Sun microsystems

JavaOne
Sun's 2005 Worldwide Java Developer Conference

Multitasking VMs: More Performance, Less Memory

Kyle Buza, Oleg Pliss, Mark Lam

Sun Microsystems, Inc.
www.sun.com

Session TS-7149

java.sun.com/javaone/sf

Goal

Discussion of Multi-tasking VM designs for CLDC and CDC

Agenda

- Multi-tasking Java™ Technology:
  - **Why Multi-tasking?**
  - How does Java Technology Multi-tasking Work?
  - CDC and CLDC Design Decisions
  - Some Implementation Details
- Q&A

Why Multi-tasking?

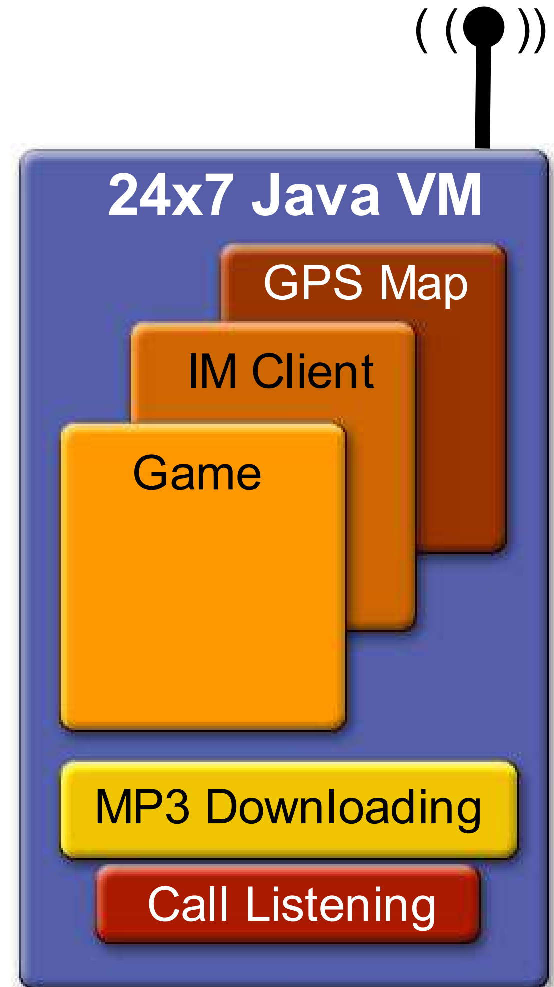
- Java ME platform maturity increasing
  - Not just for games anymore!
- Customers want to do more with Java technology
  - Run Calendar app 24x7, bring to foreground as needed
  - Contact manager
  - Messaging
  - Music, video, other multi-media apps
- Maybe even run some of the phone functions

Functionality

- Run multiple apps simultaneously!
- Switch between apps quickly
- Concurrent services as Java technology-based programs
  - (Incoming call listener, downloading, audio streaming, ...)

Alternating Foreground Applications

Background Tasks



Functionality

- Centralized screen management
- Efficient resource management
- No stopping/restarting the JVM™ software
- Quick application startup
- **No changes to existing applications! They "should just work" on multitasking VM**

Agenda

- Multi-tasking Java Technology:
  - Why Multi-tasking?
  - **How Does Java Technology Multi-tasking Work?**
  - CDC and CLDC Design Decisions
  - Some Implementation Details
- Q&A

Technical Challenges

- Concurrency
- Isolation
- Robustness
- Efficiency

Concurrency

Multiple Java technology-based applications can run on the same device at the same time. Ensure fair allocation of resources between apps.

Isolation

Each Java technology-based application has its own VM environment. It will not interfere with other concurrently running applications.

Robustness

Handle bugs and misbehavior in apps

- Quarantine bad effects: Isolate other apps
- Reliable termination of bad apps
- Reclamation of resources

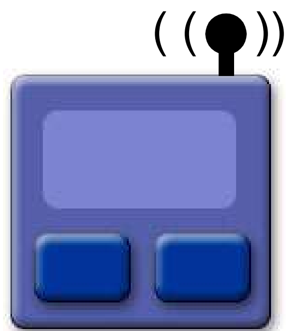
Efficiency

Reduce redundancy in resource usage and maximize sharing between apps.

