# Checkpoint/Restart de procesos
# Un análisis del diseño e implementacion

Matías Zabaljáuregui - Marcos E. Urbaneja Sánchez

22 de noviembre de 2005

## 1. Intoducción

Una imagen es una descripción de una computación que puede ejecutarse por una computadora. Un proceso es una imagen en algún estado de ejecución. En cualquier momento, el estado de un proceso puede representarse como dos componentes: el estado inicial (la imagen) y los cambios ocurrieron durante la ejecución. La información total, que es el estado inicial junto con los cambios, nos brinda el estado de un proceso. Podría ser deseable preservar este estado en determinados puntos en el tiempo, tal vez, debido a la cantidad de computaciones que se necesitaron para alcanzarlo.

Checkpoint se define como la acción de grabar un estado de un proceso computacional para que el proceso pueda reiniciarse en el punto de progreso indicado por este estado. El mecanismo de checkpoint es útil para migración de procesos, gang scheduling, hibernación (para preservar un estado entero de la máquina cuando esta se haya apagado), o suspensión (se implementa en software de virtualización como Vmware Workstation (tm), para salvar espacio de memoria o volver a estados conocidos), y como un mecanismo para permitir tolerancia a fallos.

La migración de procesos es las transferencia de algún subconjunto de información significativa a otra localidad de ejecución, de manera tal que la computación que estaba corriendo localmente pueda continuar correctamente en el destino seleccionado. La migración de procesos resulta más interesante en sistemas donde los procesadores involucrados no tienen memoria compartida, ya que de otra forma la transferencia de estado es trivial. Un ambiente típico donde la migración de procesos es interesante es el de clusters de computadoras.

La necesidad de soportar la tolerancia a fallos dentro de las actuales computadoras paralelas de gran escala está alcanzando una importancia crítica. Para computaciones científicas, las aplicaciones pueden correr durante días, semanas, o más, hasta que se completen. Debido a la ausencia de un mecanismo para tolerancia a fallos, un error de un componente es catastrófico para la aplicación que esta corriendo. En este escenario los mecanismos de checkpoint/restart estan avocados a una solución directa para proveer tolerancia a fallos. Se basan en salvar periodicamente el estado del proceso en un almacenamiento estable, para que ante un evento de falla, la aplicación pueda reiniciarse en el punto de checkpoint más reciente. Estos mecanismos son bastante prometedores asumiendo semánticas de fail-stop donde las fallas se pueden detectar (una suposición rasonable en la práctica).

Adicionalmente, queda implícito en las metas de los comúnmente llamados sistemas de computos autónomos (sistemas que se pueden manejar por si solos) que el mecanismo de checkpoint/restart debe ser completamente transparente para el programador y el usuario de la aplicación, esto es, el código fuente de la aplicación no necesita modificarse, recompilarse, o relinkearse. Dichos sistemas deben ser capaces de manejar tanto las operaciones de checkpoint/restart automáticas como las iniciados por el usuario en cualquier momento durante la ejecución de la aplicación. Por ejemplo, una operación de checkpoint puede iniciarse a intervalos regulares fijos, o en un esquema más sofisticado, el intervalo puede optimizarse automática y dinámicamente dependiendo de un números de parámetros, como lo es la tasa de fallas del sistema. El checkpoint iniciado por el usuario permitiría al administrador del sistema suspender de manera segura el proceso permitiendo que pueda correr otro, o realizar tareas de mantenimiento del sistema.

En su forma más simple, checkpoint salva el estado entero del proceso. Incremental checkpoint es una optimización bien conocida en donde solamente se guarda la parte del estado de un proceso que ha sido cambiada desde la última operación de checkpoint. La optimización se alcanza cuando el tamaño de delta (subconjunto de memoria de la aplicación que cambio desde la última operación de checkpoint) es pequeño comparado con su memoria entera. El mecanismo de protección de página implementado en los sistemas de memoria virtual se usa comunmente para monitorear las sucesivas modificaciones del estado del proceso, por lo que los cambios en la memoria de la aplicación son observadas en una granularidad de página.

## 2. Implementaciones

Los mecanismos de Checkpoint/Restart se pueden clasificar en tres dimensiones: el contexto, el agente que provee la funcionalidad checkpoint/restart, y las particularidades específicas de la implementación. Para ilustrar esta clasificación, la Figura 1 retrata el espacio de implementaciones de checkpoint/restart. Dentro de la dimensión del contexto, la implementación pueden ser a nivel usuario o sistema.

A nivel usuario la implementación puede programarse directamente en el código fuente de la aplicación o ser insertarda automáticamente por el precompilador. Usualmente en estos casos una librería específica de checkpointing provee las primitivas necesarias para checkpoint/restart, eliminando la necesidad de programarlas. En lugar de modificar el código fuente de la aplicación, las primitivas de checkpoint/restart pueden invocarse mediante manejadores de señales definidos a nivel de usuario. Otra implementación esta basada en el uso de la variable de ambiente LD_PRELOAD, en donde se instalan los manejadores de señales y la librería de checkpoint se carga sin tener que recompilar o re-linkear la aplicación.

En contraste, la implementación a nivel de sistema puede darse en el sistema operativo o en el hardware. En el sistema operativo hay varias técnicas para implementar los mecanismos de checkpoint/restart: como manejador de señales en modo kernel, como system calls o como un thread del kernel. En las subsecciones siguientes se estudian estas alternativas con más detalle.

A continuación, se revisan algunos conceptos relacionados con los requerimientos y características buscadas en los diferentes tipos de implementaciones.

La transparencia se presenta como el principal requerimiento de todo sistema de checkpoint/restart. Las aplicaciones interactúan con el kernel a través de system calls, las cuales modifican o crean estructuras dentro del kernel. El grado de posibilidad de recuperar estas estructuras que ofrezca una implementación dada es un indicador de transparencia; el sistema de checkpoint/restart no debería restringir el conjunto de system calls utilizables por las aplicaciones de usuario.

El checkpoint de aplicaciones paralelas es otro gran requerimiento. checkpoint/restart es un recurso muy importante para usuarios de clusters ejecutando aplicaciones científicas paralelas. Podemos hablar de dos clases distintas de aplicaciones paralelas para este informe: Aplicaciones multiproceso y aplicaciones multinodo. Las aplicaciones multiproceso usan más de un proceso ejecutandose en un único nodo y que se comunican a través de mecanismos de IPC tales como segmentos de memoria compartida, pipes, sockets locales. Durante un checkpoint, el sistema debe guardar el estado de los procesos y de los mecanismos IPC. La ventaja de este tipo de aplicaciones es que el sistema operativo tiene acceso a todas las estructuras necesarias para dejar a los procesos en un estado consistente. El checkpoint de aplicaciones multinodo requiere la coordinación con los sistemas operativos remotos o la colaboración activa por parte de los procesos para asegurar que se salvarán estados consistentes de los procesos. Además, si se suponen mecanismos de comunicación confiables, se debe garantizar que todos los mensajes enviados hayan sido recibidos (o almacenados en buffers) antes de suspender los procesos.

