Un mecanismo de IPC de microkernel embebido en el kernel de Linux

Pablo Pessolani
Departamento de Ingeniería en Sistemas de Información
Facultad Regional Santa Fe
Universidad Tecnológica Nacional
Santa Fe, Argentina
ppessolani@frsf.utn.edu.ar

Toni Cortes*
Barcelona Supercomputing Center y
Departamento de Arquitectura de Computadores
Universitat Politécnica de Catalunya
Barcelona - España
toni.cortes@bsc.es

Silvio Gonnet
CIDISI- Instituto de Desarrollo y Diseño
(INGAR), CONICET
Universidad Tecnológica Nacional
Santa Fe - Argentina
sgonnet@santafe-conicet.gov.ar

Fernando G. Tinetti#
III-LIDI
Facultad de Informática
Universidad Nacional de La Plata
La Plata, Argentina
Comisión de Inv. Científicas, Prov. Bs. As.
fernando@info.unlp.edu.ar

RESUMEN

Existe una marcada tendencia en la industria de comercializar procesadores con múltiples núcleos, conocidos como multi-cores. Se prevé que en el mediano plazo la cantidad de núcleos aumente significativamente hasta miles de núcleos por procesador [1] denominados many-cores. Esta tendencia requiere de sistemas operativos (OS) que puedan aprovechar éstas tecnologías y que adapten su funcionamiento de tal forma de que no se transformen en el cuello de botella de las aplicaciones que ejecutan sobre ellos como consecuencia de la contención de los recursos que comparten [2].

Los OS basados en microkernel, los OS multi-kernel, algunos exokernels, y ciertas tecnologías de virtualización ofrecen ventajas significativas en su diseño para su adaptación a sistemas many-cores. Estas arquitecturas requieren de un mecanismo de transferencia de mensajes para comunicar entidades tales como procesos, hilos, kernels o máquinas virtuales.

En este artículo se presentan el trabajo de investigación y desarrollo de un mecanismo de IPC basado en Minix 3 (denominado M3-IPC) embebido dentro del kernel de Linux. M3-IPC permite incorporar servicios de un OS basado en microkernel (Minix) dentro de un OS monolítico (Linux) conformando un sistema híbrido donde pueden convivir aplicaciones y servicios de ambas arquitecturas y adaptarlos de manera no abrupta a los sistemas many-cores.

Palabras claves: IPC, many-cores, microkernel.

* Este trabajo está siendo subvencionado parcialmente por parte del Ministerio de Ciencia y Tecnología de España por la ayuda TIN2007-60625 y del Gobierno Catalán por la ayuda 2009-SGR-980
# Investigador CIC Provincia de Buenos Aires

CONTEXTO

Este artículo forma parte de los trabajos de investigación y desarrollo del proyecto presentado en WICC 2012 [3], denominado: "*Sistema de Virtualización con Recursos Distribuidos*" (en inglés DRVS). Este proyecto de I/D involucra a investigadores de varios laboratorios y centros de investigación (ver afiliaciones de los autores), en un área que por su amplitud requiere de múltiples enfoques. Más específicamente, se deriva de las tareas de investigación en el contexto de los programas de postgrado de la Facultad de Informática de la UNLP.

INTRODUCCION

La tendencia en la industria muestra que los computadores de uso general dispondrán de cientos e incluso miles de núcleos procesadores. En su gran mayoría, el software desarrollado se basa en un modelo de memoria compartida consistente. Para mantener la consistencia de memoria entre los distintos núcleos, los sistemas multi-core utilizan protocolos de invalidación de cache o de actualización de cache. A medida que aumenta el número de núcleos, estos protocolos presentan una mayor sobrecarga y latencia lo que los hace poco escalables. Por esta razón, tal como se argumenta en [4], el hardware futuro se parecerá más a un sistema distribuido unido por una red interna dentro del mismo chip que a un sistema de memoria compartida [5].

