//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "CollectionState.h"

namespace Js

{

class Profiler;

enum Phase;

};

namespace JsUtil

{

class ThreadService;

};

class StackBackTraceNode;

class ScriptEngineBase;

class JavascriptThreadService;

#ifdef PROFILE\_MEM

struct RecyclerMemoryData;

#endif

namespace Memory

{

template <typename T> class RecyclerRootPtr;

class AutoBooleanToggle

{

public:

AutoBooleanToggle(bool \* b, bool value = true, bool valueMayChange = false)

: b(b)

{

Assert(!(\*b));

\*b = value;

#if DBG

this->value = value;

this->valueMayChange = valueMayChange;

#endif

}

~AutoBooleanToggle()

{

if (b)

{

Assert(valueMayChange || \*b == value);

\*b = false;

}

}

void Leave()

{

Assert(valueMayChange || \*b == value);

\*b = false;

b = nullptr;

}

private:

bool \* b;

#if DBG

bool value;

bool valueMayChange;

#endif

};

template <class T>

class AutoRestoreValue

{

public:

AutoRestoreValue(T\* var, const T& val):

variable(var)

{

Assert(var);

oldValue = (\*variable);

(\*variable) = val;

#ifdef DEBUG

debugSetValue = val;

#endif

}

~AutoRestoreValue()

{

Assert((\*variable) == debugSetValue);

(\*variable) = oldValue;

}

private:

#ifdef DEBUG

T debugSetValue;

#endif

T\* variable;

T oldValue;

};

class Recycler;

class RecyclerScanMemoryCallback

{

public:

RecyclerScanMemoryCallback(Recycler\* recycler) : recycler(recycler) {}

void operator()(void\*\* obj, size\_t byteCount);

private:

Recycler\* recycler;

};

template<ObjectInfoBits infoBits>

struct InfoBitsWrapper{};

// Allocation macro

#define RecyclerNew(recycler,T,...) AllocatorNewBase(Recycler, recycler, AllocInlined, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewPlus(recycler,size,T,...) AllocatorNewPlus(Recycler, recycler, size, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewPlusLeaf(recycler,size,T,...) AllocatorNewPlusLeaf(Recycler, recycler, size, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewPlusZ(recycler,size,T,...) AllocatorNewPlusZ(Recycler, recycler, size, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewPlusLeafZ(recycler,size,T,...) AllocatorNewPlusLeafZ(Recycler, recycler, size, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewZ(recycler,T,...) AllocatorNewBase(Recycler, recycler, AllocZeroInlined, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewStruct(recycler,T) AllocatorNewStructBase(Recycler, recycler, AllocInlined, T)

#define RecyclerNewStructZ(recycler,T) AllocatorNewStructBase(Recycler, recycler, AllocZeroInlined, T)

#define RecyclerNewStructPlus(recycler,size,T) AllocatorNewStructPlus(Recycler, recycler, size, T)

#define RecyclerNewStructLeaf(recycler,T) AllocatorNewStructBase(Recycler, recycler, AllocLeafInlined, T)

#define RecyclerNewStructLeafZ(recycler,T) AllocatorNewStructBase(Recycler, recycler, AllocLeafZeroInlined, T)

#define RecyclerNewLeaf(recycler,T,...) AllocatorNewBase(Recycler, recycler, AllocLeafInlined, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewLeafZ(recycler,T,...) AllocatorNewBase(Recycler, recycler, AllocLeafZeroInlined, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewArrayLeafZ(recycler,T,count) AllocatorNewArrayBase(Recycler, recycler, AllocLeafZero, T, count)

#define RecyclerNewArray(recycler,T,count) AllocatorNewArrayBase(Recycler, recycler, Alloc, T, count)

#define RecyclerNewArrayZ(recycler,T,count) AllocatorNewArrayBase(Recycler, recycler, AllocZero, T, count)

#define RecyclerNewArrayLeaf(recycler,T,count) AllocatorNewArrayBase(Recycler, recycler, AllocLeaf, T, count)

// Use static\_cast to make sure the finalized and tracked object have the right base class

#define RecyclerNewFinalized(recycler,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewBase(Recycler, recycler, AllocFinalizedInlined, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewFinalizedLeaf(recycler,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewBase(Recycler, recycler, AllocFinalizedLeafInlined, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewFinalizedPlus(recycler, size, T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewPlusBase(Recycler, recycler, AllocFinalized, size, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewFinalizedLeafPlus(recycler, size, T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewPlusBase(Recycler, recycler, AllocFinalizedLeaf, size, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewTracked(recycler,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewBase(Recycler, recycler, AllocTrackedInlined, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewTrackedLeaf(recycler,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewBase(Recycler, recycler, AllocTrackedLeafInlined, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewTrackedLeafPlusZ(recycler,size,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewPlusBase(Recycler, recycler, AllocZeroTrackedLeafInlined, size, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewEnumClass(recycler, enumClass, T, ...) new (TRACK\_ALLOC\_INFO(static\_cast<Recycler \*>(recycler), T, Recycler, 0, (size\_t)-1), enumClass) T(\_\_VA\_ARGS\_\_)

#define RecyclerNewWithInfoBits(recycler, infoBits, T, ...) new (TRACK\_ALLOC\_INFO(static\_cast<Recycler \*>(recycler), T, Recycler, 0, (size\_t)-1), InfoBitsWrapper<infoBits>()) T(\_\_VA\_ARGS\_\_)

#define RecyclerNewFinalizedClientTracked(recycler,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewBase(Recycler, recycler, AllocFinalizedClientTrackedInlined, T, \_\_VA\_ARGS\_\_)))

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC

#define RecyclerNewWithBarrier(recycler,T,...) AllocatorNewBase(Recycler, recycler, AllocWithBarrier, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewWithBarrierPlus(recycler,size,T,...) AllocatorNewPlusBase(Recycler, recycler, AllocWithBarrier, size, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewWithBarrierPlusZ(recycler,size,T,...) AllocatorNewPlusBase(Recycler, recycler, AllocZeroWithBarrier, size, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewWithBarrierArray(recycler,T,count) AllocatorNewArrayBase(Recycler, recycler, AllocWithBarrier, T, count)

#define RecyclerNewWithBarrierArrayZ(recycler,T,count) AllocatorNewArrayBase(Recycler, recycler, AllocZeroWithBarrier, T, count)

#define RecyclerNewWithBarrierStruct(recycler,T) AllocatorNewStructBase(Recycler, recycler, AllocWithBarrier, T)

