PROSO

Pràctica 1

Implementació d'un sistema operatiu

Grup YO08

Jose Cordero Rama David Guillen Fandos

Algunes decisions preses

Procés 0

Per motius de coherència amb allò que es demana a l'enunciat de la pràctica i per no haver de tractar de forma especial el procés 0, la crida a moltes funcions des de l'esmentat procés provoquen un retorn immediat i erroni. Això és pel fet que directa o indirectament, les funcions d'escriptura i lectura poden provocar el bloqueig del procés, cosa que queda explícitament prohibit al enunciat de la pràctica i que pot provocar una fallida del sistema al trobar la runqueue buida.

Una runqueue buida (suposant que el procés zero es bloqueja) és un deadlock clar, a excepció que hi hagi algun procés llegint de teclat o amb una alarma programada.

Queda exclosa l'operació d'escriptura en el display per part del procés 0, ja que és una funcionalitat requerida per passar els jocs de proves.

Llistes i PCBs

Donat que s'en ha alertat de no fer servir recorreguts sobre els PCBs per que podrien ser molt llargs hem optat per una solució similar al kernel de Linux: una taula se hash. Òbviament no té sentit amb 10 PBCs ja que és quasi més lent que ràpid, però si es vol escalar es pot fer de forma més fiable. La taula fa hashing obert i té la mida proporcional a NR_TASKS (cosa que no fa Linux ja que fa ús de hash encadenat).

Jocs de prova

Joc de prova de les pipes:

Comprovem si funcionen com calen les pipes. Per fer-ho plantegem un problema típic de SO tancant canal de lectura i escriptura

(per part de dos processos para i fill) i jugant amb els SIGPIPE emesos en trencar-se la pipe. A més es comprova de pas si les pipes bloquegen correctament els processos.

El fill es bloqueja fins que algú escriu a la pipe. Ho fa el pare. En aquest moment el pare només té el canal d'escriptura obert i el fill

només canal de lectura. El pare tard o d'hora acaba el seu quantum i es fa task_switch. El fill, que ha estat desbloquejat ja que

hi ha dades escrites a la pipe, continua la seva execució acabant (imprimint el que ha llegit).

En aquest context, en tornar l'execució al pare no hi ha cap lector de la pipe. En escriure a la pipe buida rep un SIGPIPE ja que

és una situació de broken pipe. El signal (que havia estat programat amb un handler que imprimeix un missatge) no es re-programa en cap moment, així que en intentar una nova escriptura fa que el procés mori.

Joc de prova de l'alarma

Aquest joc de proves es limita a crear un fill que programa un alarm als 2 segons. El handler es re-programa mentre no s'hi hagi entrat 10 vegades, moment en el qual no es programa. El següent signal que rep, en no tenir cap handler associat a SIGALARM, mata el procés. Cada alarma rebuda provoca la impressió d'un missatge a pantalla.

<u>Joc de prova PAUSE/CONTINUE</u>

Provarem els signals SIGSTOP i SIGCONT. Es bifurca l'execució creant un pare que envia signals de STOP i CONT al seu fill i l'esmentat fill es dedica a escriure "child working" cada cert temps (més freqüent que no pas la freqüència en que el pare llença els signals).

Pretenem comprovar com el fill escriu de forma continuada i s'aprecia com s'atura dues vegades (el correcte funcionament d'aquest joc de proves és visual, es veurà com es deixen de escriure "child working" durant un cert interval de temps per després continuar l'execució del procés).

Joc de proves bàsic de signals demo_signal

Prova de signals. Tractem de llançar dos signals sobre un procés sense re-programar-ne el handler. El procés mor (ho observem perquè no s'executa la següent instrucció, en aquest cas un printf).

També provem de programar un handler que capturi un SIGKILL i un altra que ho ignori. EL comportament esperat és que mai

aparegui a la pantalla la frase "Vivo! Error!".

<u>Joc demo_signal_stress</u>

A cada iteració un fill. El fill es queda en un bucle infinit i immediatament el pare el mata. Si el SIGKILL no funcionés bé, ja fos perquè no capta bé el signal, com perquè no fa bé el close del procés que mor, arribaríem a omplir tots els pid's y apareixeria per pantalla "Error forking", cosa que tal com dèiem no ha de passar (tal com es creen es maten).

Joc de proves waitpid

L'objectiu del joc de prova és que un procés pare esperi fins que un fill seu mori (-1 vol dir qualsevol). El joc té dos casos que comproven la correcta implementació de ZeOS. En un cas el pare es bloqueja esperant la mor del fill que, en morir (exit) desbloquejarà el pare que espera tornant-li el seu pid. L'altre cas és quan el fill mor abans que el pare, cosa que provoca que el fill hagi de guardar informació sobre la seva mort per a que quan es cridi a waitpid aquesta serveixi la informació al moment d'un fill que ja no existeix (el seu PCB és lliure). Està implementat amb bucles d'espera per a garantir quin a instrucció exit/waitpid s'executa primer.

