UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Ingeniería en Sistemas de Información

[Está CoverFlow]

Porque para entender la recursividad primero hay que comprender la recursividad

Cátedra de Sistemas Operativos

Trabajo práctico Cuatrimestral

- 1C2014 -
Versión [1.0]

Tabla de Contenidos

Introducción

El trabajo práctico consiste en simular ciertos aspectos de un sistema multiprocesador con la capacidad de interpretar la ejecución de scripts escritos en un lenguaje creado para esta ocasión. Este sistema planificará y ejecutará estos scripts (en adelante “Programas”) controlando sus solicitudes de memoria y administrando los accesos a recursos, como los dispositivos de entrada/salida y los semáforos compartidos.

Objetivos del Trabajo Práctico

Mediante la realización de este trabajo se espera que el alumno:

- Adquiera conceptos prácticos del uso distintas herramientas de programación (API) que brindan los sistemas operativos modernos
- Entienda aspectos del diseño de un sistema operativo moderno
- Afirme diversos conceptos teóricos de sistemas operativos mediante la implementación práctica de algunos de ellos
- Se familiarice con técnicas de programación de sistemas, como el empleo de makefiles, archivos de configuración y archivos de log
- Conozca con grado de detalle la operatoria de Linux mediante la utilización de un lenguaje de programación de relativamente bajo nivel como C

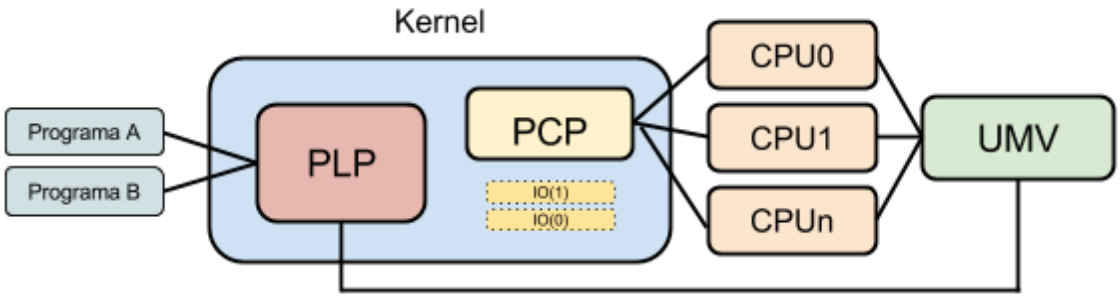
Características

- Modalidad: grupal (5 integrantes) y obligatorio
- Tiempo estimado para su desarrollo: 10 semanas
- Fecha de comienzo: sábado 12-Abril
- Fecha de entrega: sábado 12-Julio
- Fecha de primer recuperatorio: sábado 19-Julio
- Fecha de segundo recuperatorio: sábado 02-Agosto
- Lugar de corrección: Laboratorio de Medrano

Aclaraciones

Debido al fin académico del trabajo práctico, los conceptos reflejados son, en general, versiones simplificadas o alteradas de los componentes reales de hardware y de sistemas operativos modernos, a fin de resaltar algún aspecto de diseño. En algunos casos los aspectos no fueron tomados de manera literal, por lo que invitamos a los alumnos a leer las notas y comentarios al respecto que hayan en el enunciado, como así también a reflexionar y discutir con sus compañeros, ayudantes y docentes al respecto.

Arquitectura del sistema



Procesos Programa

El Programa es un proceso simple que permite enviar a ejecutar código Ansisop al sistema y funcionar como consola del mismo para recibir los resultados de su ejecución o los mensajes que el script necesite imprimir por pantalla.

Arquitectura del Proceso Programa

Al iniciar, el proceso recibirá como primer argumento el nombre del archivo a procesar. Leerá su archivo de configuración, se conectará mediante ***sockets***[1] al Proceso Kernel y, luego de un intercambio de mensajes inicial, enviará el código Ansisop del Programa. A partir de ese momento, el proceso quedará a la espera de mensajes correspondientes a las sentencias ***imprimir*** e ***imprimirTexto*** con valores que deberá mostrar por consola, hasta que su ejecución haya concluido.

Podrán existir varias instancias de este proceso en el sistema ejecutando programas Ansisop de manera independiente.

El proceso Programa deberá poder ser utilizado como intérprete de scripts Ansisop mediante el encabezado hashbang (#!)[2] y deberá terminar con una línea en blanco.

33

Proceso Kernel

[1] Siempre que en el enunciado se lea la palabra socket, se refiere a los sockets STREAM tipo AF_INET
[2] Ver http://mgarciaisaia.github.io/tutorial-c/blog/2014/03/20/she-bangs-she-bangs/

El proceso Kernel es el proceso principal del sistema. Recibirá los programas y planificará su ejecución en el sistema utilizando principalmente los hilos PLP y PCP descritos a continuación.

Planificador de Largo Plazo (PLP)

El Planificador de Largo Plazo será el encargado de mantener el grado de multiprogramación del sistema en el valor definido. También será el punto de entrada al sistema, encargado de recibir las conexiones de los nuevos Programas.

El PLP recibirá las conexiones de los nuevos Programas al sistema, y, por cada uno, creará su estructura PCB, solicitará al Proceso UMV (el encargado de manejar y gestionar la memoria en este sistema) cuatro segmentos de tamaño suficiente (código literal, índice de etiquetas y índice de funciones, índice de código y stack) y encolará el proceso según el algoritmo de planificación de largo plazo SJN.

Para estimar el tamaño del job utilizará la siguiente ecuación:

```
peso = 5 * cantidad_de_etiquetas + 3 * cantidad_de_funciones +
            cantidad_total_de_lineas_de_codigo
```

En todo momento se deberá poder ver en pantalla el estado de la cola de procesos en espera para ser ejecutados, y su correspondiente peso.

Creación del PCB

Al recibir la conexión de un nuevo Programa, el PLP intercambiará unos mensajes iniciales con el mismo, para luego recibir la totalidad del código fuente del script que se deberá ejecutar.

El PLP creará la estructura PCB, asignándole un identificador único, y usará la funcionalidad de preprocesamiento del parser, que recibirá todo el código del script y devolverá una estructura con información del programa. Esta estructura contendrá:

- Primer instrucción a ejecutar del programa
- Índice de etiquetas y funciones serializado
- Índice de código serializado
- Tamaño del índice de etiquetas y funciones
- Tamaño del índice de código

A partir de esta información, el PLP podrá reservar los segmentos para los segmentos de Índices, tanto de Instrucciones como de Etiquetas. Además, dado que el tamaño del Stack es prefijado y que el tamaño del segmento de Código coincide con el tamaño del script, el PLP estará en condiciones de reservar todos los segmentos de memoria del Programa.

Una vez reservados los segmentos, escribirá el código del script en el segmento correspondiente, y luego hará lo propio con los Índices de Instrucciones y de Etiquetas, ambos presentes en la estructura devuelta por el preprocesamiento.

Si no se pudiera obtener espacio suficiente para alguno de los segmentos que necesita usar el proceso, entonces se le rechazará el acceso al sistema, informándolo oportunamente en la consola del proceso Programa correspondiente.

Por último, el PLP inicializará el Program Counter del programa al valor que el preprocesamiento indique como la Primer Instrucción, para luego encolar al proceso en donde la planificación lo indique.

Planificador de Corto Plazo (PCP)

El subsistema Planificador de Corto Plazo es el encargado de distribuir los procesos en los distintos CPUs.

Este recibirá las conexiones de los distintos CPUs y quedará a la espera de que el PLP deposite el PCB de un nuevo programa a ejecutar en la estructura de cola que ambos subsistemas comparten.

Ante la eventual desconexión de un CPU con un Programa en ejecución, el PCP deberá notificar a la consola del Programa dicha excepción y dar por concluida la ejecución de ese programa.

Realizará las siguientes tareas:

- Recibirá los PCB del PLP y los encolará en la cola de READY, según el algoritmo de planificación de corto plazo Round Robin
- Moverá a la cola de READY todos los procesos que hayan terminado su ráfaga de CPU y concluído su entrada/salida en función de la prioridad de cada uno
- Enviará a Ejecutar los procesos que esten en READY a los procesadores disponibles, informando al cpu el quantum asignado
- Moverá a la cola de EXIT todos los procesos que hayan concluído su ejecución
- Recibirá de cada CPU la información actual de cada PCB en ejecución y enviará a cada CPU libre al primer elemento de la cola de READY

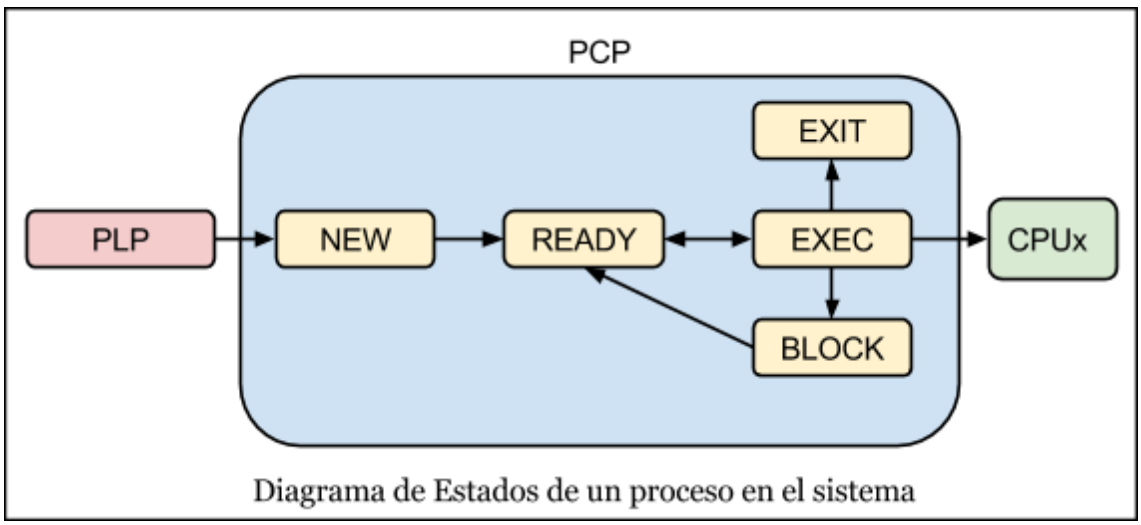


Diagrama de Estados de un proceso en el sistema

Tendrá también una responsabilidad adicional que será la de interpretar determinadas llamadas al sistema y proveer servicios que excedan sus capacidades a los Programas en

ejecución

System Calls

El lenguaje Ansisop es lo suficientemente potente para permitir que los Programas trabajen de manera colaborativa para resolver una tarea específica.