El control de la aplicación sobre el momento de ocurrencia de checkpoints puede ser un requerimiento deseable (aunque esto significará una pérdida de transparencia). Ciertas aplicaciones podrían necesitar evitar checkpoints en ciertas regiones de código, como por ejemplo en invocaciones a funciones de librerías cuyo volcado a disco no sea posible. En implementaciones checkpoint/restart basa-

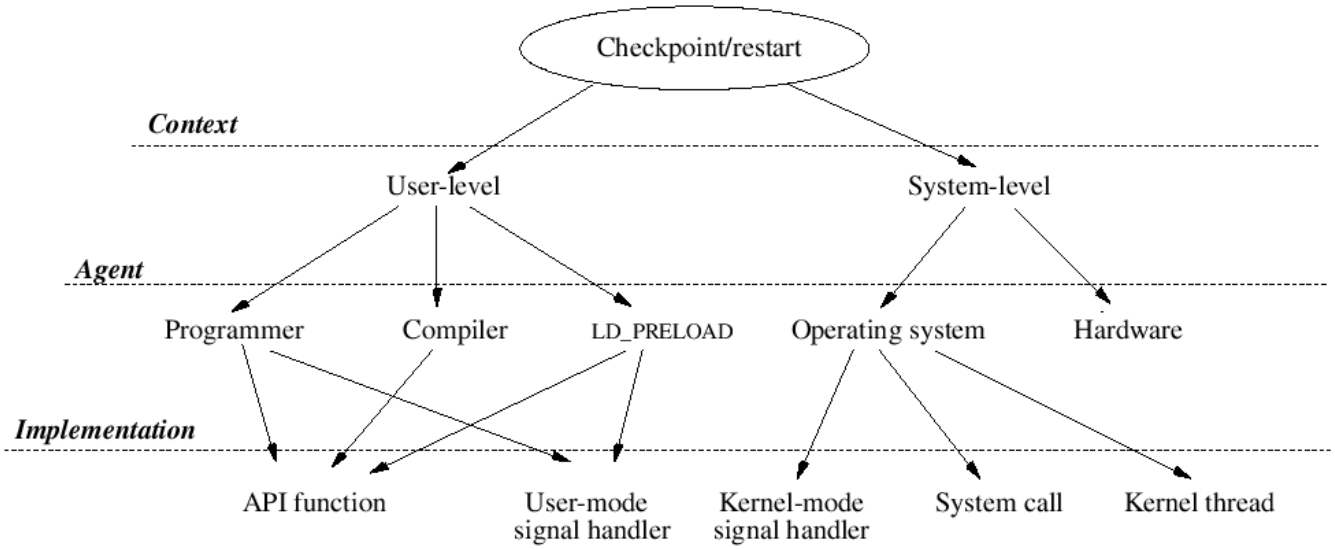


Figura 1: Clasificaciones de implementaciones de Checkpoint/Restart

das en manejadores de señales, la aplicacion puede simplemente enmascarar la señal de checkpoint a través de las system calls *signal()* o *sigprocmask()*.

Algunas implementaciones de checkpoint/restart ofrecen la posibilidad de marcar ciertas áreas de datos de los procesos como "no importantes", de modo que no sean guardadas en un checkpoint. Esta optimización permite ignorar grandes arreglos temporales, caches de la aplicación u otras estructuras de datos que pueden ser reconstruidas fácilmente (o completamente innecesarias) después de un restart y por lo tanto lograr un ahorro en espacio de disco y tiempo de checkpoint.

Durante un checkpoint debe guardarse el estado de los registros de la CPU: instruction pointer, stack pointer, registros de propósito general, registros de punto flotante, etc. Una implementación de checkpoint/restart basada en librería puede usar un manejador de señal para capturar los registros de esta forma: cuando una señal es recibida el kernel almacena los registros en la pila del proceso. El manejador de la señal usa una llamada a *setjmp()* para guardar la pila, y durante el restart invoca a longjmp para recuperarlo. De esta forma, una vez recuperada la pila del proceso, se obtienen los registros automáticamente. Por otro lado, una implementación en el kernel simplemente lee los registros de la estructura de datos correspondiente (struct task_struct).

El espacio de direccionamiento del proceso contiene la mayor cantidad de información del estado del proceso. El espacio se componen de varias secciones: las secciones de datos inicializados y sin inicializar, la heap, la pila y cualquier región mapeada (*mmap()*). Una implementación basada en librería puede obtener las direcciones de inicio y final de la mayoría de las secciones a través de system calls y algún conocimiento de cómo se alocan las secciones en memoria. Sin embargo, no hay una forma estándar de identificar las direcciones, longitudes, flags de protección (*mprotect()*), o estdo de bolqueo (*mlock()*) de las regiones mapeadas. Es posible utilizar mecanismos no estándar, como el file system */proc*, para obtener información de esas regiones. Es muy diferente la situación en el caso de una implementación en el kernel, ya que tiene acceso directo a las estructuras de datos que describen cada una de las secciones del espacio de direccionamiento del proceso.

También deben guardarse las señales pendientes y los manejadores de señales del proceso. Las implementaciones basadas en librerías pueden obtener el estado de los manejadores invocando a las system calls sigaction() o signal(). La lista de señales

pendientes se puede leer usando la system call sigpending(). Una implementación de kernel guardará directamente las estructuras que representan manejadores y señales pendientes.

Los límites de recursos y información de "accounting" tambien deben ser almacenados. Una implementación de librería puede obtener los límites utilizando la system call *getrlimit()* y volver a setearlos en el restart con *getrlimit()*. La información de los recursos usados se puede obtener con *getrusage()*, pero no existe la system call correspondiente (*setrusage*) para volver a setear esta información en el nuevo proceso. El kernel puede simplemente guardar las estructuras de datos *rlimit* y *rusage*, por lo que luego puede recuperar correctamente esta información en el nuevo proceso.

