Descripción

El presente documento es un borrador en el cual se describe la hoja de ruta del trabajo práctico “Simulación de Multitasking estático de un Sistema Operativo”.

El trabajo está dividido en tres/cuatro (3/4) entregas:

1. En la primera entrega, en base a un programa en assembler se debe generar un “ejecutable”
    a. Clases a implementar: Ensamblador, Ejecutable, clases asociadas a las instrucciones assembler (sólo las definiciones)
2. En la segunda entrega, se deben ejecutar los “ejecutables” en el procesador (monotarea) y se deben agregar includes y llamados a función en los programas assembler
    a. Clases a implementar: Procesador, Sistema Operativo, la lógica de las clases asociadas a las instrucciones assembler
3. En la tercera entrega se deben implementar algunos system calls y el multitasking
4. En la cuarta entrega se deben implementar una librería y uno/dos programas en assembler

1ra. Entrega

```
Int Ax = 0;
For (int cx = 1; cx<= 10; cx++) {
        Ax = ax + cx;
}
```

Ejemplos

Ejemplo: sumar nros de 1 a 10

```
listaInstrucciones = (Mov(), Mov(), Noop(), Add(), Inc(), Jmp()...)
entryPoint = 5
lookupTable = (“Ciclo” => 3)
listaCodigoFuente=(“Mov Ax, 0”, “Mov cx, 1”, …)

0 0     Mov ax, bx -> Mov()
1 1     Mov cx, 1 -> Mov()
2 Ciclo:
3 - 2   Add ax, cx -> Add()
```

```
4 - 3    Inc cx -> Inc()
4        Cmp cx, 11 -> Cmp()
5        JNZ Ciclo -> Jnz()
6        jmp Fin -> Jmp("Fin")
7        mov ax, dx -> Mov()
…
Fin:
-> en ax tengo acumulado el resultado
```

Lista de instrucciones

Siguiendo el ejemplo anterior la lista sería:

Instrucciones = [<instancia de Mov>, <instancia de Mov>, <instancia de NOOP>, <instancia de Add>,...]

```
class Instruccion:
        procesar(self, Procesador):
                pass

class Mov(Instruccion):
        def _init_(self, param1, param2):
                Self.paramUno = param1
                Self.paramDos = param2

        Def procesar(self, procesador):
                If self.param2 in ["ax", "bx", "cx", "dx"]:
                        procesador.setRegister(self.param1,
                        procesador.getRegister(self.param2))
                Elif:
                        procesador.setRegister(self.param1, int(self.param2))

                procesador.incrementarIP()
```

Ejecución del programa

procesador.py prog.asm prog2.asm prog3.asm

Arquitectura/Clases del TP

Ensamblador:

- Toma el archivo fuente en assembler y genera un Ejecutable
- Si hay un error debe reportar un mensaje y la línea del error

(

Multiplicar:
…

Ciclo:
instrucciones

Entry_point: <-4
Mov ax.2
Mov bx,4
Call multiplicar
)

```
1 def f(param);
2       Return param+1
3
4 -> print(f(4))
```

```
Int f(int param) {
        Return param + 1;
}

Int main() {
        printtf(..., f(4));
}
```

Ejecutable:

- Variables de instancia:
    - entryPoint <- 4
    - Lista con las instrucciones del código fuente
    - Diccionario llamado “lookupTable” que va a contener como claves los labels/etiquetas del programa y como valor la dirección de ese label
        - Ejemplo (“Ciclo” => 2, “Continuar” => 6)
    - Lista de instrucciones: el i-esimo elemento va a ser la instrucción ejecutable de la i-esima instrucción del programa

Instruccion

- Que tenga un descendiente por cada tipo de instruccion
- O sea las subclases serían: Mov, Add, Cmp, Inc, Dec,