En particular permite que el Programa utilice Variables Globales al sistema, cuyo valor es accesible por cualquier Programa en ejecución. Para evitar situaciones de condiciones de carrera brinda también al programador la posibilidad de utilizar Semáforos para sincronizar la ejecución de ciertas porciones de sus Programas. Por último, gestionará los pedidos de ejecución de operaciones de entrada/salida, administrando las colas de espera de cada dispositivo.

Las System Calls serán enviadas por el Proceso CPU mediante el socket de comunicación que lo une con el PCP. Este recibirá la solicitud y responderá en consecuencia.

Existen cinco System Calls que soporta el Kernel:

obtener_valor [identificador de variable global]
grabar_valor [identificador de variable global]

wait [identificador de semáforo]
signal [identificador de semáforo]

entrada_salida [identificador de dispositivo] [unidades a utilizar]

En el código Ansisop el identificador de la variable compartida, para diferenciarlo de las variables normales, comenzará con el caracter **signo de admiración (!)**, seguido del identificador de un caracter alfabético, por ejemplo: !a, !g, !q. Se observa que el sistema permite un máximo de 26 variables globales, inicializadas en cero.

Los semáforos estarán definidos por archivo de configuración con un identificador alfanumérico y un valor inicial, por ejemplo: *Semaforo1* o *redlight8*. Se crearán al iniciar el proceso Kernel junto con su correspondiente hilo para encolar los Programas que se encuentren bloqueados esperando un semáforo. Se considera una excepción abortiva intentar acceder a una variable global o a un semáforo inexistente y no está dentro del alcance de la evaluación.

Hilos de entrada/salida (HIO)

Por archivo de configuración se definirá la cantidad de dispositivos de entrada/salida que habrá presentes en el sistema. Para implementarlos, se lanzará un Hilo de Entrada/Salida (HIO) por cada dispositivo.

Cada instancia del HIO tendrá un identificador y un valor de retardo en milisegundos por

unidad de ejecución.

Como se observa en el Anexo, desde el lenguaje Ansisop se puede solicitar que un programa realice entrada/salida en un dispositivo mediante su identificador y una cantidad de unidades de ejecución, por ejemplo: `io Disco 10`

El primer parámetro *(Disco)* corresponde al identificador del dispositivo de entrada/salida, por lo que debería existir en el sistema un HIO con identificador Disco

El segundo parámetro *(10)* es la cantidad de unidades de ejecución que deben realizarse. Por cada unidad, el HIO debe generar una demora definida en milisegundos. Por ejemplo, si el HIO Disco tiene configurado un retardo de 1000ms, para completar la operación el hilo deberá esperar 10 segundos.

Archivo de Configuración

Al iniciar, el proceso Kernel deberá leer los siguientes parámetros de un archivo de configuración, el cuál podrá ser parametrizable como primer argumento del programa.

Parámetro	**Valor**	**Descripción**
Puerto_programas `PUERTO_PROG`	[numérico]	Puerto TCP utilizado para recibir las conexiones de los Programas
Puerto_CPUs `PUERTO_CPU`	[numérico]	Puerto TCP utilizado para recibir las conexiones de los CPUs
Quantum `QUANTUM`	[numérico]	Valor del Quantum (en instrucciones a ejecutar) del algoritmo Round Robin
Retardo Quantum `RETARDO`	[numérico]	Valor de retardo en milisegundos que el CPU deberá esperar luego de ejecutar cada sentencia
Grado Multiprogramación `MULTIPROGRAMACION`	[numérico]	Grado de multiprogramación del sistema
Identificadores Semaforos `VALOR_SEMAFORO`	[array: alfanumérico]	Identificador de cada semáforo del sistema. Cada posición del array representa un semáforo
Valores Iniciales Semaforos `SEMAFOROS`	[array: numérico]	Valor inicial de cada semáforo
Retardos Entrada Salida `HIO`	[array: numérico]	Retardo en milisegundos de cada unidad de operación de entrada/salida. Cada posición del array representa un dispositivo de entrada/salida
Identificadores Entrada	[array:	Identificador de cada dispositivo de

Salida `ID_HIO`	alfanumérico]	entrada/salida Cada posición del vector está asociada a su correspondiente retardo
Variables Compartidas `COMPARTIDAS`	[array: alfanumérico]	Identificador de cada variable compartida

Ejemplo:

```
ID_HIO=[Disco, Impresora, Scanner]
HIO=[1000, 2000, 1000]
SEMAFOROS=[SEM1, SEM2, SEM3]
VALOR_SEMAFORO=[0, 0, 5]
MULTIPROGRAMACION=6
PUERTO_PROG=5000
PUERTO_CPU=5001
QUANTUM=3
RETARDO=2000
COMPARTIDAS=[Global, UnaVar, tiempo3]
```

Proceso Unidad de Memoria Virtual (UMV)

El Proceso UMV[3] es el responsable en el sistema de brindar a los Programas espacio en memoria para que estos realicen sus operaciones. Para esto utilizará una serie de estructuras administrativas internas que deberá crear y mantener.

La UMV permitirá que los Programas funcionen abstraídos de la ubicación final de los datos en la memoria principal.

Arquitectura de la UMV

Al iniciar solicitará un único bloque de memoria (`malloc`) de un tamaño configurable por archivo de configuración, para simular la memoria principal. Luego creará las estructuras administrativas necesarias para poder gestionar dicho espacio permitiendo que cada proceso pueda crear segmentos de tamaño variable y simular que dispone de todo el espacio de direcciones.

A la UMV se conectarán, mediante sockets, el proceso Kernel y los diversos CPUs. Por cada conexión, la UMV creará un hilo dedicado a atenderlo, que quedará a la espera de solicitudes de operaciones. La UMV deberá validar cada pedido recibido, y responder en consecuencia.

Es importante destacar la naturaleza multi-hilo del proceso, por lo que será parte del

[3] *A pesar de que la implementación de la memoria virtual en la vida real es una combinación entre el sistema operativo y el CPU (MMU), se implementa en este caso como un solo proceso aparte, para facilitar su comprensión e integración con el sistema.*

desarrollo del trabajo atacar los problemas de concurrencia que surgieran.

Para simplificar el desarrollo de este proceso, **a diferencia de la realidad,** las estructuras administrativas no deben ser almacenadas en el espacio de memoria utilizado por los Programas.

Se recomienda leer y profundizar el **Anexo I - Segmentación**, y validar el diseño con el ayudante antes de iniciar el desarrollo de este proceso.

Operaciones de la UMV

El Proceso UMV, simulando aspectos de un controlador de memoria real, maneja una interfaz reducida, que no puede ser ampliada, y permite únicamente:

- Solicitar bytes desde una posición de memoria: [base], [offset] y [tamaño]
- Enviar bytes para ser almacenados: [base], [offset], [tamaño] y [buffer]

También soporta algunas operaciones que simplifican el desarrollo del trabajo práctico y su operatoria:

- Handshake: [Identificador del Programa] y [Tipo: Kernel/CPU]
- Cambio de proceso activo: [Identificador del Programa]
- Crear segmento: [Identificador del Programa], [tamaño]
- Destruir segmentos del programa: [Identificador del Programa]

Validaciones de la UMV

Es tarea de este proceso controlar que las solicitudes de memoria sean válidas y coherentes. En caso de intentar escribir en algún segmento que no existe o fuera de los límites del mismo la UMV deberá notificar la correspondiente excepción, como ser:

- *Segmentation Fault*: El acceso a memoria esta por fuera de los rangos permitidos.
- *Memory Overload*: No existe espacio suficiente para crear un segmento

Consola

El Proceso UMV al iniciar quedará a la espera de comandos enviados por teclado permitiendo al menos las siguientes funcionalidades. El diseño de la misma y la sintaxis de los comandos queda a criterio del equipo.

> ***operación****:* Este comando permitirá, dado un proceso, una base, un offset y un tamaño solicitar una posición de memoria o escribir un buffer por teclado en una posición de memoria. El resultado de la misma deberá ser mostrado en pantalla y opcionalmente grabado en un archivo. También permitirá crear y destruir los segmentos de un Programa.
>
> ***retardo***: Este comando modificará la cantidad de milisegundos que debe esperar el proceso UMV antes de responder una solicitud. Este parámetro será de ayuda para evaluar el funcionamiento del sistema.

***algoritmo**:* Deberá permitir cambiar el algoritmo entre Worst-Fit y First-Fit.

compactación: Deberá forzar el proceso de compactación.

dump: Este comando generará un reporte en pantalla y opcionalmente en un archivo en disco del estado actual de:

- **Estructuras de memoria**: tablas de segmentos de todos los procesos o de un proceso en particular.
- **Memoria principal**: indicando los segmentos de los Programas y el espacio libre.
- **Contenido de la memoria principal**: Indicando un offset y una cantidad de bytes.

Proceso CPU

Este proceso es uno de los más importantes del trabajo ya que es el encargado de interpretar y ejecutar las operaciones escritas en código Ansisop de un Programa.

Estará en permanente contacto con el Proceso UMV, tanto para obtener información del Programa en ejecución, como para actualizar las estructuras requeridas luego de ejecutar una operación.

Al iniciar se conectará al proceso Kernel y quedará a la espera de que el PCP le envíe el PCB de un Programa para que este sea ejecutado.

