WebSphere Education

Introduction to JVM-related problems

Unit objectives

After completing this unit, you should be able to:

- Describe components of JVM and overall architecture (Java Development Kit V6.0)
- Describe garbage collection (GC) and GC tuning policies
- Describe JVM command-line arguments
- Describe javacore files and how to obtain them
- Identify a sluggish JVM and detect bottleneck problems
- Explain how to tune the heap size
- Use JVM-related tools: Garbage Collection and Memory Visualizer (GCMV), Memory Analyzer tool (MAT), and Java Health Center

Topics

- JVM introduction
- Introduction to JVM problem determination
- JVM tuning

JVM introduction

Java virtual machine (JVM) features

Almost every WebSphere process runs in a JVM

The JVM provides:

- Class loading
  - A class loader verifies and loads classes into memory
  - Multiple class loaders are involved in loading the required libraries for an application to run
  - Each class loader loads its own classes
- Garbage collection (GC)
  - Garbage collection takes care of memory management for the entire application server
  - The GC process searches memory to reclaim space from program segments or inactive Java object
- Execution management
  - Manages the bookkeeping work for all the Java threads
- Execution engine (Interpreter)
  - Interprets the Java methods

Just-in-time compiler (JIT) basics

- The just-in-time compiler (JIT) is essential for a high-performing Java application
  - Java is write-once-run-anywhere; thus it is interpretive by nature and without the JIT, cannot compete with native code applications
- The JIT works by compiling bytecode that is loaded from the class loader when an application accesses it
  - Because different operating systems have different JIT compilers, there is no standard procedure for when a method is compiled
  - As your code accesses methods, the JIT determines how frequently specific methods are accessed
  - Methods that are used often are compiled to optimize performance

Ahead-Of-Time (AOT) compiler basics

- Ahead-Of-Time (AOT) compiles Java classes into native code for subsequent executions of the same program
  - The AOT compiler works with the class data sharing framework
- The AOT compiler generates native code dynamically while an application runs and caches any generated AOT code in the shared data cache
  - Subsequent JVMs that run the method, can load and use the AOT code from the shared data cache without incurring the performance decrease experienced with JIT-compiled native code
- The AOT compiler is enabled by default, but is only active when shared classes are enabled
  - By default, shared classes are disabled so that no AOT activity occurs
  - The **`-Xshareclasses`** command-line option can be used to enable shared classes
- When the AOT compiler is active, the compiler selects the methods to be AOT compiled with the primary goal of improving startup time

JVM version (1 of 2)

- WebSphere supports several JVMs based on version and operating system type
  - Windows, AIX, and Linux: IBM supplied
  - Can use only IBM SDK that zWSAS provides on z/OS
  - Solaris and HP-UX: Hybrid of IBM add-ons and vendor supplied JVM
- For a comprehensive list of the supported JVMs, check
  - `http://www.ibm.com/support/docview.wss?uid=swg27038218`
- To determine the JVM version in use:
  - Look in the `SystemOut.log` file of one of the profile instances

    `<profile_home>/logs/server1/SystemOut.log`
- JVM version can be found in the server job logon z/OS

JVM version (2 of 2)

- To determine the JVM version in use:
  - Run `java -fullversion` (IBM JVMs only) from the command line

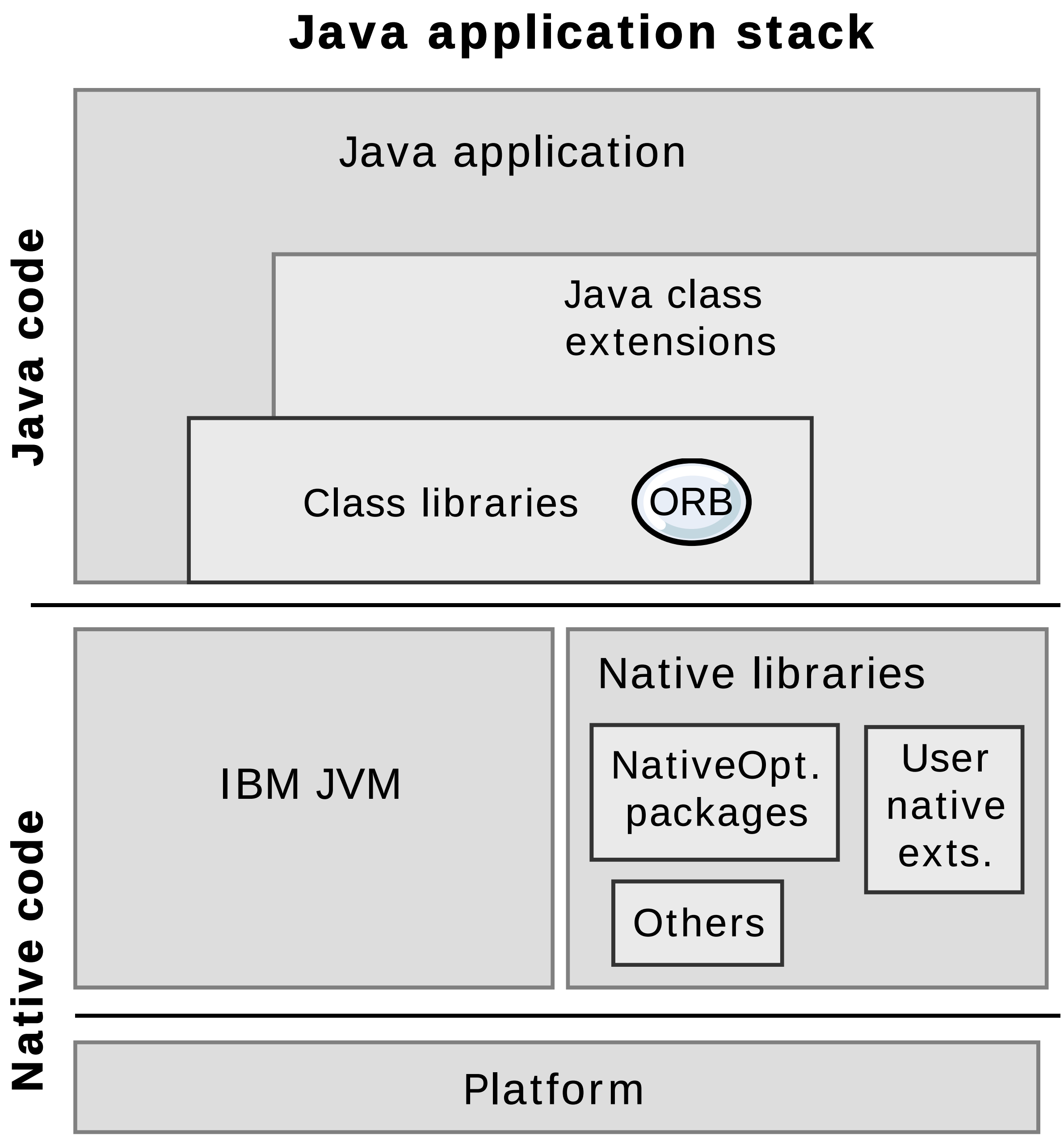
```
[root@washost bin]# ./java -fullversion
java full version "JRE 1.6.0 IBM Linux build pxa6460_26sr5fp1ifx-
20130408_02 (SR5 FP1)"
```