Instrucciones:

```
Mov:
    • Mov ax, bx
    • Mov ax, 1
    • Mover un valor literal o el valor de otro registro al registro de la
      izquierda

Add:
    • Add ax, bx
    • Add ax, 3
    • suma dos registros o un registro y un valor literal y deja el
      resultado en el registro de la izquierda
Jmp:
    • Jmp <label/etiqueta>
    • Saltar al lugar del programa que está referenciado por la etiqueta
Jnz:
    • Jnz <label/etiqueta>
    • Saltar al lugar del programa que está referenciado por la etiqueta
      si el flag está en 1 (si no pasa a la siguiente instrucción)
Cmp:
    • Cmp ax, bx
    • Cmp ax, 5
    • Compara por igual (=) dos registros o un registro y un valor literal
      y setea el flag en 0 si la comparación da verdadera si no lo setea
      en 1
Inc:
    • Inc ax
    • Incrementa en 1 el valor de un registro
Dec:
    • Dec ax
    • Decrementa en 1 el valor de un registro
```

Ejemplo

Contexto: el parseo del archivo fuente sucede en el método ensamblar(nombreArchivo) de la clase Ensamblador

Pasos:

1. Crear una instancia Ejecutable
2. Recorrer el archivo línea por línea
3. Discernir si la línea contiene una instrucción o una etiqueta
    a. Si contiene un etiqueta tenemos que agregar en el diccionario lookupTable el par <etiqueta, posición de la instrucción correspondiente>
    b. Si contiene una instrucción, tendremos que indicar el tipo y los parámetros para generar la instrucción correspondiente

```
(
        ...
        Jnz fin

fin:
)
```

```
Entry_point:
        Mov Ax, 0
        Mov cx, 1
Ciclo:
        Add ax, cx
        Inc cx
        Cmp cx, 10
        Jnz ciclo
Continuar:
        ...
```

Qué pasa si hay que saltar a una etiqueta que está más abajo en el programa?

Ejemplo

```
Jmp ciclo
Ciclo:
```

Indice = 2

```
Ciclo:
```

```
        Add ax, cx
        Inc cx
        Cmp cx, 10
        Jnz ciclo
Entry_point:
        Mov Ax, 0
        Mov cx, 1
Continuar:
```

```
        Mov Ax, 0
        Mov cx, 1
        Add ax, cx
        Inc cx
        Cmp cx, 10
        Jnz 2
```

2da. Entrega (1ra. parte)

Instrucciones:

- El método procesar()

Procesador:

- Contiene los registros del procesador (ax, bx, cx, dx, ip, flag)
- getters/setters de cada uno
- Implementa el método procesar() que ejecuta el Ejecutable en el Procesador

Visualizador

- Va a mostrar en pantalla la ejecución del Ejecutable en el Procesador

SistemaOperativo:

- Recibe en el constructor un Ejecutable y un Procesador

```
def procesar(self):
        self.procesador.ejecutar(self.ejecutable)
```

```
class SistemaOperativo:
def __init__(self, ejecutable, procesador):
        self.ejecutable = ejecutable
        self.procesador = procesador
```

```
        Def procesar(self, ejecutable, procesador):
                #opcional: setear IP
                procesador.procesar(ejecutable)

(1ra. Versión con Ejecutable)
class Procesador:
        Def procesar(self, ejecutable):
                self.setIP(ejecutable.getEntryPoint())
                while (procesador.getIP() < len(ejecutable.getListaInstrucciones())):
                        indiceInstruccion = procesador.getIP()
                        ejecutable.getListInstrucciones()[indiceInstruccion].procesar(self)
                        visualizador.mostrar(ejecutable, procesador)
                        time.sleep(1)

                        #print(ejecutable.getCodigoFuente()[indiceInstruccion])
                        #procesador.mostrar()

(2da. Versión con Proceso)
class Procesador:
        Def procesar(self, proceso):
                # opcional procesador.setIP() = ejecutable.getEntryPoint()
                Self.proceso = proceso
                Ejecutable = proceso.getEjecutable()
                while (procesador.getIP() < len(ejecutable.getListaInstrucciones())):
                        indiceInstruccion = procesador.getIP()
                        ejecutable.getListInstrucciones()[indiceInstruccion].procesar(procesador)
                        #visualizador.mostrar(ejecutable, procesador)
                        #print(ejecutable.getCodigoFuente()[indiceInstruccion])
                        #procesador.mostrar()

        Def getProceso():
                Return Self.proceso
```

2da. Entrega (2da. parte)

Funcionalidad nueva:

- Nivel sintáctico: agregar “includes”, o sea la posiblidad de que una archivo assembler incluya otro/s archivo/s assembler. Va a permitir tener bibliotecas de funciones + PUSH, POP, RET, CALL
- Nivel semántico: implementar llamados a función (push, pop, call, ret)