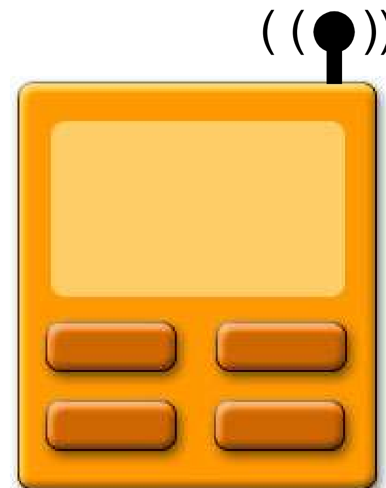
Terminology

- Isolate
  - A VM environment for executing each application
  - Analogous to an OS process context record
- Task
  - An isolate which has been scheduled for execution
  - May have multiple Java technology threads running in it
  - Analogous to an OS process
- Application Management System (AMS)
  - System which manages the running of the apps

Targeting a Range of Devices



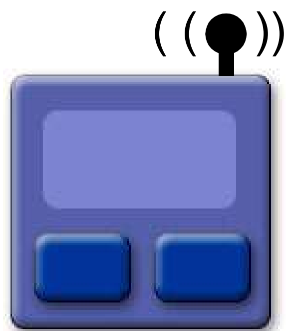
vs.



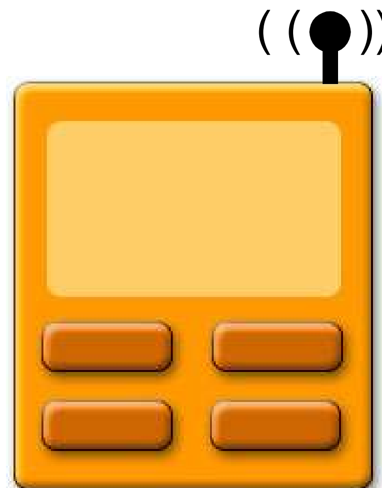
- Slower CPU
- Limited resources
- Simple OS
  - No processes
  - No native threads

- Faster CPU
- More resources
- Feature rich OS
  - Process-based
  - Native threads

Targeting a Range of Devices



vs.



CLDC **CDC**

CLDC and CDC Approaches

- CLDC employs the non-process OS approach
- CDC employs the process-based OS approach

Non-Process-Based OS

- Native applications execute as functions
  - No virtual memory address space isolation
- Maybe no native thread library
- VM may need to provide it's own thread library
  - Tasks are VM internal constructs

Process-Based OS

- Native applications execute as processes
  - Virtual memory address space isolation
- Usually has native thread library
- VM can make use of process isolation
  - Tasks are mapped to processes
- Linux, Symbian OS, etc.

Concurrency Issues

- In a non process-based VM, tasks and threads are scheduled by the VM
- In a process-based VM, tasks and threads are scheduled by the OS

VM Scheduling

Using a VM specific task and thread library

- VM has direct control over Java technology-based scheduling
- Ensure fair scheduling of Java technology-based apps
- May not be fair against native apps
  - All Java technology-based apps appear as a single thread to OS
  - Concurrent native apps may get larger time slice
- Less OS dependency for porting VM
  - Can port to more OSes independent of capability

OS Scheduling

Relying on OS processes and the native thread library

- No direct control over scheduling algorithm
  - Relies on OS for fair scheduling
- Fair scheduling for native and Java technology-based apps
  - All Java technology tasks and threads are visible to OS
  - OS can schedule them fairly with native apps
- Portability limited by OS capability
  - Need support for processes and threads

Isolation Issues

- Applications may have bugs
- Applications may misbehave
- Need to prevent one app from affecting another
  - Causing hangs
  - Crashing the VM
  - Hogging resources
  - Denial of service attacks

Isolation Issues

Native state

- Isolating state
  - Non-process OS: Not available
  - Process OS: Each process has its own address space
- Needed if libraries have native code
  - Have no control over content of native code in 3rd party libraries and application code
  - Process address spaces only isolates native state in different applications
    - Only one address space per process

Isolation Issues

Java technology state

- Isolating Java technology state
  - Non-process OS: VM does the isolation
  - Process OS: Automatic with process isolation
- ClassLoaders provide some namespace isolation
  - ClassLoaders unavailable in CLDC
  - State of application classes are isolated by namespace
  - State of system classes cannot be isolated this way
    - They are all loaded by the NULL classloader