  - Run `java –version` from the command line

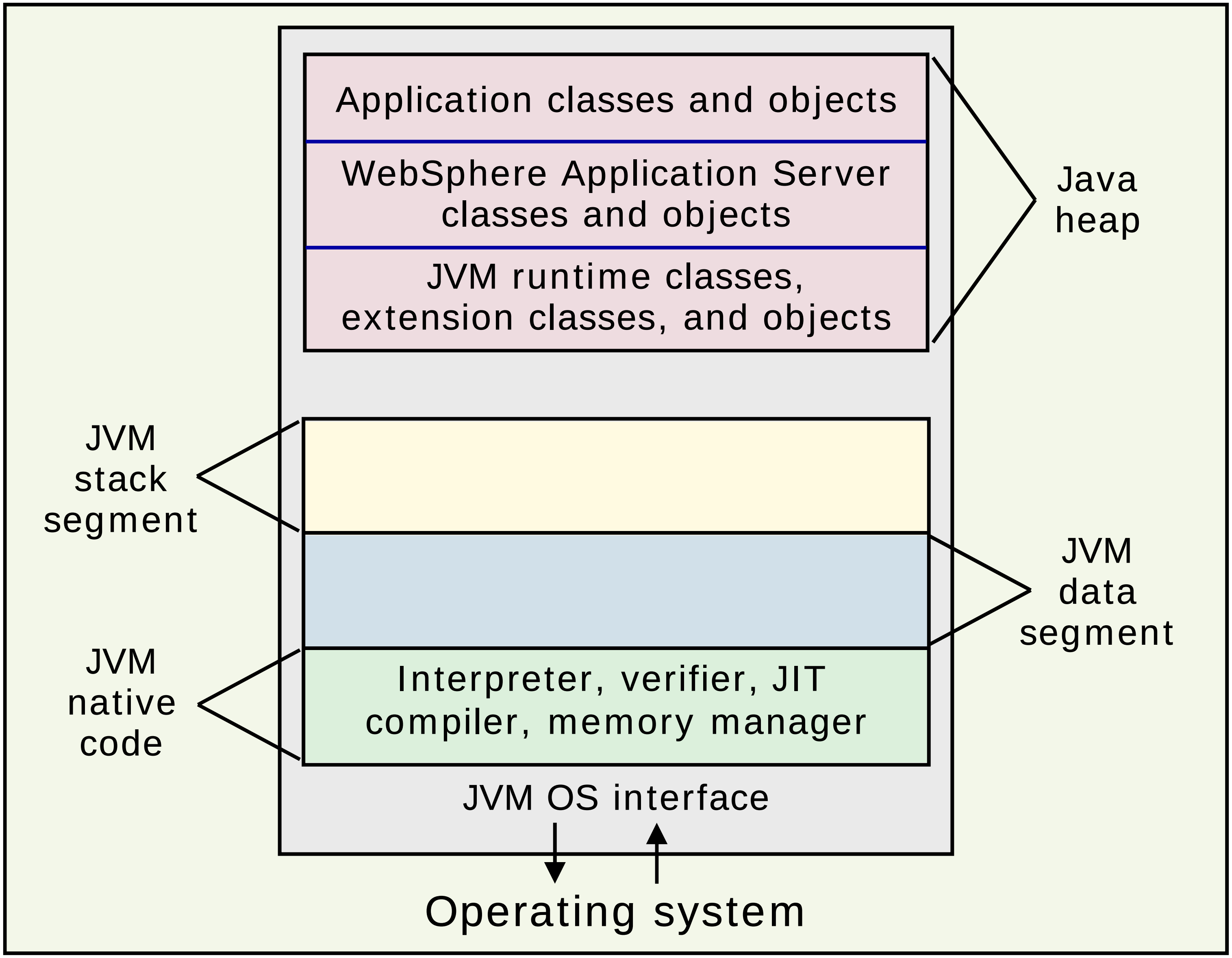
```
[root@washost bin]# ./java -version
java version "1.6.0"
Java(TM) SE Runtime Environment (build pxa6460_26sr5fp1ifix-
20130408_02(SR5 FP1+IV38399+IV38578))
IBM J9 VM (build 2.6, JRE 1.6.0 Linux amd64-64 Compressed References
20130301_140166 (JIT enabled, AOT enabled)
J9VM - R26_Java626_SR5_FP1_20130301_0937_B140166
JIT  - r11.b03_20130131_32403
GC   - R26_Java626_SR5_FP1_20130301_0937_B140166_CMPRSS
J9CL - 20130301_140166)
JCL  - 20130408_01
```

JVM overview

- JVM is built by using object-oriented design
- The core run times of JVMs are developed in C and run most functions in native code
  - Garbage collector and memory management
  - JIT
  - I/O subroutines, OS calls
- The J2SE and Java EE APIs all exist at the Java code layer
  - Makes data structures available
  - Gives users access to needed function
  - Allows interactions with system



