Determinación y Restauración Automática de los Estados Intermedios de una aplicación

Marco Mercinelli

Laboratorio de Electrónica Digital
Universidad Católica Nuestra Señora de la Asunción
Asunción - Paraguay

Los sistemas operativos modernos permiten gestionar aplicaciones muy complejas. Si es necesario una aplicación puede ser desarrollada para un conjunto de varios procesos (programas en ejecución) que se comunican entre ellos y que pueden ser ejecutados en distintas computadoras de una red. En este ambiente aparece importante la disponibilidad de un mecanismo para "congelar" y despues "restaurar" el estado de ejecución de un proceso y sus relaciones con el ambiente. En efecto un mecanismo de este tipo puede ser usado para la realización de muchas funcionalidades de alto nivel como mecanismos para aumentar la fiabilidad de un sistema, para la "migración" de procesos, para el *debugging* de aplicaciones distribuidas y para nuevos algoritmos de *backtracking*.

Este artículo describe los problemas técnicos y prácticos enfrentados en el desarrollo de un mecanismo de este tipo para un sistema operativo standard largamente utilizado, como es el UNIX™†. Este mecanismo, llamado de *snapshot* y *restart*, permite a los usuarios, bajo algunas hipótesis, de "congelar" y escribir en archivos el estado de un proceso en ejecución y de retomar la ejecución de aquel proceso, desde el punto del "congelamiento", en cualquier momento posterior, también despues de una parada de la computadora.

† UNIX es un nombre registrado por los Laboratorios Bell

1. Introducción

El principal objetivo de este proyecto era el de explorar la factibilidad de un mecanismo automático para extraer informaciones sobre el estado y el ambiente de ejecución de un proceso, en una forma bastante completa para poder restaurar el proceso en el mismo estado en un ambiente y/o en un tiempo distinto.

Un mecanismo de este tipo, fiable y eficiente, puede ser usado como "bloque de construcción" para facilitar y acelerar el desarrollo de nuevas aplicaciones y funcionalidades. Algunos ejemplos de los posibles campos de aplicación son los siguientes:

a - Es posible aumentar la fiabilidad de un sistema, guardando periodicamente en memoria estable el estado de ejecución de eventuales procesos "delicados". En caso de necesidad será posible restaurar la ejecución de los procesos a partir del último estado guardado.

b - Los sistemas operativos más avanzados afrontan a menudo el problema de la subdivisión y del balance de la carga de trabajo entre computadoras conectadas en red. Una de las técnicas propuestas es la utilización de mecanismos para hacer "migrar" los procesos de una computadora a otra de la red que tenga menor carga de trabajo. Estos mecanismos se pueden realizar "congelando" un proceso, transfiriendo las informaciones "congeladas", con todos los datos necesarios, a otra computadora de la red menos cargada y restaurando entonces su ejecución en la otra computadora.

c - Algunos algoritmos preveen la posibilidad del *backtracking*, es decir un programa puede retornar a un estado de ejecución alcanzado anteriormente, modificar el valor de algunas variables y retomar la ejecución, pero siguiendo un camino diferente. La disponibilidad de un mecanismo automático para reconstruir un estado precedente de la aplicación puede ayudar en la implementación de estos algoritmos, sobre todo cuando es muy complejo volver a un estado precedente del algoritmo.

d - Algunas técnicas nuevas para el *debugging*, particularmente en ambientes distribuidos, están basadas en la posibilidad de extraer y examinar el estado de los diferentes procesos envueltos en una aplicación sin interferir con su ejecución.

Se optó por desarrollar la investigación sobre UNIX™ porque es un sistema operativo bien conocido, cuyas fuentes están disponibles y son modificables y que se está afirmando como standard sobre todo para las *workstations* y los mini computadores. Además UNIX™ tiene muchas caracteristicas similares con otros sistemas operativos de la misma generación permitiendo la generalización de los resultados de la investigación.

Hemos llamado en general *snapshot* al conjunto de informaciones que es necesario guardar para definir de modo unívoco la ejecución de un proceso. El *snapshot* es una copia en disco del ambiente de ejecución del proceso y comprende una imágen de la memoria, el valor de los registros, el estado de los archivos en uso, el *directory* de trabajo corriente y otras informaciones sobre las relacciones con el sistema y con otros processos. La creación de un *snapshot* no debe modificar en modo alguno el comportamiento del proceso, pudiendo él continuar su ejecución como si nada hubiese pasado. Las informaciones guardadas deben ser suficientes para restaurar la ejecución del proceso, de manera que pueda continuar en manera totalmente transparente desde el punto en que fué efectuado el *snapshot*. En la fase de *restart* todos los recursos utilizados por el proceso tienen que estar disponibles de nuevo y sus estados deben ser adecuadamente restaurados.

El trabajo de investigación mostró que, por algunas razones que serán ilustradas seguidamente, un mecanismo de *snapshot* y *restart* aplicable en forma totalmente general y automática no es realizable en un sistema operativo como el UNIX™. Es necesario entonces definir unas hipótesis para caracterizar el tipo de los procesos previstos y, en algunos casos, direccionar el mecanismo de *restart* según informaciones adjuntas por el usuario. Además surgen múltiples dificultades cuando se quiere tratar grupos de procesos cooperantes, sobre todo a causa de los problemas asociados con la definición del estado de una aplicación

distribuida.