Isolation Issues

Native bugs

- Isolating native bugs
  - Non-process OS: Not available
  - Process OS: Protected by virtual address space
- Virtual address space protection
  - Crashes are isolated in originating process
  - Deadlocks and hangs are isolated
  - Other Java technology-based apps unaffected

Isolation Issues

Java technology bugs

- Isolating Java technology bugs
  - Non-process OS: VM does the isolation
  - Process OS: Automatic with process isolation
- VM isolation
  - Isolate monitors to prevent cross-application deadlocks
  - Java technology programming bugs cannot crash VM by design
  - Isolation of Java technology-based state also prevents interference and bug propagation between apps

Robustness Issues

Tracking resources

- Tracking native memory usage
- Tracking Java technology memory usage
- Tracking other resources

Robustness Issues

Tracking native memory usage

- Native code in libraries can allocate memory
  - Need to be tracked in order to be reclaimed later
- Tracking native memory
  - Non-process OS: Done by VM and class libraries
  - Process OS: Tracked automatically by processes
- Reclaiming memory
  - Normally done by native code that did allocation
  - Async termination cleanup handled by OS process
  - Async termination not allowed for non-process OS

Robustness Issues

Tracking Java technology memory usage

- Java technology memory usage comes from the Java technology-based heap
- Tracking Java technology memory
  - Non-process OS: Tracked by the Java technology heap
  - Process OS: Also tracked by the Java technology heap
- OSes do not know about Java technology memory allocation
  - Hence, not able to track
- Reclamation done by garbage collector

Robustness Issues

Tracking other resources

- File handles, graphics widgets, etc.
- Tracking and reclamation handled in the same way as native memory

Robustness Issues

Other robustness issues

- Terminating tasks
- Reclaiming resources
- Load-balancing resources

Robustness Issues

Terminating tasks abnormally

- AMS may want to force termination on bad apps
- Non-process OS
  - AMS requests task termination by VM
  - VM throws non-catchable exception in all task threads
  - System classlibs handle this exception carefully
    - Clean-up resources and rethrow exception
  - Application code cannot have native code
    - Native code can ignore exceptions
- Process OS
  - AMS requests process termination by OS

Robustness Issues

Reclaiming resources

- Need to reclaim resources when terminating task
- Non-process OS
  - Classlib native code check for termination condition
  - Native code will release all used resources
  - Reclamation is cooperative between VM and classlibs
  - All native code including 3rd-party libraries need to behave properly for this to work
- Process OS
  - Automatically done by OS process clean-up

Robustness Issues

Load-balancing resources

- Prevent resource hogging by single apps
- Non-process OS
  - Java technology heap is shared
  - AMS enforces quota on Java technology heap usage
  - VM and AMS enforces quota on native resources
- Process OS
  - Java technology heaps are resizable
  - AMS tells each isolate to size its heaps accordingly
  - OS enforces quota on native resources

Efficiency Issues

Efficient memory usage

- MVM Implementations
  - Maximize sharing between isolates
  - Minimize redundancies
- Isolates can share some memory content
  - Native code and constant data automatically shared
  - Keep, as much as possible, data as constants
  - ROMization of classfiles

Efficiency Issues

Efficient memory usage on a non-process OS

- Isolates exist within same address space
  - All loaded Java code and constant data can be shared
  - All JIT compiled code can be shared
- Isolates share the same heap
  - Less internal fragmentation
  - May reduce GC overhead memory in some case

Efficiency Issues

Efficient memory usage on a process-based OS

- If an OS mechanism for sharing memory between processes is supported
  - VM can employ a warm up list
    - Preload some classes
    - Pre-JIT compile some methods
  - New isolates start as a copy of original with sharing
    - Inherits preloaded classes and pre-JITed methods
  - OS recognizes unmodified pages of memory
    - Unmodified pages are not duplicated in each isolate process

Efficiency Issues

Efficient VM startup

- Part of VM startup is the same for all isolates
- Reduce redundant startup processing
  - Faster application startup
- Non-process OS
  - For each new isolate, create a new isolate, and start a new task and thread to run the app main method
- Process OS
  - For each new isolate, start a new VM process with sharing, initialize the isolate, invoke the app main method in the main thread