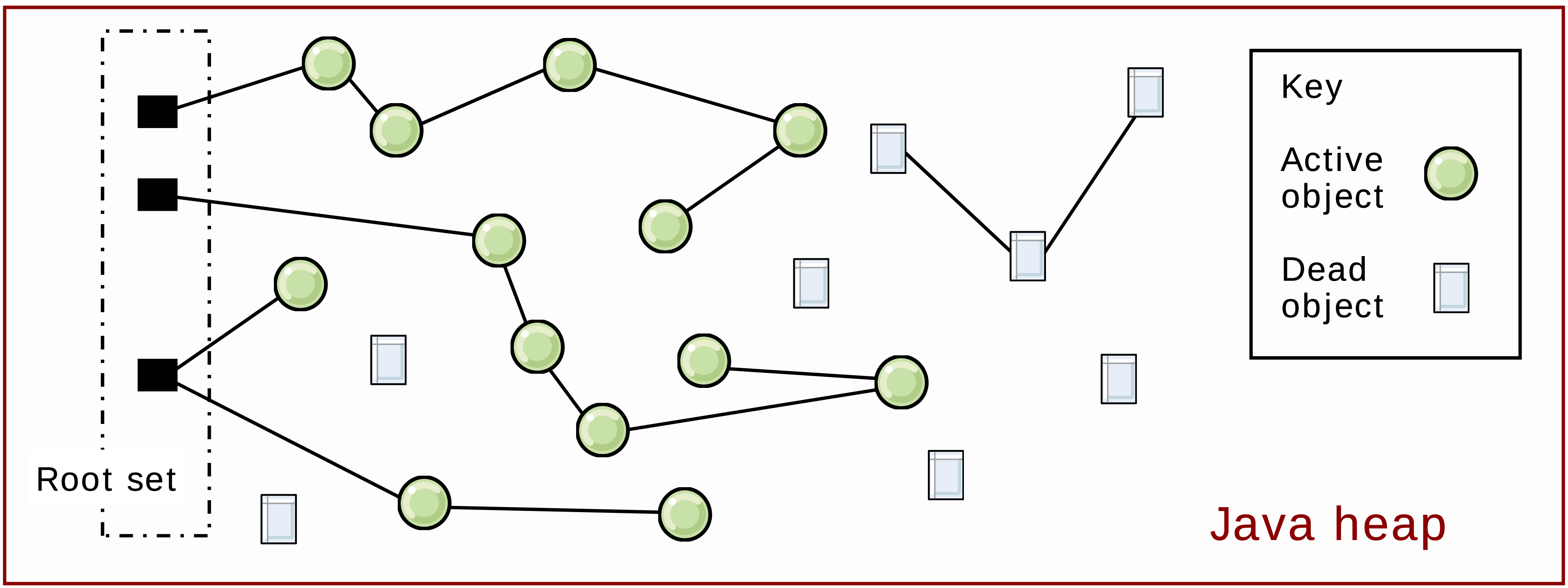
JVM memory segments



Typical JVM architecture

Understanding garbage collection (GC)

- JVM manages the Java heap and does garbage collection
- Object is eligible for GC when there are no more references from root to that object
- Root set: references in the stack and registers
- Reference is dropped when variable goes out of scope



IBM JDK GC process

- **Mark**: Recursively marks all the live objects, starting with the registers and thread stacks
- **Sweep**: Frees all the objects that were not marked in the mark phase
- **Compaction:** Reduces heap fragmentation
  - This phase attempts to move all live objects to one end of the heap, freeing up large areas of contiguous free space at the other end
  - Compaction stops JVM activity while it occurs
  - Not every GC cycle results in a compaction
- **Parallel** mark and sweep process uses main and multiple helper threads (number of processors minus one) to process tasks
- **Concurrent** mark and sweep can be configured
  - It starts a concurrent marking and sweeping phase before the heap is full
  - The mark and sweep phase runs while the application is still running

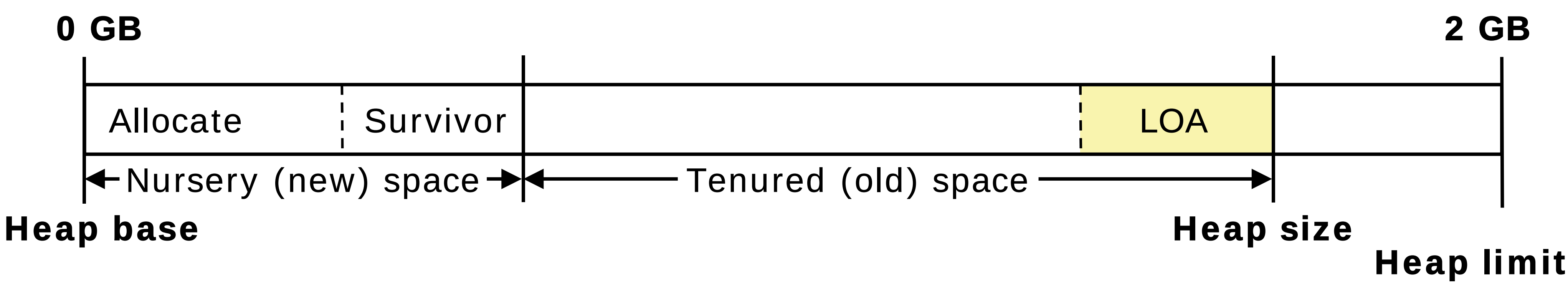
Garbage collection policies

Memory management is configurable by using four different policies with varying characteristics

- **Generational concurrent**: (new default) divides heap into “nursery” and “tenured” segments that provide fast collection for short lived objects
  - Can provide maximum throughput with minimal pause times
- **Optimize for throughput**: Flat heap collector that is focused on maximum throughput
- **Optimize for pause time**: Flat heap collector with concurrent mark and sweep to minimize GC pause time
- **Balanced**: New policy
  - Uses a region-based layout for the Java heap
  - These regions are individually managed to reduce the maximum pause time on large heaps and increase the efficiency of garbage collection
- **Subpool**: This option is now deprecated and is treated as an alias for optimize for throughput