A pesar de estas dificultades se realizó un mecanismo experimental que puede "congelar" y "restaurar", en forma completa y transparente, el estado de cualquier proceso que no tenga comunicaciones abiertas con otros procesos (por lo menos en el momento del *snapshot*), no tenga relacciones con el ambiente codificadas en forma implicita en el algoritmo ejecutado y no tenga dependencias del tiempo de ejecución. Este mecanismo es suficientemente general para tratar ya muchas aplicaciones interesantes.

La extensión del sistema UNIX™ para realizar el mecanismo de *snapshot* y *restart* fué desarrollado sobre una *workstation* SUN 2/120 en el ambito de una colaboración con el centro de investigación italiano CSELT y el Departamento de *Computer Science* de la Universidad de California en Berkeley.

2. Determinabilidad del estado de ejecución de una aplicación

El conjunto de informaciones necesario para definir en forma completa un proceso en ejecución está constituida por su código, que normalmente no es modificable y está definido por el contenido del archivo ejecutable, por el contenido de la parte de datos y del *stack* que, en cambio, son modificadas durante la ejecución, por el contenido y estado de los archivos usados, por el estado de los recursos del sistema correlacionadas en alguna forma con la ejecución del proceso, por las informaciones relativas a las relaciones del proceso con otros procesos residentes en la misma computadora o además de ellos ejecutados en otras computadoras conectadas en red, por el estado y la configuración actual de la computadora y eventualmente de toda la red donde se inserta.

Las informaciones necesarias para componer un *snapshot* "completo" pueden, entonces, ser muchas y no facilmente encontrables. Hay también relaciones con otros procesos y con el ambiente de ejecución que en general no pueden ser deducidas en forma autómatica de los datos del sistema porque están implicadas y codificadas en el particular algoritmo ejecutado por el proceso. Como ejemplos las relaciones entre el interprete de comandos de UNIX™ (*shell*) y los procesos que genera son codificadas en el programa mismo; otros programas, como los dos instrumentos de comunicación de UNIX™, *uucp* y *tip*, sincronizan sus accesos a las lineas de comunicación creando unos *lock files*, cuya existencia es obviamente parte del ambiente de ejecución de los programas, aunque la relación con ellos no sea ni explícita ni deducible autómaticamente con un mecanismo general.

Por consiguiente parece no posible, en forma automática, realizar un mecanismo totalmente general de *snapshot* y *restart* en grado de gestionar todas las posibles aplicaciones que los procesos pueden ejecutar en un sistema complejo como el UNIX™. Entonces es necesario hacer algunas hipótesis para caracterizar la clase de procesos que deben ser considerados, de manera a poder definir exactamente y limitar las informaciones necesarias para formar un *snapshot* consistente de estos procesos.

3. Hipótesis para el mecanismo de *snapshot* y *restart*

Las hipótesis de trabajo utilizadas para realizar un primer mecanismo autómatico de *snapshot* y *restart* son las siguientes:

1 - el ambiente de ejecución y las relaciones de un proceso debe ser determinable totalmente, con un método de tipo general, a partir del contenido de las estructuras de datos del sistema operativo;

2 - el algoritmo ejecutado por el proceso no debe ser dependiente del tiempo, es decir puede ser interrumpido por un período de tiempo de cualquier longitud sin provocar problemas de temporización.

Hemos entonces clasificado las aplicaciones que satisfacen a las hipótesis precedentes en cuatro "clases". Para cada una de estas clases las informaciones necesarias para formar un *snapshot* de los procesos envueltos en la aplicación son limitadas y bien determinadas. Cada clase fué definida como un subconjunto de la precedente de manera que el mecanismo de *snapshot* y *restart* para una clase pudiese ser realizado como una extensión del mecanismo para la precedente:

A) aplicaciones ejecutadas por un solo proceso que no utiliza los mecanismos de comunicación entre procesos, y no utiliza archivos o particulares dispositivos de E/S sino solo los canales standard de input, output y error de UNIX™ conectados a la terminal del usuario;

B) aplicaciones ejecutadas por un solo proceso que utiliza archivos y/o dispositivos de E/S y puede tener necesidad de setear los dispositivos de E/S en una forma particular (por ejemplo la supresión del eco de caracteres en una línea de transmisión);

C) aplicaciones ejecutadas por una o más jerarquías de procesos (un proceso "padre" y sus "descendientes") cada uno del tipo descrito en la clase B, que residen en la misma computadora y se comunican entre ellos, pero usando solo los mecanismos standard de comunicación entre procesos previstos por sistema operativo (IPC);

D) aplicaciones ejecutadas por muchas jerarquías de procesos, del tipo de las descritas en la clase C, que pueden ser ejecutadas en paralelo con varias computadoras de un sistema distribuido o varios procesadores de un sistema con multiprocesador, y que se pueden comunicar a través de los mecanismos standard de comunicación entre procesos (IPC) o por medio de memoria compartida.

Las aplicaciones de clase A no tienen interés desde el punto de vista práctico pero representan un punto de partida conveniente para el desarrollo del mecanismo de *snapshot* y *restart*.

El mecanismo implementado actualmente gestiona las aplicaciones de la clase B y ya puede tratar muchos casos de interés práctico. En el resto del artículo se describe la implementación de este mecanismo.

Las aplicaciones de clase C necesitan de un mecanismo más complejo en cuanto requieren guardar y restaurar informaciones que se refieren a las relaciones entre los procesos. Además es necesario poner particular atención en evitar que alguno de los procesos continúe su ejecución antes que el *snapshot* completo haya sido escrito en memoria estable, pudiendo esto hacer inconsistentes partes de las informaciones guardadas.

