentification) único
- ✓ Asignarle memoria para regiones
  - Stack, Text y Datos
- ✓ Crear estructuras de datos asociadas
  - Fork (copiar el contexto, regiones de datos, text y stack)



# *Relación entre procesos Padre e Hijo*

Con respecto a la Ejecución:

- ✓ El padre puede continuar ejecutándose concurrentemente con su hijo
- ✓ El padre puede esperar a que el proceso hijo (o los procesos hijos) terminen para continuar la ejecución.



# *Relación entre procesos Padre e Hijo (cont.)*

Con respecto al Espacio de Direcciones:

- ✓ El hijo es un duplicado del proceso padre (caso Unix)
  - ✓ Se crea un nuevo espacio de direcciones copiando el del padre
- ✓ Se crea el proceso y se le carga adentro el programa (caso Windows)
  - ✓ Se crea un nuevo espacio de direcciones vacío



# Creación de Procesos

## ☑ En UNIX: (2 System Calls)

- ✓ system call **fork()** crea nuevo proceso igual al llamador
- ✓ system call **execve()**, generalmente usada después del fork, carga un nuevo programa en el espacio de direcciones.

## ☑ En Windows: (1 System Call)

- ✓ system call **CreateProcess()** crea un nuevo proceso y carga el programa para ejecución.



# ¿Como funciona fork? (1)

*Padre*

**//Instrucciones previas**

```
PC → nue = fork()  
    if nue == 0  
        // hijo  
    elsif nue > 0  
        // padre  
        // nue es el PID del hijo  
    else  
        // error  
    end
```

**PC = Program Counter**





# ¿Como funciona fork? (2)

## Padre (el que invoca fork)

PC→

```
//Inst. previas
nue = fork()
if nue == 0
    // hijo
elseif nue > 0
    // padre
    // nue es PID
hijo
else
    // error en el
    llamado, el proceso
    hijo nunca se creo
end
```

PC→

## Hijo (el creado)

```
//Inst. previas
nue = fork()
if nue == 0
    // hijo
elseif nue > 0
    // padre
    // nue = PID hijo
else
    // error
end
```

**PC = Program Counter**

Fork retorna: 0 en el proceso hijo, valor mayor a cero en el padre (PID del hijo), valor negativo para error



# Ejemplo SysCall fork

```
#
# El padre puede terminar antes que los hijos
#

import os, time
hijos = 0
print '\n\nSoy el PROCESO PADRE. PID: ', os.getpid() , 'y tengo', hijos, 'hijos\n'
print '\n\nQuiero tener un hijo? (sn)'
respuesta = raw_input()
while (respuesta <> 'n'):
    newpid = os.fork()
    if newpid == 0:
        #
        # Seccion del hijo
        #
        time.sleep(30)
        print '\t\t\t\t\t', os.getpid(), ' - Me aburro. Me voy a jugar a la PLAY'
        exit(0)
    else:
        #
        # seccion del padre
        #
        hijos = hijos + 1
        print '\t\tTuve un hijo!!!! Tiene el PID: ', newpid

    print '\n\nQuiero tener otro hijo? (s|n)'
    respuesta = raw_input()

print '\nBueno, hasta aca llegue. Me voy a dormir. Ya con', hijos , 'hijos es suficiente'
exit(0)
```



# Terminación de procesos

- ☑ Ante un (**exit**), se retorna el control al sistema operativo
  - ✓ El proceso padre puede esperar recibir un código de retorno (via **wait**). Generalmente se lo usa cuando se requiere que el padre espere a los hijos.
- ☑ Proceso padre puede terminar la ejecución de sus hijos (**kill**)
  - ✓ La tarea asignada al hijo se terminó
  - ✓ Cuando el padre termina su ejecución
    - ◆ Habitualmente no se permite a los hijos continuar, pero existe la opción.
    - ◆ Terminación en cascada