#define RecyclerNewWithBarrierStructZ(recycler,T) AllocatorNewStructBase(Recycler, recycler, AllocZeroWithBarrier, T)

#define RecyclerNewWithBarrierFinalized(recycler,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewBase(Recycler, recycler, AllocFinalizedWithBarrierInlined, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewWithBarrierFinalizedPlus(recycler, size, T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewPlusBase(Recycler, recycler, AllocFinalizedWithBarrier, size, T, \_\_VA\_ARGS\_\_)))

#else

#define RecyclerNewWithBarrier RecyclerNew

#define RecyclerNewWithBarrierPlus RecyclerNewPlus

#define RecyclerNewWithBarrierPlusZ RecyclerNewPlusZ

#define RecyclerNewWithBarrierArray RecyclerNewArray

#define RecyclerNewWithBarrierArrayZ RecyclerNewArrayZ

#define RecyclerNewWithBarrierStruct RecyclerNewStruct

#define RecyclerNewWithBarrierStructZ RecyclerNewStructZ

#define RecyclerNewWithBarrierFinalized RecyclerNewFinalized

#define RecyclerNewWithBarrierFinalizedPlus RecyclerNewFinalizedPlus

#endif

#ifdef TRACE\_OBJECT\_LIFETIME

#define RecyclerNewLeafTrace(recycler,T,...) AllocatorNewBase(Recycler, recycler, AllocLeafTrace, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewLeafZTrace(recycler,T,...) AllocatorNewBase(Recycler, recycler, AllocLeafZeroTrace, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewPlusLeafTrace(recycler,size,T,...) AllocatorNewPlusBase(Recycler, recycler, AllocLeafTrace, size, T, \_\_VA\_ARGS\_\_)

#define RecyclerNewArrayLeafZTrace(recycler,T,count) AllocatorNewArrayBase(Recycler, recycler, AllocLeafZeroTrace, T, count)

#define RecyclerNewArrayTrace(recycler,T,count) AllocatorNewArrayBase(Recycler, recycler, AllocTrace, T, count)

#define RecyclerNewArrayZTrace(recycler,T,count) AllocatorNewArrayBase(Recycler, recycler, AllocZeroTrace, T, count)

#define RecyclerNewArrayLeafTrace(recycler,T,count) AllocatorNewArrayBase(Recycler, recycler, AllocLeafTrace, T, count)

// Use static\_cast to make sure the finalized and tracked object have the right base class

#define RecyclerNewFinalizedTrace(recycler,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewBase(Recycler, recycler, AllocFinalizedTrace, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewFinalizedLeafTrace(recycler,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewBase(Recycler, recycler, AllocFinalizedLeafTrace, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewFinalizedPlusTrace(recycler, size, T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewPlusBase(Recycler, recycler, AllocFinalizedTrace, size, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewTrackedTrace(recycler,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewBase(Recycler, recycler, AllocTrackedTrace, T, \_\_VA\_ARGS\_\_)))

#define RecyclerNewTrackedLeafTrace(recycler,T,...) static\_cast<T \*>(static\_cast<FinalizableObject \*>(AllocatorNewBase(Recycler, recycler, AllocTrackedLeafTrace, T, \_\_VA\_ARGS\_\_)))

#else

#define RecyclerNewLeafTrace RecyclerNewLeaf

#define RecyclerNewLeafZTrace RecyclerNewLeafZ

#define RecyclerNewPlusLeafTrace RecyclerNewPlusLeaf

#define RecyclerNewArrayLeafZTrace RecyclerNewArrayLeafZ

#define RecyclerNewArrayTrace RecyclerNewArray

#define RecyclerNewArrayZTrace RecyclerNewArrayZ

#define RecyclerNewArrayLeafTrace RecyclerNewArrayLeaf

#define RecyclerNewFinalizedTrace RecyclerNewFinalized

#define RecyclerNewFinalizedLeafTrace RecyclerNewFinalizedLeaf

#define RecyclerNewFinalizedPlusTrace RecyclerNewFinalizedPlus

#define RecyclerNewTrackedTrace RecyclerNewTracked

#define RecyclerNewTrackedLeafTrace RecyclerNewTrackedLeaf

#endif

#ifdef RECYCLER\_TRACE

#define RecyclerVerboseTrace(flags, ...) \

if (flags.Verbose && flags.Trace.IsEnabled(Js::RecyclerPhase)) \

{ \

Output::Print(\_\_VA\_ARGS\_\_); \

}

#define AllocationVerboseTrace(flags, ...) \

if (flags.Verbose && flags.Trace.IsEnabled(Js::MemoryAllocationPhase)) \

{ \

Output::Print(\_\_VA\_ARGS\_\_); \

}

#define LargeAllocationVerboseTrace(flags, ...) \

if (flags.Verbose && \

(flags.Trace.IsEnabled(Js::MemoryAllocationPhase) || \

flags.Trace.IsEnabled(Js::LargeMemoryAllocationPhase))) \

{ \

Output::Print(\_\_VA\_ARGS\_\_); \

}

#define PageAllocatorAllocationVerboseTrace(flags, ...) \

if (flags.Verbose && flags.Trace.IsEnabled(Js::PageAllocatorAllocPhase)) \

{ \

Output::Print(\_\_VA\_ARGS\_\_); \

}

#else

#define RecyclerVerboseTrace(...)

#define AllocationVerboseTrace(...)

#define LargeAllocationVerboseTrace(...)

#endif

#define RecyclerHeapNew(recycler,heapInfo,T,...) new (recycler, heapInfo) T(\_\_VA\_ARGS\_\_)

