



Specialized Memory Pool for C and C++

Version 1.6
June 14, 2006

PathMATE Technical Notes

Pathfinder Solutions LLC
33 Commercial Drive, Suite 2
Foxboro, MA 02035 USA
www.PathfinderMDA.com
888-662-7284

Table Of Contents

1. Introduction.....	1
2. Feature Overview.....	1
Types of Pools	1
<i>Task-Specific General Purpose Pool</i>	<i>1</i>
<i>Task-Specific Incident Pool</i>	<i>1</i>
<i>Default Pool.....</i>	<i>2</i>
<i>Fault Handling</i>	<i>2</i>
Specification of Pool Block Sizes.....	2
<i>General Purpose Task Pool Sizing</i>	<i>2</i>
<i>Incident Pool Sizing</i>	<i>3</i>
<i>Default Pool.....</i>	<i>4</i>
Overall Feature Disabling	4
Marking Summary	4
3. Feature Implementation	5
Mechanism Extensions	5
Template Extensions.....	5
<i>When to Apply General Purpose Task Pool</i>	<i>5</i>
Task-Safety Report.....	6
Usage Statistics Reporting	6
4. Feature Design.....	6
Classes	7
<i>MemoryBlock</i>	<i>7</i>
<i>PfdAllocationCluster.....</i>	<i>7</i>
<i>PfdBufferManager <<singleton>></i>	<i>7</i>
<i>PfdClusterManager</i>	<i>8</i>
<i>PfdCriticalSection</i>	<i>8</i>
<i>PfdLocalPool</i>	<i>8</i>
<i>PfdPoolBlock</i>	<i>8</i>
<i>PfdPoolBlockSpecification</i>	<i>9</i>
<i>PfdPoolSpecification.....</i>	<i>9</i>
<i>PfdTask.....</i>	<i>9</i>
Associations	9
Domain Scenarios	10
Domain Services	21
PfdAllocationCluster Operations.....	21
PfdBufferManager Operations.....	21
PfdClusterManager Operations	22

PfdCriticalSection Operations	23
PfdLocalPool Operations	23
PfdPoolBlock Operations.....	23
5. System Types.....	23
Enumerates	23
User Defined Types	23

1. Introduction

This Technical Note describes extensions to PathMATE's C and C++ Maps to improve static buffer management. These extensions support task-local memory management pools. They are based on the buffer manager.

2. Feature Overview

The goals of this feature are to:

- Reduce the run-time overhead associated with dynamic memory allocation, which normally goes through the BufferManager, an intertask-safe resource.
- Allow for the specific sizing and allocation of dynamic memory blocks for specific classes.

This feature is built upon the base capabilities and characteristics of the BufferManager mechanism. This is a simple block management mechanism, where memory blocks of a small number of different sizes are arranged such that a request for a specific size block will result in the allocation of one of the smallest sized block available that will satisfy the request. Deallocation marks the originally allocated block as available again. One consequence of such a simple facility is that there is no notion of fragmentation – only contiguous blocks are allocated and deallocated, so no loss of efficiency over time is realized from varying usage and release patterns. Also, when a larger sized block is used to satisfy a smaller sized request, the entire block is allocated, so when it is deallocated it is again immediately available for use up to its complete size.

Types of Pools

All dynamically allocated memory elements come from one of three types of pools:

- A task-specific general purpose pool
- A task-specific incident pool
- The default pool

Note - the allocation of class instances from a pre-declared array is currently supported via the `MaxIndex` marking and the InstanceTable mechanism.

Task-Specific General Purpose Pool

You may define a memory pool for use within a single task. The pool provides task-local allocation and deallocation of memory blocks of a variety of user-specified sizes, in support of class instance data. Because it is task-local, the task-specific pool avoids the overhead of intertask safety mechanisms.

Note: If a task-specific pool is defined and a class instance is created for a class that did not specify a `MaxIndex` (and thereby got its own array space), the task-specific pool is used.

Task-Specific Incident Pool

A task-specific incident pool may be specified. This pool is an array within the Task structure that holds pointers to available Incident instances. These are

used to satisfy any incident allocations within the task. Once allocated, the address of the new incident is removed from the Task pool. Any Incidents freed within a Task that has an incident pool will go into that task's pool, regardless of where they came from. If a Task with an incident pool runs out of Incidents and a request comes in for another, the incident pool is replenished from the default pool. If a Task with an incident pool runs fill up with available Incidents and another is released, 1/2 of the available incidents in the incident pool are returned to the default pool.

To support possible incident parameters, when a task-specific incident pool is used, a task-specific data container pool is also used. The allocation/deallocation mechanics of this pool mirror that of the incident pool.

Default Pool

The default pool for dynamic memory allocation is the standard BufferManager pool. One instance of the BufferManager is allocated to each process in the system. Each instance provides intertask-safe allocation and deallocation of memory blocks of a variety of user-customizable sizes. If no other pool is specified for a dynamic memory request, or if an Incident is destined for the InterTaskIncidentQueue, the BufferManager pool is used.

Fault Handling

In the case of running out of memory the default action is to shut down the process in which the outage occurred. A new SW Service SW:RegisterForErrorGroup(enum fault_type, IncidentHandle handler) will be provided to allow for the user configurable handling of the SW_FAULT_DEPLETED_MEMORY fault.

Specification of Pool Block Sizes

General Purpose Task Pool Sizing

Task-specific pools are allocated by specifying the "TaskPool-<process label>-<task name>" marking. The designer should identify the specific sizes of the dynamic elements within their task, and specify block sizes and counts appropriately. For example consider a scenario within the Robochef.FoodPrep domain where many recipes can be loaded and unloaded. Let's say the base RecipeStepSpec class instance overhead is 40 bytes, and a maximum of 900 instance of this class are expected. Assume it is allocated to its own domain and this specifies a domain-specific pool. One way to allocate space for those events is to specify 900 blocks of 40 bytes. A partial specification of this would be:

```
System,Robochef,TaskPool-MAIN-SYS_TASK_ID_FP,... 900|40; ...
```

The sample specification below creates task-specific pools for the tasks SYS_TASK_ID_MAIN and SYS_TASK_ID_AUX. Each pool contains 1024 blocks in each of three block sizes:

```
System,CarShuffle,TaskPool-MAIN-
SYS_TASK_ID_MAIN,1024|16;1024|48;1024`|196
System,CarShuffle,TaskPool-MAIN-SYS_TASK_ID_AUX,1024|88;1024|148;1024|306
```

Optionally you may specify the reload and unload thresholds for each block size in the pool. When the number of blocks in the pool falls below the reload threshold, the pool will try to lock the buffer manager. If the buffer manager can't be locked, the pool will allocate the block. If the buffer manager can be locked, it will replenish the pool and then allocate the block. If the pool is empty, it will wait to obtain a lock on the buffer manager.

When the number of free spots in the pool falls below the unload threshold, the pool will try to lock the buffer manager. If the buffer manager can't be locked, the pool will deallocate the block. If the buffer manager can be locked, the pool will release memory to the buffer manager and deallocate the block. If the pool has no free spots, the pool will wait to obtain a lock on the buffer manager and release memory back to the buffer manager.

The default reload and unload thresholds are 5 or one less than the maximum count of items in the pool if the pool contains less than 5 items.

Specify the reload and unload thresholds as follows:

```
System,CarShuffle,TaskPool-MAIN-SYS_TASK_ID_MAIN,1024|16|2|4;1024|48;1024|196
System,CarShuffle,TaskPool-MAIN-SYS_TASK_ID_AUX,1024|88;1024|148;1024|306|20
```

In the example above, the main task pool of size 16 has a reload threshold of 2 and an unload threshold of 4. The size 48 and 196 pools use the default unload and reload threshold of 5.