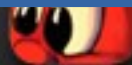
# Ejemplo SysCall fork+wait+exit

```
#
# El padre espera que terminen sus hijos antes de retirarse
#
import os, time
hijos = 0
print '\n\nSoy el PROCESO', os.getpid() , 'y tengo', hijos, 'hijos\n'
while True:
    newpid = os.fork()
    if newpid == 0:
        #
        # Seccion del hijo
        #
        time.sleep(30)
        print '\t(Hijo)', os.getpid(), ' - Se va a jugar a la play'
        exit(0)
    else:
        #
        # Seccion del padre
        #
        hijos = hijos + 1
        print '\t(Padre) Tuve un hijo!!!! Se llama', newpid
        if raw_input( ) == 'q': break

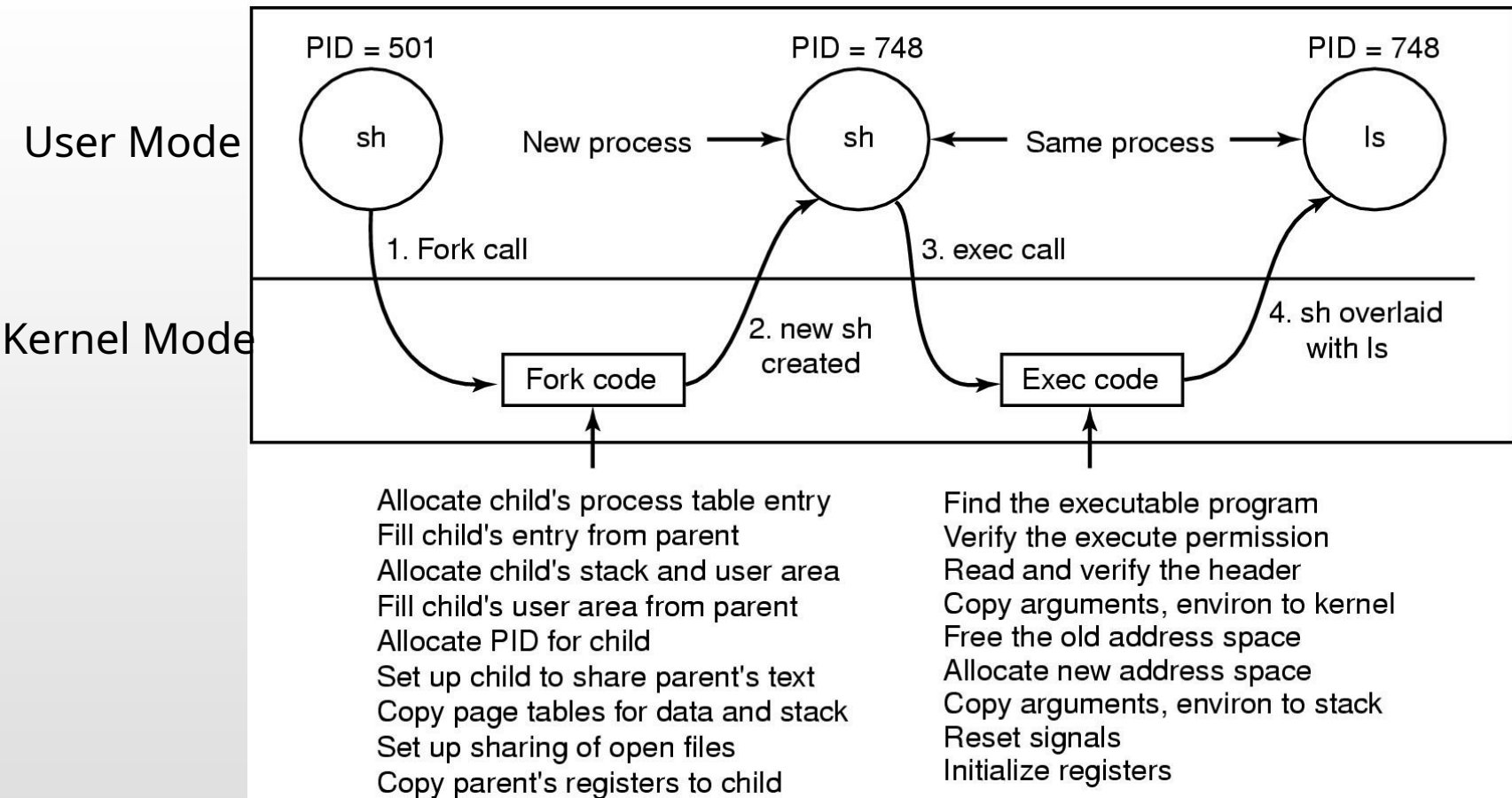
print '\n(Padre) - VAYAN A JUGAR A LA PELOTA!!!!'

while hijos > 0:
    os.wait()
    print '\t(Padre) - Joya, uno menos para cuidar!!!\n'
    hijos = hijos - 1

print '\n(Padre) - Listo, se fueron todos, me voy a dormir'
exit(0)
```