Los OS basados en microkernel como Minix[6] o L4 [7], el exokernel Corey[8], los OS multi-kernel como Barrelfish [4], Nix [9] o FOS [10] y algunas tecnologías de virtualización [11] ofrecen ventajas significativas en su diseño para su adaptación a sistemas many-cores. Estas arquitecturas utilizan la transferencia de mensajes como mecanismo para comunicar entidades tales como procesos, hilos, kernels, hipervisores o máquinas virtuales. Los

sistemas de microkernel como FOS[10] asignan un núcleo o core para cada servidor o device driver por lo que se evita el cambio de contexto del emisor al receptor durante la transferencia de mensajes. Los sistemas multikernel, asignan un kernel a cada núcleo comunicándose entre sí mediante IPC.

Se hace evidente la importancia y criticidad que representan para estos sistemas los mecanismos de IPC que utilizan.

LÍNEAS DE INVESTIGACIÓN Y DESARROLLO

La línea de I/D de este proyecto refiere a desarrollar un modelo de DRVS [3]. Un DRVS es un Sistema de Virtualización con Recursos Distribuidos que comparte características con los Sistemas de Imagen Unica (SSI), con la virtualización basada en OS y con los OS de microkernel multiservidores. Las aplicaciones y procesos servidores se ejecutan en un entorno o compartimento que se denomina Máquina Virtual (VM). A diferencia de otras tecnologías de virtualización estas VMs no están confinadas a un solo nodo, sino que pueden abarcar un subconjunto de nodos. Para el DRVS un nodo es un computador de un cluster o un núcleo de un sistema many-cores.

Dado que los procesos de una VM se comunicarán mediante transferencia de mensajes, son fundamentales la robutez, sencillez y rendimiento del mecanismo de IPC. Por sus características, el DRVS requiere comunicar procesos co-residentes en el mismo nodo (locales) y procesos residentes en diferentes nodos (remotos).

El proyecto del DRVS establece como requerimiento para el mecanismo de IPC su compatibilidad con las APIs de Minix 3, de alli surge la denominación de M3-IPC.

La decisión de crear un nuevo mecanismo de IPC dentro del kernel de Linux y no utilizar los mecanismos de IPC existentes se debió básicamente a que estos últimos no ofrecen la totalidad de las características requeridas por el proyecto de I/D:

- *Messages Queues, Pipes, FIFOs, Unix Sockets:* No disponen de la posibilidad de comunicarse con procesos remotos de forma transparente y uniforme.
- *RPC, UDP/TCP Sockets:* Tienen gran latencia en la inicialización de los canales de comunicación y bajo rendimiento de las transferencias entre procesos co-residentes (ver Micro-benchmarks).

Para el diseño de los mecanismos de IPC del DRVS se establecieron los siguientes principios:

- *Simplicidad*: Debe proveerse un número pequeño de APIs básicas, de rápida compresión por parte de los programadores y que permita una sencilla traza de mensajes para la depuración de las aplicaciones.
- *Transparencia*: Las APIs de IPC deben ser transparentes a la ubicación de los procesos, es decir, no debe haber distinción entre procesos locales y remotos a fin de facilitar la programación.
- *Aislamiento*: Debe soportar VMs como compartimentos de procesos de tal forma que solo se admitirá la comunicación entre procesos pertenecientes a la misma VM.
- *Escalabilidad*: Debe aprovechar eficientemente los recursos de un cluster o de un sistema many-cores.
- *Rendimiento*: El rendimiento en computadores de pocos núcleos debe ser equivalente a los mecanismos de IPC más veloces de los que se dispone en Linux.

Un DRVS es un contenedor de VMs que abarca a todos los nodos (computadores de un cluster o núcleos de un sistema many-cores). Una VM es un contenedor de procesos que abarca a un subconjunto de nodos, por lo que los procesos de una VM pueden estar ejecutando en diferentes nodos. Un nodo puede ser compartido por diferentes VMs, y por lo tanto pueden ejecutarse en él procesos de diferentes VMs, pero los mismos no tienen posibilidad de comunicarse entre sí (aislamiento).

En la primera etapa (que es la que se presenta en éste artículo) se procedió al diseño, desarrollo, implementación y pruebas de rendimiento de un mecanismo de IPC para comunicar procesos locales, es decir que se encuentran compartiendo un nodo. Para esta etapa, se estableció como objetivo desarrollar el software de IPC, programas de prueba y micro-benchmarks sobre Linux para plataforma x86.