Generational concurrent GC (gencon)

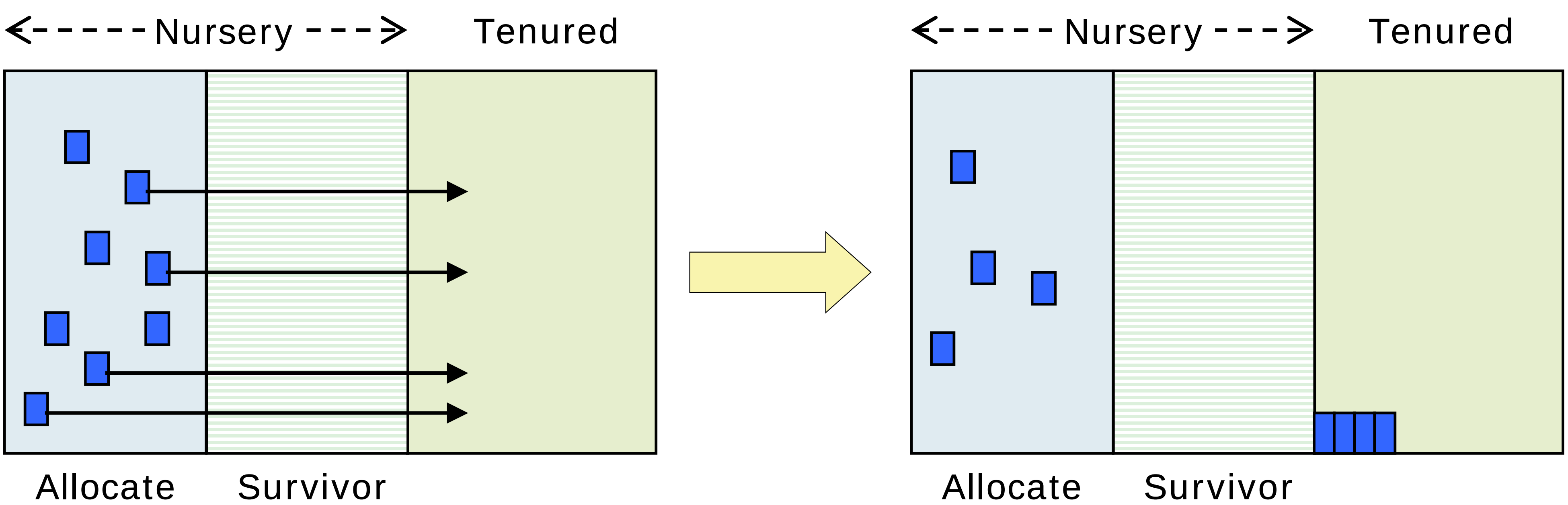
- Similar in concept to the mode that Solaris and HP-UX uses
  - Parallel copy and concurrent global collects by default
- Motivation: Objects die young so focus collection efforts on recently created objects
  - Divide the heap up into a two areas: “new” and “old”
  - Perform allocations from the new area
  - Collections focus on the new area
  - Objects that survive a number of collects in new area are promoted to old area (tenured)



- Ideal for transactional and high data throughput workloads

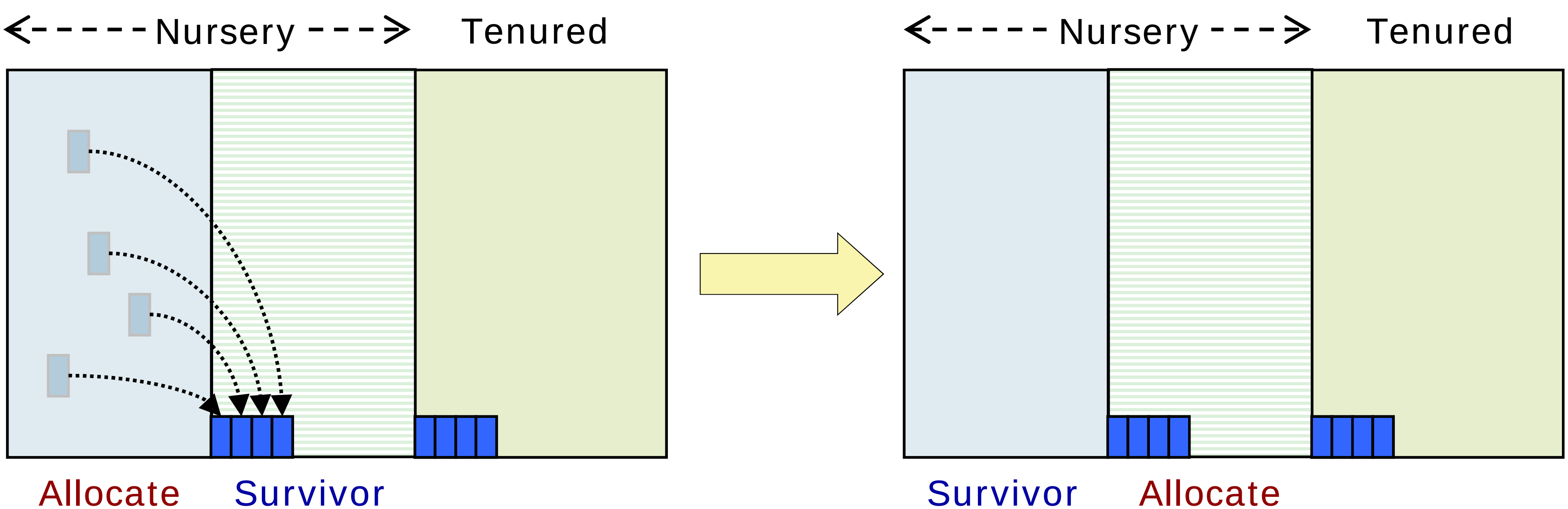
Generational concurrent GC: Scavenge (1 of 2)

- When the allocate space is full, a GC is triggered
- The allocate space is traced
- Objects that reach the tenured age (survived a specific number of scavenge operations; maximum is 14) are copied into the tenured area