Incrementará el valor del registro Program Counter del PCB y utilizará el índice de código para solicitar a la UMV la próxima sentencia a ejecutar. Al recibirla, la parseará, ejecutará las operaciones requeridas, actualizará los segmentos del Programa en la UMV, actualizará el Program Counter en el PCB y notificará al PCP que concluyó un **quantum**.[4]

Ejemplo:

Luego de recibir de la UMV la instrucción "`a = b + 3`", el parser la interpretará y ejecutará las siguientes Primitivas:

1) `obtener_direccion('b')`: utilizando el diccionario de datos, devolverá el desplazamiento respecto al stack de la variable `b`. Por ejemplo, `0x4245`.
2) `dereferenciar(0x4245)`: pedirá a la UMV los 4 bytes a partir de `offset_b`, correspondientes al valor de la variable `b`. Por ejemplo, `8`.
3) `obtener_direccion('a')`: como 1), pero para saber dónde guardar el resultado. Por ejemplo, `0x424c`.
4) `almacenar(0x424c, 11)`: almacenará en la UMV el resultado de la suma.

El ingreso a funciones o procedimientos requiere que se asienten datos como el punto de

[4] En este punto el alumno ya puede notar que si el proceso fuera desalojado y su PCB enviado a otro CPU, este tendría ahí y en la UMV toda la información necesaria para continuar su normal ejecución

retorno o las variables locales en el segmento de ***Stack*** - Ver detalle en el Anexo IV - Stack

Hot plug

Nuevas instancias del proceso CPU pueden ingresar al sistema en cualquier momento. El PCP deberá reconocer dicha disponibilidad y agregarla a la planificación.

Mediante la señal SIGUSR1, se le podrá notificar a un CPU que deberá desconectarse una vez concluída la ejecución del Programa actual, dejando de dar servicio al sistema.

Fin de la ejecución

Al ejecutar la última sentencia el CPU deberá solicitar la destrucción de todas las estructuras correspondientes, mostrar en la consola del Programa el estado final de las variables y notificarle al PCP que el proceso finalizó.

Excepciones

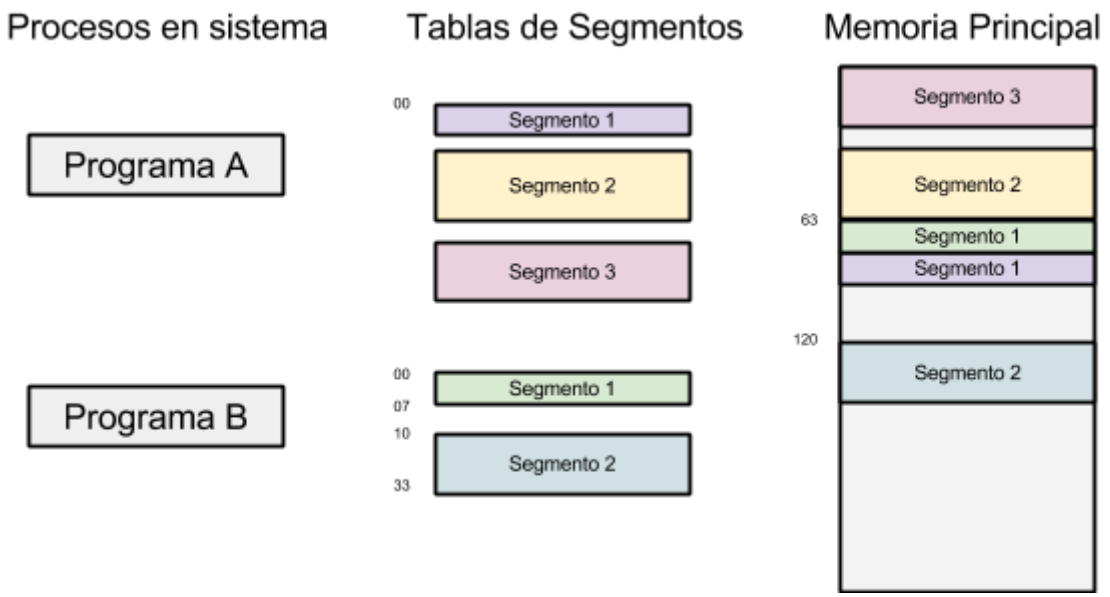
Existe la posibilidad que el Proceso CPU reciba un mensaje de excepción como resultado de una solicitud a la UMV. En este caso deberá notificarla por pantalla y concluir la ejecución del Programa actual.

Anexo I - Modelo de Segmentación

El modelo de Segmentación permite que el programador solicite segmentos de memoria de cualquier tamaño, abstraído de la cantidad total o libre de memoria en la computadora.

Internamente se utilizan estructuras administrativas para llevar control de las solicitudes de memoria y donde se encuentra finalmente almacenada. En otras palabras este mecanismo permite que el proceso crea que dispone del total de la memoria del sistema.

Diagrama de Memoria con Segmentación



En el ejemplo de esta figura, el Proceso B consta de dos Segmentos. El Segmento 1 de 8 bytes iniciando en la posición virtual 0 y a continuación se observa el Segmento 2 de 23 bytes ubicado en la posición virtual 10.

Se observa luego que dichos segmentos están realmente almacenados en las posiciones 63 y 120 de la Memoria Principal.

Los registros en la tabla de segmentos del Programa B se verían así:

Identificador	Inicio	Tamaño	Ubicación en Memoria Principal
Segmento 1	00	8	63
Segmento 2	10	23	120

Entonces, por ejemplo, si el Programa B solicita 5 bytes iniciando en la posición 25 (15 bytes de offset del inicio del segmento 2), utilizando la tabla de segmentos la UMV devolvería 5

bytes empezando de la posición 135 (15 bytes de offset del inicio del segmento en memoria principal).

Creación y destrucción de los segmentos

Los segmentos serán creados a solicitud del PLP cuando crea el PCB del Programa.

Como se observa en la figura los segmentos de un Programa no necesariamente son contiguos. Por razones de seguridad las direcciones de inicio de los segmentos en su creación debe ser seleccionada de manera aleatoria, lógicamente evitando que esta caiga dentro de un segmento existente en ese Programa.

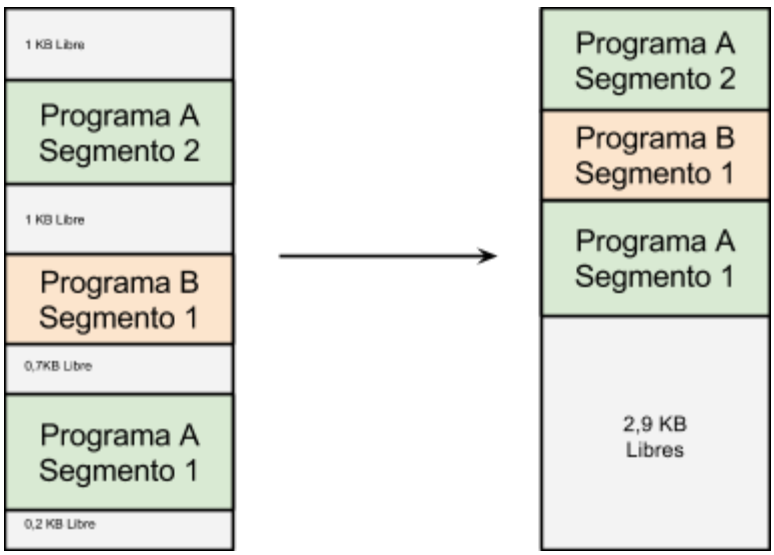
La ubicación del segmento en Memoria Principal deberá seleccionarse mediante los algoritmos Worst-Fit y First-Fit definido por archivo de configuración. El mismo podrá ser cambiado en tiempo de ejecución por consola.

La destrucción de los segmentos sólo se dará en conjuntos, cuando el UMV reciba la orden de destruir todos los segmentos asociados a un programa.

Compactación

Existirán situaciones donde la creación de un segmento no podrá realizarse por no encontrar un espacio libre contiguo lo suficientemente grande para contenerlo.

En este caso, antes de responder con una excepción, el Proceso UMV deberá realizar el proceso de Compactación, cuyo objetivo consiste, como se observa en la figura, en ubicar todos los segmentos ocupados de manera contigua en las posiciones de memoria superiores logrando así que el espacio libre quede agrupado en las posiciones inferiores.



Es importante aclarar que la Compactación es el único mecanismo disponible para unir

segmentos de memoria libre.

Concluído este proceso deberá validar nuevamente si el espacio libre es suficiente para resolver la solicitud y responder en consecuencia.

Anexo II - Bloque de Control del Programa (PCB)

Al igual que en un sistema operativo convencional, todo Programa estará identificado por una estructura denominada PCB (Process Control Block - Bloque de Control del Programa) la cual contendrá el identificador único del proceso, el ***program counter (PC)*** del programa y las referencias a los segmentos del programa: Código y Stack (Pila), y los segmentos auxiliares Índice de Código e Índice de Etiquetas.

Estructura	Información
Identificador Único	Número identificador del proceso
Segmento de código	Código Ansisop del programa
Segmento de stack	Pila de llamadas a funciones
Cursor del stack	Puntero al inicio del contexto de ejecución actual
Índice de código	Estructura auxiliar que contiene el offset del inicio y del fin de cada sentencia del Programa.
Índice de etiquetas	Estructura auxiliar utilizada para conocer las líneas de código correspondientes al inicio de los procedimientos y a las etiquetas
Program Counter	Número de la próxima instrucción del Programa que se debe ejecutar
Tamaño de Contexto Actual	Cantidad de variables (parámetros y variables locales) del contexto de ejecución actual

Índice de código

Debido a que el UMV comprende únicamente solicitudes relacionadas con posiciones de memoria y que las líneas en el código en Ansisop, a diferencia de un programa ejecutando en una computadora real[5], son de longitud variable y además podría contener comentarios o líneas en blanco, este índice es una estructura que almacena el desplazamiento respecto al inicio del código del programa y la longitud de cada línea ejecutable.