Si bien el objetivo primario es satisfacer los requerimientos del proyecto, también se pretende brindar una nueva herramienta de comunicación para Linux que facilite nuevos desarrollos, como por ejemplo un microkernel como módulo instalable de un OS co-residente con el propio Linux de forma equivalente a como lo hacen algunos sub-kernels de tiempo real tales como RTLinux [12] o RTAI [13].

La configuración de los parámetros de operación del DRVS (número de VMs, número de nodos, etc.) puede hacerse en forma dinámica especificándolos como argumentos del comando de inicialización u obtenerse a partir de registros de un directorio LDAP.

Para poder utilizar M3-IPC el DRVS debe estar inicializado. Como el DRVS estará conformado por N nodos, debe asignársele un *nodeid* a cada nodo durante la inicialización. Esto puede hacerse directamente al iniciar Linux incluyendo el *nodeid* en la línea de parámetros de booteo, en un script de arranque, directamente ingresando el comando en forma manual o como parte de un programa invocando a *mnx_drvs_init()*.

Luego de inicializado el DRVS, se deben inicializar las VMs. Cada VM tiene asignado un *vmid*, una serie de parámetros de configuración, los nodos que abarca y un nombre que facilita su identificación a los administradores del DRVS. Los parámetros de configuración de las VMs pueden establecerse por línea de comandos, como parte de un programa utilizando *mnx_vm_init(),* o a partir de registros de un directorio LDAP.

Los procesos Linux ordinarios no pueden utilizar M3-IPC, previamente deben registrase a una VM mediante *mnx_bind().* Un proceso solo puede pertenecer a una VM. A cada proceso registrado se le asigna un identificador denominado *endpoint* que es único dentro de la VM y que será utilizado en las primitivas de IPC como origen/destino de los mensajes y demás APIs. Luego de registrarse se establece una relación biunívoca entre el PID de Linux y el

endpoint de una VM (PID⇔(vmid, endpoint)) y el proceso queda habilitado para utilizar las APIs de M3-IPC refiriéndose a los otros procesos mediante sus *endpoints* (sin mencionar el *vmid*, ya que es idéntico al propio).

Los mensajes en M3-IPC, al igual que en Minix, tienen diferentes formatos y tamaño fijo de 36 bytes para una plataforma x86 de 32 bits. La copia de bloques de datos entre procesos admite tamaños de bloques de hasta 4 MB.

La de-registración de un proceso de una VM se hace en forma explícita mediante *mnx_unbind()*. Si el proceso finaliza en forma normal con *exit()* o porque ha recibido una señal que lo finaliza sin realizar *mnx_unbind(),* el sistema lo hace en forma implícita. Si el proceso finalizado esperaba enviar mensajes a otro proceso, u otros procesos esperaban recibir mensajes de éste, los códigos de retornos de las APIs le informarán que el *endpoint* ha finalizado.

Para finalizar una VM se utiliza *mnx_vm_end(),* que envía un SIGPIPE a todos los procesos registrados en esa VM, y luego retorna sus recursos (memoria) a Linux.

Para finalizar el DRVS se utiliza *mnx_drvs_end(),*que finaliza todas las VMs, y por consecuencia todos los procesos pertenecientes a las VMs.

Las siguientes son las APIs para la transferencia de mensajes y copias de bloques de datos:

- *mnx_send():* Envía un mensaje de tamaño fijo desde el espacio de direcciones del proceso emisor al espacio de direcciones del proceso receptor. Si el proceso receptor no está esperando el mensaje, el emisor se bloquea.
- *mnx_receive():* Recibe un mensaje en el espacio de direcciones del proceso emisor. Si no hay pendiente de entrega un mensaje requerido, el receptor se bloquea.
- *mnx_sendrec():* Envía un mensaje a un proceso y espera por un mensaje de respuesta. El proceso permanece bloqueado hasta recibir la respuesta.
- *mnx_notify():* Envía un mensaje de notificación (sin contenido, similar a una señal o signal) a un proceso. Si el proceso receptor no está esperando dicho mensaje, queda registrada la notificación y el proceso receptor la recibirá cuando ejecute el próximo *mnx_receive().*
- *mnx_vcopy():* Permite copiar bloques de datos entre los espacios de direcciones de dos procesos.