The aux task pool of size 306 has an unload threshold of 20 and a default reload threshold of 5. There is no way to specify a default reload threshold and a non-default unload threshold.

This feature is not supported on Windows using Visual C++ 6.0 and on OSE. These platforms do not support an unblocked acquisition of a mutex.

NOTE: Reload and unload thresholds are only supported in C++.

Incident Pool Sizing

Task-specific incident pools are allocated by specifying the "TaskIncidentPool-<process label>-<task name>" and "TaskIncidentParameterPool-<process label>-<task name>" markings. The value of these markings is the maximum number of pre-allocated incident or parameter blocks to be made available. They should be tuned to support the incident and incident parameter needs of your system during high usage scenarios. The compiler flag PATH_MEMORY_STATISTICS will turn on statistics collection in the BufferManager and the LocalPools. Each of these classes has a reportStatistics operation that can be called to print out statistics on memory usage which can help the designer tune the memory pool sizes.

The sample specification below creates task-specific incident and parameter pools for the tasks SYS_TASK_ID_MAIN in the MON process, and SYS_TASK_ID_AUX in the PIO. Each task has 100 incidents and 50 parameters:

```
System,CarShuffle,TaskIncidentPool-MON-SYS_TASK_ID_MAIN,100
System,CarShuffle,TaskIncidentParameterPool-MON-SYS_TASK_ID_MAIN,50
System,CarShuffle,TaskIncidentPool-PIO-SYS_TASK_ID_AUX,100
```

```
System,CarShuffle,TaskIncidentParameterPool-PIO-SYS_TASK_ID_AUX,50
```

Default Pool

Block sizes for the standard BufferManager pool can now be specified by the system marking "DefaultPoolSizes." This property takes a list of sizes (numeric literals or symbolic constants) in increasing order. If

"DefaultPoolSizes" is not specified the standard BufferManager pool will retain the default pool configuration as specified in the BufferManager mechanism module. The sample specification below explicitly calls out block sizes matching the delivered BufferManager implementation:

```
System,CarShuffle,DefaultPoolSizes,2048|32;1638|40;1365|48;1024|64;
512|128;128|512;32|2048;1|0xFFFF
```

Overall Feature Disabling

All task-local pools (general, incident and parameter) are gated with the PATH_USE_LOCAL_POOLS compiler flag. This flag is set by default in generated project files and makefiles. Setting the system marking DisableLocalPools to "TRUE" prior to generating project files and makefiles will cause the PATH_USE_LOCAL_POOLS compiler flag not to be set, disabling all task-local pools.

Marking Summary

The following markings control this feature:

<u>Model Element</u>	<u>Name</u>	<u>Value</u>
System	DefaultPoolSizes	<count> <size>;<count> <size>...
System	DisableLocalPools	TRUE, FALSE (FALSE)
System	TaskPool-<process id>-<task id>	<count> <size>;<count> <size>...
System	TaskIncidentPool-<process id>-<task id>	<size>
System	TaskIncidentParameterPool-<process id>-<task id>	<size>

Where:

<count> and <size> are integer literals or symbolic constants,
 <process id>-<task id> specify a valid process label and task id (from domain marking)

In addition to marking-based controls, all BufferManager capabilities, excluding specialized pools, can be replaced by native malloc and free calls by setting the compile switch PATH_NO_BUFFER_MANAGER.

3. Feature Implementation

Mechanism Extensions

Extensions to the C-Maps base mechanisms and system files take the existing `SW_BufferManager` mechanism and generalize it. These extensions allow for multiple instances, either task-local or intertask-safe, and configurable block sizes. To specify the block size for the default pool, the `main()` function calls a generated pool allocation and configuration function.

The `SW_Task` structure adds the data member `memoryPool` to allow for a task-specific pool. If specified, this task-specific pool would be used in class instance and link structure allocations instead of the default pool.

The `SW_Task` structure adds the data members `incidentPool` and `parameterPool` to allow for a task-specific incident pools. If specified, the task-specific pool would be used in `SW_Incident_new`, `SW_Incident_deallocate`, `SW_DataContainer_new` and `SW_DataContainer_deallocate` instead of the buffer manager.

To support the use of class-specific pools, the functions `SW_BufferManager_allocBlockForPool` and `SW_BufferManager_freeBlockForPool` are added.

To streamline operations on sets of allocations and deallocations (supporting task-specific incident and parameter pools) by reducing the overhead of entering and leaving the `SW_BufferManager` critical section, a transactional-style interface will be added to the `SW_BufferManager`. `SW_BufferManager_startTransaction`, `SW_BufferManager_endTransaction`, `SW_BufferManager_fastAlloc` and `SW_BufferManager_fastFree` functions will be added

Template Extensions

Extensions to the C-Maps code generation templates are applied to ensure the proper memory mechanisms are used in accord with the marking values.

A pool allocation and configuration function is generated to specify the block sizes for the default pool. This function is called from the `main()` function.

The generated function `System_Run()` is modified to create, configure, and delete task-specific pools as specified by markings.

When to Apply General Purpose Task Pool

Template modifications are required to ensure that only truly local requests are made of the general purpose task pool. In order for a class to use this pool the following conditions must be met:

- The class is in a specific fixed task, and not `SYS_TASK_ANY`.
- The class is not statically allocated, and does not have a `MaxIndex` value specified indicating direct array storage.
- The class is allocated to a task with a general purpose pool defined.

- All actions where the class is CREATED and DELETED are allocated to the same fixed task.

Task-Safety Report

If only domain multi-task is implemented, then there will be no circumstances where the use of task-specific pools could result in an unsafe intertask memory access. However with domain operation multi-task or class multi-task support, it may be possible to specify that a single domain can run in multiple tasks. In this case, a set of templates will be run as part of code generation pre-processing to look for uses of task-specific pools that may incorrectly span task boundaries.

For example, assume the Carshuffle VehicleHousing domain has some domain operations that run in `SYS_TASK_ID_MAIN`, and some domain operations that run in `SYS_TASK_ID_AUX`. If the class `VH.Car` does not specify a `MaxIndex` and is not statically initialized, task-specific pools are specified, and `Car` instances are CREATED in the task `SYS_TASK_ID_MAIN` and DELETED in the task `SYS_TASK_ID_AUX`, this will be highlighted as a run-time issue via a transformation Engine LOGERROR message.

Allocating memory in one task and then deallocating it in another task is problematic if the pool specifications do not support the exact same size blocks. For example, if task T1 supports blocks of 16, 32 and 64 bytes and T2 supports blocks of 32 and 64 bytes, a problem occurs if a block of size 16 or less were allocated in T1 and deallocated in T2.

For example, T1 receives a request to allocate a block of size 8. T1 chooses a block size of 16 because it is the closest block size to 8. When the block is deallocated in T2, the pool block chooses the first pool with size \geq to 8. For T2, this is 32. The block, which is actually size 16, is now in a pool of size 32 blocks. If the size 16 block is allocated as if it is 32, memory corruption will occur as 16 bytes of the adjacent block in the buffer manager will be overwritten.