Los archivos y los descriptores de archivos han demostrado ser un desafío para las soluciones checkpoint/restart, y es conocida la importancia de la interfaz que representan los archivos en los sistemas operativos tipo Unix. Los archivos podrían ser modificados entre un checkpoint y el correspondiente restart (posiblemente por la propia aplicación si esta es recomenzada dos veces desde el mismo checkpoint, o desde un checkpoint periódico después de una caída). Incluso peor, si un descriptor de archivo ha sido cerrado, o un archivo fue borrado, no hay estructuras de datos disponibles el estado del archivo. Existen soluciones parciales a este problema, pero estas guardan copias ocultas de todos los archivos cuando estos son abiertos y también cuando el proceso es checkpointed. Incluso así, la garantía de ejecución correcta no es total. Los descriptores de archivos son el único nexo entre un proceso y algún archivo. Los descriptores asociados con archivos regulares y terminales, deberían ser reconectados con esos archivos y dispositivos respectivamente, cuando la aplicación es relanzada. Los directorios también son accedidos a través de descriptores de archivos. Cualquier flag seteada en el descriptor, el modo de acceso y el offset deben ser recuperados antes del restart. Las implementaciones de librerías se basan en redirecciones de system call para mantener una copia separada de la tabla de descriptores de archivos del kernel. En algunos casos, como en accesos read-only, alcanza con tener la tabla de descriptores. En los casos de acceso read/write, puede ser necesario guardar el contenido del archivo en el checkpoint.

Finalmente, los sockets siempre han representado un serio problema para los sistemas de checkpoint/restart. A diferencia de los casos anteriores, no existe una solución estándar para el checkpoint y restart de sockets, aunque se han desarrollado varias alternativas. La mayoría de las implementaciones decidieron ignorar completamente a los sockets. En algunos casos, se le da a la aplicación la oportunidad de cerrar correctamente y reconectar sus sockets a través de funciones callback, que son invocadas en el momento de checkpoint y restart. Una solución se basa en buffering de mensajes y una conexión separada de control para mantener una señal de heartbeat y recuperar los mensajes perdidos después de una falla de conexión. Existe otro esquema donde los sockets son puestos en estado TCP_TIME_WAIT para evitar la caída de la conexión. En el restart, se crea un nuevo socket y se modifica la estructura de datos del kernel que representa al socket para que apunte al extremo remoto en la conexión. El socket remoto es modificado, a través de un mecanismo de señalización fuera de banda, para apuntar al nuevo socket creado.

## 2.1. Implementaciones en espacio de Usuario

Como se indicó anteriormente, las implementaciones en espacio de usuario sufren de falta de transparencia ya que las aplicaciones deben modificarse y recompilarse, o linkeditarse con una librería de checkpoint. Sin embargo, la ventaja de estas soluciones es que su implementación es más simple y portable que realizar modificaciones al kernel. La mayoría de ellas son iniciadas automáticamente a nivel de usuario ya que es la misma aplicación la que realiza periódicamente invocaciones explícitas a operaciones checkpoint. En consecuencia, la falta de flexibilidad es la principal preocupación sobre estas implementaciones. Por otro lado, solamente unas pocas implementan el inicio automático a nivel de sistema y checkpoint incremental.

Un esquema común es instalar un manejador de señal para una señal estándar y, de esta forma, iniciar automáticamente la operación de checkpoint a nivel de sistema. Los manejadores de señales se definen en espacio de usuario y son invocados por el kernel. Estas señales pueden ser disparadas por un timer que periódicamente interrumpa a la aplicación con la señal SIGALARM. Otra opción es usar señales de propósito general como *SIGUSR1*, *SI-*

*GUSR2*, y *SIGUNUSED*. Aunque estas implementaciones suelen ser diseñadas principalmente para una iniciación automática, pueden ser invocadas por el usuario a través del uso explícito del comando *kill*.

Desafortunadamente, estas soluciones no son generales porque en muchos casos los manejadores de señales interfieren con la aplicación. Otro problema inherente al checkpointing en espacio de usuario es la eficiencia: se generan demasiados context switches entre modo usuario y modo kernel por el gran número de system calls que se necesitan para extraer cierta información del estado del proceso. Aunque el context switching se ha optimizado bastante en Linux todavía representa una solución de mayor costo que acceder directamente a las estructuras de datos del kernel porque la mayoría de los registros de la CPU deben salvarse y restaurarse cada vez que una se ejecuta una system call. Por ejemplo, en Linux la system call *sbrk(0)* se usa para extraer los límites de la heap, *lseek()* se usa para extraer los índices de un archivo, y *sigispending()* se usa para extraer las señales pendientes sobre el proceso. Toda esta información es accesible directamente dentro del kernel, mediante la estructura de datos que representa el estado del proceso.

Aún peor, resulta el hecho de que algunas estructuras del kernel que continenen estado de los procesos no son accesibles, ni siquiera indirectamente, a nivel de usuario. Para solucionar esto sin hacer grandes modificaciones al kernel, es necesario replicar estas estructuras en el espacio de usuario interceptando las system calls, por ejemplo *mmap()* y *unmmap()* para hacer un trace de la memoria dinámica, *dlopen()* para hacer un trace de las librerías dinámicas compartidas, y *open()* o *dup()* para extraer atributos de un archivo. Esta alternativa es extremadamente indeseable porque agrega overhead en tiempo de ejecución. Mas aún, las implementaciones a nivel de usuario estan limitadas a aplicaciones que no dependan de estados persistentes perteneciente al sistema operativo, por ejemplo, sockets, memoria compartida, otra información de red, etc. Además, el esquema de señales en espacio de usuario representa un escenario más complejo en el cual programar porque el uso de funciones no reentrantes en el contexto de señales puede causar deadlock o estados inconsistentes en el sistema. Por ejemplo, algunas funciones de la librería de C, como *malloc()* y *free()* son no reentrantes. En contraste, el kernel fue diseñado para ser reentrante.

En la implementación a nivel usuario los checkpoints incrementales son realizados mediante el seguimiento de las modificaciones del estado del proceso con una granularidad de página. La protección de cada página de memoria se setea como solo lectura usando la system call *mprotect()* al principio del intervalo de checkpoint. Cuando la aplicación intenta un acceso de escritura, el sistema operativo le envía al proceso la señal *SIGSEGV* la cual puede usarse para registrar las modificacione de las páginas.

## 2.2. Implementaciones a nivel de sistema.

Hay dos principales acercamientos de checkpointing a nivel de sistema: una implementación enteramente realizada en el sistema operativo, y otra basada en el co-diseño entre hardware/software que involucra el sistema operativo y hardware de propósito especial. En la primera opción, el Checkpoint incremental se implementa con el uso de mecanismos de protección de páginas: cuando el proceso trata de acceder a una página protegida contra escritura se genera una exepeción de page fault y los cambios en la aplicación se siguen con una granularidad de página. Con el soporte de hardware de propósito especial las modificaciones del estado del proceso pueden rastrearse con una granularidad mucho más fina.

### 2.2.1. Implementaciones en el kernel.

En el espacio del kernel cada estructura de datos relevante para el estado del proceso es accesible directamente: esto incluye registros, regiones de memoria, descriptores de archivos, y señales, entre otras. Esta accesibilidad simplifica enormemente la implementación de operaciones de checkpoint/restart, sin embargo requiere más conocimiento interno del kernel. Aunque, en teoría, decrese la complejidad y se incrementa la eficiencia, en la práctica este acercamiento es un camino a seguir si se desean alcanzar los objetivos de eficiencia, transparencia, y generalidad.