#define RecyclerHeapDelete(recycler,heapInfo,addr) (static\_cast<Recycler \*>(recycler)->HeapFree(heapInfo,addr))

typedef void (\_\_cdecl\* ExternalRootMarker)(void \*);

enum CollectionFlags

{

CollectHeuristic\_AllocSize = 0x00000001,

CollectHeuristic\_Time = 0x00000002,

CollectHeuristic\_TimeIfScriptActive = 0x00000004,

CollectHeuristic\_TimeIfInScript = 0x00000008,

CollectHeuristic\_Never = 0x00000080,

CollectHeuristic\_Mask = 0x000000FF,

CollectOverride\_FinishConcurrent = 0x00001000,

CollectOverride\_ExhaustiveCandidate = 0x00002000,

CollectOverride\_ForceInThread = 0x00004000,

CollectOverride\_AllowDispose = 0x00008000,

CollectOverride\_AllowReentrant = 0x00010000,

CollectOverride\_ForceFinish = 0x00020000,

CollectOverride\_Explicit = 0x00040000,

CollectOverride\_DisableIdleFinish = 0x00080000,

CollectOverride\_BackgroundFinishMark= 0x00100000,

CollectOverride\_FinishConcurrentTimeout = 0x00200000,

CollectOverride\_NoExhaustiveCollect = 0x00400000,

CollectOverride\_SkipStack = 0x01000000,

CollectMode\_Partial = 0x08000000,

CollectMode\_Concurrent = 0x10000000,

CollectMode\_Exhaustive = 0x20000000,

CollectMode\_DecommitNow = 0x40000000,

CollectMode\_CacheCleanup = 0x80000000,

CollectNowForceInThread = CollectOverride\_ForceInThread,

CollectNowForceInThreadExternal = CollectOverride\_ForceInThread | CollectOverride\_AllowDispose,

CollectNowForceInThreadExternalNoStack = CollectOverride\_ForceInThread | CollectOverride\_AllowDispose | CollectOverride\_SkipStack,

CollectNowDefault = CollectOverride\_FinishConcurrent,

CollectNowDefaultLSCleanup = CollectOverride\_FinishConcurrent | CollectOverride\_AllowDispose,

CollectNowDecommitNowExplicit = CollectNowDefault | CollectMode\_DecommitNow | CollectMode\_CacheCleanup | CollectOverride\_Explicit | CollectOverride\_AllowDispose,

CollectNowConcurrent = CollectOverride\_FinishConcurrent | CollectMode\_Concurrent,

CollectNowExhaustive = CollectOverride\_FinishConcurrent | CollectMode\_Exhaustive | CollectOverride\_AllowDispose,

CollectNowPartial = CollectOverride\_FinishConcurrent | CollectMode\_Partial,

CollectNowConcurrentPartial = CollectMode\_Concurrent | CollectNowPartial,

CollectOnAllocation = CollectHeuristic\_AllocSize | CollectHeuristic\_Time | CollectMode\_Concurrent | CollectMode\_Partial | CollectOverride\_FinishConcurrent | CollectOverride\_AllowReentrant | CollectOverride\_FinishConcurrentTimeout,

CollectOnTypedArrayAllocation = CollectHeuristic\_AllocSize | CollectHeuristic\_Time | CollectMode\_Concurrent | CollectMode\_Partial | CollectOverride\_FinishConcurrent | CollectOverride\_AllowReentrant | CollectOverride\_FinishConcurrentTimeout | CollectOverride\_AllowDispose,

CollectOnScriptIdle = CollectOverride\_FinishConcurrent | CollectMode\_Concurrent | CollectMode\_CacheCleanup | CollectOverride\_SkipStack,

CollectOnScriptExit = CollectHeuristic\_AllocSize | CollectOverride\_FinishConcurrent | CollectMode\_Concurrent | CollectMode\_CacheCleanup,

CollectExhaustiveCandidate = CollectHeuristic\_Never | CollectOverride\_ExhaustiveCandidate,

CollectOnScriptCloseNonPrimary = CollectNowConcurrent | CollectOverride\_ExhaustiveCandidate | CollectOverride\_AllowDispose,

CollectOnRecoverFromOutOfMemory = CollectOverride\_ForceInThread | CollectMode\_DecommitNow,

CollectOnSuspendCleanup = CollectNowConcurrent | CollectMode\_Exhaustive | CollectMode\_DecommitNow | CollectOverride\_DisableIdleFinish,

FinishConcurrentOnIdle = CollectMode\_Concurrent | CollectOverride\_DisableIdleFinish,

FinishConcurrentOnIdleAtRoot = CollectMode\_Concurrent | CollectOverride\_DisableIdleFinish | CollectOverride\_SkipStack,

FinishConcurrentOnExitScript = CollectMode\_Concurrent | CollectOverride\_DisableIdleFinish | CollectOverride\_BackgroundFinishMark,

FinishConcurrentOnEnterScript = CollectMode\_Concurrent | CollectOverride\_DisableIdleFinish | CollectOverride\_BackgroundFinishMark,

FinishConcurrentOnAllocation = CollectMode\_Concurrent | CollectOverride\_DisableIdleFinish | CollectOverride\_BackgroundFinishMark,

FinishDispose = CollectOverride\_AllowDispose,

FinishDisposeTimed = CollectOverride\_AllowDispose | CollectHeuristic\_TimeIfScriptActive,

ForceFinishCollection = CollectOverride\_ForceFinish | CollectOverride\_ForceInThread,

#ifdef RECYCLER\_STRESS

CollectStress = CollectNowForceInThread,

#ifdef PARTIAL\_GC\_ENABLED

CollectPartialStress = CollectMode\_Partial,

#endif

#ifdef CONCURRENT\_GC\_ENABLED

CollectBackgroundStress = CollectNowDefault,

CollectConcurrentStress = CollectNowConcurrent,

#ifdef PARTIAL\_GC\_ENABLED

CollectConcurrentPartialStress = CollectConcurrentStress | CollectPartialStress,

#endif

#endif

#endif

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

CollectNowFinalGC = CollectNowExhaustive | CollectOverride\_ForceInThread | CollectOverride\_SkipStack | CollectOverride\_Explicit | CollectOverride\_AllowDispose,

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

CollectNowExhaustiveSkipStack = CollectNowExhaustive | CollectOverride\_SkipStack, // Used by test

#endif

};

class RecyclerCollectionWrapper

{

public:

typedef BOOL (Recycler::\*CollectionFunction)(CollectionFlags flags);

virtual void PreCollectionCallBack(CollectionFlags flags) = 0;

virtual void PreSweepCallback() = 0;

virtual void PreRescanMarkCallback() = 0;

virtual size\_t RootMarkCallback(RecyclerScanMemoryCallback& scanMemoryCallback, BOOL \* stacksScannedByRuntime) = 0;

virtual void RescanMarkTimeoutCallback() = 0;

virtual void EndMarkCallback() = 0;

virtual void ConcurrentCallback() = 0;

virtual void WaitCollectionCallBack() = 0;

virtual void PostCollectionCallBack() = 0;

virtual BOOL ExecuteRecyclerCollectionFunction(Recycler \* recycler, CollectionFunction function, CollectionFlags flags) = 0;

virtual uint GetRandomNumber() = 0;