Las aplicaciones de clase D son las más difíciles de gestionar principalmente por dos razones: es muy complicado reconstruir el estado de los protocolos de comunicación, y además no es posible aplicar el mecanismo de *snapshot* "exactamente" al mismo tiempo a los procesos en ejecución en máquinas diferentes, a causa de la casualidad de los retardos de transmisión a través de la red. Entonces, según como sea efectuado el *snapshot*, puede no ser posible guardar un estado global consistente de los procesos envueltos en la aplicación. Una metáfora creada por los investigadores Chandy y Lamport ilustra claramente este problema:

> *El algoritmo para guardar el estado de los procesos en un ambiente distribuido puede ser comparado con un grupo de fotógrafos que observan una escena muy grande y dinámica como un cielo cubierto por una bandada de pájaros migradores. La escena es tan grande que no puede ser completamente captada por una sola fotografía. Los fotógrafos deberán tomar muchas fotografías (snapshots) al mismo tiempo y luego juntarlas para poder tener una imagen de toda la escena. Con todo eso, las fotografías no pueden ser captadas "exactamente" en el mismo instante, a causa de la dificultad de sincronizar la acción de los fotógrafos. Además los fotógrafos no pueden "perturbar" el proceso que están fotografiando, por ejemplo no pueden hacer de manera que todos los pájaros permanezcan inmóviles en el cielo mientras son captadas las fotografías. Sin embargo si se quisiera que la imagen final fuese "significativa", es decir describa el fenómeno en modo completo y adecuado. El problema es definir que quiere decir "significativo" en un cierto contexto y entonces determinar como las fotografías deben ser realizadas.*

Chandy y Lamport describieron un algoritmo que permite obtener un estado global consistente. Además Dave Presotto, investigador de los Laboratorios Bell identifica los vinculos, que respetandolos, permiten verificar la consistencia de un estado global. Estos y otros trabajos de investigación razonablemente hacen suponer poder realizar en el futuro un mecanismo suficientemente general también para las aplicaciones de clase D.

4. La gestión de los procesos en UNIX™

El usuario de un sistema UNIX™ considera los procesos como entidades "activas" que ejecutan cada uno una secuencia propia de instrucciones. Cada proceso lleva adelante la tarea asignádale en forma concurrente con todos los otros procesos activos del sistema. A cada proceso está asociado un número, llamado identificador de proceso (*pid*), que es único entre todos los identificadores de los procesos presentes en el sistema en un cierto instante.

En cambio, desde el punto de vista del sistema operativo un proceso es una entidad "inactiva", formada por varias áreas de memoria y estructuras de datos (que forman la "imagen" del proceso para el sistema operativo) que son manipuladas por las rutinas de su *kernel* y ejecutadas por el procesador. El sistema operativo controla al procesador, haciendolo pasar muy frecuentemente por la ejecución de una "imagen" a otra, de manera a dar a cada proceso la posibilidad de llevar adelante la tarea asignádale.

La "imagen" de un proceso está compuesta por un conjunto de informaciones que comprenden las estructuras de datos del sistema operativo que se refieren directamente al proceso, más una parte llamada "parte de usuario" formada por código, datos, *stack* y contenido de los registros del programa en ejecución. El sistema operativo utiliza unas cuantas estructuras de datos para identificar un proceso y los recursos que utiliza. Algunas de estas estructuras contienen datos globales, que deben ser siempre accesibles por el sistema aún cuando el proceso está temporalmente suspendido (por ejemplo tabla de procesos activos, tabla de segmentos de códigos compartidos, etc.). Cada proceso tiene además una estructura de datos asociada llamada impropiamente *User Area*, que contiene los datos y las referencias necesarias cuando el proceso está ejecutando su código (por ejemplo informaciones sobre los archivos abiertos, acciones asociadas a las señales que puede recibir, definición de *timers*, etc.).

El sistema operativo asocia un conjunto de estados a las diferentes situaciones en que un proceso puede encontrarse en el curso de su existencia. La ejecución de las rutinas del *kernel* del sistema puede causar transiciones de un estado a otro. En el curso de este articulo haremos referencia solo a tres estados principales: *running*, *ready* y *sleeping*. El proceso *running* (llamado también proceso corriente) es el único proceso ejecutado por el procesador en un cierto instante; los procesos *ready* son los que esperan en una cola para poder ser ejecutados por el procesador; los procesos *sleeping* son los que están suspendidos en espera de que ocurra algún "evento" significativo para la continuación de su ejecución (por ejemplo la terminación de una operación de E/S).

El *kernel* utiliza una politica de *scheduling* de la ejecución de los procesos basada en una "prioridad" asociada a cada proceso. La prioridad es modificada dinámicamente de manera a asegurar que todos los procesos *ready* puedan obtener atención del procesador dentro de una cantidad de tiempo finita. La politica de variación de las prioridades privilegia los procesos que tienen frecuentes periodos de inactividad (por ejemplo los programas interactivos). El mecanismo de *scheduling* de UNIX™ fué realizado de manera que una substitución del proceso corriente no pueda ocurrir dentro del *kernel* en el curso de la ejecución de una llamada al sistema operativo. Un proceso puede ser forzado a ceder la utilización del procesador solo cuando está por ejecutar el código usuario y si hay en cola otro proceso *ready* que tiene una prioridad más alta que la suya. Un proceso que está ejecutando una rutina del *kernel* del sistema operativo puede dejar espontaneamente la utilización del procesador, pasando al estado *sleeping*, si ha alcanzado un punto más allá del cual su ejecución no puede continuar inmediatamente. Los mecanismos de substitución del proceso en ejecución adoptados por UNIX™ simplifican notablemente los problemas de sincronización dentro de su *kernel*.