Hay tres alternativas principales para proveer la funcionalidad de checkpoint/restart a nivel de sistema: via system call, por señales en modo kernel, o threads del kernel.

- **System Call.** Esta aproximación ocasiona la introducción de nuevas llamadas al sistema operativo para invocar las operaciones de checkpoint y restart. Una práctica común es ejecutar la inicialización automática a nivel de usuario, esto es, que la aplicación invoque directamente system calls, con lo cual la falta de transparencia y flexibilidad se transforman en el problema principal.

- **Manejadores de señales en modo kernel.** Esta aproximación se basa en los mecanismos de señales que ofrece el kernel, pero ahora en lugar de usar señales de propósito general a nivel usuario, se agrega una nueva señal específica al kernel para este propósito. La acción por default de estas señales es hacer checkpoint de la aplicación. La ventaja es que el checkpoint se ejecuta a nivel de sistema en vez de a nivel de usuario. La señal puede generarse con el comando *kill* a nivel de usuario o directamente a nivel de sistema.

- **Thread del kernel.** Aquí se crea un thread en el kernel para ejecutar las actividades de checkpoint/restart. La interacción del espacio de usuario con el thread del kernel puede llevarse a cabo a través de tres interfaces posibles: (1) usando las operaciones estándar de archivos como *read()*, *write()*, y *ioctl()* para comunicarse con el archivo de dispositivo (usualmente dentro de */dev*); (2) via el pseudo file system */proc* usando las operaciones de *read()* y *write()*; o (3) una nueva system call que puede invocarse por otro proceso de usuario (como un proceso monitor) para informar al thread del kernel que haga un checkpoint de un proceso escpecífico. Alternativamente, las operaciones de checkpoint pueden iniciarse a nivel de sistema usando mecanismos internos para activar el thread del kernel.

Todas estas diferentes ideas requieren algunos cambios sobre el kernel, aunque a menudo es posible escribir la mayoría del código como un módulo. Esto mejora la portabilidad y la modularidad, y es útil durante el desarrollo porque un módulo puede cargarse o descargarse dinámicamente.

Las aproximaciones con system calls y manejadores de señales en modo kernel tienen la ventaja de poder ejecutarse en el contexto del proceso que esta por ser suspendido. Por lo tanto el espacio de direccionamiento del proceso *current* [1] es el del proceso que sera suspendido. En contraste, el thread del kernel no tiene el espacio correcto de direcciones (porque los threads del kernel siempre usan direcciones del kernel que son independientes del proceso), y usan la tabla de páginas de la tarea que fue interrumpida, la cual puede no ser la seleccionada para hacer el chekpoint. En esos casos se requiere un intercambio de espacios de direccionamiento de procesos y esto puede invalidar la TLB y de esta forma perjudicar la performance del sistema.

En los primeros dos planteos la misma aplicación está ejecutando el código de checkpoint (tanto si es una system call como un manejador de señal), entonces los datos no cambian durante la operación de checkpoint. Un thread del kernel, en cambio, es un proceso diferente que, en sistemas multiprocesadores, podría correr en paralelo con la aplicación. Esta aplicación podría estar modificando sus datos cuando el thread del kernel esta salvándolos. En este caso se necesita un mecanismo para suspender la ejecución de la aplicación (como quitar el proceso de la run queue) con el fin de garantizar la consistencia de los datos. Una alternativa es hacer un fork del proceso y dejar que el proceso padre corra mientras se esta salvando al proceso hijo (que no empezó a ejecutarse) antes de destruírlo.

Un proceso puede estar en un estado que es difícil de guardar o reproducir, por ejemplo puede estar esperando por un evento externo como una interrupción de un dispositivo. Este problema afecta al manejador de señales en modo kernel pero no al mecanismo de system calls (siempre que el proceso no esté usando alguna función asincrónica) porque en este último caso es la aplicación por si misma la que llama a la función de checkpoint.

La implementación con systems calls requiere algunos cambios en el código fuente de la aplicación para poder llamar a la función de checkpoint, perdiendo transparencia. La flexibilidad también se pierde con este enfoque: dado que es la aplicación la que invoca a las system calls, hay pocas posibilidades de control externo, por lo que el control global de computaciones paralelas a gran escala puede ser dificil de implementar.

El método de manejador de señales en modo ker-

[1] En el kernel Linux, *current* es una macro que devuelve un puntero al *task_strut* del proceso que se está ejecutando actualmente

nel es más transparente que el de system calls pero la ejecución de un manejador de señales es diferido hasta la próxima transición desde el modo kernel a modo usuario. Debido a que es difícil de estimar con precisión cuantos procesos estarán corriendo en un determinado momento, no hay forma de conocer cuando se ejecutará el manejador de señales.

Otro problema esta relacionado con el algoritmo de scheduling de time-sharing: el proceso puede ser suspendido por el kernel porque hay otro proceso con mayor prioridad esperando por la CPU (la prioridad es dinámica y cambia durante la ejecución del proceso). Las interrupciones también pueden afectar al checkpointing. Un nuevo algoritmo de schedulling podría aliviar este problema, pero todos los procesos deberían tener la misma prioridad (alta). Un thread del kernel es un proceso diferente que puede tener una mayor política de prioridad (como la prioridad *SCHED_FIFO*), esto asegura que el thread se ejecutará tan pronto como se despierte y estará corriendo hasta que haya completado su trabajo. Los procesos no pueden interrumpir un thread del kernel si no tienen la misma prioridad. Podría introducirse una nueva prioridad para asegurar que el thread del kernel no se interrumpirá. Pero las interrupciones aún podrían parar el thread, con lo que se torna necesario tener un mecanismo para retrasar estos eventos y de esta forma asegurar que el thread jamás será interrumpido.

El desarrollo de la funcionalidad de checkpoint/restart a nivel de sistema para Linux es un fenómeno relativamente reciente, su primera aparición ronda el año 2001. Las primeras implementaciones fueron desarrolladas principalmente para proveer migración de procesos en clusters. Mas tarde, las implementaciones brindaron funcionalidades avanzadas para gang scheduling, hibernación y tolerancia a fallos.

#### 2.2.2. Implementaciones en el hardware.

La realización del checkpoint puede soportarse mediante hardware diseñado para éste propósito. Como sucede con la implementación a nivel de sistema operativo, esta puede ser completamente transparente al usuario. Pero el checkpointing a nivel de hardware esta limitado precisamente porque esto depende de hardaware a medida, en contraste con la tendencia a construir clusters con componentes convencionales.

Los esquemas basados en hardware tipicamente implementan checkpointing incrementales con granularidad más fina de como se hace a nivel de sistema operativo: modificaciones de espacio de direcciones de la aplicación se rastrean con una granularidad de entrada de cache.