De esta manera es posible saber la ubicación de cada línea *útil* en el código para poder solicitarla a la UMV.

Estos datos se almacenarán como un arreglo ordenado de pares de enteros, [desplazamiento de inicio] y [longitud]. En otras palabras, variables enteras de 4 bytes de forma contigua donde cada par identificará de forma ordenada una sentencia del código.

De este modo, los primeros 8 bytes a partir de la dirección indicada por Índice de Código en el PCB, serán dos variables numéricas. La primera será el desplazamiento dentro del

[5] Las instrucciones en un programa compilado son generalmente de longitud fija: el tamaño de una palabra del CPU

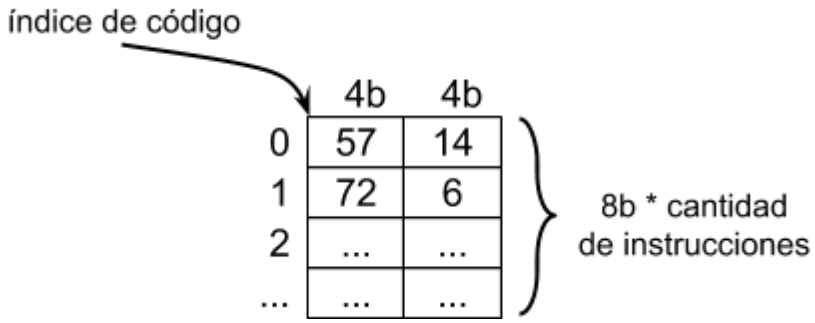
Segmento de Código en el que comienza la primer instrucción de nuestro programa (Program Counter = 1). La segunda variable de 4 bytes indicarán la longitud en caracteres de dicha instrucción. Luego los siguientes 8 bytes corresponderán a la segunda instrucción ejecutable de nuestro programa (Program Counter = 2), y así sucesivamente.

Es vital recordar que el índice no es de líneas sino de instrucciones: el código que genere este índice recorrerá el código de un programa ignorando las primeras líneas vacías o con comentarios hasta encontrar la primer instrucción válida. Registrará su inicio y su longitud, luego buscará la siguiente ignorando saltos y comentarios y sucesivamente.

Por ejemplo, para un script como el siguiente:

```
#!/usr/bin/ansisop
begin
# primero declaro las variables
variables a, b
a = 20
print a
end
```

La instrucción 1 (`variables a, b`) comienza en el byte 57, con una longitud de 14 bytes, mientras que la instrucción 2 (`a = 20`) comienza en el byte 72 y mide 6 bytes. En memoria, se almacenarán así:



Al momento de ejecutar una instrucción, la CPU solicitará a la UMV la entrada del Índice correspondiente a la instrucción a ejecutar, determinada por el Program Counter. Teniendo el byte de inicio y longitud de la instrucción, realizará la petición a la UMV en el segmento de código del Programa.

Al obtener la línea de código a interpretar, la procesará usando el Parser, el cual invocará las Primitivas Ansisop correspondientes.

Índice de Etiquetas

Al preprocesar el código del script, el Parser generará una estructura Índice de Etiquetas que el PLP deberá almacenar en la UMV.

El Índice de Etiquetas constará de la serialización de un diccionario que contiene el

identificador de cada función, procedimiento o etiqueta asociado con la primer instrucción ejecutable del mismo (es decir, el valor que el Program Counter deberá tomar cuando el programa deba pasar a ejecutar ese bloque).

A efectos prácticos, la CPU deberá leer este segmento desde la UMV cada vez que cambie el Programa en ejecución y deserializarlo, de modo de tener un diccionario en memoria mediante el cual poder buscar las direcciones a las que cambiar el Program Counter ante un salto o llamada.

Dado que esta información se mantiene fija durante toda la ejecución del Programa, no debería en ningún momento volver a modificarse, ni habrá necesidad de preservar el diccionario deserializado en memoria.

Tamaño del Contexto Actual

El campo Tamaño del Contexto Actual guardará la cantidad de variables existentes en el contexto de ejecución actual, necesario para poder recrear el Diccionario de Variables al reanudar la ejecución de un Programa.

En cada cambio de contexto, la CPU generará una estructura auxiliar para indexar la ubicación de las variables en el segmento de Stack. Es decir, almacenará que, por ejemplo, que la *variable a* del contexto de ejecución actual se encuentra a partir de cierto offset dentro del Stack, y que la *variable c* de ese contexto está en otro offset determinado.

Cuando un Programa se vuelve a planificar y debe reanudar su ejecución en una CPU, necesita conocer la cantidad de variables locales que debe leer del segmento de Stack. Ese dato se encuentra en el campo Tamaño del Contexto Actual del PCB.

Anexo III – Especificación del Lenguaje AnSISOP

AnSISOP es un lenguaje de programación interpretado de propósito general y bastante bajo nivel. Su sintaxis es simple y, en general, no es muy recomendable para utilizar en tareas productivas. En cambio, su objetivo es principalmente académico: ayudar a entender los conceptos y mecanismos que un sistema operativo debe tener en cuenta para manejar la ejecución de los programas.

El lenguaje se divide en dos capas: la sintaxis de alto nivel, utilizada para escribir los scripts, y las operaciones Primitivas que la CPU deberá ejecutar para llevar a cabo las instrucciones de los primeros. Para la realización del trabajo práctico, la cátedra proveerá un Parser AnSISOP, capaz de interpretar una instrucción de código y requerir la ejecución de las Primitivas necesarias para realizarla. Será responsabilidad de los grupos cumplimentar la interfaz de Primitivas requerida, asegurándose de cumplir en su totalidad y sin alteraciones (modificaciones ni agregados) la especificación de cada una de ellas.

Para la evaluación del trabajo práctico no se proveerán programas con errores de sintaxis ni de semántica.

Sintaxis

- El código principal del programa estará comprendido entre las palabras reservadas *begin* y *end*. *Begin* solo indica por donde comenzará a ejecutar el programa.
- Las sentencias finalizan con un salto de línea. Los saltos adicionales son ignorados.
- Toda línea comenzada por un caracter numeral (#) es un comentario y debe ser ignorado.
- Todo programa y deberá terminar con una línea en blanco.
- Un ":" seguido de una palabra es una etiqueta que será utilizada para permitir saltos dentro del código con *jump, jz* y *jnz*, explicados más adelante.
- Todas las variables dentro de una función son locales.
- Una función puede llamar a otra función.

Variables

Las variables locales se declaran luego de la sentencia *variables*. Son solamente de tipo entero con signo y su identificador es un caracter alfabético [a-zA-Z]. Su valor no debe ser inicializado. Se las indica en el codigo solo con su nombre.

Las variables dadas como parámetros de funciones se nombraran con un único dígito [0-9] y se accedera a ellas en el código como **$#** (siendo # el nombre de la variable).

Las variables compartidas[6] se declaran e inicializan en la configuración del kernel, se nombran como cadenas sin restricción de nombre, y se las indica en el código como **!identificador**.

Asignación

Con el nombre de la variable a la izquierda de un signo igual a una variable se le podrá asignar como valor:

- Un número entero u otra variable (local, parámetro o compartida; no semaforos)
- El resultado de una operación aritmética la cual podrá ser suma o resta

Salto condicional

Las instrucciones de salto condicional *saltar-si-no-es-cero* (jnz - jump on not-zero) y *saltar-si-es-cero* (jz - jump on zero) recibirán como parámetro una variable que evaluarán y una etiqueta a la que deberán saltar en caso de que se cumpla la condición. Estas instrucciones por definición del enunciado tendrán como origen y el destino el procedimiento actual. En otras palabras el código de una función o procedimiento no podrá saltar dentro del código de otro.

[6] Llamamos "*variables compartidas*" a aquellas manejadas por el kernel, de las que todos los CPUs tiene acceso. No son "*variables globales*" ya que "*global*" refiere al scope/contexto de cada programa. No existen en AnSISOP las variables globales.

Este código de ejemplo incrementa la variable i de uno a diez e imprime dichos valores en pantalla.

```
#!/usr/bin/ansisop
begin
variables     i,b
      i = 1
      :inicio_for
      i = i + 1
      print i
      b = i - 10
      jnz b           inicio_for
      #fuera del for
end
```

Observe que si la variable *i* no es igual a 10 entonces b = i - 10 no es cero, entonces la instrucción de salto condicional iría a la etiqueta *inicio_for* hasta llegar a la 10ma iteración, donde no saltará más.

Funciones

La definición de las funciones estará dada por la palabra reservada *function* seguida del nombre de la misma. No hay diferencia sintáctica entre funciones y procedimientos[7].

io

Un caso especial es la función ***io*** la cual recibirá dos parámetros.

1. Una cadena como identificador del sistema del dispositivo de entrada/salida
2. El tiempo que se accederá a este dispositivo de entrada/salida

Impresión en pantalla

Existen dos formas de impresion. *textPrint* seguido de una cadena imprimirá la cadena tal cual aparece en el código fuente. La palabra reservada *print* será utilizada para mostrar el valor de la variable que reciba como parámetro.
La información deberá ser mostrada en la terminal del programa y registrada en el log del sistema.

Ejemplo:

```
a = 0
print a
```

Resultado:

```
VARIABLE a: 0
```

Código de ejemplo

[7] Distinguimos una *función* de un *procedimiento* por la habilidad de retornar un valor.

Este código de ejemplo imprime variables.

```
#!/usr/bin/ansisop
function prueba
      variables a,b
      a = 2
      b = 16
      print b
      print a
      a = a + b
end

begin
variables a, b
      a = 20
      print a
      call prueba
      print a
end
```

Lo que se ve por pantalla sería (Nótese la localidad/scope de las variables):

VARIABLE a: 20

VARIABLE b: 16

VARIABLE a: 2

VARIABLE a: 20

Anexo IV – Stack

Resulta complicado explicar el ***stack*** dado que no necesariamente se asemeja a algún concepto de la realidad que se pueda usar como metáfora, por lo que vamos a describirlo con el mayor detalle posible.