El procesador, cuando ejecuta un código por cuenta del proceso corriente, utiliza diferentes "privilegios" y métodos de acceso a la memoria según que se ejecute el código de un programa usuario (modo usuario) o el código de alguna rutina del sistema operativo (modo sistema). El "espacio de direcciones" al cual un proceso puede acceder cuando está en modo usuario es totalmente disjunto del accesible en modo sistema. En el modo usuario un proceso puede acceder solo a las partes usuario de su imagen, es decir a su código, datos y *stack*, mientras que en el modo sistema puede acceder directamente solo al código de las rutinas del sistema operativo y a sus estructuras de datos. Sin embargo, en el modo sistema, un proceso puede también transferir datos en forma indirecta de y hacia el espacio de direcciones del usuario, utilizando procedimientos del *kernel* del sistema operativo a muy bajo nivel.

El *stack* utilizado en el modo sistema es diferente del *stack* usuario y es accesible en una misma dirección por todos los procesos, entonces cada proceso, cuando está suspendido, debe mantener una copia de su *stack kernel*, para utilizar durante su ejecución. Una copia de los registros del procesador utilizados en el modo usuario es guardado en el *stack kernel* cada vez que un proceso comienza a ejecutarse en modo sistema y es restaurada cuando el proceso retorna al modo usuario.

El código y las estructuras de datos globales del sistema operativo residen siempre en la memoria central y son compartidos por todos los procesos cuando ejecutan rutinas del sistema operativo en el modo sistema. En cambio el código, los datos, los *stack* usuario y *kernel* y los datos de sistema (*User Area*) propios de cada proceso pueden ser mantenidos completamente o parcialmente en disco cuando el proceso no está en ejecución. En efecto, el sistema puede decidir copiar en disco parte de la imagen de un proceso suspendido a fin de liberar espacio en la memoria central (mecanismos de *swapping* y memoria virtual). De todas maneras el proceso que está en ejecución puede acceder a cada ubicación dentro de su espacio de direcciones sin preocuparse si esta está actualmente residente en memoria central o si fué copiada en disco: si el proceso hace referencia a una parte de código o a datos que se encuentran en disco, el sistema operativo se preocupa de hacer accesibles nuevamente en la memoria central las partes de la imagen que interesan. El proceso no se da cuenta que parte de su imagen es copiada de y hacia el disco, porque las transferencias son gestionadas en forma "transparente" por los mecanismos de gestión de memoria del sistema operativo.

5. Implementación del mecanismo de *snapshot* y *restart*

Todas las funciones del mecanismo de *snapshot* y *restart* implementado en el curso del proyecto de investigación fueron realizadas dentro del *kernel* de UNIX™.

A primera vista parecería más simple el uso de un mecanismo a nivel usuario, que podría ser realizado en UNIX™ mediante un programa que acceda a las estructuras de datos del *kernel*, leyendolas directamente de la memoria central de la computadora (que en UNIX™ se puede usar como cualquier archivo de sistema). El problema principal es que los procesos a nivel usuario son entidades concurrentes, así que la imagen del proceso del que se quiere tomar el *snapshot* podría modificarse en el curso de la operación de escritura de las informaciones, por el hecho que esto continua su ejecución en paralelo con el que efectua el *snapshot*, y hacer inconsistente las informaciones guardadas. Aunque el proceso fuese suspendido el mecanismo de gestión de la memoria virtual podría todavía mover partes de su imagen de y hacia el disco mientras el programa de *snapshot* está tratando de leerlas. También se podría pensar en insertar un procedimiento para efectuar el *snapshot* directamente dentro del programa usuario, pero esta solución haría al mecanismo mucho menos flexible requiriendo la modificación y la recompilación de todos las aplicaciones que se quisiera poder tratar con el mecanismo de *snapshot*.

6. Informaciones que forman un *snapshot*

Las informaciones que se necesita guardar para realizar un *snapshot* de un proceso de clase B deben comprender: una imagen de la memoria usada, el valor de los registros, el estado de los archivos abiertos, el directorio de trabajo corriente y, en general, todos los datos del sistema operativo relacionados con la ejecución del proceso y los recursos utilizados.

Hemos decidido guardar estas informaciones en dos archivos, uno en el formato del sistema llamado *core* y un otro en un nuevo formato que hemos llamado *snap*.

Los archivos en formato *core* son normalmente generados por el sistema, con fines de *debugging* cuando se verifican particulares errores en un proceso y comprenden copias de: datos de sistema asociados directamente al proceso(*User Area*), contenido de los registros, *stack kernel* y usuario, área de memoria conteniendo los datos del programa usuario. Se eligió mantener este formato standard, poniendo las demás informaciones necesarias en otro archivo, de manera que su contenido pueda ser examinado usando los instrumentos de *debugging* ya disponibles para UNIX™ (adb, sdb, dbx).

El archivo *snap* contiene todas las demás informaciones, necesarias para la definición del ambiente de ejecución del proceso, que no son previstas para el archivo *core:* informaciones extraidas de la tabla de procesos, datos del sistema operativo que describen los archivos y dispositivos de E/S utilizados por el proceso, informaciones sobre el estado de los dispositivos de E/S y en particular el *setting* de los terminales, fecha del *snapshot*, etc.