### 2.3. Migración de Procesos.

La migración de procesos está muy relacionada con el checkpoint/restart de procesos. Ambas deben guardar los registros, heap y pila del proceso. La migración puede ser implementada a través de checkpoint/restart de procesos utilizando un file system compartido, un protocolo de tranporte de archivos o directamente un socket para transferir el archivo de checkpoint a un nodo remoto para un restart inmediato.

Existen, sin embargo, dos grandes diferencias entre la migración de procesos y el checkpoint/restart para tolerancia a fallos. Primero, un archivo de checkpoint normalmente es escrito de forma tal que sea válido después de una caída o reinicio del nodo donde la aplicación fue iniciada (generalmente llamdo el *home node*). Por el contrario, un esquema de migración puede dejar datos residuales en el *home node*, por lo que el proceso migrado puede seguir requiriendo que el *home node* siga online. La segunda gran diferencia está relacionada con el número de instancias de la aplicación. Mientras que un esquema de migración puede asumir que la aplicación se ejecuta con éxito, un esquema checkpoint/restart debe ser capaz de hacer roll back de una aplicación a un estado conocido después de una caída. Es posible que una aplicación que requiera mucho tiempo de ejecución (muy común en HPC) necesite más de un roll back, debido a múltiples caídas.

## 3. CRAK una implementación de Checkpoint/Restart para migración de procesos

CRAK es una implementación de checkpoint/restart para Linux. Está pensado para hacer migración de procesos y tiene dos objetivos principales de diseño que lo diferencian de otras alternativas:

- Se implementa como un módulo del kernel;

- Puede reconectar sockets de un proceso después del checkpoint.

Es una implementación que no requiere modificación del código de las aplicaciones de usuario, y se encapsula en un módulo del kernel que puede ser cargado dinámicamente en memoria. También se implementa algo de funcionalidad en espacio de usuario: principalmente, la identificación de los procesos a los cuales se les va a realizar la operación de checkpoint y la reconexión de decriptores de archivos y pipes después del restart. Es posible hacer un checkpoint de un único proceso, a todos los hijos de un proceso o a un proceso y a todos sus hijos. Luego de identificar los procesos sobre los cuales se realizará la operación de checkpoint, se les envía una señal *SIGSTOP*, de forma tal que todos queden suspendidos en un estado consistente. Luego se invoca al código del kernel, el cual escribirá la imagen en un archivo o la enviará por un socket. Posteriormente, se enviará la señal *SIGCONT* a los procesos suspendidos o directamente se los eliminará (generalmente es así en una migración), dependiendo de los parámetros enviados por el usuario. En algunos proyectos, el código de checkpoint es invocado por el mismo proceso a ser suspendido, por lo que el kernel puede usar el puntero al proceso *current* y acceder fácilmente a las estructuras relevantes, como a las tablas de páginas. En cambio en CRAK, el proceso actual es el que requiere el checkpoint, pero no el proceso suspendido. Por lo tanto es necesario recorrer la lista de procesos del sistema para recuperar el descriptor de proceso correspondiente. También es necesario buscar y recorrer las tablas de páginas manualmente para obtener la ubicación física de la memoria del proceso.

CRAK suele ser bien criticado cuando se lo compara con otras alternativas de checkpoint/restart. Se destacan la transparencia, el manejo de sockets, su implementación como módulo que evita tener que parchear y recompilar el kernel, y la posibilidad de hacer checkpoint a múltiples procesos. Sin embargo tiene algunas falencias tales como:

- La mayoría del soporte para archivos abiertos, pipes y sockets todavía se implementa en espacio de usuario.
- Sólo se salvan los path de los archivos abiertos, pero no sus contenidos.
- Durante el restart, no se chequea si los archivos abiertos del proceso suspendido fueron borrados o modificados.
- El soporte de aplicaciones paralelas está limitado a un único proceso padre y sus hijos inmediatos
- Sólo soporta pipes entre procesos paralelos cuando la operación de checkpoint se realiza sobre todos los procesos paralelos.
- No soporta direcciones de loopback.
- Sólo soporta sockets TCP.
- No recupera ningún valor de PID.

## 3.1. Implementación

El módulo registra un *char device* en el kernel cuando es cargado en memoria, de modo que las aplicaciones de usuario pueden interactuar con el módulo utilizando las operaciones estándar que generalmente se usan con drivers para dispositivos orientados a carácteres: *open()*, *close()*, *read()*, *write()*, *ioctl()*, etc. Otra opción hubiese sido utilizar el pseudo file system */proc* para comunicar las aplicaciones con el kernel. Para que los usuarios no tengan la necesidad de invocar directamente las operaciones del módulo a través de llamadas a *ioctl()*, también se implementó una librería llamada *ckptlib.c*, que proveé una interface bastante más amigable:

a)

```
int checkpoint (int fd, int pid, int
flags);
```

La función recibe tres parámetros: el descriptor del archivo donde debe ser escrita la imagen de checkpoint generada; el pid del proceso y los flags que pueden ser una operación OR de:

- *CKPT_KILL*: El proceso se matará despues del checkpoint.
- *CKPT_NO_BINARY_FILE*: La sección de código no será volcada en el archivo de checkpoint.
- *CKPT_NO_SHARED_LIBRARIES*: Las librerías compartidas no serán volcadas.

Los flags se heredaron de otro proyecto conocido llamado: *Epckpt* [4]. Su significado se explicó en la sección 4-5 (agregar esta explicación). Otra idea interesante tomada de *Epckpt* es la de pasar un descriptor de archivo a la función, en lugar del path de un archivo. De esta forma, cuando el kernel escribe la imagen del proceso generada, lo hace en el flujo representado por el descriptor, ya sea que éste representa un archivo en algún file system local, o el extremo de una conexión TCP representada por un socket.

b)

```
int restart (const char * filename, int
pid, int flags);
```

También recibe tres parámetros, el nombre del archivo de checkpoint, un pid y los flags. Esta función carga la imagen en memoria y luego reemplaza el proceso current por el proceso suspendido. Los últimos dos parámetros están principalmente para checkpoint y restart de grupos de procesos:

- Si el flag *RESTART_NOTIFY* esta seteado, cuando el restart termina se envia una señal *SIGUSR1* al proceso especificado por pid.
- *RESTART_STOP* indica al kernel que pare inmediatamente al proceso reiniciado (usualmente se utiliza junto con *RESTART_NOTIFY* para que otra tarea pueda reactivar al proceso reiniciado)

Como mencionamos anteriormente, es posible pasarle un descriptor de socket a la rutina de checkpoint y de esta forma enviar directamente la imagen del proceso a una máquina remota. Lamentablemente, no es posible realizar la operación de restart directamente desde un socket, debido a que la system call *mmap()* requiere un archivo real en el file system.

