#include <stdio.h>

#include <stdlib.h>

int a; //(1)

double b; //(1)

int c=10; //(2)

char \*ch="Hola mundo"; //(2)

char d; //(1)

char e; //(1)

char h='a'; //(2)

f(int arg){

int a; double b;

printf("f: &a: %p, &b: %p, &arg: %p\n",&a,&b,&arg);

g(arg);

return 0;

}

g(int arg){

int a; double b;

printf("g: &a: %p, &b: %p, &arg: %p\n",&a,&b,&arg);

return 0;

}

main(){

int \*m1, \*m2;

m1 = malloc(sizeof(int));

m2 = malloc(sizeof(int));

printf("Main: main: %p, f: %p g: %p, printf: %p\n", main, f, g, printf);

printf("Main: &a: %p, &b: %p\n",&a,&b);

printf("Main: m1: %p,m2: %p\n",m1,m2);

printf("Main: &c: %p, &ch: %p, ch: %p\n", &c, &ch, ch);

printf("Main: &d: %p, &e: %p, &h: %p\n", &d, &e, &h);

f(10);

g(20);

}

(1)-> Se guardan en memoria a una distancia cercana entre si

(2)-> Al ser valores inicializados, se guardan en memoria a una distancia cercana entre si

**Memoria:**

---------

funciones

---------

malloc

---------

c,d,e

---------

Hola mundo \0 (en ASCII, carácter juntos con diferencia de un bit)

---------

main

---------

g

---------

f

---------

printf

**Distribución de la memoria:**

Segmento de texto: contiene código máquina para las funciones y suele estar en las direcciones más bajas de memoria. En x86\_64, en la configuración más usual el segmento de código comienza en 0x40000. Todos los procesos (salvo el SO) trabajan con una memoria virtual: disponen de un rango determinado de direcciones a la que pueden acceder como si fuera una gran memoria continua. Luego del seg de texto vienen segmentos de datos

Segmentos de datos: x86\_64 usa datos inicializados y datos sin inicializar. Luego hay memoria dinámica (Ej. malloc), a este segmento (que se extiende a medida que pido memoria) se lo llama heap (montículo)

Al tope del espacio de direcciones está el segmento de pila. La pila mantiene información sobre las llamadas a función y usualmente crece hacia direcciones más bajas (push) \*Mirar modelo de memoria en apunte transp\_arq\_x86\_64\*

**(Complementario a la transparencia Arquitectura x86\_64 y Apunte 3)**

Registros: se usan para fines específicos, no es obligatorio pero el uso habitual se relación con determinadas funcionalidades. Hay instrucciones que usan implícitamente un registro en particular.

* rax: Acumulador, operaciones aritméticas y lógicas, cálculos generales
* rbx: Puntero a la base de una estructura de datos complejas (struct de C, un arreglo, etc…)
* rcx: Contador (hay instrucciones de loops, bluces)
* rdx: Datos, puede usarse como extensión de rax

Registros de segmentos: tienen que ver con la administración de la memoria, en la configuración habitual casi son todos vacíos

* rip: Puntero a la próxima instrucción
* rflgas: Banderas y control

*Control*: determinados bits permiten controlar determinadas funciones del procesador (ej: modos de protección, incremento y decremento automático de punteros)

*Banderas*: indican información sobre el procesador y en particular, sobre el resultado de la última operación

Subregistros: (mirar transparencia)

Más registros:

* rbp: base pointer, puntero a la base de la pila (usado en llamadas a función)
* rsp: stack pointer, puntero al último elemento de la pila (hay instrucciones que usan implícitamente este registro)
* rsi: source index, puntero a un dato fuente
* rdi: destination index, puntero a un dato destino. Hay instrucciones (de cadena) que usan estos registros
* Little-endian y big-endian: Intel (desde el 8080) usa arq "little endian": los bytes menos significativos se guardan en las zonas(posiciones) de memoria más bajas
* Otras arquitecturas funcionan a la inversa, llamada "big endian" y hay incluso algunas que utilizan ordenamientos más complicados. Otras permiten elegir la convención utilizada
* x86\_64 permite tomar un solo argumento en las operaciones de la memoria (es más rápido el acceso a registro)
* Ciclos de instrucción: fetch (acceso a memoria para leer la instrucción), decode y execute (acceso a memoria para traer o almacenar datos)

.data

s: .asciz "Hola mundo\n";

.text

.global main

main:

movq s, %rax

ret

// #gcc -g [NAME].s -p [NAME]

// print &s (imprime dir de memoria)

// x/s [DIR] (imprime lo que hay en esa dir como string)

**Factorial iterativa**

int fact(ind d){

int res;

for(res=1; d>0, res\*= d)

return res;

}

Todas las funciones recursivas de cola pueden convertirse en iteraciones. Una función es recursiva de cola (tail recursive) si:

- devuelve una expresión simple

- devuelve invocaciones a si misma pero sin operaciones posteriores a la invocación

Ej:

int fact(int d)

{

if(d==0) return 1;

else return d\*fact(d-1)

}

No es recursiva de cola ya que tiene operaciones posteriores a la invocación

Una forma de escribir fact para que sea tail recursive es:

inf factr(int d, int res)

{

if (d==0) return 1

else return factr(d-1, res\*d)

}

fact(n) = factr(n, 1), donde fact(n) es una recursive tail

Como convertimos factr en una iteración:

inf facti (int d, int res){

while(d!=0)

{

res\*=d;

d--;

}

return res;