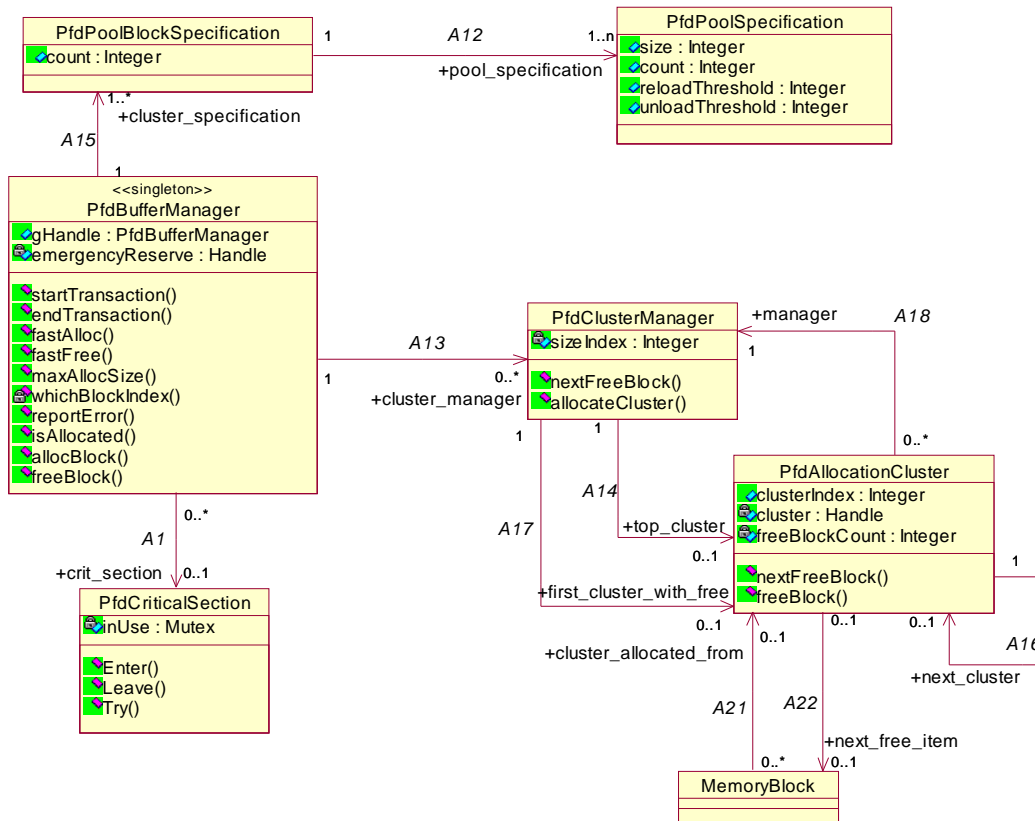
Usage Statistics Reporting

As a tool to designers to aid in the proper allocation of pools, simple statistics will be kept throughout each execution of the system, and reported upon system shutdown. For the default process pool and for each task-specific general purpose pool, the maximum number of blocks on a per-cluster (block-size) basis will be kept – a “high water” mark will be reported. In addition, the current number of blocks (per cluster) still in use at shutdown time will be reported.

For the task-specific incident pool, the total number of incidents allocated and released will be reported per-task. (It is thought that deficit trends – where a large imbalance exists in this area might be useful information).

All of the above statistic collection and report code is activated by the `PATH_MEMORY_STATISTICS` compile flag.

4. Feature Design



Memory Management Software Mechanisms Class Diagram

Classes

MemoryBlock

(U) A block of free memory residing in a local pool or buffer manager.

PfdAllocationCluster

(U) A block of memory divided up and allocated out in equal sized blocks.

cluster (Handle): (U) Block of space managed by the cluster. This space is divided up into blocks of the appropriate size and allocated.

clusterIndex (Integer): (U) The index of the cluster into the list of clusters held by the cluster manager.

freeBlockCount (Integer): (U) The number of free blocks left within this cluster.

PfdBufferManager <<singleton>>

(U) Allocates and frees dynamic memory blocks efficiently by using a limited set of block sizes. The buffer manager is a singleton instance that can be accessed by any of the tasks in the program. The buffer manager is protected by a mutex so it can be accessed by multiple tasks.

(U) The population of block sizes can be customized by specifying design markings.

emergencyReserve (Handle): (U) Reserved memory to be released if memory runs out. Releasing the reserved memory will allow the system to terminate gracefully.

gHandle (PfdBufferManager): (U) The singleton instance of the buffer manager.

PfdClusterManager

(U) Tracks all of the clusters holding memory blocks of the same size. The cluster manager creates clusters at initialization time and allocates new clusters as the existing clusters run out of memory blocks.

sizeIndex (Integer): (U) The size of the block held by this cluster.

PfdCriticalSection

(U) A synchronization mechanism to support concurrent access to a shared resource.

inUse (Mutex): (U) The mutex or critical section providing locking to the client tasks. The implementation is platform specific.

PfdLocalPool

(U) A pool of memory blocks of itemSize allocated from the buffer manager. The memory pool avoids the overhead of acquiring the critical section for the singleton buffer manager each time an allocation is done.

(U) At startup, the pool acquires the buffer manager critical section and loads the pool. When the number of free blocks drops below the reloadThreshold, the pool attempts to acquire the buffer manager critical section and reload the pool.

(U) If the number of empty slots in the pool falls below the unloadThreshold, the pool attempts to acquire the buffer manager critical section and returns memory to the buffer manager.

itemSize (Integer): (U) The size of the memory blocks held in the pool in bytes.

maxCount (Integer): (U) The maximum number of memory blocks that can be stored in the pool.

nextAvailable (Integer): (U) The index of the next free block or -1 if no free blocks are available.

reloadThreshold (Integer): (U) Start trying to acquire the lock on the buffer manager to reload the pool when the number of free blocks falls below this number.

unloadThreshold (Integer): (U) Start trying to acquire the buffer manager lock to unload the pool when the number of free spaces falls below this number.

PfdPoolBlock

(U) A set of local memory pools of different sizes used exclusively by a particular task.

poolCount (Integer): (U) The number of local pools in this block

PfdPoolBlockSpecification

(U) Defines the parameters for a set of memory pools.

count (Integer): (U) The number of pools of different sizes available.

PfdPoolSpecification

(U) Defines the parameters for the memory blocks held by a memory pool.

count (Integer): (U) The number of blocks of memory in the pool.

reloadThreshold (Integer): (U) When the number of free blocks in the pool falls below this threshold, start attempting to acquire the lock on the buffer manager.

size (Integer): (U) The size of each block of memory in the pool.

unloadThreshold (Integer): (U) When the number of empty spots in the pool falls below this threshold, start attempting to acquire the lock on the buffer manager.

PfdTask

(U) A thread of control for dispatching events to state models and executing actions.

Associations

A1 :

PfdCriticalSection crit_section (0..1) \leftarrow A1 \rightarrow (*) PfdBufferManager

description: (U) The critical section that protects the buffer manager as memory is allocated by multiple threads.

SortOrder: FIFO

A11 :

PfdLocalPool pool_block (1..*) \leftarrow A11 \rightarrow (1) PfdPoolBlock

description: (U) A set of memory pools holding items of different sizes.

SortOrder: FIFO

A12 :

PfdPoolSpecification pool_specification (1..*) \leftarrow A12 \rightarrow (1)

PfdPoolBlockSpecification

description: (U) Defines all the memory pool sizes supported by the pool block. PoolSpecifications are order by ascending size across this association

SortOrder: ascending

A13 :

PfdClusterManager cluster_manager (*) \leftarrow A13 \rightarrow (1) PfdBufferManager

description: (U) The cluster managers that perform allocations on the different block sizes.

SortOrder: FIFO

A14 :

PfdAllocationCluster top_cluster (0..1) \leftarrow A14 \rightarrow (1) PfdClusterManager

description: (U) The first cluster of the specified size.