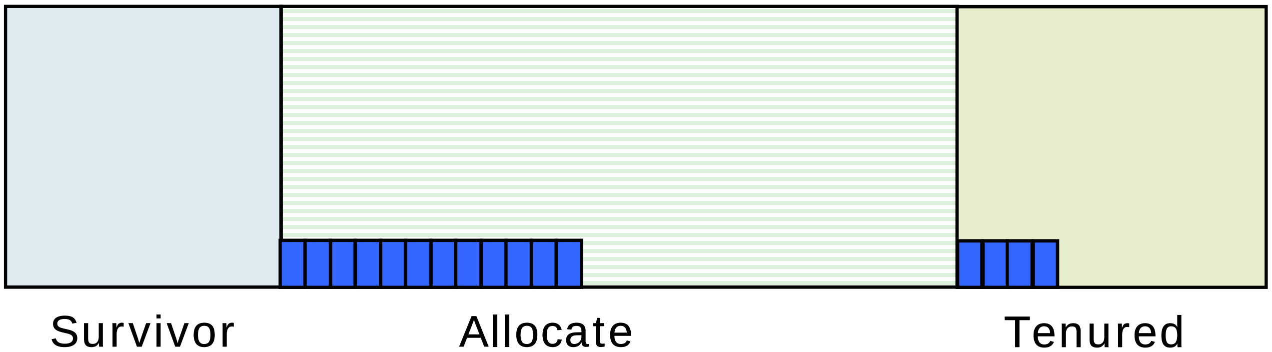
Generational concurrent GC: Scavenge (2 of 2)

- Remaining live objects are copied from the allocate space into the survivor space, and a count of the number of times each object was scavenged is incremented
- The copied objects are placed contiguously in the survivor space so that an effective compaction occurs
- At the end of scavenge, the survivor space and allocate space pointers are flipped
  - The survivor space becomes the new allocate space and is empty



Generational concurrent GC: Self-adjusting

- The scavenger automatically adjusts the relative sizes of the allocate and survivor spaces by using history and predictions (Tilt)



- The generational collector is designed to respond dynamically to varying conditions so that its behavior is adaptive
- These mechanisms are:
  – Concurrent collection runs in the tenured area
  – In many cases, one rarely or never sees a full classic collection (mark, sweep, compact) in the tenured area
  – The sizes of the young and tenured generations will self-adjust over time based on memory pressure
  – The flip count (number of scavenges that an object must survive before tenure) is adjusted by the scavenger that is based on history
  – The tilt ratio self-adjusts based on history

Using IBM Java 7

- In WebSphere Application Server V8.5 and V8.5.5, IBM Java 6 is the default Java runtime
  - You have the option of installing and switching to IBM Java 7
- A new command line tool that is called `managesdk` is added to the `<profile_root>/bin` folder
- List installed run times:
  - `managesdk.sh -listAvailable`
- List run times that are configured for all profiles:
  - `managesdk.sh -listEnabledProfileAll`
- Change all configured run times to Java 7, 64-bit:
  - `managesdk.sh -enableProfileAll -sdkname 1.7_64 -enableServers`
- Change default configured runtime for future servers that are created in a profile:
  - `managesdk.sh -setNewProfileDefault -sdkname 1.7_64`

Introduction to JVM problem determination

JVM problem determination: Symptom analysis

- Application server stops responding under the following conditions:
  - Server crash
  - Hung process
  - Out of memory condition
- Server crash
  - Application server stops or exits unexpectedly
  - Tools are available to determine the cause of crash
- Hung process
  - Verify that application server process is still running
  - Threads might be deadlocked
  - Code might be running in a loop
  - Tools available to determine cause of the hang
- Out of memory condition
  - Errors and exceptions that are logged without process exit
  - At times, can result in unexpected process exit
- Performance degradation
  - Application server might crash and the node agent restarts it
  - Check to see whether the process ID is continually changing

JVM problem determination: Data to collect

- Core files
  - Also known as process memory dumps or system core files
  - Complete memory dump of the virtual memory for the process
  - Can be large
  - Usually required by IBM support
  - Tools available to parse these files
- Javacore files
  - Also known as javadump or thread dump files
  - Text file that is created during an application server failure
  - Can also be generated manually
  - Error condition is given at top of memory dump
  - Format of the file is specific to the IBM JDK
- VerboseGC logs
  - Provides detailed information about garbage collection cycles
- Heap memory dump
  - Shows the objects that use Java heap memory
  - Needed for memory leak determination

Javacore overview

- What is a javacore?
  - A snapshot of the running Java process
  - Small diagnostic text file that the JVM produces contains vital information about the running JVM process
  - Lists the JVM command line, environments, and loaded libraries
  - Provides a snapshot of all the running threads, their stack traces, and the monitors (locks) held by the threads
  - GC history and storage management (memory) information
  - Necessary for detecting hung threads and deadlock conditions
  - Useful for detecting some categories of native memory leaks

- Also, helpful for detecting performance problems
  - Take at least three snapshots of the JVM (about 2 – 3 minutes apart)
  - Analyze the javacores to see what different threads are doing in each snapshot
  - **Example:** A series of snapshots where container threads are in the same method or waiting on the same monitor or resource might indicate a bottleneck, hang, or a deadlock

Javacore file location and naming

- The javacore file is stored in the first viable location of:
  - `Xdump:java:file=<path_name>/<filename.txt>` (command line argument)
  - The setting of the IBM_JAVACOREDIR environment variable (deprecated)
  - `<`*`WAS_install_root`*`>/profiles/<`*`profile`*`>`
  - `TMPDIR` or `TEMP` environment variable
  - Windows only: If the javacore cannot be stored in any of the above, it is put to STDERR
  - A new option `-Xdump:java:defaults:file=<path/filename>` can be used to change the default path and name of all javacore dump agents
- Javacore naming
  - Windows and Linux: `javacore.YYYYMMDD.HHMMSS.PID.txt` where `YYYY` = year, `MM` = month, `DD` = day, `SS` = second, `PID` = processID
  - AIX: `javacorePID.TIME.txt` where `PID` = processID, `TIME` = time since 1/1/1970
  - z/OS: `Javadump.YYYYMMDD.HHMMSS.PID.txt` where `YYYY` = year, `MM` = month, `DD` = day, `SS` = second, `PID` = processID