1 - Agregado de “includes”

En el Ensamblador

- Agregar una función que recorra el programa y detecte “include”
- Para cada “include” llamar recursivamente a la función

Main -> lib 1 -> lib 2 -> .. -> lib n

[instr lib n
Instr lib n-1
….
Lib 1
main
]

main.asm:

Include “lib1.asm”

Mov ax, bx

—-
Lib1.asm:
Include “lib2. Asm”
Mov ax, 1

—--
Lib2.asm
Mov ax,2

[Mov ax,2
Mov ax, 1
Mov ax, bx]

Ejemplo:

Main.asm:

Include “funciones.asm”

Push 4
Push 5

Include “codigo_repetido.asm”

Call multiplicar

Include “codigo_repetido.asm”

Funciones.asm:
Multiplicar:
…

Al final la función nos va a devolver la siguiente lista de tuplas:
[
[“Multiplicar:”, “Funciones.asm”],
[..],
[..],
[“Push 4”, “main.asm”],
[“Push 5”, “main.asm”],
[“call multiplicar”, “main.asm”]
]
Lista = generarLista(“main.asm”)
For tupla in lista:
    ...

2- Implementación de funciones

- Requerirá una pila o stack
- En un llamado a función por el stack se pasan los parámetros y la dirección de retorno
- Hay que implementar las siguientes instrucciones (en assembler)
    - PUSH <entero/registro>: agregar un nro entero en el tope del stack o el valor de un registro

- POP <registro>: saca el nro entero que está en el tope del stack y se los asigna al registro
- CALL <etiqueta>: llama a la función “etiqueta”, el llamado a función implica agregar al tope del stack la dirección de retorno de la función
- RET: retorno de una función, saca del tope del stack el entero que haya y lo interpretará como la dirección a la que tiene que ir para seguir la ejecución

Llamados a función:

- En assembler se realiza mediante la instrucción “Call”
- Se utiliza el <stack>

SP

Stack/Pila

-> Direccion de retorno
...

```
Sumar:
        Pop bx -> saca la dirección de retorno
        Pop cx -> 7
        Pop dx -> 5
        Push bx -> pongo en el tope de la pila la dir. de retorno
        Add cx, dx
        Mov ax, cx
        Ret

Ciclo:
        Add ax, cx
        Inc cx
        Cmp cx, 10
        Jnz ciclo
Etiqueta_1:
        Mov Ax, 0
        Mov cx, 1

        Push ax = 5
        Push bx = 7
        Call sumar
        cmp ax, 4 <-
Continuar:
```

```
Include “librería.asm”

Entry_point:
        push 4
        Push 8
        Call sumar
        Cmp ax, 10
```

```
Class Proceso:
        Def __init__(ejecutable):
                Self.ejecutable = ejecutable
                Self.stack = []

        Def getEjecutable(...):
                Return self.ejecutable

        Def getStack():
                …

Class SistemaOperativo:
        Def __init__():

        Def procesar(ejecutable, procesador):
                proceso= Proceso(ejecutable)
                procesador.procesar(proceso)
```

Resumen de la 2da. Entrega (2C 2022)

- Implementar las instrucciones
- Visualizador
- Includes en los programas assembler
- Llamados a función (PUSH, POP, CALL, RET)
- Ejecución en el Procesador (monotarea)

    - método ejecutar(Proceso), recorre las instrucciones del Ejecutable y las ejecuta hasta el final
- SistemaOperativo: procesar(Ejecutable, Procesador):
    - Crear la clase Proceso (debería contener un Ejecutable y la pila/stack)
    - procesador.procesar(proceso)

3ra. Entrega

Multitasking estático

El multitasking es la capacidad de un par procesador/sistema operativo que permite la ejecución de varias tareas/procesos a la vez.

El multitasking estático es la ejecución de un conjunto predefinido de tareas/procesos.

Un cambio de contexto es el procedimiento mediante el cual se saca a un proceso del procesador y se pone el siguiente proceso a ejecutar en el procesador.