SortOrder: FIFO

A15 :

PfdPoolBlockSpecification cluster_specification (1..*) \leftarrow A15 \rightarrow (1)

PfdBufferManager

description: (U) Defines the set of memory block sizes supported by the buffer manager.

SortOrder: FIFO

A16 :

PfdAllocationCluster next_cluster (0..1) \leftarrow A16 \rightarrow (1) PfdAllocationCluster

description: (U) The next cluster allocating blocks of the same size.

SortOrder: FIFO

A17 :

PfdAllocationCluster first_cluster_with_free (0..1) \leftarrow A17 \rightarrow (1)

PfdClusterManager

description: (U) First cluster managed by the manager that may contain a free block.

SortOrder: FIFO

A18 :

PfdClusterManager manager (1) \leftarrow A18 \rightarrow (*) PfdAllocationCluster

description: (U) Defines the parent manager for this cluster.

SortOrder: FIFO

A19 :

MemoryBlock item_pool (*) \leftarrow A19 \rightarrow (1) PfdLocalPool

description: (U) The set of free blocks in the local pool. This association is implemented as an array of size maxCount. The array holds the set of available blocks. Blocks may be freed to an empty spot in the array. The nextAvailable attribute keeps track of the first memory block stored in the array.

SortOrder: FIFO

A20 :

PfdPoolBlock general_pool_block (0..1) \leftarrow A20 \rightarrow (1) PfdTask

description: (U) The general purpose local memory pool for this task.

When linked, use this pool when allocating memory from this task.

SortOrder: FIFO

A21 :

PfdAllocationCluster cluster_allocated_from (0..1) \leftarrow A21 \rightarrow (*)

MemoryBlock

description: (U) A memory block when it is allocated points back to its cluster. When it is deallocated, the pointer is used to determine its owning cluster.

SortOrder: FIFO

A22 :

MemoryBlock next_free_item (0..1) \leftarrow A22 \rightarrow (0..1) PfdAllocationCluster

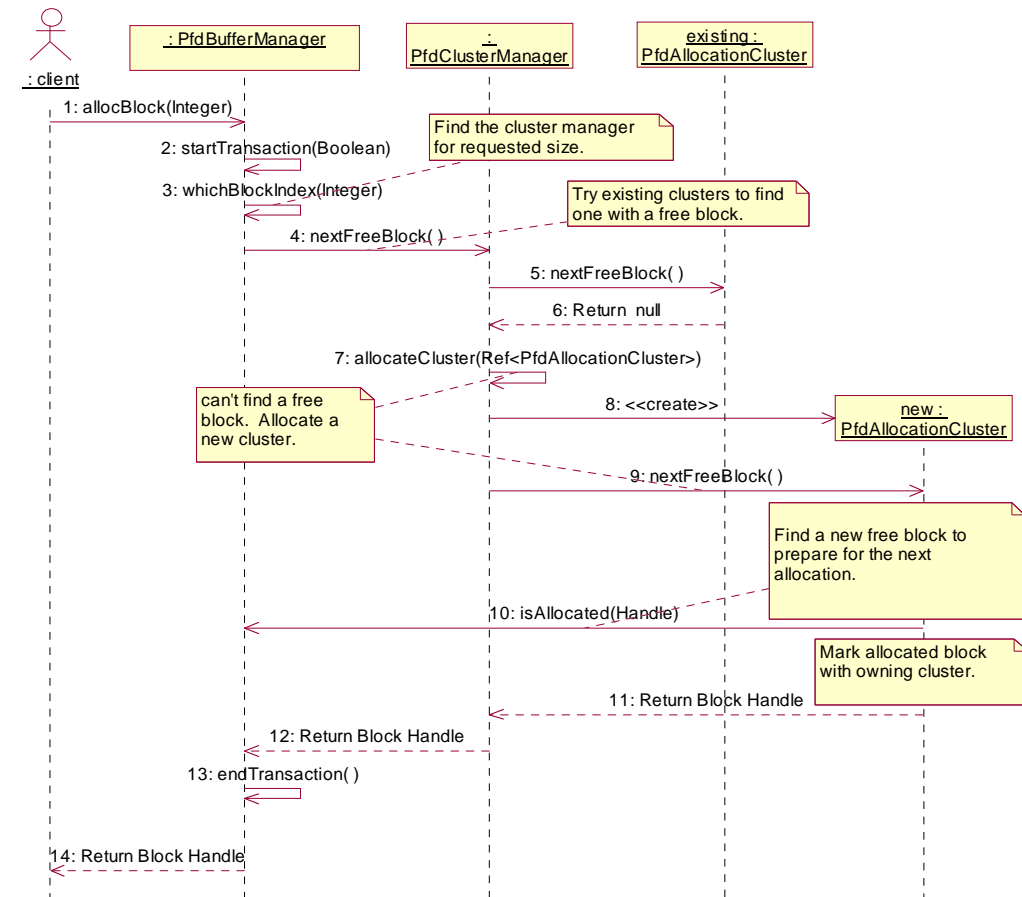
description: (U) The free memory block in this cluster with the lowest address. The next memory block to be allocated.

SortOrder: FIFO

Domain Scenarios

UNCLASSIFIED

Allocate from Buffer Manager - New Cluster Required



Description:
Allocate a block of memory from the buffer manager. For this nominal flow, a block of memory is successfully allocated from a newly allocated cluster.

Preconditions:

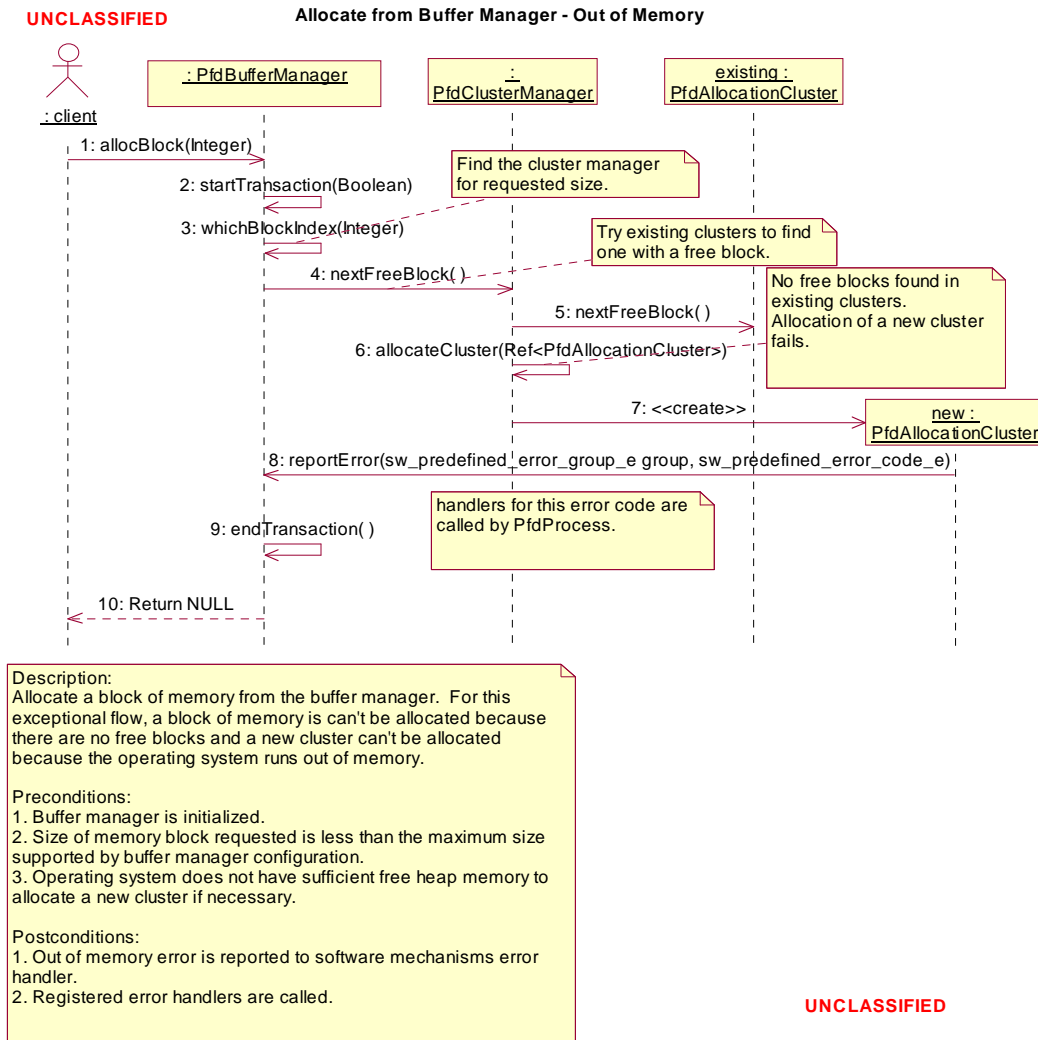
1. Buffer manager is initialized.
2. Size of memory block requested is less than the maximum size supported by buffer manager configuration.
3. Operating system has sufficient free heap memory to allocate a new cluster.
4. Existing clusters of the size requested do not have any free blocks.