También se desarrolaron un conjunto de utilidades de usuario:

- **ck**: hace checkpoint de un proceso y lo guarda en un archivo
- **restart**: reinicia un proceso o un grupo
- **dump**: Hace un volcado de la información de un archivo-imagen de checkpoint.

### 3.1.1. Espacio de direccionamiento

En GNU/Linux (y arquitecturas de 32 bits), todos los procesos tienen un espacio de direccionamiento de 4GB, pero sólo una parte es realmente utilizada. Cada proceso está representado en el kernel por una estructura llamada *task_struct*, la cual contiene toda la información relevante sobre el proceso. Desde esta estructura es posible recorrer todas las secciones de memoria del proceso, llamadas Virtual Memory Area o VMA. Durante un checkpoint, es necesario acceder al espacio de direccionamiento del proceso a suspender. Linux proveé un conjunto de funciones helper (*copy_from_user()*, *copy_to_user()*, etc) que permiten la transferencia de datos entre el espacio de kernel y el proceso *current*. Pero en el caso de CRAK, estas funciones no son de mucha ayuda ya que no es el proceso *current* al cual se necesita acceder.

Cada proceso tiene su propio directorio y tabla de páginas inicializados en un fork y apuntados por el *current task_struct*. De esta forma, el espacio de direccionamiento del proceso se mapea a diferentes regiones de memoria física.

El kernel también tiene sus propias tablas y directorios de páginas, con la diferencia de que sólo mapea direcciones físicas a sí mismo. Por ejemplo, un puntero que apunta a 0x00004000 en un proceso, puede estar apuntando a la dirección física 0x01234000, pero en el kernel una dirección virtual es exactamente la dirección física [2].

Ahora, supongamos que se desea acceder a la dirección dir en un proceso p. La dirección física ligada a dir es una función de p y dir. Teniendo en cuenta que esta direción física se corresponde con la dirección virtual en el kernel, es posible acceder directamente (por ejemplo, desreferenciando un puntero). En CRAK se implementa una función llamada *get_kernel_address()* que toma p y dir como parámetros y retorna su dirección física, usando los directorios y tablas de páginas de p.

### 3.1.2. Conjunto de Registros

Para que sea posible reiniciar un proceso, es necesario que todos los registros de la CPU sean cargados con los valores que tenían en el momento de

[2] En la versión 2.6 del kernel Linux, esto no es exactamente así. Debe tenerse en cuenta que CRAK está implementado para el kernel 2.2.

la operación de checkpoint. En un context switch, el conjunto de valores de los registros es guardado y el kernel cambia a otro proceso seleccionado para ejecutarse. Para lograr un volcado consistente del estado del procesador es necesario que el proceso no esté ejecutandose.

Para recuperar el estado del procesador, simplemente se reemplaza el conjunto de registros en la pila (de modo kernel) actual con los valores volcados en el checkpoint. Luego, cuando se retorna de la invocación a *restart()*, éstos seran cargados en los registros. El único problema es dónde se encuentra. En Linux, la estructura *task_struct* se encuentra en el mismo frame de 8 KB junto con la pila de modo kernel, y el conjunto de registros se guarda en el tope de la misma [3].

Por lo tanto, existe una conexión sencilla entre la ubicación de la estructura *task_struct* y la ubicación del conjunto de registros. Asumiendo que p es un puntero a una estructura *task_struct* de un proceso, entonces la ubicación del conjunto de registros correpondiente es:
$structpt\_regs * regs = ((structpt\_regs*)(2 * PAGE\_SIZE + (unsignedlong)p)) - 1;$

### 3.1.3. Archivos Abiertos

Básicamente hay tres tipos de archivos: archivos nombrados, pipes y sockets. Cada tipo tiene una forma distinta de identificarse. Los archivos nombrados, lo hacen mediante paths. Los pipes son anónimos (sin referencia en el file system), y se identifican directamente por números de inodos. La identidad de los sockets depende del protocolo, pero típicamente consiste en una tupla de dirección de red y puerto. Es necesario salvar la identificación de los archivos de acuerdo a su tipo y los valores de las estructuras de datos asociadas con esos archivos, para asegurarnos que cuando reiniciamos el proceso, cada descriptor de archivo referencie a los archivos originales.

Si se dispone de un file system distribuido de acceso global (Coda, AFS), sólo será necesario guardar el path name. En cualquier otro caso, se deberá guardar todo el contenido de los archivos. Cuando se reinicia un proceso y para volver a asociar cada file descriptor a su archivo original, CRAK utiliza la system call *dup2()*. Este system call duplica un descriptor de archivo de manera tal que tanto el original como su replica hacen referencia al mismo archivo. A continuación se muestra como CRAK utiliza *dup2()*:

```
int open_force (int fd, char * filename,
                int flags, int mode)
{
int ret = open(filename, flags, mode);
// open the file

if (ret < 0 || ret == fd) return ret;
//if error or already the desired descriptor
//just return

if (dup2(ret, fd) < 0) return -1;
// dup the descriptor, note this shouldn't fail

close(ret);
// close the original descriptor
return fd;
}
```

Esta función fuerza a abrir un archivo con el descriptor de archivo especificado.

Para pipes el problema es más complejo porque involucra un grupo de procesos, pero la idea es la misma. Mas adelante se discute la migración de procesos paralelos.

La implementación actual de CRAK sólo soporta guardar los paths de los archivos abiertos, pero no sus contenidos. Por lo tanto, sólo funciona con un file system global o distribuido.

### 3.1.4. Otros Estados

CRAK, también hace checkpoint y restart de los siguientes estados:

- **Directorio de trabajo actual**. Es recuperado llamando *chdir()* en espacio de usuario.
- **Modo del Terminal**. Ciertas aplicaciones setean modos especiales de terminal, como vi o emacs. Esto es hecho desde espacio de usuario con las funciones *tcgetattr()* y *tcsetattr()*.
- **Manejadores de Señanles**. Esto se hace en espacio de kernel.

[3] Idem anterior.

### 3.1.5. Procesos paralelos

Pocos paquetes soportan checkpoint/restart para procesos paralelos. *Epckpt* lo soporta pasando un flag en la invocación a la rutina para indicar que debe hacerse un volcado de determinado proceso y todos sus hijos directos. Esto se implementa completamente en el kernel. Pueden mencionarse dos grandes desventajas de esta alternativa:

- **Inflexibilidad**: sólo soporta una estructura de familia de procesos, un padre y varios hijos. Ya que está harcodeado en el kernel, es dificil de extender.
- **Complejidad**: La complejidad en la operación de checkpoint de grupos de procesos reside en la sincronización.