(n) = facti(n, 1), donde facti(n) es recursive tail

.data

i: .long 0

.text

calculo\_muy\_complicado:

movl i, %eax

imull $2, %eax

movl %eax, i

jmp vuelve

.global main

main:

movq $0, i # Inicializa i = 0

cond : cmpq $1000, i

jge fin # i >= 1000

jmp calculo\_muy\_complicado

vuelve : incq i # i++

jmp cond

fin : ret

\*/

Para poder separar la función en otro modulo (archivo) hay que definir como global los símbolos que serán accedidos desde otro modulo (archivo)

/\*

#Como retornar a la funciona llamante en el lugar correcto

.data

i: .long 0

.text

calculo\_muy\_complicado:

movl i, %eax

imull $2, %eax

movl %eax, i

jmp \*%rbx

.global main

main:

cmpq $1000, i

jge fin

movq $sigue, %rbx

jmp calculo\_muy\_complicado

sigue: incq i

imp main

fin: movq $sigue2, %rba

jmp calculo\_muy\_complicado

sigue2: ret

cond: cmpq $1000, i

jge fin # i >= 1000

movq $sigue, %rbx

jmp calculo\_muy\_complicado

vuelve: incq i # i++

jmp cond

fin: jmp calculo\_muy\_complicado

sigue2: ret

Para implementar una forma más general de funciones (con funciones recursivas) y variables automáticas se usa la pila. La pila en x86\_64 crece hacia las dir de memoria más bajas. Se mantiene un puntero al tope de la pila el cual puede ser el próximo lugar vacío o el último elemento que se ingresó (facilita el push & pop). x86\_64 utiliza un puntero al último lugar utilizado (%rsp)

El pop involucra la actualización del puntero, así, el pop completo será:

movq (%rsp), destino

addq $8, %rsp

push entonces será:

add $8, %rsp

movq origen, (%rsp)

Assambler provee las instrucciones push y pop que son equivalentes a las antes vistas. En x86\_64, todos los accesos al stack deben estar alineados en 8 bytes. Un push de un long (también de un unsigned long y de un puntero) de C sera entonces:

pushq long #usa rsp

pushq %rax

¿Cómo guardar un char? Convertirlo a 64 bits utilizando las instrucciones para convertir a tipos enteros más grandes

*Instrucciones de conversión* (enteras):

- movslq orig, dest # convierte un .long en orig a un .quad extendiendo el signo

- movsbq

- movswq # ¿Existe?

Las instrucciones anteriores convierten un quad a word extendiendo el signo

Otra opción (para los unsigned) es movz

*Extensiones* (rd:rax)

- cltq # %eax -(sig ext)-> %rax

- cqto # %eax -(sig ext)-> %rdx:%rax

- cbw # al -> ax

- cbde # ax -> eax

- cwd # ax -> dx:ax

- cdq # eax -> edx:eax

Volviendo a las llamadas a funciones, la llamada en si es un salto pero antes hay que informar (a la función llamada) la dirección del retorno (la dir de la siguiente instrucción a ejecutar).

Ej:

movq $sigue, %rbx

jmp func

sigue:

x86\_64 usa el stack para guardar la dirección de retorno, o sea: en lugar del "movq $sigue, %rbx" hace "pushq $sigue"

Retornar de una función equivale entonces a:

jmp (%rsp) # y pop

popq %rbx

jmp \*%rbx # esto pierde un registro también

Para evitar usar registro, assambler provee una inst que hace el pop y el salto (sin usar registros adicionales) que es "ret", o sea, “ret” equivale a hacer “popq” y luego “jmp”

La llamada a una función:

call func = pushq $prox\_inst

jmp func

Una función a llamada, entonces, es "call func" y para retornar "ret"

Ej:

calculo\_muy\_complicado

movl i, %eax

imul $2, %eax

ret

main:

movl $0, i

sigue:

cmpl $1000, i

jge

call calculo\_muy\_complicado

incl i

jmp sigue

fin:

call calculo\_muy\_complicado

ret

**¿Cómo se pasan argumentos y se devuelven valores?**

Se usan convenciones de llamadas: todos los programas en C en linux, saben que determinados registros y el stack se usan para pasar argumentos.

Los seis primeros argumentos (si son todos de tipo entero) van a parar a los registros rdi, rsi, rdx, rcx, r8, r9 y el resto ira al stack (salvo flotantes), de derecha a izquierda. El valor de retorno se guarda en rax

**Solución a jmp en diferentes lugares:**

.

.

.

movq $sigue, %rbx

jmp calculo\_muy\_complicado

sigue: incq i

.

.

.

movq $sigue2, %rbx

jmp calculo\_muy\_complicado

sigue2: ret

#include <stdio.h>

main(){

char a=28;

printf("%d %d\n",printf("1\n"),printf("2\n"));

printf("%d %d\n",a=a+1,a=a+1);

}

/\*

Resultado:

$2

$1

$2 2 (Retorno de printf, cant de arg)

$30 29

Los argumentos se evalúan de derecha a izquierda

\*/

void f(int a, char b) {return ;}

int a(unsigned a1, char a2, unsigned char a3,

long a4, char \*a5, void (\*a6)(int, char), int a7, char a8, unsigned char a9) {

return 10;

}

unsigned b1=0;

char b2=128;

unsigned char b3 = 128;

long b4=0xaaaabbbbccccddddL;

char \*b5="hola mundo\n";

int b7 = 0xffffeeee;

char b8= 128;

unsigned char b9=128;

main() {

return a(b1,b2,b3,b4,b5,f,b7,b8,b9);