Postconditions:

1. Buffer manager located and returned a handle to memory block of the appropriate size.

UNCLASSIFIED

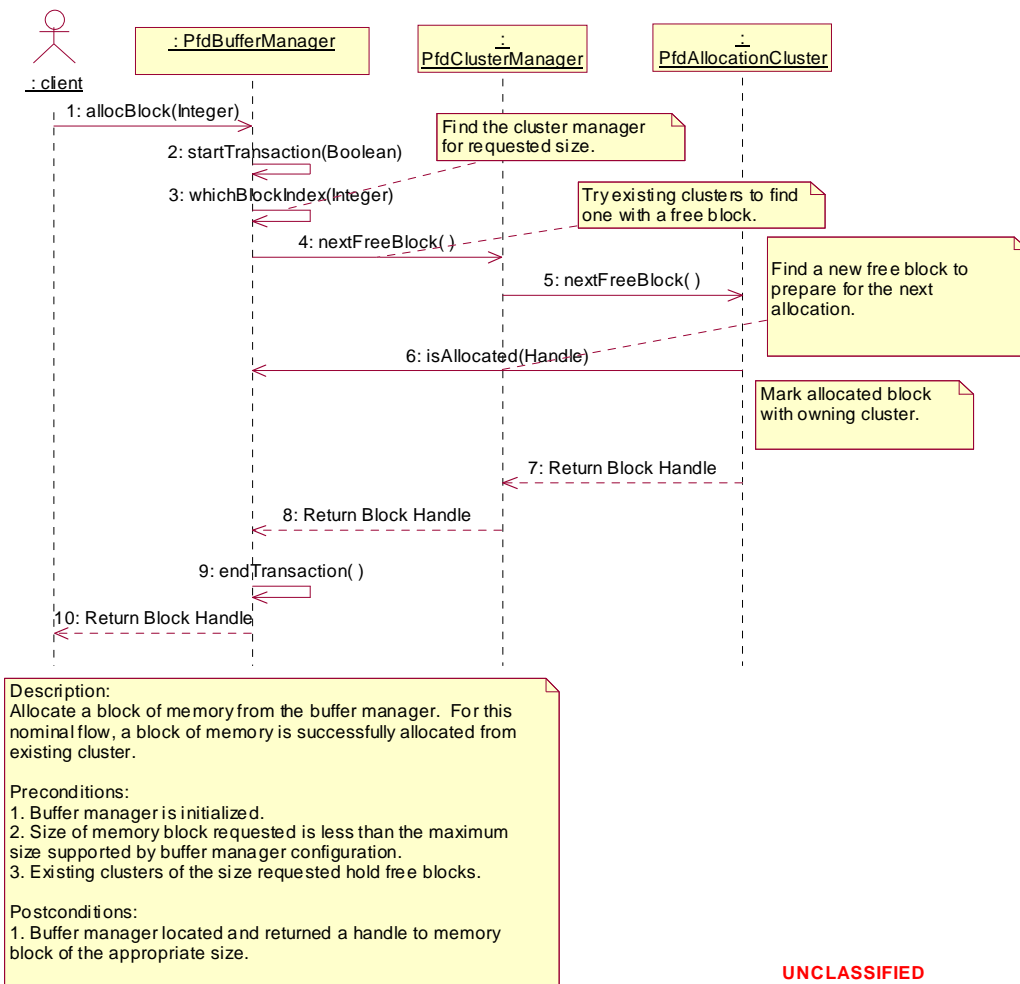
Domain Scenario - Allocate from Buffer Manager - New Cluster Required



Domain Scenario - Allocate from Buffer Manager - Out of memory

UNCLASSIFIED

Allocate from Buffer Manager - Nominal

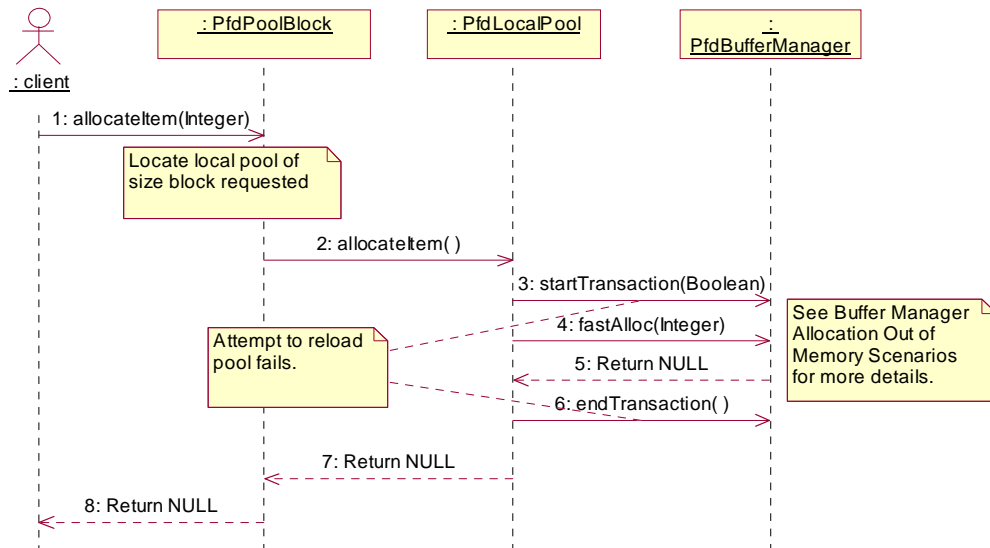


UNCLASSIFIED

Domain Scenario - Allocate from Buffer Manager

UNCLASSIFIED

Allocate from Local Pool - Out of Memory



Description:
Allocate a block of memory from a local pool. For this exceptional flow, the pool is empty and there is insufficient memory to reload the local pool. The allocation fails.

Preconditions:

1. Buffer manager is initialized.
2. Local pools are initialized.
3. Size of memory block requested is less than the maximum size supported by local pool configuration.
4. Local pool of correct size is empty.
5. Buffer manager can't allocate enough memory to reload the local pool.