Las aplicaciones de clase B requerirían guardar copias de todos los archivos abiertos por el proceso y también del código del programa, puesto que podrían ser modificados o removidos luego del *snapshot* y antes del *restart*. Estas copias de los archivos no son siempre necesarias y podrían ser muy "costosas" en terminos de recursos elaborativos y de memoria de masa requerida. Entonces hemos decidido dar al usuario la posibilidad de elegir si efectuarlas o no.

El mecanismo que realiza el *snapshot*, implementado a nivel del *kernel* de sistema operativo, puede acceder solo a la representación "interna" de los diferentes objetos de un sistema y por esta razón hace referencia a los archivos por medio del número del *inode* (la estructura de datos del *file system* de UNIX™ asociada a cada archivo) y no a través de su nombre (*pathname*), como, en cambio, lo hacen los programas normales de los usuarios. Cuando un archivo es copiado, el número de su *inode* cambia aunque las informaciones contenidas sean siempre las mismas. Entonces se tuvo que proveer algunos programas de utilidad para permitir al usuario controlar y modificar las informaciones contenidas en los archivos *snap* y *core*, efectuando eventualmente las transformaciones necesarias entre la representación de los archivos a nivel usuario (*pathname*) y la interna del sistema (*inode*).

7. Realización del *snapshot*

El *snapshot* de un proceso es producido por una rutina del sistema operativo cuando a este proceso se envia una "señal" particular. Las señales en UNIX™ son un mecanismo para interrumpir temporalmente el flujo de ejecución normal de un programa y pedir la ejecución de particulares procedimientos asociados a la señal que, según los casos, pueden ser funciones del sistema operativo o del programa del usuario. Hemos implementado dos nuevas señales en el *kernel* de UNIX™, para dar al usuario una posibilidad mínima de control sobre el mecanismo de *snapshot:* PSNAP y FSNAP. Mientras la señal PSNAP provoca la escritura de la imágen del proceso (en los archivos *core* y *snap*), la señal FSNAP provoca también la realización de las copias de los archivos abiertos.

El uso del mecanismo de las señales tiene muchos inconvenientes. Por ejemplo las señales no son gestionados inmediatamente siendo tratados por el mismo proceso receptor, que tiene que esperar su turno de ejecución. Además si un proceso está suspendido en espera de un "evento" con alta prioridad (o si fué suspendido por el usuario) no es reclamado inmediatamente por el envio de la señal y la gestión de este último será muy retardada. Otro problema no resolvible utilizando el mecanismo de las señales es que estos no proveen una manera de informar el proceso que requiere el *snapshot* (enviando la señal) si este fué efectuado correctamente o si se verifica algún error.

El motivo por el cual se eligió utilizar el mecanismo de señales para gestionar el *snapshot* es que esto permite conocer exactamente "dónde" y "cuándo" puede ser efectuado la gravación de la imagen de un proceso. Sin esta posibilidad de ubicar el *snapshot* el

mecanismo para efectuar el *restart* del proceso no sería realizable, como se verá en seguida. El mecanismo de señales permite localizar el *snapshot* porque hay un único y particular código del *kernel* que controla la llegada de una señal y ejecuta las acciones asociadas. En efecto las señales son recibidas por el proceso solo mientras ejecuta en el modo sistema en algunos puntos muy bien identificados del código del *kernel*. En particular en UNIX™ 4.2BSD la llegada de las señales es controlada siempre y solo antes que un proceso retome su ejecución en modo usuario, luego de un llamado al sistema operativo o de cualquier interrupción síncrona (debida por ejemplo a la rotación en el uso del procesador impuesta por la política de *scheduling)*.

El uso del mecanismo de señales además permite simplificar notablemente la implementación, puesto que la función de *snapshot* es ejecutada mientras que está en ejecución el mismo proceso implicado. En efecto, por los motivos descritos en el capítulo anterior, el proceso puede acceder en forma directa a todas las ubicaciones de la propia imágen sin importar si parte de esta pudieron ser copiadas en disco a causa de los mecanismos de memoria virtual.

En realidad no habría grandes dificultades en tomar un *snapshot* en cualquier momento, sin embargo una solución de este tipo llevaría a encontrarse con dos grandes problemas en la fase de *restart*, haciendo practicamente imposible su realización. El primer problema está ligado a la seguridad del sistema mientras que el segundo es el problema de restaurar los punteros a los elementos de las estructuras de datos del sistema operativo.

El problema de seguridad surge porque, pudiendo también efectuar el *snapshot* en cualquier punto dentro del código del sistema operativo, se obligaría al mecanismo de *restart* a restaurar el estado de toda la cadena de las llamadas de rutinas del sistema operativo que el proceso tenía eventualmente activas en el momento de su *snapshot*. Para efectuar esta restauración sería necesario utilizar en forma directa las informaciones contenidas en la copia del *stack kernel* guardado en el archivo *core*. Desgraciadamente no hay ningún método practicamente realizable que permita asegurar que el contenido de los archivos que contienen las informaciones del *snapshot* no haya sido modificado en forma peligrosa por un usuario experto. Por ejemplo la copia del *stack* podría ser modificada memorizando direcciones de rutinas o datos del sistema operativo que podrían romper la seguridad del sistema o causar un *crash* del mismo.