#ifdef FAULT\_INJECTION

virtual void DisposeScriptContextByFaultInjectionCallBack() = 0;

#endif

virtual void DisposeObjects(Recycler \* recycler) = 0;

#ifdef ENABLE\_PROJECTION

virtual void MarkExternalWeakReferencedObjects(bool inPartialCollect) = 0;

virtual void ResolveExternalWeakReferencedObjects() = 0;

#endif

#if DBG || defined(PROFILE\_EXEC)

virtual bool AsyncHostOperationStart(void \*) = 0;

virtual void AsyncHostOperationEnd(bool wasInAsync, void \*) = 0;

#endif

};

class DefaultRecyclerCollectionWrapper : public RecyclerCollectionWrapper

{

public:

virtual void PreCollectionCallBack(CollectionFlags flags) override {}

virtual void PreSweepCallback() override {}

virtual void PreRescanMarkCallback() override {}

virtual void RescanMarkTimeoutCallback() override {}

virtual void EndMarkCallback() override {}

virtual size\_t RootMarkCallback(RecyclerScanMemoryCallback& scanMemoryCallback, BOOL \* stacksScannedByRuntime) override { \*stacksScannedByRuntime = FALSE; return 0; }

virtual void ConcurrentCallback() override {}

virtual void WaitCollectionCallBack() override {}

virtual void PostCollectionCallBack() override {}

virtual BOOL ExecuteRecyclerCollectionFunction(Recycler \* recycler, CollectionFunction function, CollectionFlags flags) override;

virtual uint GetRandomNumber() override { return 0; }

#ifdef FAULT\_INJECTION

virtual void DisposeScriptContextByFaultInjectionCallBack() override {};

#endif

virtual void DisposeObjects(Recycler \* recycler) override;

#ifdef ENABLE\_PROJECTION

virtual void MarkExternalWeakReferencedObjects(bool inPartialCollect) override {};

virtual void ResolveExternalWeakReferencedObjects() override {};

#endif

#if DBG || defined(PROFILE\_EXEC)

virtual bool AsyncHostOperationStart(void \*) override { return false; };

virtual void AsyncHostOperationEnd(bool wasInAsync, void \*) override {};

#endif

static DefaultRecyclerCollectionWrapper Instance;

private:

static bool IsCollectionDisabled(Recycler \* recycler);

};

#ifdef RECYCLER\_STATS

struct RecyclerCollectionStats

{

size\_t startCollectAllocBytes;

#ifdef PARTIAL\_GC\_ENABLED

size\_t startCollectNewPageCount;

#endif

size\_t continueCollectAllocBytes;

size\_t finishCollectTryCount;

// Heuristic Stats

#ifdef PARTIAL\_GC\_ENABLED

size\_t rescanRootBytes;

size\_t estimatedPartialReuseBytes;

size\_t uncollectedNewPageCountPartialCollect;

size\_t partialCollectSmallHeapBlockReuseMinFreeBytes;

double collectEfficacy;

double collectCost;

#endif

// Mark stats

size\_t tryMarkCount; // # of pointer try mark (\* pointer size to get total number byte looked at)

size\_t tryMarkNullCount;

size\_t tryMarkUnalignedCount;

size\_t tryMarkNonRecyclerMemoryCount;

size\_t tryMarkInteriorCount;

size\_t tryMarkInteriorNullCount;

size\_t tryMarkInteriorNonRecyclerMemoryCount;

size\_t rootCount;

size\_t stackCount;

size\_t remarkCount;

size\_t scanCount; // non-leaf objects marked.

size\_t trackCount;

size\_t finalizeCount;

size\_t markThruNewObjCount;

size\_t markThruFalseNewObjCount;

struct MarkData

{

// Rescan stats

#if defined(PARTIAL\_GC\_ENABLED) || defined(CONCURRENT\_GC\_ENABLED)

size\_t rescanPageCount;

size\_t rescanObjectCount;

size\_t rescanObjectByteCount;

size\_t rescanLargePageCount;

size\_t rescanLargeObjectCount;

size\_t rescanLargeByteCount;

#endif

size\_t markCount; // total number of object marked

size\_t markBytes; // size of all objects marked.

} markData;

#ifdef CONCURRENT\_GC\_ENABLED

MarkData backgroundMarkData[RecyclerHeuristic::MaxBackgroundRepeatMarkCount];

size\_t trackedObjectCount;

#endif

#ifdef PARTIAL\_GC\_ENABLED

size\_t clientTrackedObjectCount;

#endif

// Sweep stats

size\_t heapBlockCount[HeapBlock::BlockTypeCount]; // number of heap blocks (processed during swept)

size\_t heapBlockFreeCount[HeapBlock::BlockTypeCount]; // number of heap blocks deleted

size\_t heapBlockConcurrentSweptCount[HeapBlock::SmallBlockTypeCount];

size\_t heapBlockSweptCount[HeapBlock::SmallBlockTypeCount]; // number of heap blocks swept

size\_t objectSweptCount; // objects freed (free list + whole page freed)

size\_t objectSweptBytes;

size\_t objectSweptFreeListCount; // objects freed (free list)

size\_t objectSweptFreeListBytes;

size\_t objectSweepScanCount; // number of objects walked for sweeping (exclude whole page freed)

size\_t finalizeSweepCount; // number of objects finalizer/dispose called

#ifdef PARTIAL\_GC\_ENABLED

size\_t smallNonLeafHeapBlockPartialReuseCount[HeapBlock::SmallBlockTypeCount];

size\_t smallNonLeafHeapBlockPartialReuseBytes[HeapBlock::SmallBlockTypeCount];

size\_t smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::SmallBlockTypeCount];

size\_t smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::SmallBlockTypeCount];

#endif

// Memory Stats

size\_t heapBlockFreeByteCount[HeapBlock::BlockTypeCount]; // The remaining usable free byte count

size\_t largeHeapBlockUsedByteCount; // Used byte count

size\_t largeHeapBlockTotalByteCount; // Total byte count

// Empty/zero heap block stats

uint numEmptySmallBlocks[HeapBlock::SmallBlockTypeCount];

uint numZeroedOutSmallBlocks;

};

#define RECYCLER\_STATS\_INC\_IF(cond, r, f) if (cond) { RECYCLER\_STATS\_INC(r, f); }

#define RECYCLER\_STATS\_INC(r, f) ++r->collectionStats.f

#define RECYCLER\_STATS\_INTERLOCKED\_INC(r, f) { InterlockedIncrement((LONG \*)&r->collectionStats.f); }