El mecanismo de IPC resultante, su rendimiento y la sencillez de sus APIs permiten construir nuevos servicios equivalentes a los brindados en los OSs de microkernel en un OS monolítico como es Linux.

Trabajos Relacionados

Existen proyectos similares a M3-IPC. Estos son:

- *Synchronous Interprocess Messaging Project for LINUX (SIMPL) [14]:* Es un proyecto open source para Linux que permite disponer de las APIs estilo QNX de transferencia síncrona de mensajes. SIMPL es una librería que utiliza FIFOs para las transferencias locales y TCP para las transferencias remotas. El rendimiento es bastante pobre dado que utiliza *pipes* para la sincronización de procesos y shared memory (*shmem*) para la copia de mensajes y datos.
- *Send/Receive/Reply (SSR)[15]:* Es una implementación de las APIs de QNX4 en Linux. A diferencia de SIMPL, SSR es un módulo instalable de kernel. La comunicación entre los procesos de usuario y el modulo de kernel se realiza utilizando operaciones IOCTL de un archivo de dispositivo.

RESULTADOS Y OBJETIVOS

En esta sección se resumen las Características de M3-IPC detallando algunos aspectos referentes al diseño, y a fin de contrastar resultados con los objetivos de rendimiento establecidos para esta etapa del proyecto se realizaron una serie Micro-benchmarks. Estos tests permiten comparar el desempeño de M3-IPC frente a otros mecanismos equivalentes disponibles en Linux, y frente al IPC de Minix en dos versiones que utilizan diferentes tecnologías de administración de memoria.

Características de M3-IPC

Uno de los objetivos de M3-IPC es brindar un mecanismos de transferencia de mensajes entre procesos sencillo y fácil de comprender construido en base a componentes de software existentes en Linux.

Las características del M3-IPC resultante de esta etapa del proyecto de I/D son las siguientes:

- Es un mecanismo de IPC sincrónico sin utilización de buffers en el kernel y compatible con APIs de Minix 3.
- Utiliza el mismo vector interrupciones que Minix, que es diferente al utilizado por Linux para las llamadas al sistema. Esto permite que ambas APIs convivan en el mismo kernel y que aplicaciones desarrolladas para Minix seguir utilizando el mismo vector. Para habilitar este nuevo vector de interrupción se debió modificar el kernel de Linux, lo que seguramente dificultará el mantenimiento de M3-IPC en kernels futuros.
- Permite a los programadores configurar VMs para aislar el IPC entre sus aplicaciones.
- A fin de aprovechar el paralelismo en sistemas SMP y many-cores:
  - Las APIs soportan hilos (*threads*) y utilizan *mutexes* del kernel de Linux para realizar la exclusión mutua.
  - Se maximizó la granularidad de las regiones críticas a nivel de procesos utilizando *spinlocks y mutexes.* Esto permite el paralelismo de transferencias de mensajes entre múltiples pares de procesos diferentes.
  - La utilización de VMs permite el paralelismo de transferencias de mensajes dentro de VMs diferentes.
- Las estructuras de datos de los procesos registrados en una VM (*struct proc*), se encuentran alineadas con las líneas de cache L1 de CPU a fin de reducir el tiempo de acceso.
- Utiliza *Task Queues* y el mecanismo de *Event waiting* de Linux para la sincronización de procesos e hilos.
- Utiliza *Reference Counters* del kernel de Linux para controlar el número de procesos registrados en una VM.
- La copia de bloques de datos entre procesos y la transferencia de mensajes se realiza desde el espacio de

direcciones del proceso origen al espacio del proceso destino sin copia intermedia en un buffer del kernel.

- La copia de bloques de datos distingue entre la copia de bytes y la copia de páginas (más eficiente).
- La información del sistema M3-IPC, el DRVS, sus VMs, nodos y procesos se encuentra integrada al sistema de archivos de Linux */proc*.
- Desarrollado inicialmente para plataforma Linux x86 en lenguaje C como Open Source.