}

/\*

Por convención, cada arg tiene que ir separados en distintos registros

unsigned a1 -> rdi

char a2 -> rsi

unsigned char a3 -> rdx

long a4 -> rcx

char \*a5 -> r8

void (\*a6)(int, char) -> r9

Los sgtes arg van al stack pero del ultimo al primero (der a izq)

int a7 -> pusq (3) (rsp+24)

char a8 -> pusq (2) (rsp+16)

unsigned char a9 -> pusq (1) (rsp+8)

Según el argumento sea signed o unsigned se usara movzbl o movsbl

\*/

.text

.global main

main:

movq $0, %rax # Val tipo ent se devuelve en rax

ret # main es una funcion

/\*

int main(int argc, char \*\*argv)

{

return 0;

}

Usar echo $? para ver el retorno

\*/

.data

s: .asciz "Hola\n"

.text

.global main

main:

movq $0, %rax # En rax debo poner la cant de arg en pto flotante por la func printf

movq $s, %rdi # Con $ guarda en rdi la dir / Sin el peso en rdi estaría la "H"

call printf # Definida en la librería C

movq $0, %rax # Es necesario ya que printf seguramente modifique rax

ret

/\*

int main()

{

printf("Hola");

return 0;

}

\*/

.data

s: .asciz "%ld\n"

.text

.global main

main:

movq $0, rax # En rax debo poner la cant de arg en pto flotante por la func printf

movq $s, %rdi

movq $3, rsi

call printf # Definida en la librería C

movq $0, rax # Es necesario ya que printf seguramente modifique rax

ret

/\*

int main()

{

printf("%ld\n", 3);

return 0;

}

\*/

# Posible implementación del siguiente programa

# f(long a1, long a2, long a3, long a4, long a5, long a6, long a7, long a8) {

# return a8;

# }

#

# main() {

# return f(1,2,3,4,5,6,7,8);

# }

.text

f: # argumentos en rdi,rci,rdx,rcx,r8,r9 + stack (push 8, push 7)

movq 16(%rsp), %rax # return a8

ret

.global main

main:

#Cargar argumentos que van al stack

cargar\_pila: # Los demas arg van al stack de derecha a izq

pushq $8 # 1 oper arit + 1 mov

pushq $7 # 1 oper arit + 1 mov

# O (lo que es lo mismo), más eficiente

# subq $16, %rsp # 1 oper arit

# movq $8, 8(%rsp) # 1 mov

# movq $7, (%rsp) # 1 mov

cargar\_regs: # Los 6 primeros arg van a registros de izq a derecha

movl $6, %r9d # Al usar movl, solo estoy usando la parte de baja (32 bits)

movl $5, %r8d # movl me rellena la parte alta con 0

movl $4, %ecx

movl $3, %edx

movl $2, %esi

movl $1, %edi

callf:

call f

deshace\_pila: # Por los dos pushq en la línea 22 (7to y 8vo arg)

addq $16, %rsp # Más eficiente, no uso un reg auxiliar

#O (lo que es casi lo mismo)

#popq %rcx

#popq %rcx

retorna:

ret

/\*

- En la línea 27, se podría hacer movq $8, -8(%rsp); movq $7, -16(%rsp); subq $16, %rsp pero si se produce una interrupción, al haber dejado para lo último el aumento del rsp, esos dos elementos se podrían perder ya que no estaban dentro de mi stack

\*/

- La función llamante es la que se encarga de preparar los argumentos y también de liberar el espacio usado por esta en el stack.

- En general, los valores que han sido desapilados es porque no se usaran más y por lo tanto no se puede garantizar que sigan estando ahí ya que el SO u otras funciones podrían usar el stack para sus tareas.

- En x86\_64, el stack tiene una zona roja (128 bits) debajo del valor actual del rsp que una función puede utilizar como almacenamiento temporario (siempre que no llame a otras funciones o así misma)

main:

pushq %rbp # Meto base pointer en el stack

movq %rsp, %rbp # Igual el stack pointer con el base pointer

subq $16, %rsp # Reservo espacio en el stack para los dos arg

movq $8, 8(%rsp) # Argumentos que van al stack

movq $7, (%rsp)

movl $6, %r9d # Los primeros 6 arg van en registros

movl $5, %r8d

movl $4, %ecx

movl $3, %edx

movl $2, %esi

movl $1, %edi

call f

leave

ret

f:

pushq %rbp # Guardo el rbp para poder usarlo (para poder moverme por el stack) #y no perderlo

movq %rsp, %rbp

movq %rdi, -8(%rbp) # Guardo los registros en el stack

movq %rsi, -16(%rbp)

movq %rdx, -24(%rbp)

movq %rcx, -32(%rbp)

movq %r8, -40(%rbp)

movq %r9, -48(%rbp)

movq 24(%rbp), %rax # Guardo en rax el 8vo arg

popq %rbp # Restauro el valor original de rbp

ret # Equivale a movq %rbp, %rsp; popq %rbp

/\*

Estado del stack

ret adress main

rbp (como la recibió main)

8

7

rbp (como la recibió f)

rdi

rsi

rdx

rcx

r8

r9

\*/

**Convención de uso de registros:**

Hay un conjunto de registros que una función puede usar sin preocuparse de restaurar su valor. Estos registros no son preservados cuando se llama a una función y por lo tanto la función llamante debe estar preparada ya que posiblemente su valor cambie cuando realice un llamado a otra función

Si hay un valor útil en estos registros, conviene que la función llamante lo guarde en otro lado

Ej:

addq %rbx, %rax

pushq %rax # Guardar rax

call printf # rax = nro de chars

popq %rax # Restaurar rax

addq %rcx, %rax

A ese conjunto de registros (preservados por el llamante) se los llama "caller save"