# Fork / Exec - Ejemplo



# Creación y Terminación de Procesos

- ✓ Un ejemplo de una CLI (command line interface) o shell

```
1 while (TRUE) { /* repeat forever */
2     type_prompt( ); /* display prompt */
3     read_command (command, parameters) /* input from terminal */
4
5     if (fork() != 0) { /* fork off child process */
6         /* Parent code */
7         waitpid( -1, &status, 0); /* wait for child to exit */
8
9     } else {
10        /* Child code */
11        execve (command, parameters, 0); /* execute command */
12    }
13 }
14
15
```



# Ejemplo SysCall fork+execv

```
#
# SHELL
#
import os, time
print '''
----- Esta es la ISO DIR/LS SHELL -----
-----

# exit (para salir)
'''

cmd = raw_input("iso:\> ")
while cmd <> 'exit':

    if cmd == '':
        cmd = raw_input("iso:\> ")
    else:
        newpid = os.fork()
        if newpid == 0:
            # Seccion del hijo
            lista = cmd.split(' ')
            os.execvp(lista[0], lista)
            print "Imprimir AAAAAAAAAA"
            exit(0)
            print "Imprimir BBBBBBBBBB"
        else:
            # Seccion del padre
            #os.wait()
            cmd = raw_input("iso:\> ")

exit(0)
```

¿Qué pasa con este código?





# System Calls - Unix

System call	Description
<code>pid = fork( )</code>	Create a child process identical to the parent
<code>pid = waitpid(pid, &amp;statloc, opts)</code>	Wait for a child to terminate
<code>s = execve(name, argv, envp)</code>	Replace a process' core image
<code>exit(status)</code>	Terminate process execution and return status
<code>s = sigaction(sig, &amp;act, &amp;oldact)</code>	Define action to take on signals
<code>s = sigreturn(&amp;context)</code>	Return from a signal
<code>s = sigprocmask(how, &amp;set, &amp;old)</code>	Examine or change the signal mask
<code>s = sigpending(set)</code>	Get the set of blocked signals
<code>s = sigsuspend(sigmask)</code>	Replace the signal mask and suspend the process
<code>s = kill(pid, sig)</code>	Send a signal to a process
<code>residual = alarm(seconds)</code>	Set the alarm clock
<code>s = pause( )</code>	Suspend the caller until the next signal

## *Syscalls de Procesos*





# System calls - Windows

Win32 API Function	Description
CreateProcess	Create a new process
CreateThread	Create a new thread in an existing process
CreateFiber	Create a new fiber
ExitProcess	Terminate current process and all its threads
ExitThread	Terminate this thread
ExitFiber	Terminate this fiber
SetPriorityClass	Set the priority class for a process
SetThreadPriority	Set the priority for one thread
CreateSemaphore	Create a new semaphore
CreateMutex	Create a new mutex
OpenSemaphore	Open an existing semaphore
OpenMutex	Open an existing mutex
WaitForSingleObject	Block on a single semaphore, mutex, etc.
WaitForMultipleObjects	Block on a set of objects whose handles are given
PulseEvent	Set an event to signaled then to nonsignaled
ReleaseMutex	Release a mutex to allow another thread to acquire it
ReleaseSemaphore	Increase the semaphore count by 1
EnterCriticalSection	Acquire the lock on a critical section
LeaveCriticalSection	Release the lock on a critical section

## *Syscalls de Procesos*