Micro-benchmarks

Uno de los principios de diseño establece que el rendimiento esperado en computadores de pocos núcleos debe ser equivalente a los mecanismos de IPC mas veloces de los que se dispone en Linux.

A fin de evaluar el rendimiento de M3-IPC se prepararon micro-benchmarks sobre Linux 2.6.32 x86 de 32 bits para M3-IPC y para otros mecanismos de IPC tales como:

- *Message Queues:* Se realizaron benchmarks propios.
- *FIFOs, pipes, Unix Sockets, TCP Sockets:* Se utilizó *ipc-bench* [16] (modificado).
- *SRR, SIMPL:* Se utilizaron los benchmarks que forman parte de la distribución.

En todos los casos, los programas de pruebas fueron realizados o modificados de tal forma que las transferencias de mensajes y copias de bloques de datos se asemejen a la semántica M3-IPC.

Por otro lado se realizaron micro-benchmarks para Minix 3.1.2 que utiliza administración de memoria segmentada y Minix 3.2.0 que utiliza administración de memoria por paginación. En ambos versiones debieron alterarse ligeramente los microkernels y el administrador de procesos (PM) para que admita la realización de tests equivalentes a los de M3-IPC. Es importante aclarar que Minix 3.1.2 soporta sistemas monoprocesadores y Minix 3.2.0 se encuentra en etapa experimental de soporte SMP.

El equipo utilizado para realizar los micro-benchmarks es un dual-core Intel Core 2 E7400 @ 2.80GHz con cache de 3072 kB y línea de cache de 64 bytes.

Transferencias de Mensajes

Las pruebas de rendimiento de transferencia de mensajes fueron denominadas:

- *IPC1*: Un proceso cliente envía un mensaje con *mnx_send()* a un proceso servidor que espera con un *mnx_receive()*. El servidor responde con un *mnx_send()* al cliente que espera con un *mnx_receive()*.
- *IPC2*: Un proceso cliente envía un mensaje con *mnx_sendrec()* a un proceso servidor y espera por la respuesta. El servidor espera la petición del cliente con un *mnx_receive()* y le responde con un *mnx_send()*.
- *IPC3*: Un proceso cliente envía un mensaje con *mnx_notify()* a un proceso servidor y continúa su ejecución hasta hacer un *mnx_receive()* esperando por la respuesta. El servidor que espera con un *mnx_receive()* responde con un *mnx_notify()* al cliente.

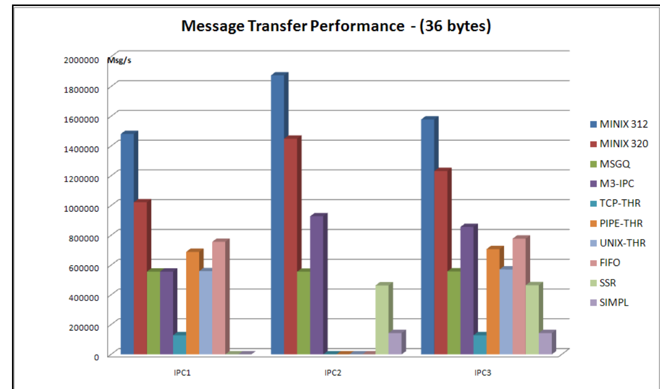

Figura 1. Rendimiento Comparativo de Transferencia de Mensajes.

En la Figura 1 se presentan los resultados comparativos para los tres tests de transferencia de mensajes. A fin de realizar los tests equitativos entre Linux y Minix, se anuló en Linux el segundo núcleo de la plataforma dual-core. Se puede apreciar el rendimiento marcadamente superior de las dos versiones de Minix frente a todas los mecanismos de IPC de Linux. Esto básicamente se debe a la sencillez del microkernel de Minix y la complejidad del kernel de Linux (mayor camino crítico). Aún así, M3-IPC muestra el máximo desempeño en dos de los tres tests frente al resto de los mecanismos de IPC de Linux. Cuando se habilitaron los dos núcleos y múltiples pares cliente/servidor, el rendimiento fue de 1.460.000 [Msg/s] para IPC2.