Javacore file subcomponents

- **TITLE**: Shows basic information about the event that caused the generation of the javacore, the time it was taken, and the file name
- **GPINFO**: Shows general information about the OS
  - If the memory dump resulted from a general protection fault (GPF), information about the fault module is provided
- **ENVINFO**: Shows information about the JRE level, details about the command line that started the JVM process, and the JVM environment
- **NATIVEMEMINFO**: Provides information about the native memory that the Java Runtime Environment (JRE) allocates
  - Requested from the OS by using library functions such as `malloc()` and `mmap()`
- **MEMINFO**: Shows the free space in heap, the size of current heap, details on other internal memory that the JVM is using, and garbage collection history data
- **LOCKS**: Shows the locks threads hold on resources, including other threads
  - A lock (monitor) prevents more than one entity from accessing a shared resource
- **THREADS**: Identifies the current thread, provides a complete list of Java threads that are alive, and provides their stack traces
- **CLASSES**: Shows information about the class loaders and specific classes loaded

Javacore example (JDK v6)

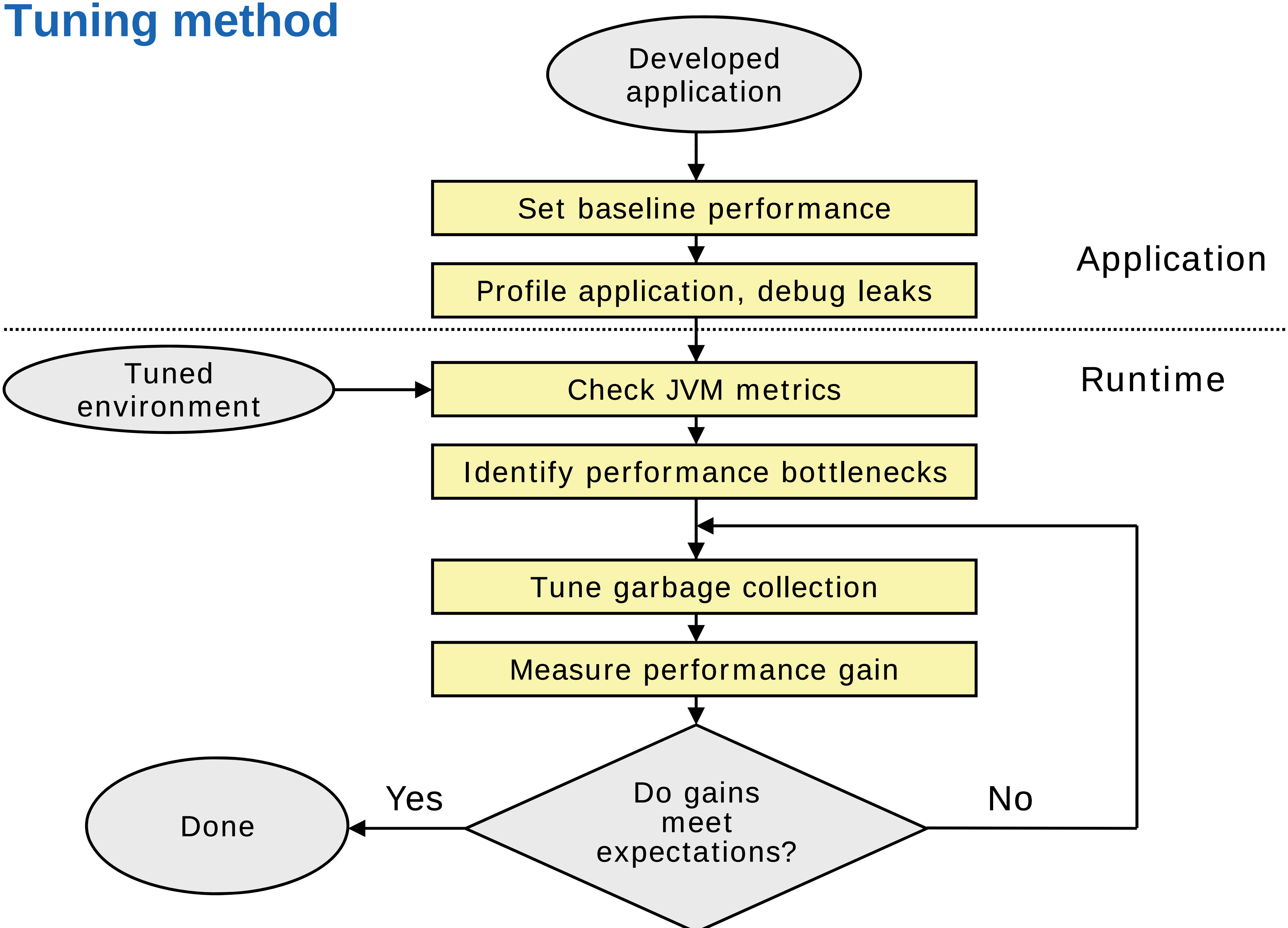
```
0SECTION       TITLE subcomponent dump routine
NULL           ===============================
1TICHARSET     1252
1TISIGINFO     Dump Requested By User (00100000) Through com.ibm.jvm.Dump.JavaDump
1TIDATETIME    Date:                      2012/03/06 at 17:08:36
1TIFILENAME    Javacore filename: C:\ProgramFiles\IBM\WebSphere\AppServer\profiles\
                                   profile1\javacore.20120306.170836.6404.0001.txt
1TIREQFLAGS    Request Flags: 0x81 (exclusive+preempt)
1TIPREPSTATE   Prep State: 0x106 (vm_access+exclusive_vm_access+)
NULL           ------------------------------------------------------------------
0SECTION       GPINFO subcomponent dump routine
NULL           ================================
2XHOSLEVEL     OS Level          : Windows XP 5.1 build 2600 Service Pack 3
2XHCPUS        Processors -
3XHCPUARCH       Architecture    : x86
3XHNUMCPUS       How Many        : 1
3XHNUMASUP       NUMA is either not supported or has been disabled by user
1XHERROR2      Register dump section only produced for SIGSEGV, SIGILL or SIGFPE.
NULL           ------------------------------------------------------------------
0SECTION       ENVINFO subcomponent dump routine
NULL           =================================
1CIJAVAVERSION JRE 1.6.0 Windows XP x86-32 build 20111113_94967 (pwi3260_26sr1-
20111114_01(SR1))
1CIVMVERSION   VM build R26_Java626_SR1_20111113_1649_B94967
1CIJITVERSION  r11_20111028_21230
1CIGCVERSION   GC - R26_Java626_SR1_20111113_1649_B94967
1CIJITMODES    JIT enabled, AOT disabled, FSD disabled, HCR disabled
1CIRUNNINGAS   Running as a standalone JVM
```

