Universidad Latina de Costa Rica

BISOF-18 Sistemas Operativos II

Análisis de caso #2 - Resúmenes de videos y presentaciones - Jaime Fung Delgado

**Multiprocessor and Real-Time Scheduling- CAP10**

**1. Concurrencia vs. Paralelismo**

* **Concurrencia**
  + Concepto lógico: múltiples hilos o procesos *progresan* en un mismo intervalo de tiempo.
  + Un solo núcleo puede simular concurrencia alternando procesos (*time-slicing*).
* **Paralelismo**
  + Estado de ejecución: dos o más hilos realmente corren al mismo tiempo.
  + Solo posible con multicore o multiprocesadores.
  + Ejemplo: una laptop con 8 núcleos puede ejecutar 8 hilos en paralelo.

**2. Tipos de Sistemas Paralelos**

**2.1. Multiprocesadores**

* **Tightly coupled (estrechamente acoplados)**:
  + Comparten memoria principal y reloj.
  + Tipos:
    - UMA (Uniform Memory Access): todos los procesadores acceden a la memoria con el mismo tiempo.
    - NUMA (Non-Uniform Memory Access): cada procesador tiene memoria “cercana” más rápida.
* **Ejemplos**: NYU Ultracomputer, RP3, SGI Origin 2000, Sun Ultra HPC.

**2.2. Multicomputadores**

* **Loosely coupled (débilmente acoplados)**:
  + Cada nodo tiene su propia memoria y no comparten reloj.
  + Se comunican mediante mensajes (LAN, Myrinet, etc.).
  + Pueden estar:
    - Colocalizados (ej. clúster HPC).
    - Remotos (sistemas distribuidos).

**2.3. NUMA y afinidad**

* SO **NUMA-aware**: asigna memoria local al CPU que ejecuta un hilo.
* **Processor affinity**: un proceso se “adhiere” a un CPU específico para mejorar rendimiento en caché.

**3. Granularidad de Sincronización**

* **Fina (<20 instrucciones)**: paralelismo dentro de instrucciones, usado en pipelines y SIMD.
* **Media (20-200 instrucciones)**: múltiples hilos de una misma aplicación que interactúan.
* **Gruesa (200-2000 instrucciones)**: procesos independientes en multiprogramación (ej. pipes en Unix).
* **Muy Gruesa (2000-1M instrucciones)**: nodos en red con baja interacción.
* **Independiente**: procesos sin sincronización (ej. time-sharing en SO).

**4. Planificación en Multiprocesadores**

**4.1. Asignación de procesos a CPUs**

* **Estática**
  + Cada proceso se queda en un CPU fijo.
  + Ventajas: menos cambios de contexto, permite *gang scheduling*.
  + Desventajas: CPUs desbalanceados, difícil redistribución.
* **Dinámica (SMP)**
  + Todos los CPUs se autogestionan.
  + Opciones:
    - Cola global compartida.
    - Cola privada por procesador.

**4.2. Modelos de coordinación**

* **Master/Slave**: un CPU gestiona y los demás ejecutan, pero hay riesgo de cuello de botella.
* **Peer**: todos los CPUs pueden ejecutar el SO, es más flexible, pero más complejo.

**5. Estrategias de Multiprocessor Scheduling**

* **Process Scheduling (Planificación de procesos)**
  + Cola única para todos los procesos o múltiples colas según prioridades.
  + Todas las colas alimentan un pool común de procesadores.
  + En sistemas con múltiples procesadores, la disciplina de planificación específica (como Round Robin o SJF) pierde relevancia.
  + FCFS (First-Come, First-Served) o FCFS con prioridad estática suele ser suficiente.
* **Thread Scheduling (Planificación de hilos)**
  + Los hilos se ejecutan independientemente del proceso principal.
  + Una aplicación puede tener varios hilos que cooperan en el mismo espacio de direcciones.
  + Ejecutar hilos en procesadores separados mejora el rendimiento significativamente.
  + Pero si los hilos requieren mucha interacción, el rendimiento puede verse afectado negativamente.
* **Load Sharing (compartición de carga)**
  + Cola global, evita CPUs ociosos.
  + Desventajas: la caché se desaprovecha, cuello de botella en cola global.
* **Gang Scheduling**
  + Hilos relacionados se ejecutan simultáneamente en varios CPUs.
  + Muy útil para aplicaciones que necesitan sincronización frecuente.
* **Dedicated Processor Assignment**
  + Cada hilo se asigna permanentemente a un CPU.
  + Ventaja: elimina cambio de contexto.
  + Desventaja: CPUs pueden quedar ociosos.
* **Dynamic Scheduling**
  + El número de hilos cambia en tiempo de ejecución.
  + SO ajusta la carga dinámicamente.