El problema de la restauración de los punteros es originado por el hecho de que el *kernel* mantiene muchas tablas cuyos *items* son cargados en forma dinámica cuando se presenta la necesidad (tabla de procesos, de los archivos abiertos, de los *inodes*, ...). Estos *items* son desechados cuando las informaciones que contienen ya no son inmediatamente necesarias. Entonces en general las tablas de datos del sistema operativo pueden contener informaciones muy diferentes en distintos momentos de la vida del sistema: pueden estar ausentes o presentes determinadas informaciones, las informaciones relativas a una misma entidad o recurso pueden ser puestas en posiciones de tabla diferentes, además las posiciones libres en que se pueden insertar nuevos elementos no son siempre las mismas. Frecuentemente las rutinas del sistema operativo acceden a recursos o a datos utilizando punteros a los correspondientes *items* en las tablas de datos de sistema, y suponen que la colocación de estos elementos no cambie hasta que sean utilizados. Durante la fase de restauración de un proceso sería muy poco eficiente y a menudo prácticamente no factible la ubicación de todas las informaciones o de todos los datos en las mismas posiciones originales dentro de las tablas del *kernel*. Entonces todas las rutinas del *kernel* que estaban activas en el momento del *snapshot*, luego de la restauración, encontraran todos los elementos de las tablas de datos que estaban utilizando ubicados en diferentes posiciones, y sus punteros serán inconsistentes.

La mejor solución es la de dejar al *kernel* la tarea de ubicar nuevamente los objetos necesarios para la ejecución del proceso que se quiere restaurar, utilizando sus normales mecanismos, y luego recargar los punteros usados en todas las rutinas de sistema activas en el momento del *snapshot*. Pero si el *snapshot* pudiese ser guardado en cualquier punto del *kernel*, esta solución sería difícilmente alcanzable en la práctica, porque requeriera la modificación de casi todo el código del sistema operativo con la introducción de un número absurdo y pesante de controles para recargar los punteros en el caso de que un proceso estuviera en la fase de restauración.

En la implementación del mecanismo de *restart* la posibilidad de localizar el *snapshot* ha permitido afrontar fácilmente el problema de la recarga de los punteros, introduciendo un único control inmediatamente luego del retorno de la rutina de sistema que gestiona las señales. Este control se preocupa simplemente de recargar el único puntero activo en este punto del código del *kernel* (un puntero a la tabla de los procesos) cuando el proceso que sale de la rutina de gestión de las señales es uno que se está restaurando con *restart*.

8. Reconstrucción de un proceso a partir de un *snapshot*

Hemos introducido una nueva llamada de sistema (*restart*) que efectua la restauración de la ejecución de un proceso a partir de las informaciones guardadas en el *snapshot*. Esta llamada de sistema prevee como argumentos los nombres (*pathnames*) de los archivos *core* y *snap* y del archivo que contiene el código del programa ejecutable, algunos parámetros de control, y la dirección de un *buffer* en que pueden ser puestas las informaciones detalladas sobre las eventuales condiciones de error.

La llamada de sistema *restart* provee, al comienzo de su ejecución, para construir un "esqueleto" del proceso que debe ser restaurado: inicializa oportunamente un elemento de la tabla de procesos, construye una *User Area*, reserva la cantidad de memoria virtual necesaria para el proceso y reserva también espacio en un área de disco para utilizar en caso de *swapping*. El proceso "esqueleto" es inicializado de manera a reflejar el ambiente de ejecución del proceso originario en el momento del *snapshot*. Puede surgir una dificultad cuando se trata de restaurar el identificador del proceso (*pid*), puesto que, en el momento del *restart* dicho número podría haber sido asignado a otro proceso. Sería muy ineficiente, a más de complejo, tratar de impedir la ubicación de todos los identificadores de los procesos congelados. La utilización de mecanismos complicados no sería además justificada puesto que la posibilidad de encontrar ya ubicado el mismo identificador de proceso es muy remota, aunque no excluible a priori. Entonces se adoptó una estrategia muy simple: si no hay requerimiento explícito de parte del usuario, cuando el procedimiento de *restart* se da cuenta de que el número identificador del proceso ya está utilizado, ubica otro entre los identificadores disponibles. Si el usuario sabe que el valor del identificador del proceso está utilizado explícitamente en el algoritmo efectuado por el programa (por ejemplo para generar los nombres de archivos temporales) puede forzar, usando un parámetro de control, el retorno de la llamada de sistema en el caso que este no esté disponible.

El nuevo proceso hereda por el proceso que requiere el *restart* la identidad del propietario y del grupo al que pertenece, los derechos de acceso a los archivos y los eventuales límites sobre la utilización de los recursos. Privilegios particulares del proceso originario no son restaurados por motivos de seguridad del sistema. Por este motivo los programas que utilizan explícitamente derechos particulares de acceso o privilegios, ligados a la identidad o al grupo del propietario pueden encontrar problemas si son restaurados por un usuario diferente del originario.

Luego de haber preparado el proceso "esqueleto" la llamada de sistema restaura el directorio de trabajo corriente del proceso, reabre los archivos y los dispositivos de E/S utilizados y reposiciona adecuadamente los punteros de lectura y escritura dentro de cada uno de estos. Entonces restaura el estado y el *setting* de todas las líneas de terminales utilizadas.

La terminal de control del proceso no debe ser necesariamente el original si no el del usuario que pide el *restart*, en esta forma toda la interacción entre el proceso y el usuario puede ser efectuada correctamente aunque el proceso sea restaurado por una terminal diferente de la originaria. Obviamente un programa que utiliza en modo explícito las características particulares de un cierto tipo de terminal no podrá operar correctamente si la terminal de control, a más de estar en una línea diferente, es de tipo diferente respecto a la originaria.