Cosas nuevas:

- El Procesador durante el ciclo de ejecución va a llamar al “SistemaOperativo”, mediante el método “clockHandler()”
- El “SistemaOperativo” tendrá una lista de “Procesos” en ejecución
- Se definirá la clase Proceso:
    - Ejecutable
    - Contexto: guardar el estado del procesador cuando hay un cambio de contexto (diccionario cuyas claves sean los registros: ej. “Ax” -> 0, …, “Ip”->ejecutable.entry_point, “flag” -> “0”)
    - Estado (ejecutando, finalizado, bloqueado)

```
Def procesar():
    While (estado == “Activo”):
            Instruccion = self.proceso.ejecutable.listaInstrucciones()[self.ip]
            instruccion.procesar(...)
            self.sistema.clockHandler()
            self.visualizador.mostrarPantalla()
            ….
```

Ejemplo de clockHandler() en SistemaOperativo:

```
Def clockHandler():
```

```
self.contadorInstrucciones++
If (self.contadorInstrucciones == self.RAFAGA_INSTRUCCIONES or “proceso termino”)
        If (“proceso termino”)
                Setear el estado en finalizado
        Else:
                Setear estado en bloqueado

        # sacar del procesador el proceso actual: guardar en el Contexto del Proceso los valores de los registros
        #cambio de contexto: buscar  el próximo proceso que debe ejecutar
        # setear el procesoActivo como el nuevo proceso que debe ejecutar
        # inicializamos contadorInstrucciones (contadorInstrucciones = 0)
        # restituimos el Contexto del nuevo proceso en el Procesador
        procesoActivo = procesoActivo++ % len(self.listaProcesos)
        #self.procesador.setProceso() = self.listaProcesos[procesoActivo]
        self.procesador.ejecutarProceso(self.listaProcesos[procesoActivo])

Else:
        Pass
```

contexto = {“ax” => 1, “bx” => “2”, …, “ip” => 4, “flags” => 0}

Pasos de la ejecución del sistema

1. sistema.py <prog1.asm> <prog1.asm> <prog2.asm> <prog3.asm> <prog4.asm>
2. Crear el Ensamblador y el Procesador
3. Para cada programa Assembler pasado por línea de comando el Ensamblador generará un Ejecutable
4. Se creará el SistemaOperativo y se le pasará el conjunto de Ejecutables como parámetro al constructor, y también se le pasará el Procesador
5. El SistemaOperativo en el constructor debería hacer lo siguiente:
    a. Armar una lista de Procesos en base a los Ejecutables que recibe
    b. Cargar en el Procesador el primer proceso que debe ejecutarse. Sugerencia que Procesador implemente un método “procesador.ejecutarProceso(<proceso>)”
    c. Dicho método carga los registros con los valores del Contexto del proceso que recibe como parámetro
    d. Pasarle la referencia propia (self) al Procesador: procesador.sistemaOperativo = self
6. En el main deberíamos poner a ejecutar el Procesador (procesador.procesar())
7. Dicho método tiene las siguiente características:

a. Tiene el while(True); (o while (self.estado == “Activo”)) es el método principal de toda la ejecución
b. Ejecuta cada instrucción del Ejecutable del proceso que está ejecutando en este momento: self.procesoActivo.ejecutable.listaInstrucciones[self.ip]
c. Dentro del cuerpo del while(), el Procesador deberá llamar a un método del SistemaOperativo (que denominamos sistemaOperativo.clockHandler())
d. El método sistemaOperativo.clockHandler() tendrá que:
    i. Fijarse si hay que hacer un cambio de contexto
    ii. Si hay que hacer un cambio de contexto se debería ejecutar el método procesador.sacarProcesoActual() y luego ejecutar procesador.ejecutarProceso(<proceso>) con el siguiente proceso que haya que ejecutar

Llamadas al sistema operativo. Impresión por pantalla

```
Mov ax, 2
Mov bx, 3
Mov cx,2
Int 1
```

La idea es que cada proceso tenga una matriz (10x10) que simule la memoria de video.

En cada paso el sistema tendrá que mostrar en pantalla el contenido de la memoria de video

- Las instrucciones “conocen” al procesador (porque reciben un referencia en el procesar)
- A partir de esa referencia, la instrucción (int) debería pedir al procesador la referencia del sis op
- Finalmente ejecutar el método de sis op “syscallHandler(nro de servicio, lista de parámetros)”

```
Def ejecutar(self, proceso):
        …
                instruccion.procesar(self)
```

Bosquejo del main:

1) Ensamblar el programa -> genera un Ejecutable
2) Crear un Proceso con el Ejecutable
3) Ejecutar el Proceso en el Procesador

3 - Implementar Multitasking (estático)

El multitasking es la capacidad de un par procesador/sistema operativo que permite la ejecución de varias tareas/procesos a la vez.