#define RECYCLER\_STATS\_DEC(r, f) --r->collectionStats.f

#define RECYCLER\_STATS\_ADD(r, f, v) r->collectionStats.f += (v)

#define RECYCLER\_STATS\_INTERLOCKED\_ADD(r, f, v) { InterlockedAdd((LONG \*)&r->collectionStats.f, (LONG)(v)); }

#define RECYCLER\_STATS\_SUB(r, f, v) r->collectionStats.f -= (v)

#define RECYCLER\_STATS\_SET(r, f, v) r->collectionStats.f = v

#else

#define RECYCLER\_STATS\_INC\_IF(cond, r, f)

#define RECYCLER\_STATS\_INC(r, f)

#define RECYCLER\_STATS\_INTERLOCKED\_INC(r, f)

#define RECYCLER\_STATS\_DEC(r, f)

#define RECYCLER\_STATS\_ADD(r, f, v)

#define RECYCLER\_STATS\_INTERLOCKED\_ADD(r, f, v)

#define RECYCLER\_STATS\_SUB(r, f, v)

#define RECYCLER\_STATS\_SET(r, f, v)

#endif

#ifdef RECYCLER\_TRACE

struct CollectionParam

{

CollectionFlags flags;

bool finishOnly;

bool repeat;

bool priorityBoostConcurentSweepOverride;

bool domCollect;

int timeDiff;

size\_t uncollectedAllocBytes;

size\_t uncollectedPinnedObjects;

#ifdef PARTIAL\_GC\_ENABLED

size\_t uncollectedNewPageCountPartialCollect;

size\_t uncollectedNewPageCount;

size\_t unusedPartialCollectFreeBytes;

bool inPartialCollectMode;

#endif

};

#endif

#include "RecyclerObjectGraphDumper.h"

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

// Macro to be used within the recycler

#define ForRecyclerPageAllocator(action) { \

this->recyclerPageAllocator.##action; \

this->recyclerLargeBlockPageAllocator.##action; \

this->recyclerWithBarrierPageAllocator.##action; \

this->threadPageAllocator->##action; \

}

// Macro that external objects referencing the recycler can use

#define ForEachRecyclerPageAllocatorIn(recycler, action) { \

recycler->GetRecyclerPageAllocator()->##action; \

recycler->GetRecyclerLargeBlockPageAllocator()->##action; \

recycler->GetRecyclerWithBarrierPageAllocator()->##action; \

recycler->GetRecyclerLeafPageAllocator()->##action; \

}

#else

// Macro to be used within the recycler

#define ForRecyclerPageAllocator(action) { \

this->recyclerPageAllocator.##action; \

this->recyclerLargeBlockPageAllocator.##action; \

this->threadPageAllocator->##action; \

}

// Macro that external objects referencing the recycler can use

#define ForEachRecyclerPageAllocatorIn(recycler, action) { \

recycler->GetRecyclerPageAllocator()->##action; \

recycler->GetRecyclerLargeBlockPageAllocator()->##action; \

recycler->GetRecyclerLeafPageAllocator()->##action; \

}

#endif

class RecyclerParallelThread

{

public:

typedef void (Recycler::\* WorkFunc)();

RecyclerParallelThread(Recycler \* recycler, WorkFunc workFunc) :

recycler(recycler),

workFunc(workFunc),

concurrentWorkReadyEvent(NULL),

concurrentWorkDoneEvent(NULL),

concurrentThread(NULL)

{

}

~RecyclerParallelThread()

{

Assert(concurrentThread == NULL);

Assert(concurrentWorkReadyEvent == NULL);

Assert(concurrentWorkDoneEvent == NULL);

}

bool StartConcurrent();

void WaitForConcurrent();

void Shutdown();

bool EnableConcurrent(bool synchronizeOnStartup);

private:

// Static entry point for thread creation

static unsigned int StaticThreadProc(LPVOID lpParameter);

// Static entry point for thread service usage

static void StaticBackgroundWorkCallback(void \* callbackData);

private:

WorkFunc workFunc;

Recycler \* recycler;

HANDLE concurrentWorkReadyEvent;// main thread uses this event to tell concurrent threads that the work is ready

HANDLE concurrentWorkDoneEvent;// concurrent threads use this event to tell main thread that the work allocated is done

HANDLE concurrentThread;

bool synchronizeOnStartup;

};

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

class AutoProtectPages

{

public:

AutoProtectPages(Recycler\* recycler, bool protectEnabled);

~AutoProtectPages();

void Unprotect();

private:

Recycler\* recycler;

bool isReadOnly;

};

#endif

class Recycler

{

friend class RecyclerScanMemoryCallback;

friend class RecyclerSweep;

friend class MarkContext;

friend class HeapBlock;

friend class HeapBlockMap32;

friend class RecyclerParallelThread;

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

friend class AutoProtectPages;

#endif

template <typename T> friend class RecyclerWeakReference;

template <typename T> friend class WeakReferenceHashTable;

template <typename TBlockType>

friend class SmallHeapBlockAllocator; // Needed for FindHeapBlock

#if defined(RECYCLER\_TRACE)

friend class JavascriptThreadService;

#endif

#ifdef HEAP\_ENUMERATION\_VALIDATION

friend class ActiveScriptProfilerHeapEnum;

#endif

friend class ScriptEngineBase; // This is for disabling GC for certain Host operations.

friend class CodeGenNumberThreadAllocator;

public:

static const uint ConcurrentThreadStackSize = 300000;

static const bool FakeZeroLengthArray = true;

#ifdef RECYCLER\_PAGE\_HEAP

// Keeping as constant in case we want to tweak the value here

// Set to 0 so that the tool can do the filtering instead of the runtime

static const int s\_numFramesToSkipForPageHeapAlloc = 0;

static const int s\_numFramesToSkipForPageHeapFree = 0;

static const int s\_numFramesToCaptureForPageHeap = 20;

#endif

uint Cookie;

class AutoEnterExternalStackSkippingGCMode

{

public:

AutoEnterExternalStackSkippingGCMode(Recycler\* recycler):

\_recycler(recycler)

{

// Setting this in a re-entrant mode is not allowed

Assert(!recycler->isExternalStackSkippingGC);

#if DBG

\_recycler->isExternalStackSkippingGC = true;

#endif

}

~AutoEnterExternalStackSkippingGCMode()

{

#if DBG

\_recycler->isExternalStackSkippingGC = false;

#endif

}

private:

Recycler\* \_recycler;

};

private:

class AutoSwitchCollectionStates

{

public:

AutoSwitchCollectionStates(Recycler\* recycler, CollectionState entryState, CollectionState exitState):

\_recycler(recycler),

\_exitState(exitState)

{

\_recycler->collectionState = entryState;

}

~AutoSwitchCollectionStates()

{

\_recycler->collectionState = \_exitState;

}

private:

Recycler\* \_recycler;

CollectionState \_exitState;

};

CollectionState collectionState;

IdleDecommitPageAllocator \* threadPageAllocator;

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

RecyclerPageAllocator recyclerWithBarrierPageAllocator;

#endif

RecyclerPageAllocator recyclerPageAllocator;

RecyclerPageAllocator recyclerLargeBlockPageAllocator;

JsUtil::ThreadService \*threadService;

HeapBlockMap heapBlockMap;

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

struct PinRecord

{

PinRecord() : refCount(0), stackBackTraces(nullptr) {}

PinRecord& operator=(uint newRefCount)

{

Assert(stackBackTraces == nullptr); Assert(newRefCount == 0); refCount = 0; return \*this;

}

PinRecord& operator++() { ++refCount; return \*this; }

PinRecord& operator--() { --refCount; return \*this; }

operator uint() const { return refCount; }

StackBackTraceNode \* stackBackTraces;

private:

uint refCount;

};

#else

typedef uint PinRecord;

#endif

typedef SimpleHashTable<void \*, PinRecord, HeapAllocator, DefaultComparer, true, PrimePolicy> PinnedObjectHashTable;

PinnedObjectHashTable pinnedObjectMap;

WeakReferenceHashTable<PrimePolicy> weakReferenceMap;

uint weakReferenceCleanupId;

void \* transientPinnedObject;

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

StackBackTrace \* transientPinnedObjectStackBackTrace;

#endif

struct GuestArenaAllocator : public ArenaAllocator

{

GuestArenaAllocator(\_\_in LPCWSTR name, PageAllocator \* pageAllocator, void (\*outOfMemoryFunc)())

: ArenaAllocator(name, pageAllocator, outOfMemoryFunc), pendingDelete(false)

{

}

bool pendingDelete;

};

DListBase<GuestArenaAllocator> guestArenaList;

DListBase<ArenaData\*> externalGuestArenaList; // guest arenas are scanned for roots

HeapInfo autoHeap;

#ifdef RECYCLER\_PAGE\_HEAP

\_\_inline bool IsPageHeapEnabled() const { return isPageHeapEnabled; }

\_\_inline bool ShouldCapturePageHeapAllocStack() const { return capturePageHeapAllocStack; }

bool isPageHeapEnabled;

bool capturePageHeapAllocStack;

bool capturePageHeapFreeStack;

#else

\_\_inline const bool IsPageHeapEnabled() const { return false; }

\_\_inline bool ShouldCapturePageHeapAllocStack() const { return false; }

#endif

#ifdef RECYCLER\_MARK\_TRACK

MarkMap\* markMap;

CriticalSection markMapCriticalSection;

void PrintMarkMap();

void ClearMarkMap();

#endif

// Number of pages to reserve for the primary mark stack

// This is the minimum number of pages to guarantee that a single heap block

// can be rescanned in the worst possible case where every object in a heap block

// in the smallest bucket needs to be rescanned

// These many pages being reserved guarantees that in OOM Rescan, we can make progress

// on every rescan iteration

// We add one because there is a small amount of the page reserved for page pool metadata

// so we need to allocate an additional page to be sure

// Currently, this works out to 2 pages on 32-bit and 5 pages on 64-bit

static const int PrimaryMarkStackReservedPageCount =

((SmallAllocationBlockAttributes::PageCount \* MarkContext::MarkCandidateSize) / SmallAllocationBlockAttributes::MinObjectSize) + 1;

MarkContext markContext;

// Contexts for parallel marking.

// We support up to 4 way parallelism, main context + 3 additional parallel contexts.

MarkContext parallelMarkContext1;

MarkContext parallelMarkContext2;

MarkContext parallelMarkContext3;

// Page pools for above markContexts

PagePool markPagePool;

PagePool parallelMarkPagePool1;

PagePool parallelMarkPagePool2;

PagePool parallelMarkPagePool3;

bool IsMarkStackEmpty();

bool HasPendingMarkObjects() const { return markContext.HasPendingMarkObjects() || parallelMarkContext1.HasPendingMarkObjects() || parallelMarkContext2.HasPendingMarkObjects() || parallelMarkContext3.HasPendingMarkObjects(); }

bool HasPendingTrackObjects() const { return markContext.HasPendingTrackObjects() || parallelMarkContext1.HasPendingTrackObjects() || parallelMarkContext2.HasPendingTrackObjects() || parallelMarkContext3.HasPendingTrackObjects(); }

RecyclerCollectionWrapper \* collectionWrapper;

bool inDispose;

#if DBG

uint collectionCount;

#endif

#if DBG || defined RECYCLER\_TRACE

bool inResolveExternalWeakReferences;

#endif

bool allowDispose;

bool inDisposeWrapper;

bool needOOMRescan;

bool hasDisposableObject;

DWORD tickCountNextDispose;

bool hasPendingTransferDisposedObjects;

bool inExhaustiveCollection;

bool hasExhaustiveCandidate;

bool inCacheCleanupCollection;

bool inDecommitNowCollection;

bool isScriptActive;

bool isInScript;

bool isShuttingDown;

bool scanPinnedObjectMap;

bool hasScannedInitialImplicitRoots;

bool hasPendingUnpinnedObject;

bool hasPendingDeleteGuestArena;

bool inEndMarkOnLowMemory;

bool decommitOnFinish;

bool enableScanInteriorPointers;

bool enableScanImplicitRoots;

bool disableCollectOnAllocationHeuristics;

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

bool disableCollection;

#endif

#ifdef PARTIAL\_GC\_ENABLED

bool enablePartialCollect;

bool inPartialCollectMode;

bool hasBackgroundFinishPartial;

bool partialConcurrentNextCollection;

#ifdef RECYCLER\_STRESS

bool forcePartialScanStack;

bool recyclerStress;

#ifdef CONCURRENT\_GC\_ENABLED

bool recyclerBackgroundStress;

bool recyclerConcurrentStress;

bool recyclerConcurrentRepeatStress;

#endif

#ifdef PARTIAL\_GC\_ENABLED

bool recyclerPartialStress;

#endif

#endif

#endif

#ifdef CONCURRENT\_GC\_ENABLED

bool skipStack;