El nuevo proceso asume el *stack* y los registros *kernel* del proceso que requiere el *restart* y solamente los registros utilizados por el programa usuario son restaurados por la

copia guardada dentro del archivo *core*. Entre los registros del *kernel* hay también el *program counter*, entonces el proceso restaurado comenzará su ejecución desde un punto dentro del código de la rutina *restart* y no desde aquel en que fué guardado el *snapshot*. Este hecho, como veremos, no comporta problema alguno puesto que el *snapshot* está perfectamente localizado por los motivos expuestos anteriormente. Los registros originales del *kernel* no son restaurados por los problemas de seguridad explicados antes. La utilización de los registros usuario originales no conllevan peligro alguno para la seguridad del sistema puesto que, aunque la copia de estos fuese manipulada dentro del archivo *core*, al máximo se podría modificar el comportamiento del programa usuario.

En este punto el *restart* puede insertar el proceso en la cola de procesos *ready* a la espera de ser ejecutados por el procesador. Desde este momento en adelante la llamada de sistema *restart* ya no está en posibilidad de avisar al usuario los eventuales errores que pueden todavía ocurrir durante la última fase de la restauración del proceso.

El proceso restaurado, apenas obtiene el control del procesador comienza a ejecutar el mismo código del proceso que ha llamado el *restart*, puesto que heredó su *stack* y sus registros *kernel*, pero está en grado de autoidentificarse utilizando un mecanismo similar al de la llamada de sistema *fork* (que genera un nuevo proceso, creando una copia exacta de un proceso padre con la única diferencia de un distinto código de retorno sobre el *stack*). Entonces enseguida el proceso restaurado puede comenzar a ejecutar un bloque de instrucciones específico.

El código del *restart* ejecutado en el contexto del proceso restaurado se encarga de cargar su área de datos y su *stack* usuario originales leyendolos del archivo *core*. En el caso de que no hubiese suficiente espacio disponible en la memoria central el mecanismo de memoria virtual se encarga automáticamente de desplazar partes de las áreas de memoria en el área de disco reservada para el *swapping* del proceso. El código del programa usuario es cargado del archivo ejecutable, en el caso de que no exista ya una copia utilizada por otro proceso en ejecución (si se trata de código compartido). El código no es leido del todo inmediatamente, como ocurre con los datos y el *stack* usuario, si no que sus partes son cargadas cuando se necesitan, utilizando un mecanismo llamado de "carga por demanda" (*load on demand*). Este mecanismo es obviamente mucho más eficiente del usado por los datos y el *stack*, puesto que permite no leer partes de código que pueden no ser usadas nunca. Desgraciadamente la versión actual de UNIX™ 4.2BSD prevee la carga por demanda solamente del segmento de código del archivo ejecutable y no de los datos y del *stack*.

El proceso en fase de restauración gestiona también el *timer* en tiempo real, decrementandolo del tiempo pasado del momento del *snapshot* y ejecutando las acciones eventuales previstas por el proceso en el caso (probable) que este *timer* haya expirado. Los otros *timers* previstos por el sistema UNIX™ 4.2BSD no necesitan operaciones particulares de restauración puesto que miden tiempos "virtuales" medidos solo mientras el proceso está en ejecución.

La ejecución del programa usuario originario es retomada al termino de esta fase cuando el proceso retorna al modo usuario, luego de terminar la ejecución de la llamada de sistema *restart*, y sus registros usuario son restaurados desde la copia guardada en el archivo *core*.

El programa usuario debe retomar su ejecución sin que su comportamiento sea modificado en modo alguno. Esta total transparencia es posible a condición de que los diferentes contenidos en los registros y en el *stack* usuario sean adecuadamente preservados de ser alterados por el código de retorno de la llamada de sistema *restart*, que obviamente está privado de significado por el programa restaurado. En efecto las llamadas de sistema modifican los registros y el *stack* del programa antes de retornar al modo usuario, de manera a preparar un adecuado código de retorno. Pero el *restart* ejecutado en el contexto del proceso restaurado, no debe generar valor alguno de retorno, que alteraría los registros y el *stack* arruinando un eventual valor de retorno precedente. Pero, como se explicó antes, el *snapshot* del proceso fué guardado durante la fase de gestión de señales que siempre es ejecutada al termino de cualquier llamada de sistema o de la gestión de una interrupción síncrona: en el primer caso los registros y el *stack* usuario ya fueron modificados de manera a contener el valor de retorno de la última llamada de sistema ejecutada antes del *snapshot*, mientras que

en el segundo caso no están modificados puesto que las interrupciones síncronas son gestionados de manera que no puedan alterar el *stack* o los registros. Entonces en nuestra implementación fué necesario modificar el código original de UNIX™, en el punto que gestiona el retorno de las llamadas de sistema, de forma que se efectue un control sobre una condición particular, que es verdadera solo en la última fase del *restart* y en el contexto del proceso reconstruido. El código asociado a este test se encarga de recargar el único puntero (a la tabla de procesos), utilizado en esta parte del código y en evitar la fase de generación del valor de retorno del *restart*.

El *restart* ejecuta numerosos controles durante todas sus fases a fin de evitar las situaciones inconsistentes. Si un control tiene éxito negativo, el procedimiento deshace todo el trabajo hecho, dejando los recursos eventualmente adquiridos, y termina con un código de error. Es importante poner en evidencia que todos estos controles no tienen el objetivo de asegurar la seguridad del sistema, puesto que están basados en las informaciones contenidas en los archivos que componen el *snapshot*, que en todo caso pueden ser modificados por una persona experta, manteniendo un formato consistente en la aparencia.