El stack es -*y debe ser programado como*- una estructura de tipo ***pila***, donde los datos se cargan encima de los anteriores, se apilan. Para obtener alguno, el único método disponible es extraer el primero de la pila.

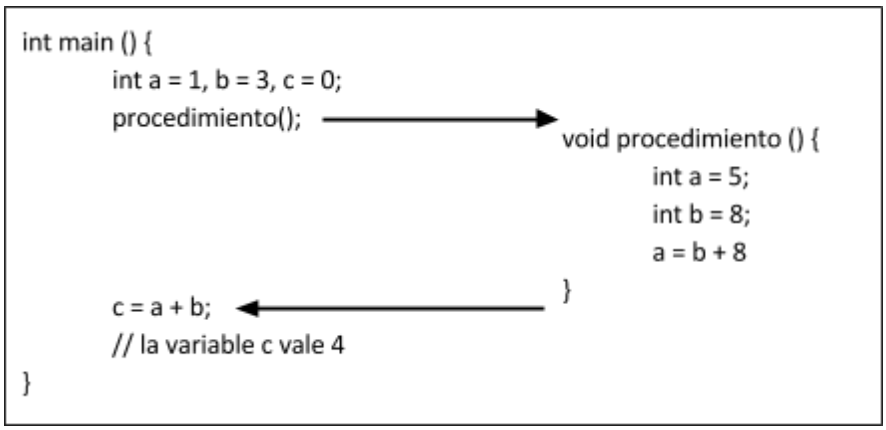
A pesar de que a simple vista daría la sensación de que una pila es una estructura arcaica o poco práctica frente a, por ejemplo, una lista ordenada o un array, en realidad la simplicidad del hecho de que su acceso esté restringido a **apilar** y **sacar** -o ***push*** y ***pop***, como se lo conoce generalmente y como lo vamos a mencionar en el enunciado- es lo que hace que este tipo de estructura sea la herramienta indicada para implementar un ***stack***.

El stack es un segmento utilizado para describir entornos de trabajo.

En un programa estructurado sabemos que, al ingresar a un procedimiento o función, las

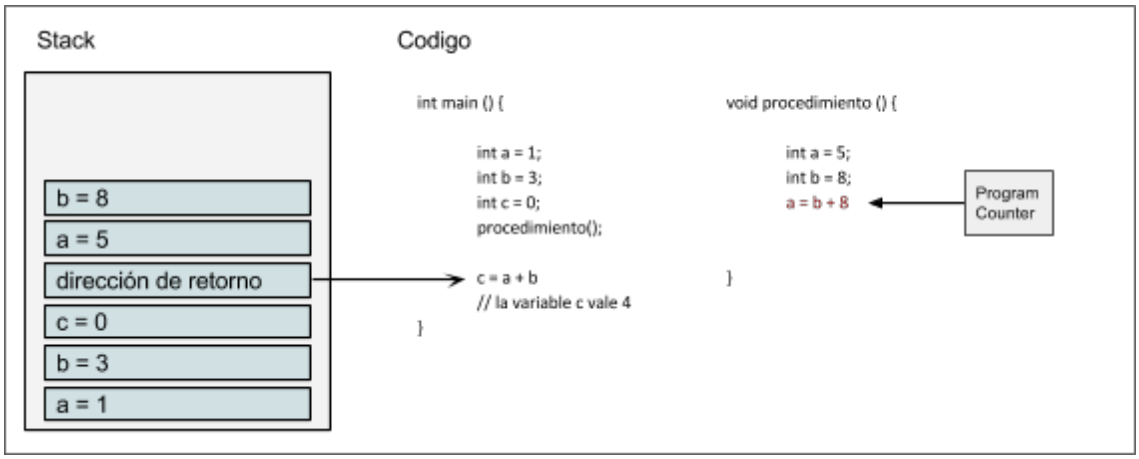
variables que estén definidas dentro de estos existirán hasta que concluya. Luego, el programa ***regresará*** a la instrucción posterior a la invocación de esa función o procedimiento.

Por ejemplo **(Ejemplo en C, no en Ansisop)**:



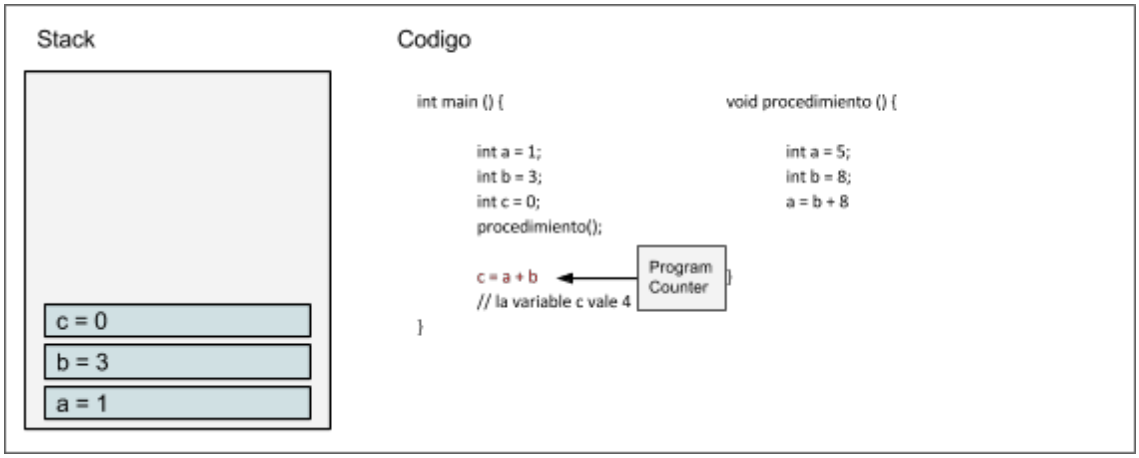
En la figura se observa que a pesar que las variables a y b existen en procedimiento() su alcance (scope) es solo dentro de él. Al regresar del procedimiento() las variables a, b y c de main() permanecen inalteradas.

Las variables se mantienen por scope porque **internamente** el sistema operativo crea un *contexto* nuevo al ingresar al procedimiento. Apila las variables nuevas en el stack y la dirección de retorno del procedimiento que lo invocó.



En la figura se ve claramente como al ingresar al procedimiento() el sistema operativo almacena en el stack la dirección donde debe volver al concluir.

Al salir de un procedimiento, el sistema operativo desapila las variables locales y la dirección de retorno la cual se la asigna al *Program Counter*.



En la figura se ve como el Programa tiene nuevamente las variables del procedimiento main()

No lo detallamos en este ejemplo, pero se entiende que las sucesivas invocaciones de un procedimiento a otro irán apilando las variables locales en el ***stack*** y des apilándolas al terminar. Si en vez de un procedimiento se tratara de una función, el resultado que devuelve la función también se apila en el stack.

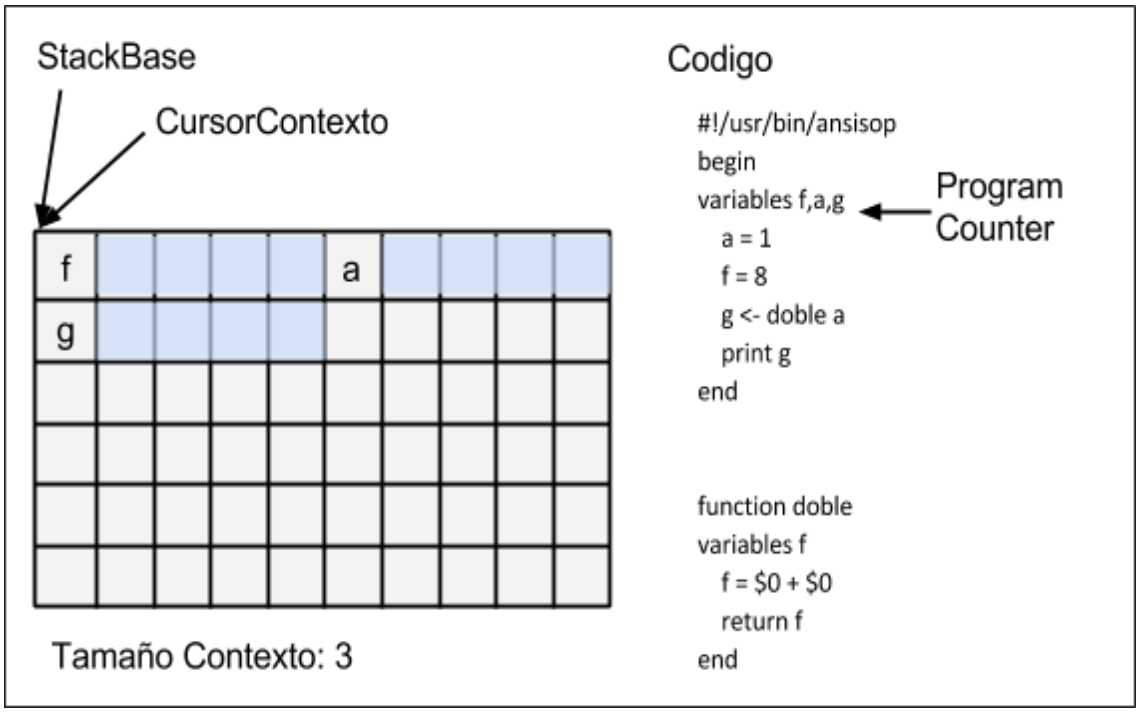
El stack, como toda estructura en memoria, tiene un tamaño determinado, fijo. En este trabajo práctico, este tamaño son 100 bytes. El lector comprenderá que podría darse la situación donde luego de una serie de procedimientos invocando otros procedimientos, **el stack se llene**. Más adelante detallaremos al respecto.

Manejando el Stack en el Trabajo Práctico

El segmento de Stack es inicializado por el PLP, encargado también de inicializar el puntero al mismo en el PCB. Además, el PLP debe inicializar el puntero del contexto actual para que coincida con el inicio del stack, y setear el contador de variables locales a 0.