El multitasking estático es la ejecución de un conjunto predefinido de tareas/procesos.

Un cambio de contexto es el procedimiento mediante el cual se saca a un proceso del procesador y se pone el siguiente proceso a ejecutar en el procesador.

Cosas nuevas:

- El Procesador durante el ciclo de ejecución va a llamar al “SistemaOperativo”, mediante el método “clockHandler()”
- El “SistemaOperativo” tedrá una lista de “Procesos” en ejecución
- Se definirá la clase Proceso:
    - Ejecutable
    - Contexto: guardar el estado del procesador cuando hay un cambio de contexto
    - Estado (ejecutando, finalizado, bloqueado)
    - contadorInstrucciones?? (Para mí debería estar dentro de Sistema Operativo este contador)

Ejemplo de acceso al stack:

```
Class Push(Instruccion):
        Def __init__(ejecutable):
                Self.ejecutable = ejecutable

        Def procesar(...):
                self.ejecutable.getStack().append(valor)
```

Ejemplo de procesar() de Procesador:

```
Def procesar():
        While (estado == “Activo”):
                While (self.ip < len(self.proceso.ejecutable.listaInstrucciones()))
                        Instruccion = self.proceso.ejecutable.listaInstrucciones()[self.ip]
                        instruccion.procesar(...)
```

```
self.sistema.clockHandler()
self.visualizador.mostrarPantalla()
….
```

Ejemplo de clockHandler() en SistemaOperativo:

```
Def clockHandler():
        self.listaProcesos[procesoActivo].contadorInstrucciones++
        If (self.listaProcesos[procesoActivo].contadorInstrucciones <
self.RAFAGA_INSTRUCCIONES)
                Pass
        Else:

                # sacar del procesador el proceso actual: guardar en el Contexto del Proceso los
valores de los registros
                #cambio de contexto: buscar  el próximo proceso que debe ejecutar
                # setear el procesoActivo como el nuevo proceso que debe ejecutar
                # inicializamos contadorInstrucciones (contadorInstrucciones = 0)
                # restituimos el Contexto del nuevo proceso en el Procesador
                self.procesador.setEjecutable() =
self.listaProcesos[procesoActivo].getEjecutable() ??????? Fijarse qué les parece mejor: que el
procesador tenga una referencia al Proceso o el Ejecutable asociado al mismo
```

Pasos de la ejecución del sistema

8. sistema.py <prog1.asm> <prog1.asm> <prog2.asm> <prog3.asm> <prog4.asm>
9. Crear el Ensamblador y el Procesador
10. Para cada programa Assembler pasado por línea de comando el Ensamblador generará un Ejecutable
11. Se creará el SistemaOperativo y se le pasará el conjunto de Ejecutables como parámetro al constructor, y también se le pasará el Procesador
12. El SistemaOperativo en el constructor debería hacer lo siguiente:
    a. Armar una lista de Procesos en base a los Ejecutables que recibe
    b. Cargar en el Procesador el primer proceso que debe ejecutarse. Sugerencia que Procesador implemente un método “procesador.ejecutarProceso(<proceso>)”
    c. Dicho método carga los registros con los valores del Contexto del proceso que recibe como parámetro
    d. Pasarle la referencia propia (self) al Procesador
13. En el main deberíamos poner a ejecutar el Procesador (procesador.procesar())
14. Dicho método tiene las siguiente características:
    a. Tiene el while(True); (o while (estado == “Activo”)) es el método principal de toda la ejecución