Postconditions:

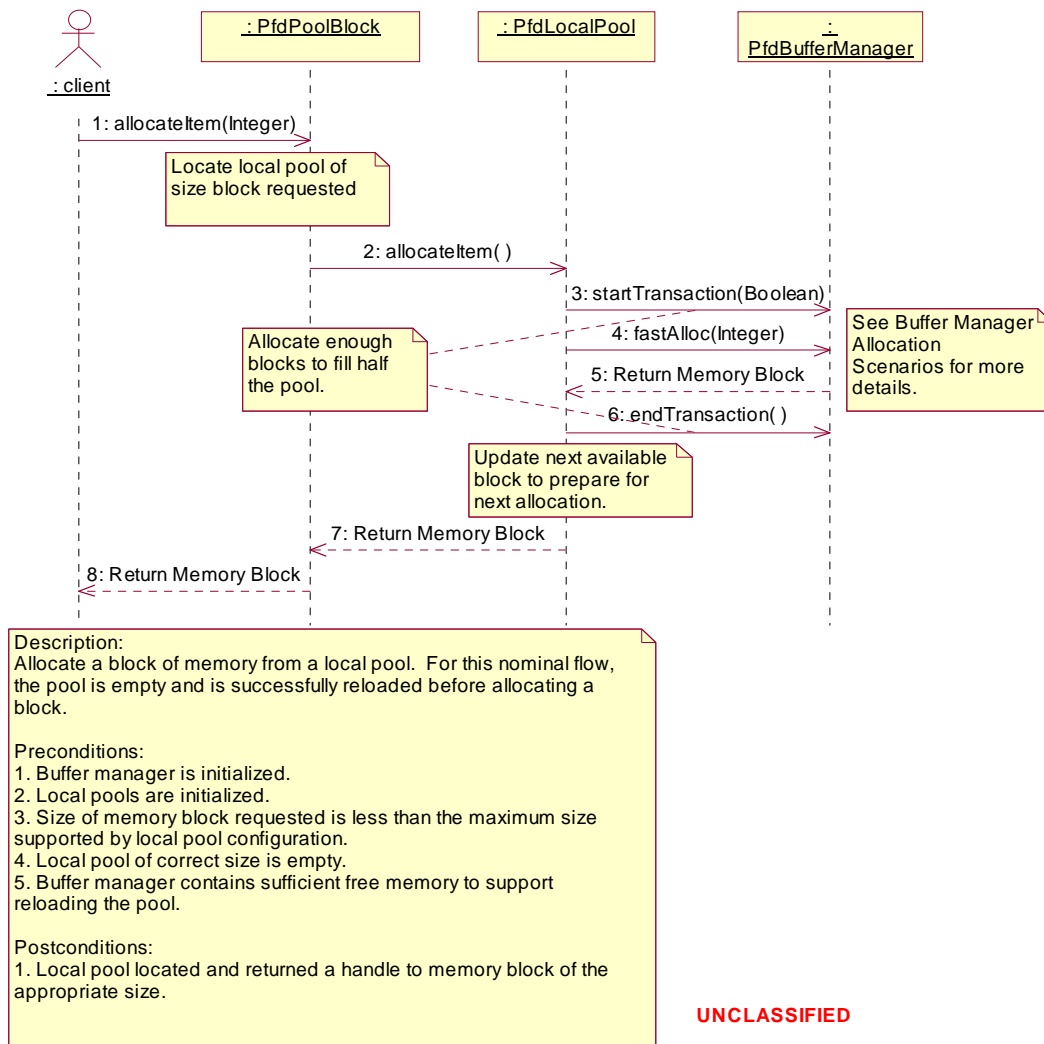
1. Local pool located and null handle is returned.