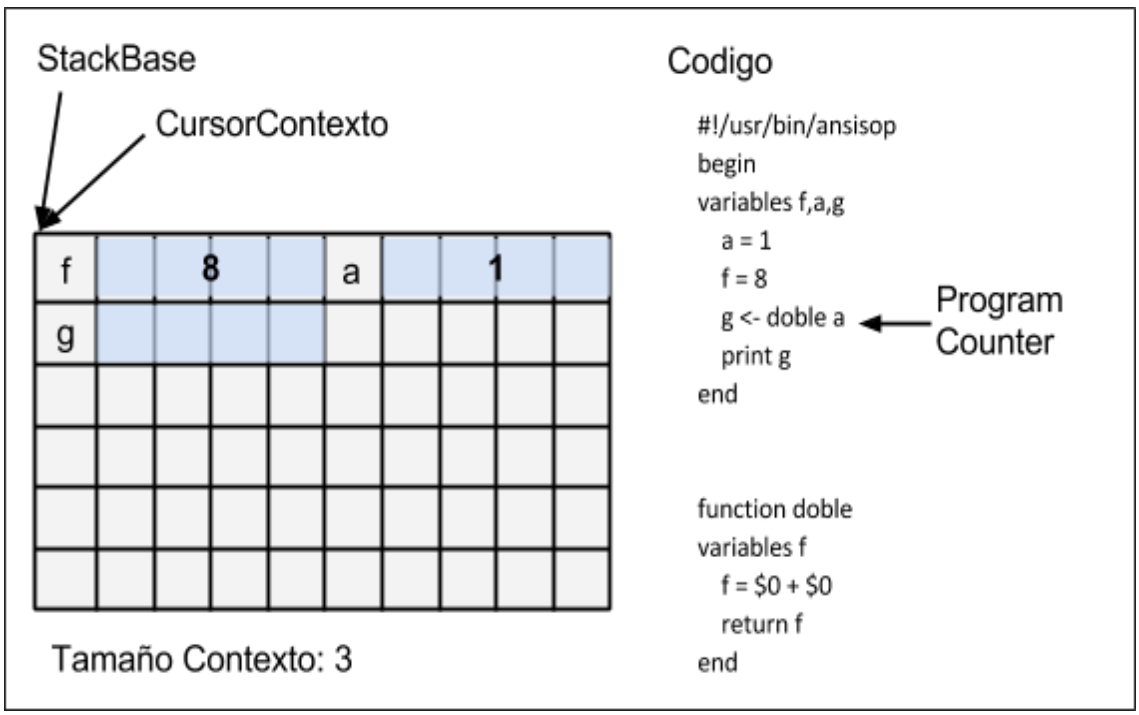
Al ejecutar una instrucción `variables`, la CPU deberá reservar la memoria para las variables locales del contexto que está comenzando a ejecutar. Por cada variable, la CPU escribirá en memoria un caracter con el nombre de la variable, y reservará los 4 siguientes para el valor de la misma. Dicho valor **no deberá inicializarse**.

De este modo, al terminar la ejecución del primer `variables` del programa, el stack contendrá las variables locales almacenadas de forma consecutiva. El Cursor del Contexto Actual seguirá apuntando al comienzo del segmento del stack, porque ahí es donde comienza el Contexto Actual. Además, la CPU contará con su Diccionario de Variables indicándole en qué posición de la memoria encontrar cada variable.

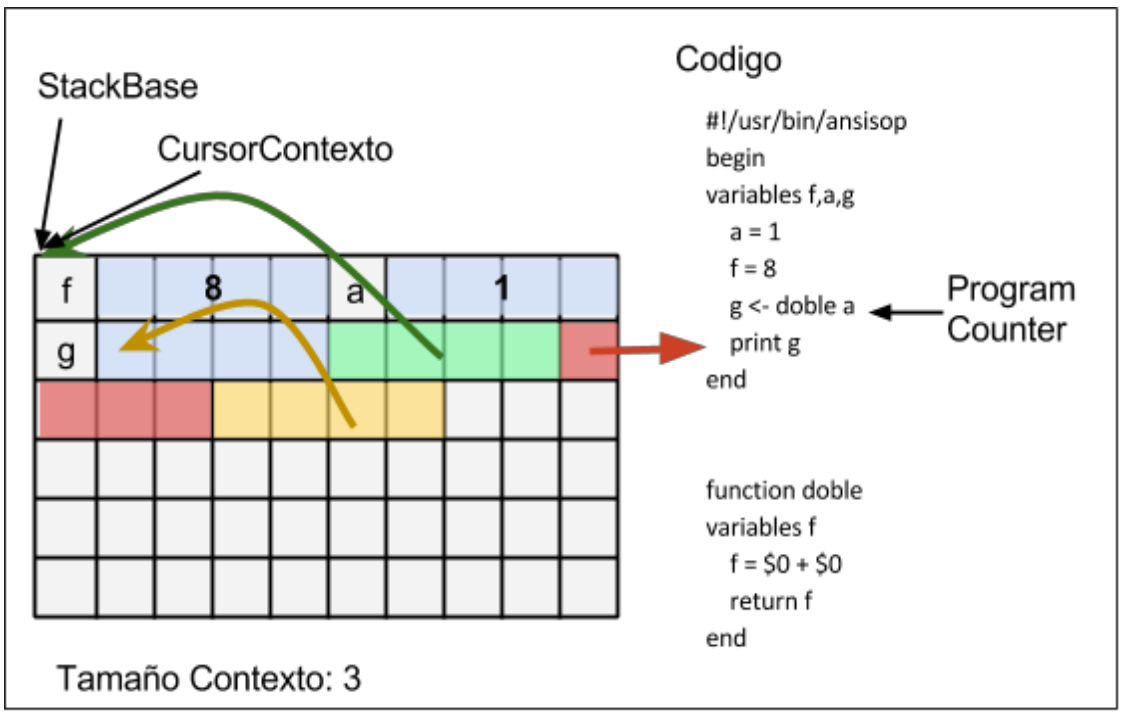


Al llamar a una función o procedimiento, la CPU deberá cambiar el contexto de ejecución actual del programa: las referencias a variables dejarán de ser sobre las actuales, y pasarán a ser sobre las propias del nuevo contexto.

Pero, además de cambiar el contexto, el Program Counter va a variar, y debe recordar a qué instrucción volver luego de concluir la ejecución del bloque de código al que saltó. Por último, debe recordar en qué posición del stack comienza el contexto que está abandonando, para poder recrearlo al volver, y, si fuera a ejecutar una función, debe recordar en qué posición de memoria debe asignar el valor que la función retorne.



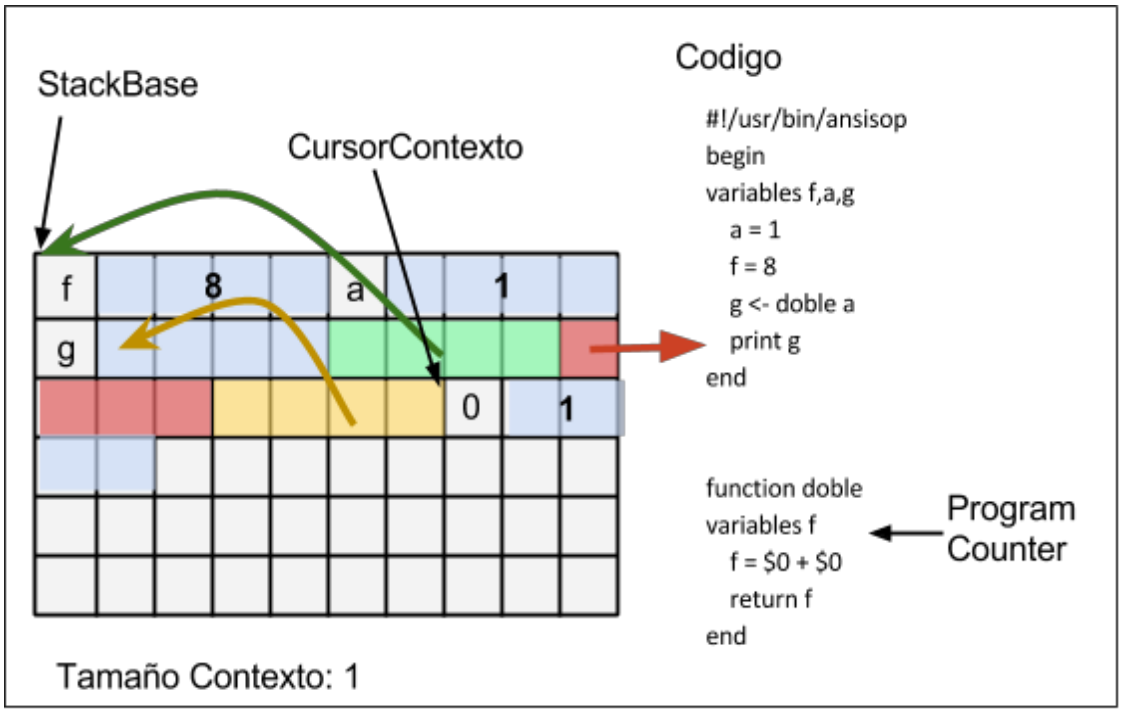
Al llamar a una función, el parser ejecutará la Primitiva `llamarConRetorno`, mientras que la invocación a un procedimiento se resolverá con `llamarSinRetorno`. Conociendo el Cursor del Contexto Actual y las variables locales a ese contexto, la CPU podrá determinar dónde termina el contexto actual. A partir de esa posición, estas llamadas deberán almacenar, en este orden, el Cursor de Contexto Actual actual, el Program Counter actual incrementado en uno (es decir, la próxima instrucción a ejecutar al retornar del nuevo contexto), y, si correspondiera, la dirección en que se deberá almacenar el retorno de la función a ejecutar. Con esto, preservaría el contexto actual para poder regenerarlo al retornar.