Extensió de ZeOS

Opcionalment hem estès ZeOS per afegir noves funcionalitats pròpies d'un sistema compatible amb POSIX. Tot el que s'ha afegit s'ha fet intentant mantenir la pràctica compatible amb l'enunciat original (i, per tant, compatible amb els jocs de prova proporcionats).

waitpid: esperar a un procés

La funció waitpid ha estat simplificada per a no tornar el codi d'estat del procés que ha mort. Així no es requereix que exit prengui cap paràmetre i és compatible amb la versió existent. A més els PCBs son alliberats instantàniament al cridar a exit, ja que si no els jocs de prova que no facin servir de waitpid ompliren tots els PCBs amb processos morts i zombis. S'ha creat una taula de PIDs de processos morts amb els seus PPIDs per a poder implementar waitpid.

kill, signal y alarm: senyals entre processos

S'ha implementat el sistema de signals d'una forma lleugerament més simple. Tota senyal es processa a l'instant enlloc d'esperar a que el procés prengui la CPU o acabi l'anterior signal. Això no és el que fa Linux però ajuda a simplificar la implementació a la vegada que manté un nivell de compatibilitat molt alt.

També s'ha fet kill d'una forma molt més simple. No es poden fer servir les seves opcions esteses com ara enviar senyals a múltiples processos a la vegada, s'ha de fer una crida per procés que rebi un signal. Això ajuda a que el codi sigui una mica més simple, a més que kill fa ús dels permisos del procés i usuari, cosa no implementada a ZeOS.

El tractament dels signals que fem és mirar l'acció que ha programat el procés i contrastar-la amb el comportament obligatori (en algunes senyals no es pot variar el comportament). En cas de tractar el signal com una crida a un handler s'apila l'estat del procés a la pila d'usuari (de fet només alguns registres, els caller-save) i es modifica la pila per aconseguir que quan el handler acabi es continuï executant el codi de forma normal. Es simula de fet un "call handler_function_pointer". El dibuix explica com queda la pila.

Cim de la pila que usarà el handler								
EIP	EIP fals que apunta a signal_return_routine							
SIGNAL No	Número de signal provocat							
EAX								
ECX	Estat dels registres del procés en el moment de rebre el signal.							
EDX	- Signal.							
EFLAGS	Estat dels flags.							
EIP	Adreça de retorn que apunta a l'instrucció que hem interromput en el moment de rebre el signal.							
Cim de la pila d'usuari								

Tambié s'ha implementat SIGCONT i SIGSTOP que paren i continuen el procés per defecte. En cas de la crida alarm és el kernel que, fent servir el rellotge, envia signals (SIGALRM) als processos que així ho han demanat.

Per a implementar el retorn del handler (vegeu la pila) cal que el retorn (primer EIP) es faci a un tros de codi que s'encarregui de desapilar l'estat i retornar. Aquest codi és la funció signal_return_routine que està ubicada al codi de kernel. Està situat a una pàgina aïllat de la resta de codi que té permisos d'usuari (per a que l'usuari hi pugui saltar). El codi es troba a payload.S.

dup2 y pipe: comunicació entre processos

Es pot fer servir pipe per a crear dos descriptors de fitxer (lectura + escriptura) i crear així una pipe. La implementació es la mateixa que la de Linux. En caso d'escriure en una pipe sense lectors s'envia SIGPIPE i es torna EPIPE como a errno. En cas de llegir d'una pipe sense escriptors es torna 0.

En general les pipes compleixen POSIX a excepció de que no hi ha accés no bloquejant (tampoc hi és als fitxers).

Els processos que llegeixen o escriuen a una pipe es bloquegen a una cua en cas que no es pugui efectuar l'operació en aquell moment. Es proden desbloquejar en cas de poder satisfer l'operació o bé en cas de broken pipe.

Cal dir que l'especificació POSIX és molt ambigua (està plena de *may* i *shall*) cosa que ens ha portat a implementar-ho tot de forma senzilla.

La crida dup2 s'encarrega de clonar un descriptor de fitxer i de tancar el canal destí si estava obert, no té massa misteri.

Errors curiosos solucionats

Error fork()

Ens ha semblat interessant explicar un problema que ens vam trobar a la segona entrega i que vam trigar en esbrinar què passava.

Què succeïa:

Codi:

int k = fork();
write("surto del fork ")
while(1);

Sortida Esperada:	Sortida Observada
surto del fork surto del fork	surto del fork

Resulta curiós que el procés no donés cap problema i s'executés de forma normal. El problema després de debugguejar-lo amb molta un error en un índex d'un bucle, precisament el que mapejava les pàgines d'un nou procés i no es mapejaven totes les pàgines del procés. La pila del procés doncs, que es mapeja des del final de la memòria de dades del procés, era compartida. Per què, doncs no apareixia cap error, ni cap fallada de pàgina o segmentació?