**6. Sistemas en Tiempo Real (RT)**

* Un sistema es correcto si cumple plazos de tiempo, es decir, el tiempo en que los resultados son producidos, no solo resultados.
* **Hard Real-Time**
  + Perder un deadline = consecuencias críticas (ej. avión, marcapasos).
* **Soft Real-Time**
  + Perder un deadline reduce rendimiento, pero no es fatal (ej. streaming de video).

**Características de un RTOS**

* Determinismo: operaciones dentro de tiempos predecibles.
* Latencias mínimas:
  + *Interrupt latency*: llegada de interrupción -> rutina de servicio.
  + *Dispatch latency*: cambio de un proceso a otro.
* Fiabilidad: tolerancia a fallos (*fail-soft*).
* Estabilidad: cumplir deadlines críticos, aunque los de baja prioridad se pierdan.

**7. Algoritmos de Real-Time Scheduling**

**7.1. Tipos**

* **Static Table-Driven**: tareas periódicas preprogramadas.
* **Static Priority Preemptive**: prioridades fijas, tareas críticas arriba.
* **Dynamic Planning**: el plan se ajusta al llegar nuevas tareas.
* **Dynamic Best Effort**: usa algoritmos como EDF.

**7.2. Earliest Deadline First (EDF)**

* Se elige la tarea cuyo deadline esté más próximo.
* Minimiza la cantidad de tareas que pierden plazos.
* Fácil de implementar, pero no siempre se garantiza factibilidad.

**7.3. Rate Monotonic Scheduling (RMS)**

* Tareas periódicas reciben prioridad inversa a su período.
* Tareas más frecuentes = mayor prioridad.
* RMS es óptimo para tareas periódicas, aunque el uso de CPU no supera 69%.

**8. Problema: Inversión de Prioridades**

* Cuando una tarea baja bloquea un recurso requerido por una de alta prioridad.
* Ejemplo: **Mars Pathfinder (1997)**
  + Sistema se reiniciaba porque tareas bajas bloqueaban un mutex crítico.

**Soluciones**

* **Priority Inheritance**: la tarea baja hereda la prioridad de la alta hasta liberar el recurso.
* **Priority Ceiling**: cada recurso tiene una prioridad máxima predefinida.
* **Deshabilitar interrupciones**: técnica extrema, pero efectiva.