Todos los procesos que se ejecutan en paralelo deberían ser volcados a disco en simultáneo. Por ejemplo, si dos procesos están hablando entre sí via un pipe, suspender y matar a uno de ellos causará un *broken pipe* y probablemente la muerte del otro antes de que pueda ser volcado a disco. *Epckpt* utiliza un esquema complejo de sincronización implementado parte en espacio de kernel y parte en espacio de usuario. CRAK ofrece un esfuerzo significativo para mejorar esta solución. En lugar de utilizar semáforos se usan señales para sincronizar los procesos, y casi todo el soporte se implementa en espacio de usuario:

- **Simplificación de la tarea del kernel:** El kernel sólo debe hacer checkpoint de un proceso por vez. No necesita conocer nada acerca de procesos paralelos.
- **Sincronización simplificada:** Se realiza mediante señales. Para hacer la operación de checkpoint, primero se envía una señal *SIGSTOP*a todos los procesos en el grupo a ser volcados. Para reiniciar los procesos, se utilizan los parámetros de la función *restart()*comentados anteriormente.
- **Mayor flexibilidad:** al mover el soporte al espacio de usuario, es potencialmente posible tratar cualquier tipo de jerarquía de proceso.

### 3.1.6. Sockets

La migración de sockets es la parte que mayor desafío presenta a las soluciones de checkpoint/restart para migración de procesos. Su complejidad radica en el hecho de que hay dos máquinas involucradas.

CRAK implementó un prototipo para sockets TCP/IPv4, logrando éxito para ciertas aplicaciones de red.

### 3.1.7. Aproximaciones para la migración de sockets

Cuando se migra un proceso, es necesario encontrar la forma de mantener la conexión de red, para que una vez reiniciado el proceso puede continuar la comunicación con el sitio remoto. Una propuesta es la de mantener un procesos *stub* en el nodo original cuya responsabilidad es la de mantener la conexión original y retransmitir el tráfico desde y hacia el nuevo nodo. El otro extremo no necesita conocer esta implementación. El gran problema de este esquema es que cada proceso queda ligado al primer nodo en el que es ejecutado, el cual suele ser llamado *home node*, y por lo tanto si éste nodo cae, la conexión y tal vez el proceso mueren. Esto contradice varios de los objetivos por los cuales se plantea la migración de procesos. En CRAK, se adoptó un mecanismo diferente. Cuando se migra un proceso, se salva la pila de red y se genera una nueva conexión en el nuevo nodo, para lo cual es necesario notificar al extremo remoto de la migración para que conozca la nueva dirección. De esta forma, cuando el proceso es migrado no sigue dependiendo del *home node*, pero existe el problema de que ahora debe ser posible acceder al nodo remoto y cambiar la información de su socket.

### 3.1.8. La estructura sock

Dentro del kernel Linux, todos los tipos de sockets (inet, unix) tienen asociada una estructura llamada *socket*. Para la familia inet, socket tiene un puntero a la estructura *sock*, la cual contiene toda la información específica relacionada con la pila TCP/IP.

Un *sock* se identifica por una tupla única (protocolo, dirección origen, puerto origen, dirección destino y puerto destino). Durante la migración, es necesario cambiar esta información. Por ejemplo, para el proceso migrado se debe modificar la dirección y puerto origen, y para el proceso remoto se debe cambiar su dirección destino y posiblemente el puerto destino. Pero no es tan sencillo como modi-

ficar estos valores en la estructura *sock*, ya que ésta tiene interacciones con algunas otras estructuras de datos del kernel.

Los socks son colocados en varias tablas de hashing para hacer búsquedas eficientes. Las más importantes son el hash de bind (sockets que invocaron la función *bind()*), el hash de listening (*listen()*) y el hash de established (sockets ya conectados). Por ejemplo, cuando llega un paquete perteneciente a una conexión establecida, el kernel buscará en el hash de established para encontrar la estructura sock correspondiente, basado en la clave de hash conformada por la tupla de direcciones incluida en el paquete.

Cuando se modifican sus direcciones o puertos, es necesario asegurarse que el *sock* siga estando en las tablas de hash correctas. Otro problema se genera con la información de enrutamiento. Cada estructura *sock* tiene una estructura llamda *rtable* que cachea la decisión de enrutamiento relacionada con los paquetes de su conexión, y de esta forma se evita hacer búsquedas en la tabla de enrutamiento global (en Linux, llamada Forwarding Information Base, o FIB) cada vez que se envía un paquete. Si se modifica la dirección remota, debe actualizarse la información en la cache.

#### 3.1.9. Tres pasos para migrar un socket

Para la siguiente explicación, llamaremos A al proceso al que se hará checkpoint, B al proceso remoto en la conexión TCP y C al nuevo nodo al cual se migrará el proceso. El proceso es sencillo: en primer lugar se hace checkpoint de A y se deja una conexión a B abierta. Luego, se reinicia A en C y se recupera el socket a un estado established, apuntando nuevamente a B. Por último, se modifica B para que apunte a C, y así recuperar completamente la conexión.

**Cerrar a medias la conexión** Cuando A es suspendido, todos los archivos abiertos serán cerrados. Para "cerrar"un socket TCP, se enviaría un paquete *FIN* a B terminando la conexión. Pero para poder migrar la conexión sin molestar a B, CRAK modifica manualmente la estructura *sock*, poniendo el socket en estado *TCP_ TIME_ WAIT*, y como consecuencia no se enviará un paquete *FIN* a B cuando A sea suspendido. También se salva la información relevante de la pila TCP, principalmente los números de secuencia, para poder continuar más tarde con el intercambio de segmentos.

**Recuperar a medias la conexión** Cuando se reinicia A se reconstruyen todos los sockets. La idea es establecer una nueva conexión con C sin invocar a *connect()* o enviar algún paquete. Esto se implementa tanto en espacio de usuario como en espacio de kernel. En espacio de usuario, se invoca a *socket()* y a *bind()* para crear el socket. En espacio de kernel, se actualizan la dirección y puertos remotos, y otra información de la pila TCP que fue guardada en el checkpoint. También será necesario acomodar a la estructura en las tablas de hash correspondientes y actualizar la cache *rtable*. Por último, se setea manualmente el estado del socket a *TCP_ ESTABLISHED*. Por otro lado, los sockets que sólo están escuchando pueden ser recuperados sencillamente invocando *listen()* en espacio de usuario, ya que no involucran una conexión con otro extremo.

**Recuperación total de la conexión** El último paso es modificar la direccion y puerto destino del socket en B para que apunten al nuevo socket creado en C. Cuando A es reiniciado, *restart()* invocará a la función *chsock()* en B, la cual debe invocar código remoto para realizar las modificaciones. Actualmente, CRAK implementa la ejecución remota usando rsh o ssh, pero esto puede ser reemplazado por algún protocolo ad-hoc.