Agenda

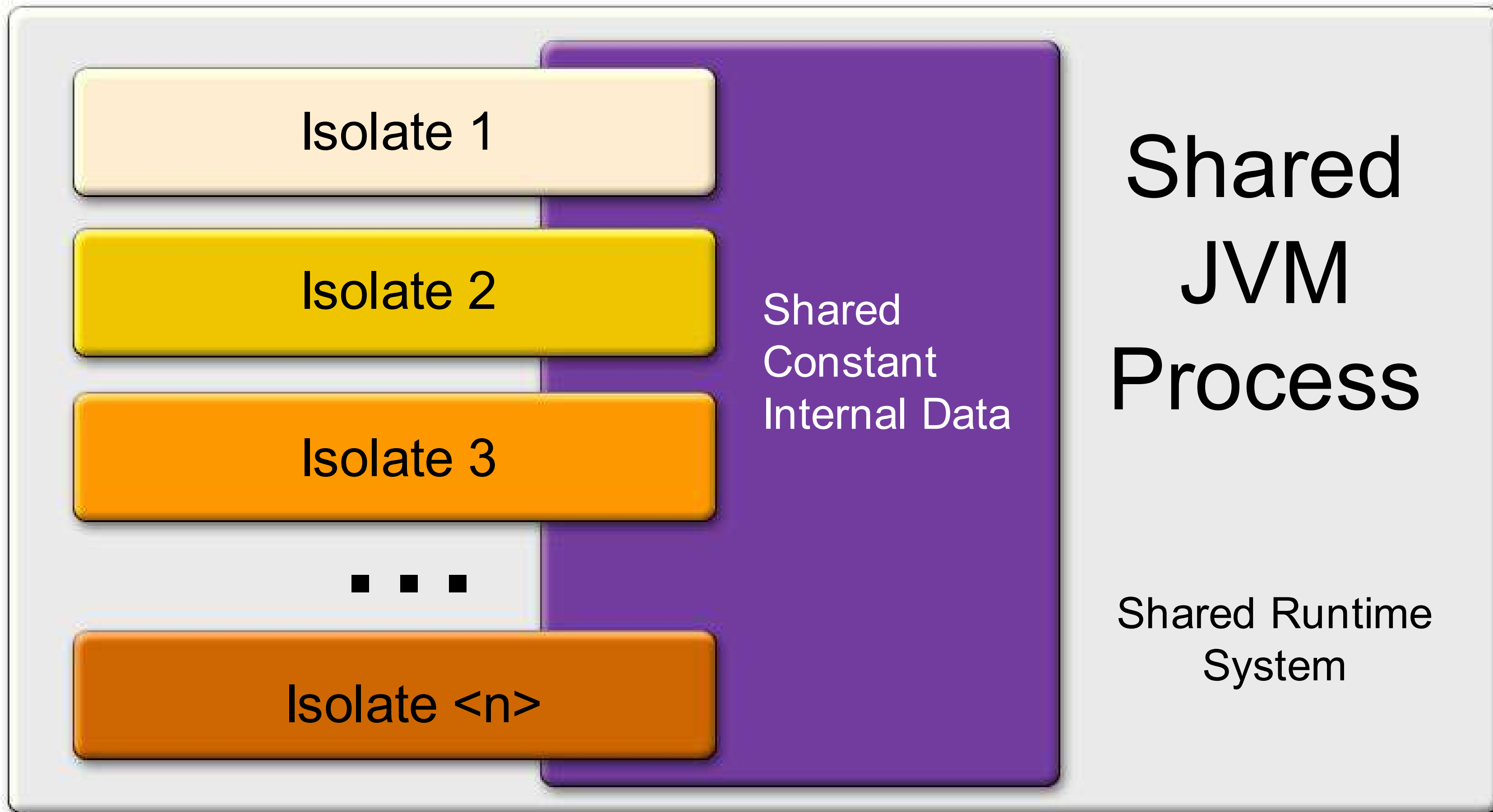
- Multi-tasking Java Technology:
  - Why Multi-tasking?
  - How Does Java Technology Multi-tasking Work?
  - **CDC and CLDC Design Decisions**
  - Some Implementation Details
- Q&A

CLDC Solution

- Assumes no processes and no native threads
- VM provides task and thread implementation
- Isolates are logical constructs within the same VM process