**9. Planificación en SO Reales**

* **VxWorks**
  + RTOS con soporte hard real-time.
  + 256 niveles de prioridad, multitarea, bajo tiempo de interrupción.
* **Linux**
  + **Comportamiento del Kernel**
    - El código estándar del kernel es no-preemptible (no se interrumpe).
    - Las interrupciones de temporizador durante ejecución en kernel establecen el flag need\_resched, lo que provoca replanificación al finalizar la llamada al sistema.
    - Durante operaciones críticas sobre estructuras de datos, se debe:
    - Evitar acceso a memoria de usuario.
    - Deshabilitar interrupciones.
  + **Rutinas de Servicio de Interrupción (ISR)**
    - **Top Half**: ejecuta con interrupciones de igual o menor prioridad deshabilitadas.
    - **Bottom Half**: permite todas las interrupciones.
    - El planificador garantiza que un bottom half no se interrumpa a sí mismo.
    - El kernel puede deshabilitar selectivamente ciertos bottom halves en secciones críticas.
  + **Prioridades y Creditos de Planificacion**
    - Basado en créditos de planificación:
    - Se selecciona el proceso con mayor número de créditos.
    - Cada interrupción de temporizador reduce un crédito.
    - El proceso se suspende cuando no tiene créditos.
    - Si ningún proceso tiene créditos:
    - Se reasignan usando:  
      Créditos = Créditos / 2 + prioridad
  + **Planificación Multiprocesador en Linux**
    - Soporte inicial en kernel 2.0.x.
    - Mejoras en bloqueos finos y subsistemas con hilos en 2.3.x.
    - El planificador otorga un “bonus” si un hilo se replanifica en el mismo CPU, favoreciendo la localidad de caché.
  + *SCHED\_FIFO*: prioridad fija (cola FIFO).
  + *SCHED\_RR*: igual que FIFO, pero con quantum.
  + *SCHED\_OTHER*: procesos normales (no RT).
  + Desde 2.6 el kernel es totalmente *preemptible*.
  + Variante: RTLinux con EDF y RMS.
* **UNIX SVR4**
  + 160 niveles de prioridad en tres rangos: tiempo real, kernel, usuario.
  + Kernel con puntos de preempción seguros.
* **Windows 2000**
  + 32 niveles de prioridad.
  + Niveles 16-31: real-time, 0-15: normales.
  + Round-robin dentro del mismo nivel.

**10. Sistemas Embebidos**

* 98% de los procesadores fabricados se usan en sistemas embebidos (no en PCs).
* **Aplicaciones**:
  + Electrodomésticos: lavadoras, hornos, aires acondicionados.
  + Automóviles: control ABS, inyección.
  + Telecomunicaciones: routers, switches.
  + Médicos: marcapasos, medidores de glucosa.
  + Entretenimiento: MP3, cámaras, consolas.

**Ventajas**

* Tamaño reducido, bajo costo, confiabilidad, bajo consumo de energía.

**Plataformas**

* ARM, PIC, MSP430, 8051, PowerPC, entre otras.
* Tecnologías: SOC (System on Chip), ASIC, FPGA.

**Arquitecturas**

* Bucle simple.
* Controlado por interrupciones.
* Multitarea cooperativa.
* Multitarea preventiva (con RTOS).

**Máquinas Virtuales - CAP**

**1. Idea general y componentes**

Una **máquina virtual (VM)** es una abstracción del hardware físico que convierte a un solo equipo en múltiples entornos de ejecución, como si cada uno fuese una computadora independiente.

* **Elementos**:
  + **Host**: hardware físico (CPU, RAM, discos, NICs).
  + **VMM / Hypervisor**: software (o firmware) que crea/gestiona VMs y presenta interfaz idéntica al hardware al guest.
  + **Guest**: proceso proporcionado con una copia virtual del host

**Beneficios**: consolidación, aislamiento, snapshots, migración en vivo, templating, entornos de desarrollo reproducibles, cloud computing.

**3. Razones para Virtualizar**

1. **Compartir y optimizar recursos:** aprovecha la capacidad del hardware evitando tiempos muertos.
2. **Entorno de desarrollo y pruebas:** se pueden correr distintos SO sin necesidad de hardware adicional.
3. **Consolidación de servidores:** menos consumo eléctrico, menor espacio físico y menos refrigeración.
4. **Maximizar disponibilidad:** recuperación ante fallos (ej. mover una VM en caso de “Blue Screen of Death”).
5. **Elasticidad:** asignar recursos dinámicamente cuando y donde se necesitan.
6. **Migración:** mover VMs entre servidores sin interrumpir el servicio.
7. **Compatibilidad con sistemas heredados (legacy).**
8. **Impacto en redes:** mejora escalabilidad, rendimiento y resiliencia.

**4. Características**

* **Aislamiento:** un fallo o virus en una VM no afecta a las demás ni al host.
* **Snapshots:** guardar el estado de una VM y restaurarlo en cualquier momento.
* **Clonación:** duplicar una VM y correr varias copias.
* **Live migration:** mover VMs en ejecución entre hosts sin interrumpir usuarios.
* **Plantillas:** crear VMs preconfiguradas (ej. SO + aplicaciones) y replicarlas.
* **Cloud computing:** la base de servicios como AWS, Azure o Google Cloud.