Verbose garbage collection (GC)

- Verbose GC is an option that the JVM run time provides
- Provides a garbage collection log:
  - Interval between collections
  - Duration of collection
  - Whether compaction was required
  - Memory size, memory that was freed, memory available
- Select **Servers > Server Types > WebSphere application servers > <*serverName*>**
- Under Server Infrastructure, click **Java and Process Management > Process Definition > Java Virtual Machine**
  - On **Configuration** tab, select **Verbose Garbage Collection** check box, restart server
  - On **Runtime** tab, select **Verbose Garbage Collection** check box, effective for current server instance
- IBM JVM writes to `native_stderr.log`

JVM tuning

Tuning method



Maximum heap size

- 32-bit Java processes have maximum heap size
  - Varies according to the OS and hardware that is used
  - Determined by the process memory layout
- 64-bit processes do not have this limit
  - Limit exists, but is so large it can be effectively ignored
  - Addressability usually between 2^{44} and 2^{64} which is 16+ terabytes

Theoretical and advised maximum heap sizes (1 of 2)

- The larger the Java heap, the more constrained the native heap
- Limits are advised to prevent native heap from becoming overly restricted, leading to OutOfMemoryErrors
- Exceeding advised limits is possible, but should be done only when native heap usage is understood
- Native heap usage can be measured by using OS tools:
  - `svmon` (AIX)
  - `ps` (Linux) (for example: `ps -o pid,vsz,rss -p <PID>` where `vsz` is total virtual address space size and `rss` is resident set size)
  - `PerfMon` (Windows)
  - `RMF` (z/OS)

Theoretical and advised maximum heap sizes (2 of 2)

Platform	Extra options	Maximum possible	Advised maximum
AIX	automatic	3.25 GB	2.5 GB
Linux		2 GB	1.5 GB
	hugemem kernel	3 GB	2.5 GB
Windows		1.8 GB	1.5 GB
	/3GB	1.8 GB	1.8 GB
z/OS		1.7 GB	1.3 GB

Tuning considerations: Heap

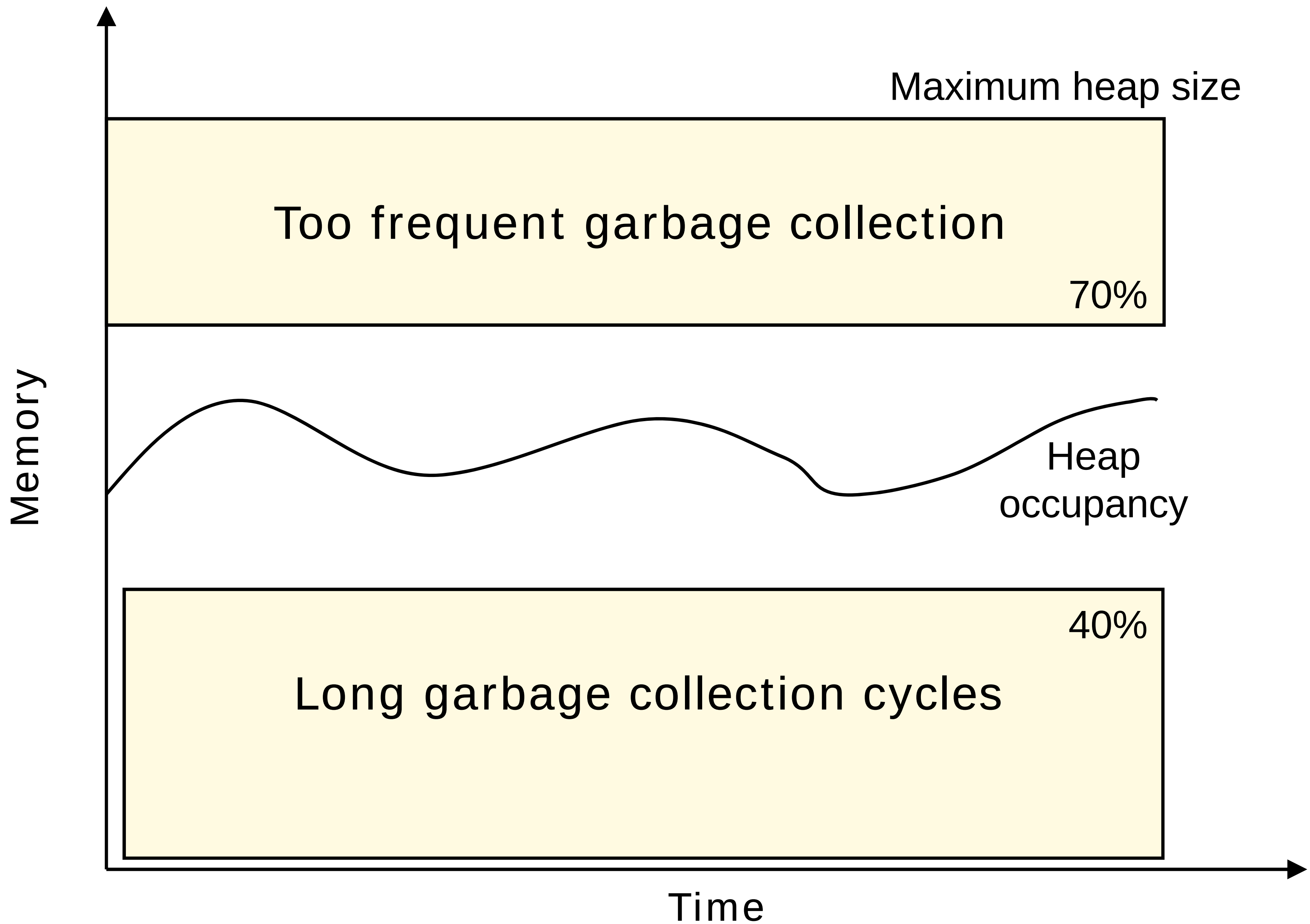
- Start with a reasonable maximum heap (-Xmx) size
  - 256 MB is the default for an application server
  - Try setting it to 512 MB
- Test different maximum heap sizes to find optimal setting
- Size the maximum heap larger than steady state to allow for peak load
- Consider opposing forces
  - The larger the heap, typically the longer the GC cycle
  - The smaller the heap, the more frequently GC is required
- Setting a larger minimum heap size (-Xms) can improve server startup time
  - If using the 50 MB default, might be resized several times during startup
- Setting the minimum too large can affect runtime performance
  - Larger value means more memory space that requires garbage collection
  - The JVM cannot compensate if you make a poor choice