Un primer grupo de controles tiene la tarea de asegurar la consistencia de las informaciones contenida en los archivos que componen el *snapshot* (*core*, *snap*, programa ejecutable, archivos en uso por el programa) y comprende los controles cruzados sobre el contenido de las copias de las estructuras de datos de sistema guardadas y sobre las fechas y las dimensiones de los archivos. El usuario tiene la posibilidad de pedir evitar los controles sobre las fechas o las dimensiones de los archivos cuando sabe que el programa que quiere restaurar no está influenciado por modificaciones de los archivos usados (por ejemplo en un programa que manipula archivos actualizados en forma secuencial) o cuando sabe que el contenido de los archivos no es diferente aunque sus fechas cambiaron por alguna razón (por ejemplo porque son copias de los originales). El único caso que nunca puede ser aceptado es el de un archivo que tenga una dimensión menor respecto al original y cuyo puntero de lectura y escritura se refiera a una posición sucesiva a la actual longitud del archivo. Otro grupo de controles es efectuado sobre el contenido de las estructuras de datos del sistema (*inodes)* que se refieren a los archivos, al directorio de trabajo corriente y a los dispositivos de E/S utilizados por el proceso restaurado. Cada *inode* debe corresponder a un objeto efectivamente presente en el *file system*, del mismo tipo del original (archivo,directorio o dispositivo de E/S), el cual debe ser todavía accesible usando los nuevos derechos de acceso que hereda desde el programa que pide el *restart*.

La seguridad está todavía basada sobre los mecanismos standard de UNIX™: los procesos restaurados no mantienen los derechos de acceso y los privilegios originarios pero asumen la identidad del usuario y del grupo del proceso que pide el *restart*. El acceso a los archivos es efectuado usando estos nuevos derechos de acceso, que son asignados y controlados en el modo usual por el sistema operativo. Además el *restart* nunca utiliza directamente las informaciones sobre las estructuras de datos del *kernel*, contenidas en los archivos del *snapshot*, pero las reinstala y las inicializa por medio de los normales procedimientos del sistema operativo. En esta forma la seguridad del sistema no es alterada, y es posible evitar que alguna modificación "maligna" a los archivos que contienen el *snapshot* pueda introducir inconsistencias en los datos del *kernel* o crear "agujeros" en los mecanismos de control del sistema.

9. Conclusiones

En un sistema operativo moderno es muy útil dar la posibilidad de identificar y guardar en disco el estado de un proceso, para poder restaurarlo en cualquier momento posterior (eventualmente luego de una parada de máquina), puesto que permite realizar muchas funciones importantes de alto nivel. Un mecanismo de este género fué realizado para el sistema UNIX™ 4.2 BSD.

Hemos confrontado la imposibilidad de realizar un mecanismo completamente general, que fuese capaz de gestionar todas las aplicaciones posibles que se pueden efectuar en un ambiente UNIX™. Entonces fué necesario hacer algunas hipótesis restrictivas y definir entonces las clases de procesos que pueden ser gestionados en forma eficiente en situaciones

reales. De todas maneras estas hipótesis son todavía suficientemente generales para permitir el tratamiento de muchos casos de interés práctico. El mecanismo para efectuar el *snapshot* y el *restart* de un proceso fué realizado dentro del *kernel* del sistema operativo UNIX™ previendo dos nuevas señales y una nueva llamada de sistema.

La implementación fué realizada en una forma de lo más posible "ortogonal" respecto al sistema originario, de manera a permitir una rápida instalación y previendo el poder transportar el mecanismo de *snapshot* y *restart* lo más facilmente posible sobre las nuevas versiones del sistema operativo. Esta filosofía de proyecto hizo modificar el sistema agregando nuevos módulos y modificando solamente una decena de lineas en el código original.

Quedan todavía muchos problemas abiertos para poder extender el mecanismo actual de manera que pueda gestionar clases de procesos más generales: operaciones conducidas por muchos procesos que se comunican, residentes en una misma computadora y operaciones distibuidas conducidas por procesos ejecutados concurrentemente en diferentes computadoras de una red. El mecanismo actual fué además realizado bajo la forma de prototipo en un ambiente de investigación y entonces debe ser optimizado y hecho más razonable desde el punto de vista de la interface con el usuario antes de poder ser insertado en un producto comercial. De todas maneras el trabajo de investigación desarrollado permitió recoger muchas informaciones importantes y experimentar en la práctica nuevas ideas, que pueden ser usadas para continuar en la investigación y para desarrollar ya unas cuantas aplicaciones interesantes.

Bibliografía

[1] K.M. Chandy, L. Lamport, *"Distributed Snapshots: Determining Global States of Distributed Systems"*, ACM Transactions on Computer Systems, 3.1, 1985, pp. 63-75

[2] D. Presotto, M. Powell, *"Publishing: a reliable broadcast communication mechanism"*, Proc. 9th ACM Symposium on Operating Systems Principles, 1983, pp. 100-109

[3] M. Spezialetti, P. Kearns, *"Efficient Distributed Snapshots"*, IEEE 6th International Conference on Distributed Computation Systems, 1986, pp. 382-388

[4] K. Thompson, *"UNIX™ Implementation"*, The Bell System Technical Journal, Vol. 57, No. 6, Julio 1978, pp. 1930-1946

[5] W. Joy, E.Cooper, R. Fabry, S. Leiffer, K. McKusick, D. Mosher, *"4.2BSD System Manual"*, UNIX™ Programming Manual, Vol. 2c, Sect. 68, University of California 94720