CLDC Solution

Isolates are logical constructs within the VM

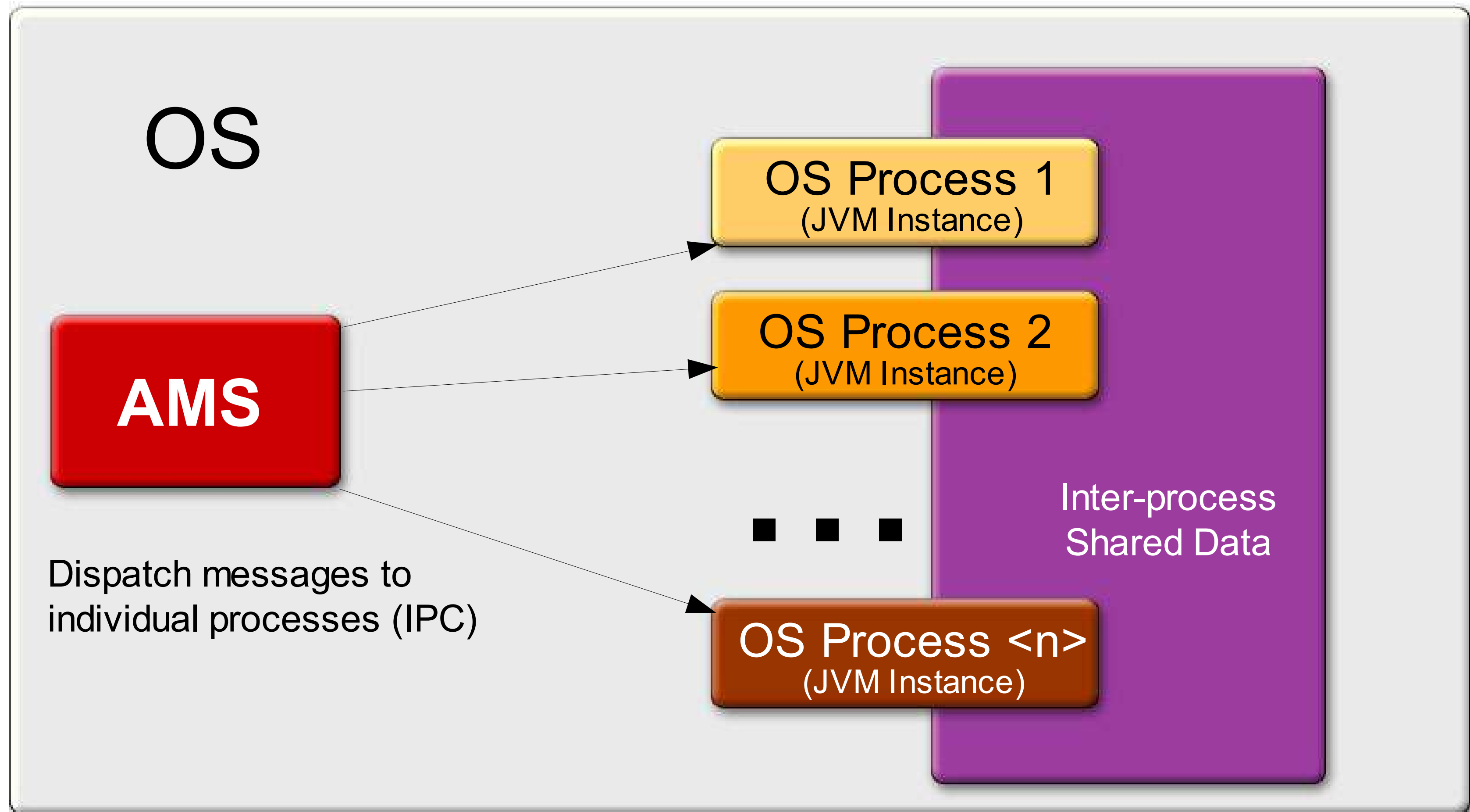


CDC Solution

- Assumes processes and native threads
- Allows co-existence with native apps
- Isolates are processes

CDC Solution

Isolates are processes



Multi-tasking Approaches

Comparing CLDC and CDC

	CLDC	CDC
Scheduling tasks	VM tasks	OS process
Scheduling threads	VM threads	OS threads
Isolating native state	not available	OS process
Isolating Java state	VM	OS process
Isolating native bugs	not available	OS process
Isolating Java bugs	VM	OS process

Multi-tasking Approaches

Comparing CLDC and CDC (Cont.)