El otro conjunto de registros (que debe ser preservado por la función llamada) se llama "callee save" (Ej: rsp rbp)

Ej:

caller save

rax, si necesito llamar a una función debo guardarlos

De argumentos: rdi, rsi, rdx, rcv, r8, r9

callee save

rbx, rbp, rsp, r10, r13, r14, r15

Esta convención está determinada por C en x86\_64 en Linux, por lo que dicha convención puede cambiar en otras arquitecturas y/o lenguajes (Ej: Pascal)

**Strcpy, implementación:**

.data

dst: .quad 0

src: .quad 0

dummy: .quad 0

len: .quad 0

.text

.global strcpy2

.strcpy2:

cmpq $0, %rcx

je fin

bucle:

lodsb

addq $1, %rdi

addq $1, %rsi

loop bucle

fin:

xorq %rax, %rax

ret

/\*

int strcpy2(char \*dst(rdi), char \*src(rsi), int dummy(rdx), int len(rcx))

-La variable dummy es para que long entre en rcx

--------------

La instrucción loop tiene dos efectos:

-Decrementa en uno el registro rcx

-Luego, salta a la etiqueta solo si el resultado de decrementar rcx dio distinto de cero. Si el resultado dio cero, el flujo del programa sigue en la siguiente instrucción al loop.

----------------

movs: Copia el dato (%rsi) (puntero) en (%rdi) e incrementa/decrementa ambos registros en la cantidad del tamaño del dato

movsl = movl (%rsi), %rax

movl %rax, (%rdi)

addq $4, %rdi

addq $4, %rsi

------------------------

Las instrucciones de cadena incrementan/decrementan los registros rsi, rdi dependiendo de la DF del registro flags, donde aumentara si DF=0 o drecrementara si DF=1

Esta bandera se modifica con

cld -> Clear direction -> DF=0

std -> Set direction -> DF=1

La DF es un "recurso" callee saved (debe ser guardado) y por convención de llamada, es 0 al entrar a una función

\*/

La anterior implementación de strcpy usa una variable len para evitar un desbordamiento de buffer

# http://x86.renejeschke.de/ -> pagina de instrucciones assembler

**Punto flotante en x86\_64:**

Tipos de datos float (en C):

- float equivale a los flotantes de simple precision de IEEE754 (32 bits)

- double equivale a doble precision de IEEE754 (64 bits)

- long double (16 bytes) IEEE754 formato extendido de 80 bits

- \_\_float128 (16 bytes) IEEE754 formato extendido de 128 bits, las operaciones son simuladas por software

-------------------------------------------

Ejemplo ejercicio:

3.14

Bit de signo: 0

En binario: 11.00100011 Parte decimal -> convertir a binario

Parte fraccionaria -> multiplicar x2 e ir guardando la parte entera

Normalizado: 1.100100011 x 2¹(^1)

Exponente: 1

Fraccion: 100100011

Pasaje a float:

Sumar 127 al exponente y quedarse con los 23 bits de la fraccion

|0|10000000|100100011|

Exponente: 128 -> 10000000

-> 0x40

SSE3: sub-unidad unidad de punto flotante disponible en todos los procesadores de x86\_64

Movimiento:

movss (%rax), %xmm7

# Copiara lo que esta apuntado por rax (32 bits) a xmm7

# Tambien permite copiar de reg a reg (flotante ambos)

Convertir:

cvts -> convert (conversiones enteras)

cvtts -> convertir con truncamiento

Convención de llamada:

- Los argumentos flotantes (o dobles) van en xmm0 a xmm7 (el primer argumento flotante en xmm0).Si hay más de 8, el resto va en el stack.

- Si la función retorna un flotante (o doble) el resultado queda en xmm0.

- Todos los registros (xmm0 a xmm7) son caller-saves

Ej:

En C:

double convert (double t){

ret t\*(1.8)+32;

}

double => retorna en xmm0

double t => 1er arg en xmm0

En assembler:

.global convert

convert: # en xmm0 viene t

# Carga el valor 1.8 en xmm1

# La constante es la representacion

# segun IEEE 754 de 1.8

movabsq $4610785298501913805, %rax # El 'abs' es necesario por el largo de la constante

movq %rax, -8(%rsp) # La red zone me permite hacerlo

movsd -8(%rsp), %xmm1

# Carga el valor 32.0 convirtiendo

# el valor entero 32 de rax a xmm2

movq $32, %rax # Cargo el 32 como un entero

cvtsi2sdq %rax, %xmm2 # Lo convierto a double

# xmm0=xmm0\*xmm1 => xmm0=t\*1.8

mulsd %xmm1, %xmm0

# xmmo=xmm0+xmm2 => xmmo=t\*1.8+32

addsd %xmm2, %xmm0

# como el valor de retorno se

# escribe en xmm0 hemos terminado

ret

Ej2:

void sum(float a[4], float b[4]) {

int i;

for(i=0;i<4;i++)

a[i]=a[i]+b[i];

}

.global sum

sum:

# Notar que en este caso los argumentos

# son punteros y vienen en rdi y rsi respectivamente

# copia los 4 floats de "a" a xmm0

movaps (%rdi), %xmm0 # Copia 128 bits a 'a'

# copia los 4 floats de "b" a xmm1

movaps (%rsi), %xmm1 # b -> xmm1

# suma los 4 floats a la vez

addps %xmm0, %xmm1 # xmm1 = a+b

# addps suma los 4 valores de precision simple a la vez

# guarda el resultado en "a"

movaps %xmm1, (%rdi) #

ret

**Corrutinas:**

Usaremos como ejemplo dos funciones que deben avanzar en sus cálculos "simultáneamente". Un ejemplo es: ls | grep "exp" (busca la expresión "exp" en la carpeta). La idea es que ft1 y ft2 se vayan turnando para ir realizando sus cálculos.