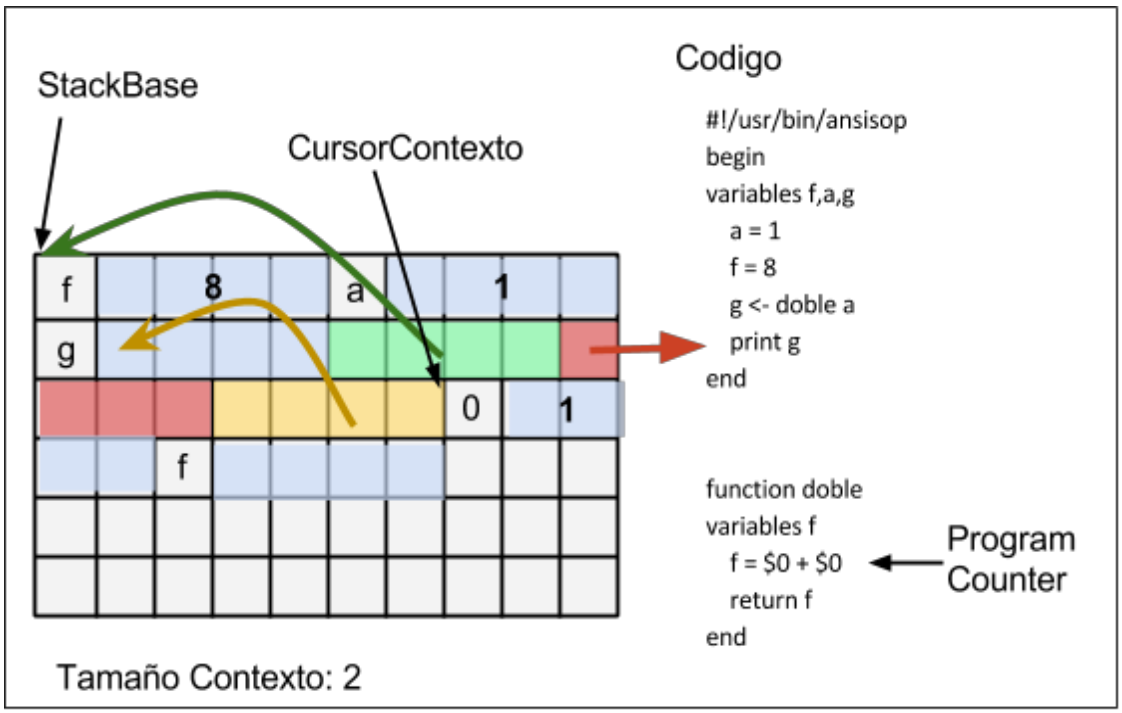
Para generar el nuevo contexto de ejecución, la CPU actualizará el valor del Cursor de Contexto Actual para que apunte al final de los datos que acaba de apilar y actualizará el Program Counter al nuevo valor, parámetro de ambas Primitivas. Además, la CPU vaciará su Diccionario de Variables actual, dado que las referencias que contiene ya no pertenecen al contexto actual.

Si la función o procedimiento recibiera parámetros, éstos se definirán inmediatamente después con llamadas a `definirVariables` (tal como lo harían las variables locales) con identificadores numéricos [0-9]. Como los parámetros serán variables a las que el bloque de código tiene acceso, la CPU los indexará en el Diccionario de Variables y, como última operación de la llamada, actualizará el valor del Tamaño del Contexto Actual a la cantidad de parámetros recibidos[8].

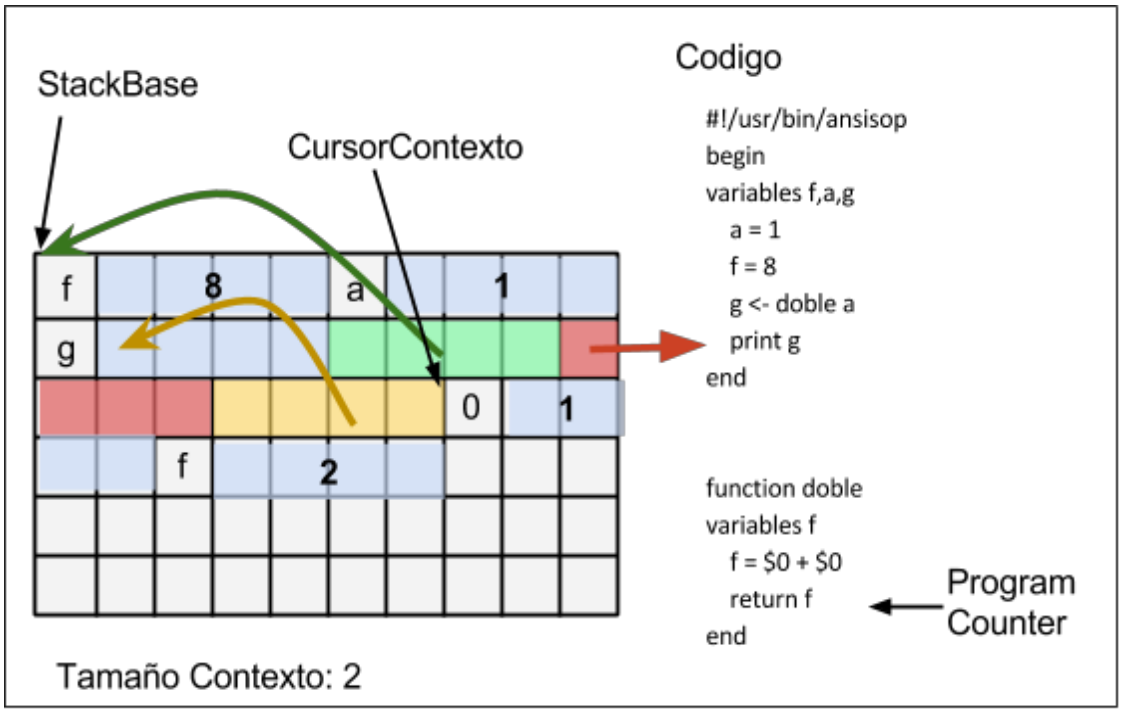
[8] Debería sobrar la aclaración de que 0 es una cantidad válida de parámetros recibidos



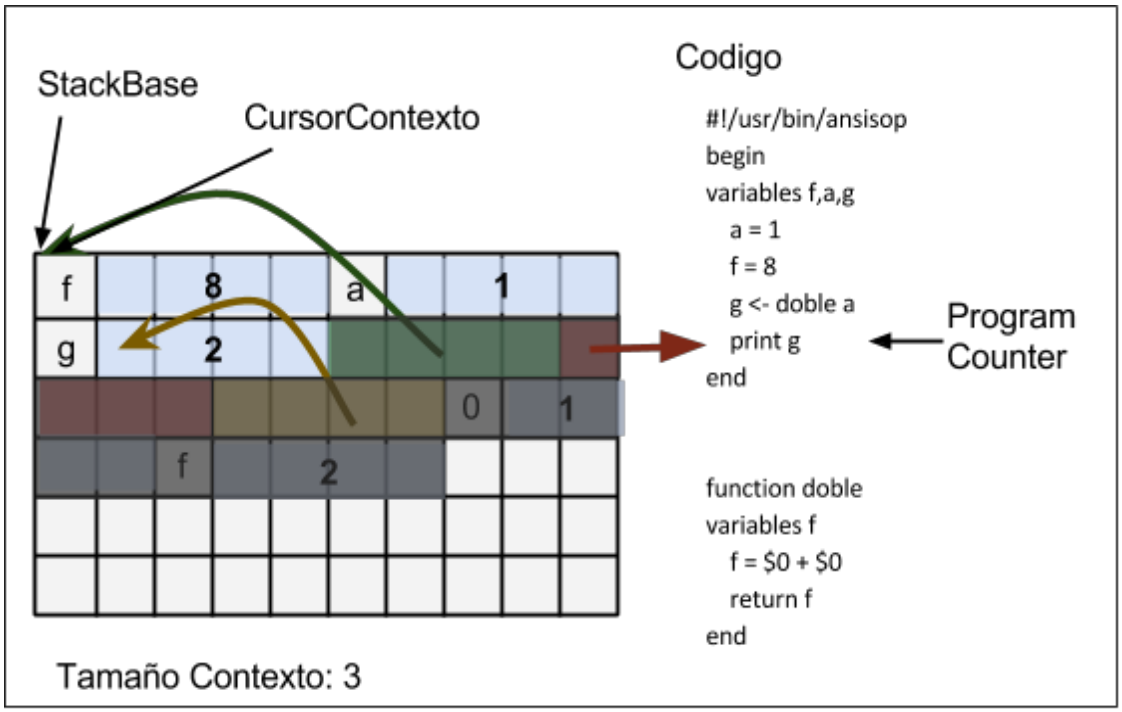
La primer instrucción ejecutable del bloque será su "`variables f`". Entonces, como ya se especificó, se reservará el espacio para las variables locales que el bloque declare, y se actualizará el Diccionario de Variables y Tamaño del Contexto Actual en consecuencia.



Las funciones terminan su ejecución con un `return`, que se resolverá con la Primitiva `retornar`.



La CPU desapilará los cuatro bytes previos al Cursor del Contexto Actual, y de ese modo obtendrá la dirección de memoria en que tendrá que almacenar el valor de retorno. En los 4 bytes anteriores encontrará el valor que deberá tomar el Program Counter luego de retornar, y los 4 anteriores a esos indicarán el nuevo Cursor del Contexto. Conociendo el Cursor del Contexto al que tiene que volver, la CPU sabrá que todos los bytes entre la dirección que acaba de leer y el valor que acaba de leer será un array de elementos de 5 bytes, cada uno con el identificador y valor de una variable del contexto al que se está regresando. Una vez regenerado el contexto, la CPU está en condiciones de ejecutar la próxima instrucción, como acostumbra.



En el caso de los procedimientos, su finalización está determinada al llegar a una instrucción de end. El comportamiento será similar, con la diferencia de que no tendrá dirección donde guardar el retorno para desapilar dado que, justamente, no retorna ningún valor.