	CLDC	CDC
Tracking native memory	VM + classlibs	OS process
Tracking Java memory	VM heap	VM heap
Tracking other resources	VM + classlibs	OS process
Terminating tasks	VM + classlibs	OS process
Reclaiming resources	VM + classlibs	OS process
Load-balancing resources	VM + AMS	OS + AMS

Agenda

- Multi-tasking Java Technology:
  - Why Multi-tasking?
  - How Does Java Technology Multi-tasking Work?
  - CDC and CLDC Design Decisions
  - **Some Implementation Details**
- Q&A

Some Implementation Details

- CLDC memory resource management
- CDC memory resource management

CLDC Memory Resource Management

- Resource reservation
  - At task startup the reserved amount of memory is transferred from the shared resource pool to private pool of the task
  - Resource allocation starts from the private pool.
    If no resources left in the private pool, the resources are borrowed from the shared pool
  - Reclaimed resources in excess of the reservation are returned to the shared pool

Memory Is a Special Resource in Java Technology

- **Object creation is vital for Java technology-based programs; all objects are allocated in the heap**
- Memory allocations are usually performed inline
  - Allocation speed and code size are essential
  - No centralized runtime routine is called at every allocation
- Memory reclamation by garbage collection (GC) is implicit
- Memory reclamation by scanning GC is delayed
  - Reference-counting GC is immediate but too restrictive and inefficient

Synchronous vs. Asynchronous Quota Management

- Memory usage accounting cannot be always exact due to implicit delayed reclamation
- Quota management is synchronous if quota violations are detected and signaled to the violating task at the moment of violation
- Asynchronous quota management:
  - Delayed detection (next GC?)
  - Delayed signaling (next allocation attempt or proper bytecode)
  - Sometimes violator(s) cannot be identified

CLDC Memory Quota Management

- Tasks allocate memory in a shared heap
- Individual limit and reservation can be set for a task
  - Cannot be dynamically modified after the task startup
- Exact synchronous quota management mechanism is integrated with object heap
- Ordinary single-tasking GC algorithm
  - Generational mark'n'compact
- Very low memory and execution speed overhead

CDC Memory Resource Management

- VM uses standard GC/heap algorithms
  - Java technology heap need to be resizable
- AMS tracks isolate VM memory usage
- AMS controls isolate VM memory usage by issuing commands over IPC
- Isolate VMs resizes heap accordingly

Summary

- CLDC employs the non-process OS approach
- CDC employs the process-based OS approach

Why the CLDC Approach Works

- Need to run on a tiny OS and tight memory
  - May not have processes or threads
- Has less native code complexity
  - Classlibs/profiles are relatively small and simple
  - CLDC does not support user native code
  - Hence, not too difficult to implement isolation and robustness with non-process approach
- Does not support classloading
  - Easier to share loaded classes in the same VM
- Non-process approach makes sense

Why the CDC Approach Works

- CDC has more memory to play with
- Has more native code complexity
  - Classlibs/profiles are far richer and more complex
  - Supports 3rd party classlibs including native code
  - Supports user native code
  - Hence, far more difficult to implement isolation and robustness with non-process approach
- Need reliability even with native code
- Process-based approach makes sense

For More Information

- The Barcelona project
  - http://research.sun.com/projects/barcelona/
- White paper on CLDC-HI
  - http://java.sun.com/j2me/docs/pdf/CLDC_mvm.pdf
- For more information on the CDC-HI MVM, contact your Sun representative

Submit Session Evaluations for Prizes!

Your opinions are important to Sun

- You can win a $75.00 gift certificate to the on-site Retail Store by telling Sun what you think!
- Turn in completed forms to enter the daily drawing
- Each evaluation must be turned in the same day as the session presentation
- Five winners will be chosen each day (Sun will send the winners e-mail)
- Drop-off locations: give to the room monitors or use any of the three drop-off stations in the North and South Halls

Note: Winners on Thursday, 6/30, will receive and can redeem certificates via e-mail.

Q&A

java.sun.com/javaone/sf

JavaOne
Sun's 2005 Worldwide Java Developer Conference

Multitasking VMs: More Performance, Less Memory

Kyle Buza, Oleg Pliss, Mark Lam

Sun Microsystems, Inc.
www.sun.com

Session TS-7149