The “correct” Java heap size (1 of 2)

- GC adapts heap size to keep occupancy between 40% and 70%
  - Heap occupancy over 70% causes frequent GC cycles
  - Which generally means reduced performance

- Heap occupancy below 40% means infrequent GC cycles, but cycles longer than they must be
  - Which means longer pause times than necessary
  - Which generally means reduced performance

The "correct" Java heap size (2 of 2)



Fixed heap sizes versus variable heap sizes

- Should the heap size be “fixed”?
  - That is, minimum heap size (-Xms) = maximum heap size (-Xmx)?
  - Does not expand or shrink the Java heap (avoids compactions)
  - Use for “flat” memory usage
- Variable heap sizes
  - GC adapts heap size to keep occupancy between 40% and 70%
  - Expands and shrinks the Java heap
  - Allows for scenario where usage varies over time, where variations would take usage outside of the 40 – 70% window
  - Provides more flexibility and ability to avoid OutOfMemoryErrors
- Each option has advantages and disadvantages
  - As for most performance tuning, you must select which is right for the particular application

Tuning considerations: Garbage collection

- “Throughput” (the JVM definition)
  - Measure of productive time
  - Time that is spent in GC is not included
- Pauses
  - Measure of time when application execution pauses during GC
- Turn on verbose GC for more information about GC operation
- Allow the JVM to determine optimum time to GC, avoid calling `System.gc()` from the application
  - Causes the least efficient, slowest GC to take place
  - Always triggers a compaction
  - `System.gc()` does not always trigger an immediate GC
  - Can be disabled in IBM JDKs using `-Xdisableexplicitgc`
  - `-Xdisableexplicitgc` must not be a permanent setting as the JVM uses `System.gc()` calls in extreme situations

Tuning considerations: GC policy

- "I want my application to run to completion as quickly as possible"

  `-Xgcpolicy:optthruput`

- "My application requires good response time to unpredictable events"

  `-Xgcpolicy:optavgpause`

- "My application has a high allocation and death rate (objects are short-lived)"

  `-Xgcpolicy:gencon`

- "I'm using a 64-bit system and need heap sizes greater than 4 GB"

  `-Xgcpolicy:balanced`

Tuning considerations: Native heap

- Beware: Garbage collection does not look into the native heap

- Ensure that JVM is not being swapped out of memory to disk
- Total memory that JVMuses > maximum heap size
- Native memory that is allocated in addition to the Java heap
- Native objects include:
  – Database connections for Type 2 JDBC drivers
  – Thread stacks
  – Compiled methods
  – JNI code and more
- Physical memory available > total memory that the JVM uses
  – JVMs do not page effectively due to garbage collection

JVM tool overview

- Assess the status of a running Java application
  – IBM Monitoring and Diagnostic Tools for Java – Health Center
  – The Health Center can also be used to monitor processor usage and lock contention
- WebSphere internal thread pools and heap usage statistics
  – Tivoli Performance Viewer
- Thread activity snapshot: Use javacore files
  – IBM Thread and Monitor Dump Analyzer for Java
- Memory and garbage collection: Use verbose GC output
  – IBM Monitoring and Diagnostic Tools for Java – Garbage Collection and Memory Visualizer (GCMV)
  – IBM Pattern Modeling and Analysis tool for Java Garbage Collector
- Memory use by object: Use heap dumps
  – Memory Analyzer tool (MAT)
  – IBM HeapAnalyzer

Unit summary

Having completed this unit, you should be able to:

- Describe components of JVM and overall architecture (Java Development Kit V6.0)
- Describe garbage collection (GC) and GC tuning policies
- Describe JVM command line arguments
- Describe javacore files and how to obtain them
- Identify a sluggish JVM and detect bottleneck problems
- Explain how to tune the heap size
- Use JVM-related tools: Garbage Collection and Memory Visualizer (GCMV), Memory Analyzer tool (MAT), and Java Health Center

Checkpoint questions

1. True or false: In addition to a mark and sweep, a compaction occurs for every garbage collection cycle.

2. True or false: The gencon garbage collection policy is enabled by default for all application servers.

3. True or false: In general, the larger the heap size, the longer the pause time during garbage collection.

Checkpoint answers

1. False: A compaction occurs only if the mark and sweep phases cannot recover enough heap space.

2. True: Starting with version 8, gencon policy is the default.

3. True

Exercise 3

Introduction to configuring garbage collection policies

Exercise objectives

After completing this exercise, you should be able to:

- Enable verbose GC for an application server
- Manually view verbose GC logs
- Configure the optthruput GC policy
- Configure the gencon GC policy
- Use IBM Monitoring and Diagnostic Tools for Java - Garbage Collection and Memory Visualizer (GCMV) to analyze verbose GC data
- Configure and use the Java Health Center