Algunos tests no fueron realizados para ciertos mecanismos de IPC (valores cero en Figura 1) porque no era posible reproducir una semántica equivalente. Para el caso de IPC3 donde se utiliza *mnx_notify(),* se asumió que la transferencia de 1 byte podría considerarse equivalente.

Copia de Bloques de Datos

Previo a la realización de las pruebas de copia de bloques de datos entre procesos se realizó el test *lmbench*[17] que brinda información acerca del rendimiento de copia de datos dentro del espacio de direcciones del mismo proceso (*bw_mem*). El rendimiento de *bm_mem* fue de 16,7 [GB/s] para copia de bloques de 4 kB, estableciendo el límite máximo de rendimiento de cualquier mecanismo de copia de bloques de datos entre diferentes procesos.

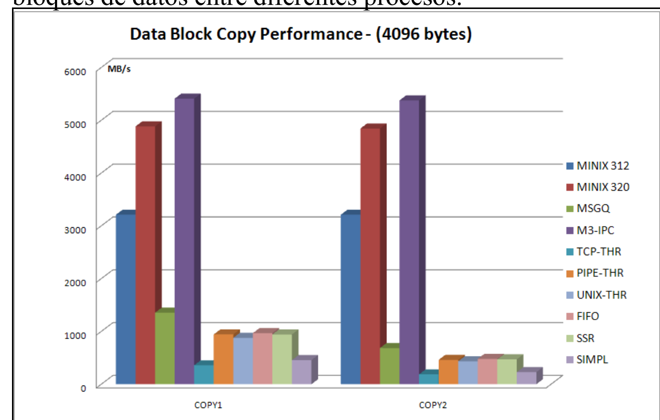

Figura 2. Rendimiento de Copia de Bloques de Datos.

Los test de copias de bloques de datos fueron denominados:

- *COPY1*: Copia un bloque de datos desde el espacio direcciones del proceso que invoca *mnx_vcopy()* hacia el espacio de direcciones de otro proceso.
- *COPY2*: Copia un bloque de datos desde el espacio direcciones de un proceso origen, hacia el espacio de direcciones de otro proceso destino. Un tercer proceso *requester* es el que invoca a *mnx_vcopy()*.

Como muestra la Figura 2, el rendimiento de M3-IPC es máximo, incluso superando a ambas versiones de Minix. Tanto Minix como M3-IPC solo realizan una copia de datos entre el espacio de direcciones del proceso origen al espacio de direcciones del proceso destino. Los otros mecanismos de Linux realizan una doble copia (Origen→Kernel→Destino).

El rendimiento superior de M3-IPC se debe a que utiliza una optimización que ofrece el kernel de Linux que permite copiar páginas completas y que no requiere de cambios de contextos entre los procesos. Esta copia se lleva a cabo en una función que utiliza instrucciones MMX, en cambio Minix utiliza la instrucción MOVSB menos eficiente.

Se deben considerar algunos aspectos referentes a la copia de bloques de datos que pueden alterar su rendimiento:

- *Tamaños de las Cache*: El tamaño de cache L1 para el hardware utilizado es de 32 kB con un tiempo acceso reportado por *lmbench* de 1.07[ns], el tiempo de acceso a cache L2 es de 5.41[ns] y a cache L3 es de 11[ns]. El test *bw-mem* reportó que hay una reducción significativa (~50%) en el rendimiento cuando el tamaño de conjunto de trabajo (2 veces el tamaño de bloque) es mayor al tamaño de cache L1.
- *Alineación de los bloques de datos con respecto a las páginas*: Si los bloques de datos a copiar están alineados con el tamaño de página, se utiliza la copia de páginas completas que es un mecanismo más eficiente debido a utiliza instrucciones MMX como se indicó anteriormente.
- *Alineación de los bloques de datos origen y destino entre sí*: Si la dirección de inicio del bloque de datos origen se encuentra desplazada *src_off* bytes de la alineación de página, y la dirección de inicio del bloque de datos destino se encuentra desplazada *dst_off* bytes, si *src_off≠dst_off*, se duplica el número de ciclos de copia que hay que realizar y se impide realiza copias de páginas completas debiéndose realizar copias de bytes.