b. Ejecuta cada instrucción del Ejecutable del proceso que está ejecutando en este momento
c. Dentro del cuerpo del while(), el Procesador deberá llamar a un método del SistemaOperativo (que denominamos sistemaOperativo.clockHandler())
d. El método sistemaOperativo.clockHandler() tendrá que:
    i. Fijarse si hay que hacer un cambio de contexto
    ii. Si hay que hacer un cambio de contexto se debería ejecutar el método procesador.sacarProcesoActual() y luego ejecutar procesador.ejecutarProceso(<proceso>) con el siguiente proceso que haya que ejecutar

```
Class Ensamblador:

Def ensamblar(codigoFuente):
ejecutable = Ejecutable()
nroLinea = 0
For linea in codigoFuente:
        If (linea es etiqueta):
                ejecutable.agregarEtiqueta(etiqueta,nroLinea)
        Elseif (linea es instruccion):
                Instrucccion = None
                If (instrucion es Mov):
                        Instruccion = Mov(...)
                elseIf (instrucion es Inc):
                        Instruccion = Inc(...)
                Ielsef (instrucion es JMP):
                        Instruccion = JMP()
                ...
                ejecutable.agregarInstruccion(instruccion)
        nroLinea+=1

Return ejecutable

Class Ejecutable:
listaInstruciones
diccionarioEtiquetas
```

4to. entregable

Llamadas al sistema operativo

1. Pedir una cierta cantidad de nros. Entero
2. Devolver una cierta cantidad de nros Entero
3. Imprimir en pantalla una cierta cantidad de nros Entero

El servicio y 2 estan relacionados y se implementan del siguiente modo:

1. El sistema op tendrá un vector
2. Cuando invoquemos al servicio, el sis op va a buscar una posición libre del vector y nos va a devolver el índice correspondiente
3. El sis op también tiene que registrar a qué proceso le asignó esa posición del vector
    a. Idea para resolver esto: tener otro vector del mismo tamaño que registre el nro. De proceso al que se le asigno cada posición de memoria

```
Entry_point:
        push 4
        Push 2
        Call mult //este proc me devuelve el resultado en “ax”
        Mov bx, 4
        Int 1 // devuelve el indice en “bx”
        Mov [bx], ax
        Add bx, 1
        Mov [bx], [cx]

        …
        mov bx, 2
        Int 2
```

```
Mov [bx+2], [bx]
```

```
memoria : [34,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
asignacionesDeMemoria:  [1,1,1,1,-1,1,1,-1,0,0,0,0,0,0,0,0,0,0,0,0]
```

```
Class SistemaOperativo:

Def getMemPos(pos):
        If (asignacionesDeMemoria[pos] == procesoActual):
                Return memoria[pos]
```

```
    Else:
        #terminar la ejecución

Def setMemPos(pos, value):
    If (asignacionesDeMemoria[pos] == procesoActual):
        Memoria[pos] = value
    Else:
        #terminar la ejecución
```

Impresión por pantalla (servicio 3)

```
Mov cx, 2
Mov dx, 3
Add bx,2
Int 3
```

La idea es que cada proceso tenga una matriz (10x10) que simule la memoria de video.

En cada paso el sistema tendrá que mostrar en pantalla el contenido de la memoria de video

---

- Las instrucciones “conocen” al procesador (porque reciben un referencia en el procesar)
- A partir de esa referencia, la instruccion (int) debería al procesador la referencia del sis op
- Finalmente ejecutar el método de sis op “syscallHandler(nro de servicio, lista de parámetros)”

```
Class Int(Instruccion):
Def __init__(nro):
    Self.nro = nro

Def procesar(self,procesador):
    sisop = procesador.getSisop();
    Parametros = []
    If (self.nro == 1): #pedir memoria
```

```
        # en ax está la cantidad de enteros que quiero pedirle al sis op
                Parametros = [procesador.getAx()]
        Elsif (self.nro == 2): # devolver memoria
        # en ax está el índice desde donde empieza el bloque de memoria que quiero devolver
                Parametros = [procesador.getAx()]

        Elsif (self.nro == 3): #imprimir por pantalla
        # en ax vamos a tener el entero que queremos imprimir, en bx tendrá la fila y cx la
columna de donde donde quiero que se imprima en la pantalla
                Parametros = [procesador.getAx(), procesador.getBx(), procesador.getCx()]
        Else:
                #Error

        sisop.syscallHandler(self.nro, parametros)


Class SistemaOperativo:
        Def syscallHandler(servicio, parametros):
                …
                elsIf (servicio == 3): #imprimir por pantalla
                        Valor = parametros[0]
                        Fila = parametros[1]
                        Columna = parametros[2]
                        self.procesos[procesoActivo].memoriaVideo[fila, columna] = valor
                        refrescarPantalla()


Mov [ax], [cx]


Class Mov(Instruccion):
        Class __init__(,origAccedeAMem, destAccedeAMem):
                self.origAccedeAMem = origAccedeAMem
                self.destAccedeAMem = destAccedeAMem

        Class procesar():
                ...
                If (origAccedeAMem):
                        Sisop = procesador.getSisop()
                        Valor = sisop.getMemPos(procesador.getCx())
                If (destAccedeAMem):
                        Sisop = procesador.getSisop()
                        sisop.setMemPos(procesador.getAx(), valor)
```

```
Add [ax],1

Mov bx, [ax]
Add bx, 1
Mov [ax], bx
```