Para esto podríamos usar jump/ goto (con etiquetas) pero hay un problema, C no permite saltar a etiquetas de otras funciones y si lo intentáramos hacer embebiendo un jmp tampoco serviría ya que el jmp no cambia el rsp y el rbp, con lo que los accesos a variables automáticas, temporales, return adress y argumentos serán erróneos.

Otra posibilidad es usar llamadas a funciones (call y ret en assembler) pero ft2 no continuaría se ejecución anterior, por lo que ft2 nunca progresaría ya que estaría siempre empezando desde el comienzo (valga la redundancia)

La solución es usar setjmp y longjmp: la idea es que ft1 avance un poco, luego hago un setjmp para guarda el estado y dsp hago un longjmp para que tome el control ft2 para que esta avance un poco, luego hago un setjmp para guarda el progreso y dsp hago un longjmp para que tome el control ft1, y así es el ciclo.

Luego el TRANSFER (que permite ceder la ejecución temporalmente a otra tarea - en C serán funciones) será entonces

TRANSFER(o, d) equivale a ((setjmp(0) == 0 ? (longjmp(d, 1), 0) : 0 )) donde o y d son jumpbuffer

void ft1() // t1

{

double d;

for(d=-1;;d+=0.001)

{

printf("%f\n", d);

TRANSFER(t1, t2);

}

}

void ft2() // t2

{

int i;

for(i=i; i<10000; i++)

{

printf("%d\n", i);

TRANSFER(t2, t1);

}

TRANSFER(t1, taskmain);

}

Antes de usar TRANSFER, obligatoriamente tengo que haber usado setjmp, podríamos hacer en el main:

#if (setjmp(t1) != 0) ft1;

if (setjmp(t2) != 0)

ft2;

ft1();

...

TRANSFER(t1, t2);

Pero todavía seguimos teniendo otro problema, el marco de act de ft1 podría pisar el marco de act de ft2. Para evitar esto, podemos dejar un espacio entre el fin de marco de act de ft1 y el comienzo del marco de ft2

¿Qué ocurre si tengo que organizar un nro. grande de corrutinas?, tengo que organizar los TRANSFER para que todas las corrutinas puedan avanzar. Una secuencia posible seria A->B; B->C; C->A y de esta manera formar un ciclo.

¿Cómo puedo hacer para que A avance más rápido que las otras tareas?, podríamos hacer:

void fA(){

double d;

for(d=-1;;d\*=0.001){

printf("d=%f\n", d);

if(((int)d\*1000)%2)

TRANSFER(A, B);

}

}

Gestionar la lógica de los TRANSFER en cada corrutina se hace complicado. La idea es hacer una corrutina que centralice la lógica de los TRANSFER. De esta forma, la administración de la ejecución de las tareas se hace en un solo punto del programa.

Las corrutinas, cuando deciden ceder la ejecución harán YIELD para que la lógica centralizada (planificador o scheduler) pueda decidir que corrutina seguirá ejecutando.

Así, ft1() quedara:

void ft1(){

double d;

for(d=-1;;d\*=0.001){

printf("d=%f\n", d);

YIELD(t1); // Hara un longjmp al planificador

}

}

El planificador será:

void schedfunc(){

for(;;){

ACTIVATE(t1);

ACTIVATE(t2);

}

}

En donde, ACTIVATE y YIELD se definirán como:

# define ACTIVATE(d) TRANSFER(schedt, d) // Tarea asociada a schedfunc ; d <- destino

# define YIELD(o) TRANSFER(o, schedt) // o <- origen

Si queremos darle más tiempo de ejecución a t1, podemos hacer:

for(;;){

ACTIVATE(t1);

ACTIVATE(t1);

ACTIVATE(t2);

ACTIVATE(t3);

}

Obs: Ahora, agregar una nueva tarea o darle más tiempo a alguna tarea es fácil, solo hay que modificar el planificador. El código del main será:

task t1, t2, taskmain, schedt

... (codigo de las t1, t2, ...)

main(){

stack(schedt, schedfunc);

stack(t1, ft1);

stack(t2, ft2);

YIELD(taskmain);

return 0;

}

Este esquema de trabajo colaborativo se utilizó en algunos sistemas operativos como en Windows 3.1 (multitarea cooperativa). ¿Qué problema puede traer?

Ej:

for(;;); // Un programa mal hecho o malicioso podría arruinar toda la multitarea

Los sistemas operativos modernos usan un modo de multitarea apropiativa, o sea, un proceso ejecuta algunas instrucciones y luego el SO cambia a otros procesos (sin que el proceso inicial lo sepa). Cuando se retome la ejecución habrá que restaurar TODOS los registros (no solo las callee saves).

Usaremos un mecanismo de señales (que provee el SO) para simular las interrupciones del reloj (que usa un SO real). Una señal es un evento asíncrono que puede recibir un proceso y a determinados eventos se les puede asociar una acción. Un ejemplo de esto es el ^C (Ctrl+C) que normalmente interrumpe un proceso. Lo que hace el SO es enviar una señal SIGINT, la cual podemos capturar e ignorar.