#if DBG

bool isConcurrentGCOnIdle;

bool isFinishGCOnIdle;

bool isExternalStackSkippingGC;

#endif

bool queueTrackedObject;

bool hasPendingConcurrentFindRoot;

bool priorityBoost;

bool disableConcurrent;

bool enableConcurrentMark;

bool enableParallelMark;

bool enableConcurrentSweep;

uint maxParallelism; // Max # of total threads to run in parallel

byte backgroundRescanCount; // for ETW events and stats

byte backgroundFinishMarkCount;

size\_t backgroundRescanRootBytes;

HANDLE concurrentWorkReadyEvent; // main thread uses this event to tell concurrent threads that the work is ready

HANDLE concurrentWorkDoneEvent; // concurrent threads use this event to tell main thread that the work allocated is done

HANDLE concurrentThread;

HANDLE mainThreadHandle;

class SavedRegisterState

{

public:

#if \_M\_IX86

static const int NumRegistersToSave = 8;

#elif \_M\_ARM

static const int NumRegistersToSave = 13;

#elif \_M\_ARM64

static const int NumRegistersToSave = 13;

#elif \_M\_AMD64

static const int NumRegistersToSave = 16;

#endif

SavedRegisterState()

{

memset(registers, 0, sizeof(void\*) \* NumRegistersToSave);

}

void\*\* GetRegisters()

{

return registers;

}

void\* GetStackTop()

{

// By convention, our register-saving routine will always

// save the stack pointer as the first item in the array

return registers[0];

}

private:

void\* registers[NumRegistersToSave];

};

void \* stackBase;

SavedRegisterState savedThreadContext;

template <uint parallelId>

void ParallelWorkFunc();

RecyclerParallelThread parallelThread1;

RecyclerParallelThread parallelThread2;

Js::ConfigFlagsTable& recyclerFlagsTable;

#if DBG

// Variable indicating if the concurrent thread has exited or not

// If the concurrent thread hasn't started yet, this is set to true

// Once the concurrent thread starts, it sets this to false,

// and when the concurrent thread exits, it sets this to true.

bool concurrentThreadExited;

bool disableConcurentThreadExitedCheck;

bool isProcessingTrackedObjects;

bool hasIncompletedDoCollect;

// This is set to true when we begin a Rescan, and set to false when either:

// (1) We finish the final in-thread Rescan and are about to Mark

// (2) We do a conditional ResetWriteWatch and are about to Mark

// When this flag is true, we should not be modifying existing mark-related state,

// including markBits and rescanState.

bool isProcessingRescan;

#endif

uint tickCountStartConcurrent;

bool isAborting;

#endif

RecyclerSweep recyclerSweepInstance;

RecyclerSweep \* recyclerSweep;

static const uint tickDiffToNextCollect = 300;

#ifdef IDLE\_DECOMMIT\_ENABLED

HANDLE concurrentIdleDecommitEvent;

DWORD needIdleDecommitSignal;

#endif

#ifdef PARTIAL\_GC\_ENABLED

SListBase<void \*> clientTrackedObjectList;

ArenaAllocator clientTrackedObjectAllocator;

size\_t partialUncollectedAllocBytes;

// Dynamic Heuristics for partial GC

size\_t uncollectedNewPageCountPartialCollect;

#endif

uint tickCountNextCollection;

uint tickCountNextFinishCollection;

void (\*outOfMemoryFunc)();

#ifdef RECYCLER\_TEST\_SUPPORT

BOOL (\*checkFn)(char\* addr, size\_t size);

#endif

ExternalRootMarker externalRootMarker;

void \* externalRootMarkerContext;

#ifdef PROFILE\_EXEC

Js::Profiler \* profiler;

Js::Profiler \* backgroundProfiler;

PageAllocator backgroundProfilerPageAllocator;

DListBase<ArenaAllocator> backgroundProfilerArena;

#endif

#ifdef PROFILE\_MEM

RecyclerMemoryData \* memoryData;

#endif

ThreadContextId mainThreadId;

#ifdef ENABLE\_BASIC\_TELEMETRY

Js::GCTelemetry gcTel;

#endif

#if DBG

uint heapBlockCount;

bool disableThreadAccessCheck;

#endif

#if DBG || defined(RECYCLER\_STATS)

bool isForceSweeping;

#endif

RecyclerWatsonTelemetryBlock localTelemetryBlock;

RecyclerWatsonTelemetryBlock \* telemetryBlock;

#ifdef RECYCLER\_STATS

RecyclerCollectionStats collectionStats;

void PrintHeapBlockStats(wchar\_t const \* name, HeapBlock::HeapBlockType type);

void PrintHeapBlockMemoryStats(wchar\_t const \* name, HeapBlock::HeapBlockType type);

void PrintCollectStats();

void PrintHeuristicCollectionStats();

void PrintMarkCollectionStats();

void PrintBackgroundCollectionStats();

void PrintMemoryStats();

void PrintBackgroundCollectionStat(RecyclerCollectionStats::MarkData const& markData);

#endif

#ifdef RECYCLER\_TRACE

CollectionParam collectionParam;

#endif

#ifdef RECYCLER\_MEMORY\_VERIFY

uint verifyPad;

bool verifyEnabled;

#endif

#ifdef RECYCLER\_DUMP\_OBJECT\_GRAPH

friend class RecyclerObjectGraphDumper;

RecyclerObjectGraphDumper \* objectGraphDumper;

public:

bool dumpObjectOnceOnCollect;

#endif

public:

Recycler(AllocationPolicyManager \* policyManager, IdleDecommitPageAllocator \* pageAllocator, void(\*outOfMemoryFunc)(), Js::ConfigFlagsTable& flags);

~Recycler();

void Initialize(const bool forceInThread, JsUtil::ThreadService \*threadService, const bool deferThreadStartup = false

#ifdef RECYCLER\_PAGE\_HEAP

, PageHeapMode pageheapmode = PageHeapMode::PageHeapModeOff

, bool captureAllocCallStack = false

, bool captureFreeCallStack = false

#endif

);

Js::ConfigFlagsTable& GetRecyclerFlagsTable() const { return this->recyclerFlagsTable; }

void SetMemProtectMode();

bool IsMemProtectMode()

{

return this->enableScanImplicitRoots;

}

size\_t GetUsedBytes()

{

size\_t usedBytes = threadPageAllocator->usedBytes;

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

usedBytes += recyclerWithBarrierPageAllocator.usedBytes;