Veiem que a priori hauria de passar. Dibuixarem l'estat de les 3 piles que ens ocupen en cada moment. La casella remarcada és la casella apuntada per l'eip.

Inicialment les 3 piles estan buides				Fork() de libc			
					%ebp		
					%eip		
	Pu	PT0	PT1		Pu	PT0	PT1
Face of facels do		T	I	Coming with sistems		T	
Fem el fork de sys_fork(), no copiem		SAVED REGS		Copiem pila sistema		SAVED REGS	SAVED REGS
encara pila de sistema						-	
		-				-	
		_				_	
		-				-	
	%ebp	_			%ebp	_	
	%ebp	_			%ebp	-	
	Pu	PT0	PT1		Pu	PT0	PT1
	l u	110	111		l u	110	111
Tornem a usuari, just abans de fer el return de libc		SAVE D REGS	SAVE D REGS	Al fer write apilem 3 paràmetres a pila de usuari.		SAVE D REGS	SAVES REGS
		_			%ebp		
		_			%eip		
		_			par2		
	%ebp	_			par2		
	%eip	_			par1		
	Pu	PT0	PT1		Pu	PT0	PT1
Al fer el task_switch i tornar a usuari la		SAVED REGS	SAVED REGS	Veiem que l'eip al que s	saltarà c	erà la di	irecció
posició de la pila just		KEGS	REGS	del paràmetre par2 (bu	uffer). Llavors, hauria de		
abans de fer el return és la següent:				produir-se un error. Què passa doncs?			
es la seguent.	%ebp			Que pussa dones:			
	%eip	-					
	par3						
	par2	-					
	par1						
	Pu	PT0	PT1				

Passa que el compilador fa una optimització probablement per alinear els paràmetres a 16 bytes per no produir accessos segmentats a memòria. El codi següent mostra com el compilador reserva aquest espai tot i que la crida no usi paràmetres. Resulta doncs que quan es torna del fork, s'apunta justament a la direcció de retorn que hi deixa el write, de forma que la direcció de retorn és correcta (vàlida, si més no) i no es produeix cap error, però saltem directament a la direcció del jmp (while(1)) així que només s'imprimeix un missatge, i no dos com hauria de ser.

100000: 100001: 100003: 100006: 100009: 10000e: 100015: 100016:	55 89 e5 83 e4 f0 83 ec 10 e8 22 00 00 00 c7 44 24 08 04 00 00 00 c7 44 24 04 90 80 10 00	push mov and sub call movl	<pre>%ebp %esp,%ebp \$0xfffffff0,%esp \$0x10,%esp 100030 <fork> \$0x4,0x8(%esp) \$0x108090,0x4(%esp)</fork></pre>
10001e:	c7 04 24 01 00 00 00	movl	<pre>\$0x1,(%esp) 100170 <write> 10002a <main+0x2a></main+0x2a></write></pre>
100025:	e8 46 01 00 00	call	
10002a:	eb fe	jmp	

Inicialment les 3 piles estan buides	Pu	PTO	PT1	Fork() de libc, abans de fer la crida a sys_Fork()	%ebp %eip %ebp Pu	PTO	PT1
Tornem a usuari		SAVED	SAVED	Estat de la pila al cridar a write		SAVED	SAVED
després de la crida a sys_fork(), just abans	%ebp	REGS	REGS	Cridar a Write	%ebp	REGS	REGS
de fer el return de libc	%eip				%eip		
					par3		
					par2		
					par1		
	%ebp				%ebp		
	Pu	PT0	PT1		Pu	PT0	PT1

				,1			
Al tornar al main després de fer el		SAVED	SAVED SAVED REGS REGS	Al fer el task_switch i tornar a usuari la posició de la pila just abans de fer el return és la següent:		SAVED REGS	SAVED REGS
write, queden valor	%ebp	REGS			%ebp	REGS	
residuals a la pila	%eip				/ocip		
	par3				par3		
	par2			Els paràmetres de la	par2		
	par1			pila son residuals: han quedat a la pila al fer	par1		
				el write i a causa a la optimització del compilador, casualment els			
	%ebp				%ebp		
	Pu	PT0	PT1		Pu	PT0	PT1
				registres casen de forma que no salta cap error de pàgina tot i que l'execució no és correcta.			

Insuficients pàgines de codi

El codi que hem fet no compila bé (no linka de fet) en cas de linkar amb libjp2.a, ja que la quantitat de codi és massa gran. En aquest cas, donat que les adreces de codi i de dades són fixades ens indica que les dues seccions es solapen. És per això que entreguem el codi dels JP comentat.