UNCLASSIFIED

Domain Scenario - Allocate from Local Pool - Out of Memory

UNCLASSIFIED

Allocate from Local Pool - Reload Required

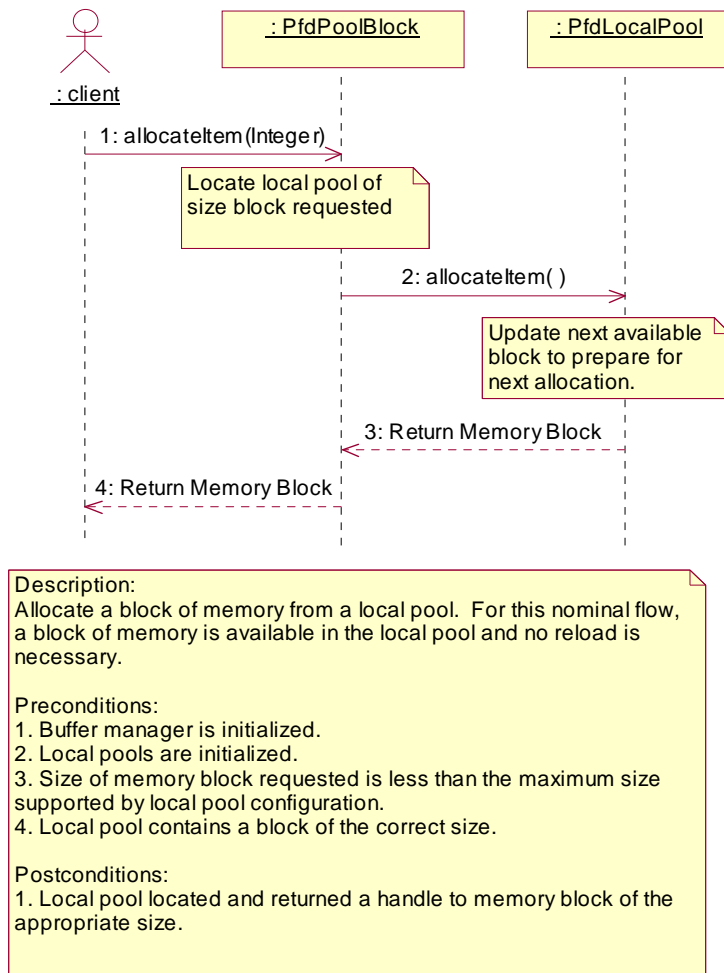


UNCLASSIFIED

Domain Scenario - Allocate from Local Pool - Reload required

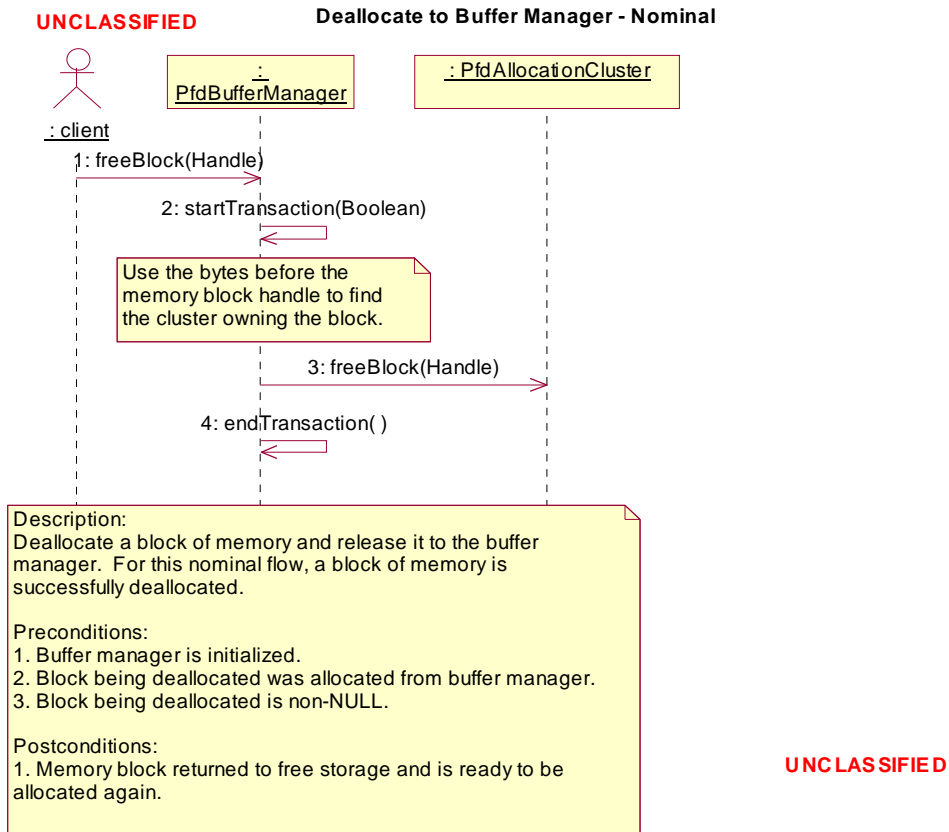
UNCLASSIFIED

Allocate from Local Pool - Nominal

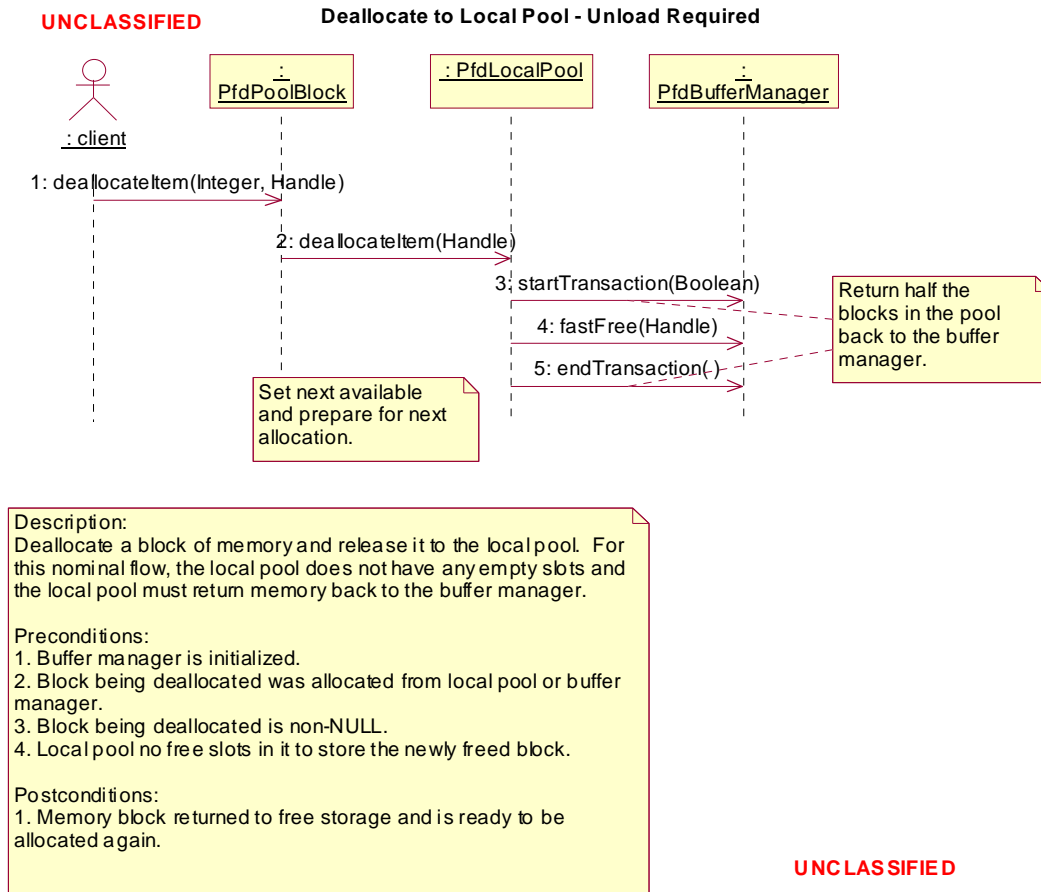


UNCLASSIFIED

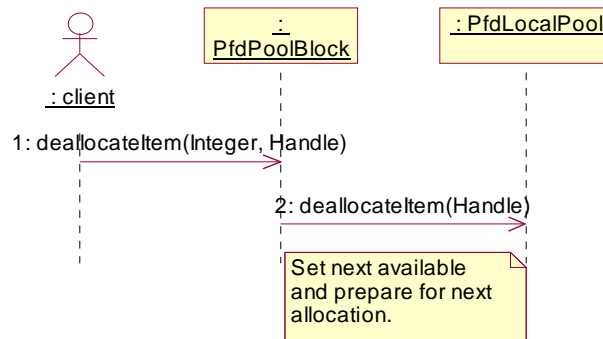
Domain Scenario - Allocate from Local Pool



Domain Scenario - Deallocate to Buffer Manager



Domain Scenario - Deallocate to Local Pool - Unload required

UNCLASSIFIED**Deallocate to Local Pool - Nominal**

Description:
Deallocate a block of memory and release it to the local pool. For this nominal flow, the local pool has empty slots in it and no unload is required.

Preconditions:

1. Buffer manager is initialized.
2. Block being deallocated was allocated from local pool or buffer manager.
3. Block being deallocated is non-NULL.
4. Local pool has at least one free slot in it to store the newly freed block.

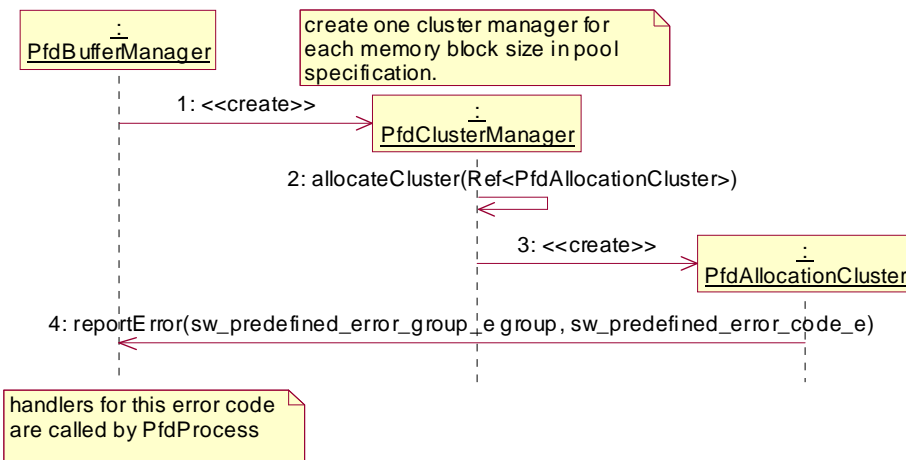
Postconditions:

1. Memory block returned to free storage and is ready to be allocated again.

UNCLASSIFIED**Domain Scenario - Deallocate to Local Pool**

UNCLASSIFIED

Initialize Buffer Manager - Out of Memory



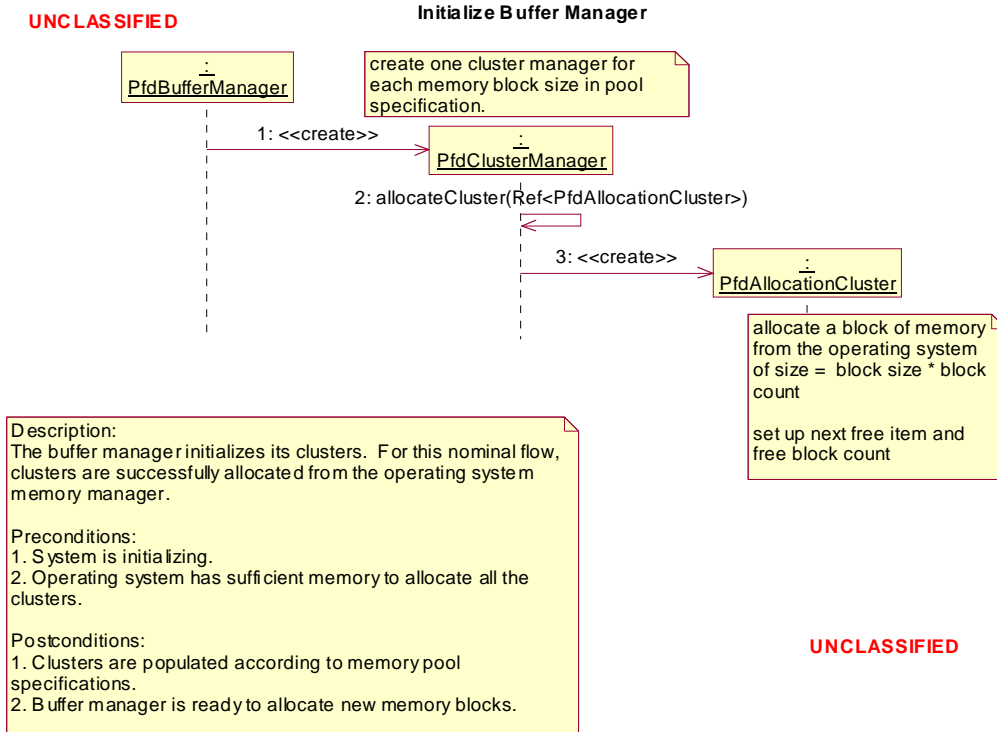
Description:
The operating system runs out of memory while the buffer manager is initializing its clusters. An out of memory error is reported.

Preconditions:
1. System is initializing.
2. Operating system has insufficient memory to allocate all the clusters.

Postconditions:
1. Out of memory error is reported to software mechanisms error handler.
2. Registered error handlers are called.

UNCLASSIFIED

Domain Scenario - Initialize Buffer Manager - Out of Memory



Domain Scenario - Initialize Buffer Manager

Domain Services

PfdAllocationCluster Operations

PfdAllocationCluster:freeBlock (instance-based): (U) Return a memory block back to the free store so it can be allocated again.

in: **item_to_free** (Handle): (U) The block to free.

PfdAllocationCluster:nextFreeBlock (instance-based): (U) Returns the next free block in the cluster. Returns 0 if none are available.

returns: (Handle)

PfdBufferManager Operations

PfdBufferManager:allocBlock (instance-based): (U) Allocate a block of memory of at least size and return a handle to it.

returns: (Handle)

in: **size** (Integer): (U) The minimum size of the memory block to allocate.

PfdBufferManager:endTransaction (object-based): (U) Unlock the mutex for the buffer manager.

PfdBufferManager:fastAlloc (object-based): (U) Allocate without locking the mutex. The client must call startTransaction before calling this member function.

returns: (Handle)

in: **size** (Integer): (U) The size of the memory block to be allocated.

PfdBufferManager::fastFree (object-based): (U) Free without locking the mutex. The client must call *startTransaction* before calling this member function.

in: **block_to_free** (Handle): (U) The handle to the block that will be returned to available free memory.

PfdBufferManager::freeBlock (instance-based): (U) Put a block to the free store. The block is ready to be allocated again.

in: **block** (Handle): (U) A handle to the block to be freed. Ignore NULL handles.

PfdBufferManager::isAllocated (object-based): (U) Return *TRUE* if the block has already been allocated. Return *FALSE* if the block is free.

returns: (Boolean)

in: **block_handle** (Handle): (U) Check this memory block to see if it is allocated or free.

PfdBufferManager::maxAllocSize (object-based): (U) Return the maximum size block that can be allocated from the buffer manager.

returns: (Integer)

PfdBufferManager::reportError (instance-based): (U) Report an memory error detected during allocation or deallocation.

in: **error_group** (sw_predefined_error_group_e group): (U) Report an error belonging to this group.

in: **error_code** (sw_predefined_error_code_e): (U) Report an error with this code.

PfdBufferManager::startTransaction (object-based): (U) Obtain the mutex required for accessing the buffer manager. Return *TRUE* if the lock could be obtained. Return *FALSE*, if *wait_for_critical* was *FALSE* and the buffer manager lock could not be obtained without waiting.

returns: (Boolean)

in: **wait_for_critical** (Boolean): (U) If *TRUE*, block while waiting for the buffer manager mutex. If *FALSE*, only lock the mutex if it can acquired without waiting.

PfdBufferManager::whichBlockIndex (object-based): (U) Return the index of the cluster used to allocate a block of the requested size.

returns: (Integer)

in: **request_size** (Integer): (U) The size of the block requested.

PfdClusterManager Operations

PfdClusterManager::allocateCluster (instance-based): (U) Allocate a new cluster for this manager. Return *TRUE* if successful.

returns: (Boolean)

in: **previous_cluster** (Ref<PfdAllocationCluster>): (U) The previous cluster in the chain of clusters.

PfdClusterManager::nextFreeBlock (instance-based): (U) Return a free block from an allocation cluster. Return 0 if all clusters are used and no new clusters can be allocated.

returns: (Handle)

PfdCriticalSection Operations

PfdCriticalSection:Enter (instance-based): (U) Lock the critical section. Suspend the task and wait if the critical section is locked by another task.

PfdCriticalSection:Leave (instance-based): (U) Unlock the critical section so that other tasks may access the shared resource.

PfdCriticalSection:Try (instance-based): (U) Try to lock the critical section. Don't wait if the critical section is locked by another task. Return TRUE if the critical section was successfully locked. Return FALSE if the critical section was not successfully locked. To acquire the lock, the client must call Enter to wait for the lock to become available or poll using Try.
returns: (Boolean)

PfdLocalPool Operations

PfdLocalPool:allocateItem (instance-based): (U) Allocated a block of memory from the pool. Return a pointer to the block or 0 if the block can't be allocated.
returns: (Handle)

PfdLocalPool:deallocateItem (instance-based): (U) Free a block of memory.
in: **item** (Handle): (U) A pointer to the block of memory to free.

PfdLocalPool:reload (instance-based): (U) Reload the pool with more blocks from the buffer manager if it is empty or almost out.

PfdLocalPool:unload (instance-based): (U) Return some of the empty memory to the buffer manager if the pool is completely full or nearly full.

PfdPoolBlock Operations

PfdPoolBlock:allocateItem (instance-based): (U) Find the correct pool containing the correct memory block size. Allocate a block of memory from the pool and return a pointer to the memory.
returns: (Handle)
in: **size** (Integer): (U) The size of the block of memory to allocate.

PfdPoolBlock:deallocateItem (instance-based): (U) Return a memory block to the pool for its size.
in: **item_size** (Integer): (U) The size of the memory block pointed to by item in bytes.
in: **item** (Handle): (U) A handle to the block of memory to be freed.

5. System Types

Enumerates

User Defined Types

Mutex : base: Integer

PfdBufferManager : base: Integer

sw_predefined_error_code_e : base: Integer

sw_predefined_error_group_e group : base: Integer