Ej:

#include <signal.h>

main(){

signal(SIGINT, SIG\_IGN);

for(;;);

}

Hay acciones predefinidas:

SIG\_IGN <- ignorar la señal

SIG\_DFL <- restaurar el comportamiento predeterminado

Signal nos permite asociar una acción a determinada señal (evento). La función signal se solía definir asi:

void (\*signal(int sig, void (\*pf)(int))(int));

// Signal es una función que toma un entero sig (nro de señal) y pt (un puntero a función que toma un entero y devuelve void) y devuelve una función que toma un entero y devuelve void

Uso: signal(nro\_de\_señal, accion) -(retorno)-> void (x)(int), acción asociada anteriormente a la función (por si hay que dejar las cosas como estaban)

Ej:

#include <stdio.h>

#include <signal.h>

void handler(int n){

printf("Auch\n");

}

main(){

signal(SIGINT, handler); // Cuando se reciba la señal SIGINT (C^), se ejecutara la funcion handler

for(;;);

}

Obs: SIGKILL no puede ser capturada, por lo que siempre funcionara para poder interrumpir programas

Podemos hacer un programa que soporte tres ^C y al cuarto sea interrumpido, solo hay que modificar el handler:

void handler(int n){

static int i=0;

printf("Sig: %d, van %d veces\n", n, i+1);

if(++i>3)

signal(n, SIG\_DFL);

}

La función kill(int pid, int signal) envía la señal signal al proceso pid. La función raise(int signal) equivale a kill(int pid, int signal)

Ej:

#include <stdio.h>

#include <signal.h>

void handler(int n){

static int i=0;

printf("\nSig: %d, van %d veces\n", n, i+1);

if(++i>3)

signal(n, SIG\_DFL);

}

main(){

signal(SIGINT, handler);

kill(getpid(), SIGINT);

//raise(SIGINT);

for(;;);

}

Luego de ejecutar el handler, el proceso continúa su ejecución normal. Si en el handler ponemos un YIELD, podríamos hacer cambios transparentes pero no queremos que el usuario tenga que hacer ^C o enviar señales con el comando kill. Usaremos (en vez de C^) la señal de un temporalizador: SIGALRM, esta llega cuando se cumple el tiempo especificado en la última llamada a alarm. Por lo tanto, alarm(5) enviara al proceso actual una señal SIGALRM dentro de 5 segundos. Otra señal es SIGSEGV, que está asociada a fallos de segmentación. (Ejercicio: hacer un programa que detecte un segfault (y solucionarlo si es posible))

Para hacer que las corrutinas cambien automáticamente, puedo poner alarmas y en la acción asociada hacer YIELD.

Ej:

#include <stdio.h>

#include <signal.h>

void alarma(int n){

printf("Alarma");

alarm(5);

}

main(){

signal(SIGALRM, alarma);

alarm(5);

for(;;){

printf("a\n");

sleep(1);

}

}

Ej:

#include <stdio.h>

#include <signal.h>

#include "guindows.h"

task t1, t2, taskmain, tsched;

static int i;

void fsched(){

for(;;){

if(i++%2) ACTIVATE(t1);

else ACTIVATE(t2);

}

}

void ft1(){

for(;;)

printf("Soy t1\n"); // YIELD no estan porque van a hacer interrumpidos por la alarma

}

void ft2(){

for(;;)

printf("Soy t2\n"); // YIELD no estan porque van a hacer interrumpidos por la alarma

}

void handler(int sig){

alarm(1);

if(i%2) YIELD(t2);

else YIELD(t1);

}

main(){

stack(t1, ft1);

stack(t2, ft2);

stack(t1, ft1);

stack(tsched, fsched);

signal(SIGALRM, handler);

alarm(1);

TRANSFER(taskmain, t1);

return 0;

}

Un SO moderno suele usar un mecanismo similar (basado en temporalizadores e interrupciones) para proveer multitarea. A esto se lo llama multitarea apropiativa (los procesos no se enteran ni tienen que hacer nada para convivir con otros procesos, por eso no incluyen el YIELD).

El SO cambia de tarea de prepo (preempt) y de forma transparente.

guindows.h

#ifndef \_\_GUINDOWS\_H

#define \_\_GUINDOWS\_H

#include <setjmp.h>

typedef jmp\_buf task;

#define TRANSFER(o,d) (setjmp(o)==0?(longjmp(d,1),0):0);

extern void stack(task, void (\*pf)());

#endif//\_\_GUINDOWS\_H

guindows.c

#include <stdlib.h>

#include <setjmp.h>

#define TPILA 4096

#define NPILAS 10 // Soporta hasta 10 corrutinas

static void hace\_stack(jmp\_buf buf, void (\*pf)(), unsigned prof, char \*dummy) {

if( dummy - (char \*) &prof >= prof) {

if (setjmp(buf)!=0) {

pf(); abort();

}

} else hace\_stack(buf, pf, prof, dummy);

}

void stack(jmp\_buf buf, void (\*pf)()) // Funcion para crear marcos de act desde el ult hacia el primero

{

static int ctas = NPILAS;

char dummy; // variable para calcular la profundidad

hace\_stack(buf, pf, TPILA \*ctas, &dummy);

ctas--; // Empiezo con el ultimo y voy subiendo

}

Corrutinas con ACTIVATE y YIELD.c

#include <stdio.h>

#include "guindows.h"

# define ACTIVATE(d) TRANSFER(schedt, d) // Tarea asociada a schedfunc ; d <- destino

# define YIELD(o) TRANSFER(o, schedt) // o <- origen

task t1, t2, t3, t4, taskmain, schedt;

void ft1(){

for(;;){

printf("Soy ft1");

YIELD(t1);

}

}

void ft2(){

for(;;){

printf("Soy ft2");