**5. Tipos de hypervisors / VMMs**

* **Type 0 hypervisors**: Implementado directamente en hardware o firmware (ej. IBM LPARs). Menos flexibles, pero muy eficientes.
* **Type 1 hypervisors**: hypervisor que corre directamente sobre hardware (VMware ESXi, Xen en modo bare, Citrix XenServer). Ofrecen servicios similares a OS pero centrados en virtualización.
* **Type 1 hypervisors**: OS general (Linux/Windows) que incorpora funciones de hypervisor (KVM, Hyper-V).
* **Type 2 hypervisors**: hypervisor como aplicación encima de un OS host (VMware Workstation, VirtualBox, Parallels). Menos performantes pero más cómodos para desktop.

**6. Otras variantes**

* **Paravirtualization**: el guest OS se modifica para cooperar con VMM (mejor I/O y rendimiento). Ej.: antiguamente Xen. Reduce necesidad de trampas/binary translation.
* **Programming-environment virtualization**: no virtualiza hardware sino una plataforma (JVM, .NET CLR). Facilita portabilidad y features (GC, JIT).
* **Emuladores**: Correr un sistema en hardware distinto (ej. emular un procesador ARM en x86). Muy flexible, pero lento.
* **Application containment / containers**: no es virtualización completa: un solo kernel con aislamiento a nivel de espacio de usuario (Zones, chroot, Docker). Menos overhead, menos aislamiento que una VM completa, pero excelente densidad.

**7. Implementación técnica - Building Blocks**

**A. Trap-and-Emulate**

* **Idea**: guest kernel se ejecuta en modo “usuario virtual”; cuando intenta ejecutar instrucción privilegiada, se produce un trap al VMM que emula la operación.
* **Ventaja**: sencillo conceptualmente.
* **Desventaja**: sobrecarga si hay muchas traps (especialmente en código kernel). Por eso el hardware moderno añade modos auxiliares para reducir overhead.

**B. Binary Translation**

* Necesario cuando la ISA tiene instrucciones *especiales* que no trappean (x86 clásico).
* **Funcionamiento**: VMM inspecciona código, traduce dinámicamente las instrucciones “problemáticas” en secuencias seguras que emulan su efecto, y cachea las traducciones para no repetir.
* **Overhead**: costoso en traducción, pero amortizable con caching. VMware fue pionero en usar traducción binaria para x86 en años tempranos.

**C. Nested Page Tables (NPT / EPT)**

* Manejo de memoria virtual dentro de VMs.
* Cada guest cree controlar la MMU, pero en realidad el VMM mantiene tablas de páginas anidadas.
* Problema: más fallos de TLB, más lento.

**D. Hardware assistance (Intel VT-x, AMD-V)**

* CPU moderns proveen modos de virtualización que reducen la necesidad de binary translation y aceleran trap/VM-exit/VM-entry.
* Ofrecen: modos de guest/host, instrucciones para VMCS/VMCB (almacenar estado de VCPU), soporte EPT/NPT, aceleración de I/O (SR-IOV), interrupciones, DMA, etc.

**8. Memoria y gestión de memoria en VMM**

* **Overcommitment**: VMM puede presentar más RAM virtual a guests que la física disponible (beneficio de densidad). Necesita técnicas para lidiar con exceso:
  1. **Double paging**: guest cree que página está en memoria física, VMM la ha movido al backing store puede generar doble paginación, costosa.
  2. **Balloon driver**: pseudo-driver en guest que solicita al guest liberar memoria (guest OS cree liberar pages, VMM las recupera). Comunicación bidireccional para ajustar uso.
  3. **Deduplicación (memory sharing / KSM)**: identificar páginas idénticas en distintos guests y mapearlas a una copia compartida (ahorro de memoria).
* **Impacto**: mal manejo de memoria produce swapping y caídas de rendimiento. Balancear overcommit con latencias aceptables es clave para hosts de nube.