# Apéndice

# A. Variable de ambiente LD_PRELOAD

Las implementaciones modernas de ld.so (el linkeador dinámico en GNU/Linux) examinan la variable de ambiente LD_PRELOAD la cual representa una lista de librerías dinámicas compartidas que serán cargadas antes que las que fueron linkeadas dinámicamente durante la compilación. Si una función se encuentra implementada tanto en una de las librerías especificadas por LD_PRELOAD como en alguna de las otras librerías (por ejemplo en la glibc), se invocará la primer implementación. Es decir, LD_PRELOAD suele utilizarse para interceptar dinámicamente llamadas a funciones de librerías o invocaciones a system calls (que también se hacen a través de librerías). Sin embargo,

es posible referfenciar a la función originial utilizando dlopen(3) o syscall(2).

Entre las desventajas de utilizar LD_PRELOAD, se pueden mencionar que su uso se considera como un acto voluntario por parte de usuario ya que un usuario puede evitar, de forma trivial, el intento de intercepción de funciones. Además, por razones de seguridad, LD_PRELOAD se deshabilita en binarios con el bit suid seteado.

# B. Autonomic Computing System

El cómputo autónomo se basa en los sistemas biológicos, concretamente del sistema nervioso central. La intención es diseñar y construir sistemas de cómputo capaces de correr por ellos mismos, ajustándose a circunstancias variables y preparando sus recursos para manejar de manera eficiente las cargas de trabajo. Estos sistemas autónomos deben anticipar las necesidades y permitir a los usuarios concentrarse en lo que quieren obtener, es decir, que los usuarios puedan decirle al sistema: .ªjusta la seguridad", por ejemplo, y que el mismo se encargue de los detalles de la implementación.

# C. Gang schedulling (Planificación de grupos)

Desde el punto de vista de la sincronización, sería ideal que las esperas se redujeran a lo que los programadores de cualquier aplicación paralela estiman. Para ello, tendrían que estar corriendo a la vez todos los procesos ejecutables de una misma aplicación. Es lo que se conoce como coscheduling, gang scheduling o task forces.

La replanificación del conjunto de procesos se hace sobre el mismo conjunto de procesadores, lo que puede ser bueno para mantener la coherencia de la cache. De hecho, cada proceso se replanifica siempre sobre el mismo procesador y así se puede puede obtener incluso un entorno estático de ejecución para el software (asignar la memoria más cercana a dicho procesador,por ejemplo) que se mantendrá durante toda la vida de la aplicación.

El problema principal del gang scheduling es su control centralizado, que puede conducir a un cuello de botella. Además, se llega con facilidad a la fragmentación de procesadores, con infrautilización del sistema. En simulaciones hechas, la utilización del sistema aumenta al aumentar el time slice, principalmente por la reducción de cambios de contexto. Hay en general pocos datos sobre esta política y los resultados pueden dar muy diferentes (opuestos) si se tiene en cuenta o no los problemas de mantener la coherencia de la cache: hay que tener en cuenta en estas políticas que, al estar corriendo a la vez todos los flujos de una aplicación, hay una posibilidad muy elevada de estar accediendo a los mismos datos y, por tanto, invalidándose la memorias caches continuamente. Generalmente, son políticas que se utilizan cuando el modelo de comunicación entre los procesos de una aplicación es el pasaje de mensajes, para reducir las esperas de sincronización en las comunicaciones.

# D. Pagina de Manual de SETJMP y LONGJMP

longjmp y siglongjmp - salto no local a un contexto de pila salvaguardado

**SINOPSIS**

```
#include <setjmp.h>

void longjmp(jmp_buf env, int val);
void siglongjmp(sigjmp_buf env, int val);
```

**DESCRIPCIÓN**

longjmp() y setjmp() son útiles para tratar con errores e interrupciones encontrados en una subrutina de bajo nivel de un programa. longjmp() restaura el entorno salvaguardado por la última llamada a setjmp() con el argumento env correspondiente. Después de que longjmp() haya acabado, la ejecución del programa continúa como si la llamada correspondiente a setjmp() simplemente hubiera devuelto el valor val. longjmp() no puede hacer que se devuelva 0. Si se llama a longjmp() con un segundo argumento de valor 0, se devuelve 1 en su lugar. siglongjmp() es similar a longjmp() excepto en el tipo de su argumento env. Si la llamada a sigsetjmp() que establece este env empleó una opción savesigs distinta de cero, siglongjmp() también restaura el conjunto de señales bloqueadas.

A continuación se incluye un ejemplo del uso de estas funciones:

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

void salto( jmp_buf saltimbanqui, int v )
{
   longjmp( saltimbanqui, v );
}

int main()
{
   int valor;
   jmp_buf entorno;
   printf( "longjmp y setjmp son una
forma de simular el \'goto\'\n\n" );

   valor = setjmp( entorno );
/* Volveremos aqu¡*/
   if( valor != 0 )
   {
      printf( "Longjmp con el
  valor: %d\n", valor );
      exit( valor );
   }
   printf( "Vamos a saltar ... \n" );
   salto( entorno, 1 );
/* Salta al momento de setjmp() */
   return 0;
}
```

# E. Kernel Threads

Los sistemas Unix tradicionales delegan algunas tareas críticas a procesos que corren intermitentemente. Entre estas tareas se pueden mencionar hacer flush de las caches de disco, swapear páginas poco usadas a disco o servir conexiones de red, etc. No sería eficiente realizar estas tareas en un modo secuencial, se obtienen mejores resultados de performance si son planificadas en backgraund. Ya que generalemte estos trabajos se realizan en modo kernel, los sistemas operativos modernos los delegan a threads del kernel, los cuales no ocasionan cambios de contextos innecesarios para pasar a modo usuario. En Linux, los threads del kernel se diferencian de los procesos tradicionales en los siguientes aspectos:

- Cada thread ejecuta una única función C del kernel, mientras que los procesos ejecutan funciones del kernel a través de invocaciones a system calls.
- Los threads corren sólo en modo kernel, mientras que los procesos pueden implicar cambios de contexto de modo usuario a modo kernel y viceversa.
- Los threads del kernel acceden únicamente al espacio de direccionamiento del kernel (direcciones virtuales mayores que la constante PAGE_OFFSET), mientras que los procesos pueden utilizar el espacio de direccionamiento completo de 4GB (por supuesto sólo accederan a estructuras del kernel a través de su api).

# Referencias


[1] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa y S. Jiang. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. Proceedings of the 19th IEEE International Prallel and Distributed Processing Symposiom 2005.

[2] Eric Roman. A Survey of Checkpoint/Restart Implementations. Lawrence Berkeley National Laboratory. 2002.

[3] Hua Zhong y Jason Nieh. CRAK: Linux Checkpoint/Restart As a Kernel Module. Department of Computer Science. Columbia University. Technical Report CUCS-014-01. November 2001. http://www.ncl.cs.columbia.edu/research /migrate/crak.html

[4] E. Pinheiro. EPCKPT. http://www.research.rutgers.edu/ẽdpin/epckpt.