YIELD(t2);

}

}

void ft3(){

for(;;){

printf("Soy ft3");

YIELD(t3);

}

}

void ft4(){

for(;;){

printf("Soy ft4");

YIELD(t4);

}

}

void schedfunc(){

for(;;){

ACTIVATE(t1);

ACTIVATE(t2);

ACTIVATE(t3);

ACTIVATE(t3);

ACTIVATE(t4);

ACTIVATE(t4);

}

}

main(){

stack(schedt, schedfunc);

stack(t1, ft1);

stack(t2, ft2);

stack(t3, ft3);

stack(t4, ft4);

YIELD(taskmain);

return 0;

}

**Administración de memoria:**

1. Acceso a registros: una instrucción puede acceder a varios registros.
2. Memoria caché: es transparente, cuando se necesita un dato de la memoria se lo copia automáticamente a la memoria caché. La próxima vez que se requiera acceder a la misma posición de memoria, el dato se buscará en la caché.

*Problemas con la caché:*

Cuando se escribe un dato modificado en la caché de vuelta en la memoria se hace automáticamente pero aparecen problemas si se necesita acceder al mismo dato desde otro procesador ya que pueden aparecer problemas de inconsistencia.

Algunas arquitecturas permiten deshabilitar las cachés, sincronizar las cachés, vaciarlas, etc.

Cuello de botella de Von Neuman: surge por el hecho de que el procesador trabaja y envía más datos de los que la memoria principal puede procesar, por lo que se produce una ralentización. Las memorias cachés se utilizan para aliviar un poco este problema.

*Memoria principal (RAM): (mirar diapositiva)*

Opción 1:

Al estar llevando y trayendo datos de memoria a disco, el tardará mucho en completar la acción

Opción 2:

* + 1. No puedo tener dos copias de un mismo proceso
    2. Limito la memoria de cada proceso
    3. Para hacer un proceso nuevo debo reservar espacio
    4. Procesos terminados desperdician espacio

*Segmentación*:

Un programa es un conjunto de componentes lógicos de tamaño variable o un conjunto de segmentos, es decir, el espacio lógico de direcciones se considera como un conjunto de segmentos, cada uno definido por un identificador, y consistente de un punto de inicio y el tamaño asignado

La segmentación de un programa la realiza el compilador y en ella cada dirección lógica se expresará mediante dos valores:

1. Número de segmento (s)
2. Desplazamiento dentro del segmento (d)

Una de las implementaciones más obvias y directas de un espacio de memoria segmentado es asignar un segmento distinto a cada una de las secciones del espacio en memoria de un proceso.

*Tabla de segmentación*

T1|20|10k

T2|50000|10k

A todas las direcciones que use T1 se le suma 20 y a todas las direcciones que use T2 se le suma 50000. Los procesadores usan ahora direcciones virtuales y al sumar el desplazamiento correspondiente se obtiene la dirección real o física.

Uno de los problemas es la fragmentación externa: luego de un tiempo quedan espacios vacíos entre los segmentos y eventualmente habrá que reordenar los segmentos

**Idea**: dividir los programas en pequeñas partes o páginas. Del mismo modo, la memoria es dividida en trozos del mismo tamaño que las páginas llamados **marcos de página**. De esta forma, la cantidad de memoria desperdiciada por un proceso es el final de su última página, lo que minimiza la fragmentación interna y evita la externa.

**Ventajas de la segmentación:** Varias instancias del mismo proceso pueden compartir el mismo segmento de código (y controlar que no sea escrito). Lo mismo puede hacer con librerías compartidas (Ej: libc)

**Desventajas de la segmentación:** Fragmentación externa

**Paginación**:

Dir. virt Dir. reales

\_\_\_\_\_\_ \_\_\_\_\_\_

P1|\_\_\_\_\_\_| |\_\_\_\_\_\_| Paginas físicas

|\_\_\_\_\_\_| |\_\_\_\_\_\_| o marcos de mem física

|\_\_\_\_\_\_| |\_\_\_\_\_\_|

\_\_\_\_\_\_ |\_\_\_\_\_\_|

P2|\_\_\_\_\_\_| |\_\_\_\_\_\_|

|\_\_\_\_\_\_| |\_\_\_\_\_\_|

|\_\_\_\_\_\_| |\_\_\_\_\_\_|

Al tener las páginas una tamaño fijo, se sigue perdiendo espacio (en promedio, media página por proceso). Ya no hay fragmentación externa pero aparece la fragmentación interna

**¿De qué tamaño conviene elegir las páginas?**

Para minimizar la fragmentación debería usar páginas pequeñas, el problema es que mientras más pequeña es la página, más grande es la página. En la práctica se busca un equilibrio con lo cual, muchas arquitecturas terminan usando páginas de 4Kb

En la tabla de paginación también se suele incluir información de permisos, si los datos están en memoria o si la memoria no está asignada (es un hueco).