**9. I/O y almacenamiento en VMs**

* **I/O**: VMM puede virtualizar dispositivos (presentar dispositivos virtuales con drivers paravirtualizados) o permitir acceso directo parcial (passthrough).
  + *Paravirtual I/O* (virtio, Xen): mejor rendimiento al cooperar guest <-> VMM.
  + *Passthrough* (SR-IOV): dispositivo PCI expuesto directamente al guest reduce overhead pero sacrifica movilidad.
* **Storage**: discos virtuales son típicamente archivos de imagen (Type2) o volúmenes gestionados por el hipervisor (Type1).
  + **P-to-V / V-to-P**: herramientas para convertir discos físicos a imágenes virtuales y viceversa (útil en migraciones/modernización).

**10. Live migration**

Proceso típico:

1. Establecer conexión entre VMM origen y VMM destino.
2. Destino crea guest con vCPU/vmem básica.
3. Origen copia páginas de memoria *read-only* al destino.
4. Origen copia páginas *read/write*, marcando las enviadas como “limpias”.
5. Repetir paso 4 (iterativo) hasta que el número de páginas dirtied entre iteraciones sea pequeño.
6. Congelar guest en origen, enviar estado final del vCPU y páginas dirties restantes, y ordenar al destino que arranque.
7. Destino confirma y origen finaliza.

**11. VMs vs containers**

* **VMs**: mayor aislamiento (cada VM tiene su propio kernel). Mejor para multi-tenant en infraestructuras con requisitos rígidos.
* **Containers**: comparten kernel host; son ligeros, start/stop rápido, elevada densidad. Menos aislamiento por diseño dependes de namespaces, cgroups, SELinux/AppArmor para seguridad.
* **Uso práctico**: contenedores para microservicios y empaquetado; VMs para aislamiento fuerte y cargas heterogéneas (diferentes kernels/arch).

**12. Ejemplos y arquitecturas**

* **VMware Workstation (Type2)**: VM como proceso en host OS; virtual disks como archivos; permite varios guests. Buen para desktop/QA/dev.
* **Xen**: originalmente paravirtualization; soporta paravirtual y HVM (hardware-assisted).
* **KVM + QEMU**: KVM transforma Linux en hypervisor (Type1 variante), QEMU emula hardware; KVM usa VT-x/AMD-V.
* **JVM**: caso de *programming environment virtualization* bytecode, JIT, GC y portabilidad. No es virtualizar hardware, sino plataforma.

**13. Problemas de rendimiento**

* **TLB misses por NPT**: mitigado por hardware (larger TLBs, EPT) y técnicas de batching.
* **Double-paging**: reducir overcommit o usar ballooning/dedupe para evitar swapping en host.
* **I/O overhead**: usar drivers paravirtualizados (virtio), o permitir passthrough para cargas I/O intensivas.
* **Context/VM-exit overhead**: hardware moderno reduce costo de VM-exit; minimizar cambios de privilegios.

**14. Investigación y tendencias**

* **Unikernels**: imagenes minimalistas que combinan aplicación + bibliotecas + kernel reduciendo superficie de ataque y footprint. Usan VMs, ideales para microservices de alto rendimiento.
* **Separation hypervisors / partitioning hypervisors (Quest-V, Xtratum)**: particionan recursos de manera estricta para RT y seguridad (útil para IoT, robótica).
* **Secure shared memory / extended page tables**: optimizaciones para IPC entre VMs seguras con latencia baja.
* **Real-time via virtualization**: hypervisors ligeros que permiten ejecución RT en VMs aisladas, como combinación de Linux non-RT con micro RTOS en VM para tareas críticas.

**15. Resumen de ventajas y casos de uso**

* **Consolidación**: menos hardware físico; ahorro de energía y espacio.
* **Dev/Test**: snapshots y clones rápidos.
* **Uptime y mantenimiento**: live migration facilita mantenimiento sin downtime.
* **Cloud / Elasticidad**: APIs que provisionan/scale recursos según demanda.