Conclusiones

Como resultado de la primera etapa del proyecto de I/D se obtuvo un mecanismo de IPC apto para la transferencia de mensajes entre procesos de un mismo nodo, con un rendimiento similar, incluso mejorado, comparado con otros mecanismos de IPC de Linux.

M3-IPC ha sido desarrollado siguiendo las recomendaciones para su operación eficiente en sistemas multi-cores y many-cores, pero dado que utiliza las facilidades que ofrece el kernel de Linux (mutexes, spinlocks, reference counters, etc.) su escalabilidad está limitada al rendimiento que éstos presentan operando sobre ese tipo de sistemas.

Formación de Recursos Humanos

El trabajo central de la línea de I/D presentada se plantea en principio como una tesis de doctorado, aunque por la amplitud del tema, seguramente dejará abiertas muchas cuestiones que podrán ser tratadas en tesinas de grado, tesis de magister/doctorado o en trabajos de otros investigadores.

En particular, el mecanismo de IPC admite, por ejemplo, la investigación los siguientes tópicos:

– *Seguridad*: M3-IPC no incluye aspectos relativos a seguridad. Podría integrarse con las *Capabilities* de Linux.

– *Integración con Linux Containers(LXC)[18]:* Las VMs de M3-IPC podrían integrarse al mecanismo de contenedores o compartimentos LXC disponible en Linux.

Referencias

1. Shekhar Borkar. "*Thousand core chips: a technology perspective*". In DAC '07 ACM, pages 746–749, 2007.
2. Siddha, S.; Pallipadi; A Mallick. "*Process scheduling challenges in the era of multi-core processors*". Intel Technology Journal, 2007.
3. Pablo Pessolani, Toni Cortes, Silvio Gonnet, Fernando G. Tinetti. "*Sistema de Virtualización con Recursos Distribuidos*". WICC 2012. Pág. 59-43. Argentina, 2012.
4. Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. "*The Multikernel: A new OS architecture for scalable multicore systems*". In Proceedings of the 22nd ACM Symposium on OS Principles, 2009.
5. Sriram Vangal, et al. "*An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS*," l. IEEE Journal of Solid-State Circuits, Vol. 43, No. 1, Jan 2008.
6. Tanenbaum A., Woodhull A., "*Operating Systems Design and Implementation, Third Edition*", Prentice-Hall, 2006.
7. Liedtke J.; "*On µ-kernel construction*"; Symposium on Operating System Principles ACM, 1995.
8. Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. "*Corey: an operating system for many cores*". In Proceedings of OSDI'08. USENIX Association, 43-57. 2008.
9. Francisco J. Ballesteros, Noah Evans, Charles Forsyth, Gorka Guardiola, Jim McKie, Ronald Minnich, Enrique Soriano-Salvador: "*NIX: A Case for a Manycore System for Cloud Computing*". Bell Labs Technical Journal 17(2): 41-54 (2012).
10. Matarneh, R., 2009. "Multi Microkernel Operating Systems for Multi-Core Processors". JCS, 5: 493-500.
11. Carl Gebhardt and Allan Tomlinson, "*Challenges for Inter Virtual Machine Communication*", Technical Report, 2010.
12. M. Barabanov and V. Yodaiken. "*Real-Time Linux*". Linux journal, February 1997.
13. E. Bianchi and L. Dozio. "*Some experiences in fast hard-real time control in user space with RTAI-LXRT*". In Workshop on Real Time Operating Systems and Applications and 2nd Real Time Linux Workshop, November 2000.
14. John Collins and Robert Findlay, "*Programming the SIMPL Way*", Lulu.com, ISBN 0557012708, 2008.
15. SRR- QNX API compatible message passing for Linux. http://www.opcdatahub.com/Docs/booksr.html
16. ipc-bench: A UNIX inter-process communication benchmark. http://www.cl.cam.ac.uk/research/srg/netos/ipc-bench/
17. Carl Staelin and Hewlett-packard Laboratories,"*lmbench: Portable Tools for Performance Analysis*", In USENIX Annual Technical Conference, 279-294, 1996.
18. LXC Linux Containers, http://lxc.sourceforge.net/