**¿Qué tamaño tendrá una tabla de paginación?**

8 bits -> 128 bytes |\_\_\_\_\_\_\_|\_\_\_\_\_\_|\_\_offset\_\_\_\_| 32 bits

10 bits -> 1K 10 10 10

12 bits -> 4 kB 20

Un proceso puede tener 220 páginas virtuales. Si cada entrada de la tabla de paginación contiene solo una dir física en RAM (32 bits), la tabla ocupará

220\*4 = 222 = 4Mb

Otra forma de implementar paginación, es utilizar una tabla de paginación invertida la cual tendrá tantas entradas como nro. de marcos físicos por lo que se ahorra bastante espacio

Un problema que tiene es la búsqueda de secuencias la cual se puede aliviar con el uso de un caché de traducción (aunque no se usa mucho)

**Tabla de paginación invertida:**

Es una técnica de paginación en la cual hay una entrada por cada página real (o frame) de la memoria y además incluye información del proceso que posee dicha página. Por lo tanto, en el sistema solo habrá una tabla de páginas y esta solo tendrá una entrada por cada frame en la memoria física. La principal ventaja de este método es que reduce la memoria física ocupada por la tabla de páginas. Pero por otra parte aumenta el tiempo de búsqueda de páginas ya que se debe explorar la tabla cada vez que hay una referencia a una página, debido a que esta se encuentra ordenada según la memoria física y las búsquedas se realizan según la memoria virtual. Para aliviar esto se utiliza una tabla de hash (Tablas de páginas hash) para limitar la búsqueda a una o (como mucho) algunas entradas de la tabla de páginas, pero cada acceso a una página requerirá dos accesos a memoria: uno para acceder a la tabla de hash y otro para la tabla de páginas (recordar que primero se explora un buffer TLB (Translation Lookaside Buffer) antes de consultar la tabla hash). Los sistemas que utilizan tablas de páginas invertidas tienen problemas para implementar el concepto de memoria compartida ya que cada entrada de la tabla de páginas corresponde a solo un frame en memoria.

**Registros relacionados a la paginación en i386:**

CR0 → Configuración

CR1

CR2

CR3 → 1er nivel de tabla de paginación

CR4 → Configuración

El Pentium permitía elegir páginas de 4Kb o 4 Mb

**Extensión de dirección física (PAE):** (Wikipedia)

Se refiere a una característica de los procesadores x86 que permite a los sistemas de 32-bit utilizar hasta 64 gigabytes (64 GiB) de memoria física, suponiendo que el sistema operativo proporcione el adecuado soporte

El hardware del procesador está ampliado con más líneas de direcciones utilizadas para seleccionar la memoria adicional, y tablas de páginas de 36 bit, pese a que las aplicaciones software normales continúan utilizando instrucciones con direcciones de 32 bit y un modelo de memoria plano limitado a 4 GiB. El sistema operativo utiliza PAE para mapear este espacio de direcciones de 32 bit en el total de 64 gigabytes de memoria, el mapa puede ser, y normalmente lo es, diferente para cada proceso. De esta manera la memoria adicional puede ser útil aunque las aplicaciones normales no puedan acceder a toda ella de forma simultánea.

**Estructuras de la tabla de páginas:**

En el tradicional modo protegido de 32 bit, los procesadores x86 utilizan un esquema de traducción de páginas de dos niveles, en el que el registro CR3 apunta a un directorio de páginas de 4K, que está dividido en 1024 entradas de 4 bytes que apuntan a una tabla de páginas de 4 KiB, que de forma similar está formada por 1024 entradas de 4 bytes que apuntan a las páginas de 4 KiB.

Activar PAE (al fijar el bit 5, PAE, del registro de control del sistema CR4) provoca importantes cambios en este esquema. Por defecto, el tamaño de cada página se mantiene en 4 KiB. Cada entrada en la tabla de páginas y en el directorio de páginas se extiende a 64 bits (8 bytes) en lugar de 32 para permitir los bits de direcciones adicionales. El tamaño de la tabla no cambia, sin embargo cada tabla ahora sólo contiene 512 entradas. Ya que se han reducido a un cuarto el número de entradas del esquema original, se añade un nuevo nivel en la jerarquía, de manera que CR3 ahora apunta al Page Directory Pointer Table, una pequeña tabla que contiene punteros a los directorios de 4 páginas.

De forma adicional, las entradas en el directorio de páginas tienen un nuevo flag, llamado 'PS' (Page Size). Si este bit (bit 7) tiene por valor 1, la entrada en el directorio de páginas no apunta a una tabla de páginas, sino a una sola página grande (de 2 MiB).

**Funciones anidadas:** (Mirar diapositiva “transparencia\_arq\_x86\_64\_funciones\_2”)

**Problema**: Si una función es recursiva, tendré varios marcos de activación y al no saber cuántas veces se llamó a sí misma, no puedo saber cuál es el marco de activación correcto.

**Posible solución**:

→ Tabla display: guardar al comienzo del marco de activación los punteros a la funciones de niveles superiores. A esto (más el rbp anterior) se lo llama “tabla (o área) display”

|  |
| --- |
| .  .  . |
| X -> | | | a |
|  | | | .  .  . |
| b |
| .  .  . |
| r.a |
| Z -> | rbp anterior |
| X |
| Y |
| Z |
| .  .  . |
| .  .  . |
| r.a |
| W -> | | rbp anterior |
| X |
| Y |
| Z |
| W |

|  |
| --- |
| p1 |

|  |
| --- |
| p2 |

|  |
| --- |
| p3 |

|  |
| --- |
| Tabla display nivel 3 |

|  |
| --- |
| p4 |

|  |
| --- |
| Tabla display nivel 3 |

|  |
| --- |
| Tabla display nivel 4 |

Y asi la tabla se va repitiendo si se agrega otro p

**Instrucción enter N, M**:

Crea una tabla display de nivel M (copia la tabla display del marco de activación anterior de M-1 niveles) y luego agrega en el stack el rbp actual.

N es el nro de bytes reservados para variables automáticas.

Obs:

1) enter N -equivale-> enter N, 0

2) enter N -equivale-> subq $N, %rsp

enter N = enter N, 0

enter N, M

3) Se puede mejorar usando "Static link"