Como puede observarse en el último gráfico, los datos del contexto que se acaba de abandonar permanecen en la memoria; es decir, no se sobrescribe el contexto al abandonarlo. Como el PCB contiene el tamaño del contexto, sabe hasta dónde debe leer los datos actuales, y que lo que sigue es memoria sin reservar, por lo que no debe utilizarla.

Cambio de Proceso

Los Programas en ejecución podrán abandonar la CPU para pasar a otro estado dentro del sistema para replanificarse luego. Al volver a ejecutarse, podría hacerlo en una CPU distinta a la que lo había ejecutado previamente.

Casi la totalidad de los datos del Programa se encuentran en la UMV, y los restantes en el PCB. Al comenzar a ejecutar el Proceso, la CPU se valdrá del Cursor del Contexto Actual y el Tamaño del Contexto Actual para poder regenerar su Diccionario de Variables.

Como la CPU actualiza su copia del PCB durante la ejecución, es vital que cuando el Proceso abandona la CPU, ésta deberá enviar al Kernel el nuevo estado del PCB del Proceso, a fin de actualizar la copia que aquel maneja.

Anexo V - Parser de AnSISOP

Para evitar la complejidad que presenta realizar un analizador de sintaxis y dado que estas tareas no son inherentes al contenido de la materia se facilita un Parser que se encargará de interpretar cada línea de código y de ejecutar las Primitivas correspondientes, cuyo código será desarrollado por el alumno.

El parser podrá obtenerse desde https://github.com/sisoputnfrba/ansisop-parser

Primitivas de Ansisop

1. `definirVariable`
2. `obtenerPosicionVariable`
3. `dereferenciar`
4. `asignar`
5. `obtenerValorCompartida`
6. `asignarValorCompartida`
7. `irAlLabel`
8. `llamarSinRetorno`
9. `llamarConRetorno`
10. `finalizar`
11. `retornar`
12. `imprimir`
13. `imprimirTexto`
14. `entradaSalida`
15. `wait`
16. `signal`

1) definirVariable

Reserva en el Contexto de Ejecución Actual el espacio necesario para una variable llamada identificador_variable y la registra tanto en el Stack como en el Diccionario de Variables, retornando la posición del valor de esta nueva variable del stack

El valor de la variable queda indefinido: no deberá inicializarlo con ningún valor default.

Esta función se invoca una vez por variable, a pesar de que este varias veces en una línea. Por ejemplo, evaluar "variables a, b, c" llamará tres veces a esta función con los parámetros "a", "b" y "c"

```
t_puntero definirVariable(t_nombre_variable identificador_variable );
```

2) obtenerPosicionVariable

Devuelve el desplazamiento respecto al inicio del segmento Stack en que se encuentra el valor de la variable `identificador_variable` del contexto actual. En caso de error,

retorna -1.

```
t_puntero obtenerPosicionVariable(t_nombre_variable
     identificador_variable );
```

3) dereferenciar

Obtiene el valor resultante de leer a partir de direccion_variable, sin importar cual fuera el contexto actual

```
t_valor_variable dereferenciar(t_puntero direccion_variable);
```

4) asignar

Copia un valor en la variable ubicada en `direccion_variable`.

```
void asignar(t_puntero direccion_variable, t_valor_variable valor )
```

5) obtenerValorCompartida

Solicita al kernel el valor de una variable compartida.

```
t_valor_variable obtenerValorCompartida(t_nombre_compartida
variable)
```

6) asignarValorCompartida

Solicita al kernel asignar el valor a la variable compartida. Devuelve el valor asignado.

```
t_valor_variable asignarValorCompartida(t_nombre_compartida
     variable, t_valor_variable valor)
```

7) irAlLabel

Devuelve el número de la primer instrucción ejecutable de etiqueta y -1 en caso de error.

```
t_puntero_instruccion irAlLabel(t_nombre_etiqueta etiqueta)
```

8) llamarSinRetorno

Preserva el contexto de ejecución actual para poder retornar luego. Modifica las estructuras correspondientes para mostrar un nuevo contexto vacío. Retorna el numero de instrucción a ejecutar.

Los parámetros serán definidos luego de esta instrucción de la misma manera que una variable local, con identificadores numéricos empezando por el 0.

```
t_puntero_instruccion llamarSinRetorno(t_nombre_etiqueta etiqueta,
     t_puntero_instruccion linea_en_ejecuccion)
```

9) llamarConRetorno

Preserva el contexto de ejecución actual para poder retornar luego al mismo, junto con la posicion de la variable entregada por `donde_retornar` . Modifica las estructuras correspondientes para mostrar un nuevo contexto vacío. Retorna el número de instrucción a ejecutar.

Los parámetros serán definidos luego de esta instrucción de la misma manera que una variable local, con identificadores numéricos empezando por el 0.

No se pretende que se pueda retornar a una variable global. Sí a un parámetro o variable local

```
t_puntero_instruccion llamarConRetorno(t_nombre_etiqueta etiqueta,
     t_puntero donde_retornar,
     t_puntero_instruccion linea_en_ejecuccion)
```

10) finalizar

Cambia el Contexto de Ejecución Actual para volver al Contexto anterior al que se está ejecutando, recuperando el Cursor de Contexto Actual y el Program Counter previamente apilados en el Stack. En caso de estar finalizando el Contexto principal (el ubicado al inicio del Stack), deberá finalizar la ejecución del programa devolviendo el valor -1.

```
t_puntero_instruccion finalizar(void)
```

11) retornar

Modifica el Contexto de Ejecución Actual por el Contexto anterior al que se está ejecutando, recuperando el Cursor de Contexto Actual, el Program Counter y la direccion donde retornar, asignando el valor de `retorno` en esta, previamente apilados en el Stack.

```
t_puntero_instruccion retornar(t_valor_variable retorno)
```

12) imprimir

Envía al Kernel el contenido de `valor_mostrar`, para que este le reenvíe a la correspondiente consola del Programa en ejecución. Devuelve la cantidad de dígitos impresos.

```
int imprimir(t_valor_variable valor_mostrar)
```

13) imprimirTexto

Envía al Kernel una cadena de `texto` para que este la reenvíe a la correspondiente consola del Programa en ejecución. No admite parámetros adicionales, secuencias de escape o variables. Devuelve la cantidad de dígitos impresos.

```
int imprimirTexto(char* texto)
```

14) entradaSalida

Informa al kernel que el Programa actual pretende utilizar el `dispositivo` durante `tiempo` unidades de tiempo.

```
int entradaSalida(t_nombre_dispositivo dispositivo, int tiempo)
```

15) wait

Informa al kernel que ejecute la función wait para el semáforo con el nombre `identificador_semaforo`. El kernel deberá decidir si bloquearlo o no.

```
int wait(t_nombre_semaforo identificador_semaforo)
```

16) signal

Comunica al kernel que ejecute la función signal para el semáforo con el nombre `identificador_semaforo`. El kernel decidirá si esto conlleva desbloquear a otros procesos.

```
int signal(t_nombre_semaforo identificador_semaforo)
```

Descripción de las entregas

Para permitir una mejor distribución de las tareas y orientar al alumno en el proceso de desarrollo de su trabajo, se definieron una serie de puntos de control y fechas que el alumno podrá utilizar para comparar su grado de avance respecto del esperado.

Checkpoint 1

Fecha: 26 de Abril

Objetivos:

- ★ Familiarizarse con Linux y su consola, el entorno de desarrollo y el repositorio
- ★ Aplicar las Commons Libraries, principalmente las funciones para listas, archivos de conf y logs.
- ★ Implementar el Parser de AnSISOP y modelar algunas de las funcionalidades
- ★ Modelar las estructuras necesarias para gestionar los segmentos de los Programas
- ★ Desarrollar un modelo de consola para la UMV
- ★ Desarrollar un binario que sirva como intérprete de un script en Linux

Lectura recomendada:

http://faqoperativos.com.ar/arrancar
Beej Guide to Network Programming - link
Linux POSIX Threads - link
SisopUTNFRBA Commons Libraries - link

Checkpoint 2

Fecha: 10 de Mayo

Objetivos:

- ★ Realizar dos programas que permitan recibir múltiples conexiones por sockets y gestionarlas utilizando por un lado un multiplexor de entrada/salida (select, epoll) y por el otro delegar su atención a hilos independientes.
- ★ Realizar un cliente simple que permita enviar mensajes estructurados a los servidores
- ★ Utilizar semáforos para sincronizar hilos que acceden a una lista compartida
- ★ Diseñar el diagrama de estados de un PCB en el sistema
- ★ Diseñar el impacto de las operaciones sobre las estructuras del Programa
- ★ Implementar el algoritmo First-Fit y la compactación en la UMV

Lectura recomendada:

Sistemas Operativos, Silberschatz, Galvin - Capítulo 4: Hilos
Sistemas Operativos, Silberschatz, Galvin - Capítulo 5: Planificación del Procesador

Checkpoint 3

Fecha: 31 de Mayo

Objetivos:

- ★ Implementar la interfaz de mensajes de la UMV y su correspondiente validación
- ★ Implementar operaciones básicas AnSISOP en el CPU en las estructuras de la UMV
- ★ Desarrollar los hilos de entrada/salida
- ★ Implementar el algoritmo Worst-Fit en la UMV

Checkpoint 4 - Presencial

Fecha: 14 de Junio

Objetivos:

- ★ Implementar las funciones y procedimientos de AnSISOP utilizando el stack
- ★ Implementar io
- ★ Desarrollar la planificación completa de un Programa en el sistema.

Checkpoint 5

Fecha: 5 de Julio

Objetivos:

- ★ Implementar las listas compartidas y los semáforos de Ansisop
- ★ Validar los requisitos funcionales del trabajo práctico.
- ★ Realizar pruebas de stress en el sistema.
- ★ Programar los Makefiles correspondientes y hacer pruebas de funcionamiento en el laboratorio en la VM server.