4to. entregable (1c 2022)

Llamada al sistema operativo

Imprimir en pantalla un nro Entero

Implementación

Assembler

Vamos a tener una nueva instrucción “int <nro. de servicio del SO que queremos invocar>”

```
Mov ax, 5 // valor que quiero imprimir
Mov bx, 1 //fila
Mov cx, 3 //columna
Int 1
```

Instrucción Int

```
Class Instrucción:
 Def __init(nroSysCall):
        self.sysCall = nroSysCall

 Def procesar(procesador):
        sisop = procesador.getSisop();
        If (self.sysCall == 1):
        # en ax vamos a tener el entero que queremos imprimir, en bx tendrá la fila y cx la
columna de donde donde quiero que se imprima en la pantalla
                Parametros = [procesador.getAx(), procesador.getBx(), procesador.getCx()]
                sisop.syscallHandler(self.sysCall, parametros)
        Else:
                Error de ejecución
        Else:
                #Error
```

Proceso

Agregar una matriz (memoria de video). Tamaño de la matriz (por ejemplo): 10x10

```
00000000
00700000
```

0000000

Sistema Operativo

```
Class SistemaOperativo:
    Def syscallHandler(servicio, parametros):
        If (servicio == 1): #imprimir por pantalla
            Valor = parametros[0]
            Fila = parametros[1]
            Columna = parametros[2]
            self.procesos[procesoActivo].memoriaVideo[fila, columna] = valor
```

Visualizador

Va a tener que mostrar en pantalla la memoria de video del proceso.

Programas

Librería de funciones matemáticas

- Multiplicar
  - a.b = a+a+a+a… (b veces)
- Resta
  - a-b = a + (-b)
- Dividir (cociente y resto)
  - a/b = b+b+b+b+b+... (hasta pasarme del valor a)
  - 8/3 = 2 * 3 + 2
  - 8/3 = 3+3 (ax = 2, bx = 2) guardarlos en 2 registros
  - 11/4 = 2 * 4 + 3
  - 4+4
  - 4 + 4 -> el cociente es 2
  - 11 - 2*4 = 3
  - Sumar el denominador consigo mismo hasta pasarme del numerador (me quedo con el mayor tal que no me paso del numerador)
- Raíz cuadrada (de enteros positivos)
  - sqrt(25) : pruebo con el 2 (2*2 = 4), pruebo con el 3 (3*3=9), pruebo con 4 (4*4 = 16), pruebo con 5 (5*5 = 25)
  - sqrt(26): hago lo mismo que en el caso anterior y cuando prueba con 6 (6*6= 36) entonces me quedo con el 5 (porque queremos calcular la parte entera de la raiz)

Implementar una nueva instrucción:

- “neg <valor literal | registro>”
- Modificar el cmp para que: 1 si a < b, 0 si a >= b

Programas

- Función que calcule las 2 raíces de una cuadrática:
    - Recibe los coeficientes (a,b,c) de una función cuadrática (ax^2+bx+c)
    - (-b +/- sqrt(b^2-4ac)) / 2a
    - Debería dejar las raíces en 2 registros (ax, bx)
    - Escribir un programa que llame a la función (con valores hardcodeados) e imprima por pantalla las 2 raíces.
- Fibonacci
    - f(n) = f(n-1) + f(n-2)
    - f(1) = f(2) = 1
    - 1,1,2,3,5,8,13,...
    - Hacer una función recursiva que calcule f(n), debería dejar en un registro el valor de f(n) (por ejemplo en ax).
    - Escribir un programa que llame a la función (con un valor hardcodeado) e imprima por pantalla el resultado