#endif

usedBytes += recyclerPageAllocator.usedBytes;

usedBytes += recyclerLargeBlockPageAllocator.usedBytes;

return usedBytes;

}

void LogMemProtectHeapSize(bool fromGC);

char\* Realloc(void\* buffer, size\_t existingBytes, size\_t requestedBytes, bool truncate = true);

void SetTelemetryBlock(RecyclerWatsonTelemetryBlock \* telemetryBlock) { this->telemetryBlock = telemetryBlock; }

void Prime();

void\* GetOwnerContext() { return (void\*) this->collectionWrapper; }

PageAllocator \* GetPageAllocator() { return threadPageAllocator; }

bool NeedOOMRescan() const

{

return this->needOOMRescan;

}

void SetNeedOOMRescan()

{

this->needOOMRescan = true;

}

void ClearNeedOOMRescan()

{

this->needOOMRescan = false;

markContext.GetPageAllocator()->ResetDisableAllocationOutOfMemory();

parallelMarkContext1.GetPageAllocator()->ResetDisableAllocationOutOfMemory();

parallelMarkContext2.GetPageAllocator()->ResetDisableAllocationOutOfMemory();

parallelMarkContext3.GetPageAllocator()->ResetDisableAllocationOutOfMemory();

}

BOOL RequestConcurrentWrapperCallback();

BOOL CollectionInProgress() const

{

return collectionState != CollectionStateNotCollecting;

}

BOOL IsExiting() const

{

return (collectionState == Collection\_Exit);

}

BOOL IsSweeping() const

{

return ((collectionState & Collection\_Sweep) == Collection\_Sweep);

}

#ifdef RECYCLER\_PAGE\_HEAP

\_\_inline bool ShouldCapturePageHeapFreeStack() const { return capturePageHeapFreeStack; }

#else

\_\_inline bool ShouldCapturePageHeapFreeStack() const { return false; }

#endif

void SetIsThreadBound();

void SetIsScriptActive(bool isScriptActive)

{

Assert(this->isInScript);

Assert(this->isScriptActive != isScriptActive);

this->isScriptActive = isScriptActive;

if (isScriptActive)

{

this->tickCountNextDispose = ::GetTickCount() + RecyclerHeuristic::TickCountFinishCollection;

}

}

void SetIsInScript(bool isInScript)

{

Assert(this->isInScript != isInScript);

this->isInScript = isInScript;

}

bool ShouldIdleCollectOnExit();

void ScheduleNextCollection();

IdleDecommitPageAllocator \* GetRecyclerLeafPageAllocator()

{

return this->threadPageAllocator;

}

IdleDecommitPageAllocator \* GetRecyclerPageAllocator()

{

return &this->recyclerPageAllocator;

}

IdleDecommitPageAllocator \* GetRecyclerLargeBlockPageAllocator()

{

return &this->recyclerLargeBlockPageAllocator;

}

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

IdleDecommitPageAllocator \* GetRecyclerWithBarrierPageAllocator()

{

return &this->recyclerWithBarrierPageAllocator;

}

#endif

BOOL IsShuttingDown() const { return this->isShuttingDown; }

#ifdef CONCURRENT\_GC\_ENABLED

#if DBG

BOOL IsConcurrentMarkEnabled() const { return enableConcurrentMark; }

BOOL IsConcurrentSweepEnabled() const { return enableConcurrentSweep; }

#endif

template <CollectionFlags flags>

BOOL FinishConcurrent();

void ShutdownThread();

bool EnableConcurrent(JsUtil::ThreadService \*threadService, bool startAllThreads);

void DisableConcurrent();

void StartQueueTrackedObject();

bool DoQueueTrackedObject() const;

void PrepareSweep();

#endif

template <CollectionFlags flags>

void SetupPostCollectionFlags();

void EnsureNotCollecting();

bool QueueTrackedObject(FinalizableObject \* trackableObject);

// FindRoots

void TryMarkNonInterior(void\* candidate, void\* parentReference = nullptr);

void TryMarkInterior(void \*candidate, void\* parentReference = nullptr);

bool InCacheCleanupCollection() { return inCacheCleanupCollection; }

void ClearCacheCleanupCollection() { Assert(inCacheCleanupCollection); inCacheCleanupCollection = false; }

// Finalizer support

void SetExternalRootMarker(ExternalRootMarker fn, void \* context)

{

externalRootMarker = fn;

externalRootMarkerContext = context;

}

HeapInfo\* CreateHeap();

void DestroyHeap(HeapInfo\* heapInfo);

ArenaAllocator \* CreateGuestArena(wchar\_t const \* name, void (\*outOfMemoryFunc)());

void DeleteGuestArena(ArenaAllocator \* arenaAllocator);

ArenaData \*\* RegisterExternalGuestArena(ArenaData\* guestArena)

{

return externalGuestArenaList.PrependNode(&NoThrowHeapAllocator::Instance, guestArena);

}

void UnregisterExternalGuestArena(ArenaData\* guestArena)

{

externalGuestArenaList.Remove(&NoThrowHeapAllocator::Instance, guestArena);

}

void UnregisterExternalGuestArena(ArenaData\*\* guestArena)

{

externalGuestArenaList.RemoveElement(&NoThrowHeapAllocator::Instance, guestArena);

}

#ifdef RECYCLER\_TEST\_SUPPORT

void SetCheckFn(BOOL(\*checkFn)(char\* addr, size\_t size));

#endif

void SetCollectionWrapper(RecyclerCollectionWrapper \* wrapper)

{

this->collectionWrapper = wrapper;

#if LARGEHEAPBLOCK\_ENCODING

this->Cookie = wrapper->GetRandomNumber();

#else

this->Cookie = 0;

#endif

}

static size\_t GetAlignedSize(size\_t size) { return HeapInfo::GetAlignedSize(size); }

HeapInfo\* GetAutoHeap() { return &autoHeap; }

template <CollectionFlags flags>

BOOL CollectNow();

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void DisplayMemStats();

#endif

void AddExternalMemoryUsage(size\_t size);

bool NeedDispose()

{

return this->hasDisposableObject;

}

template <CollectionFlags flags>

bool FinishDisposeObjectsNow();

BOOL ReportExternalMemoryAllocation(size\_t size);

void ReportExternalMemoryFailure(size\_t size);

void ReportExternalMemoryFree(size\_t size);

#ifdef TRACE\_OBJECT\_LIFETIME

#define DEFINE\_RECYCLER\_ALLOC\_TRACE(AllocFunc, AllocWithAttributesFunc, attributes) \

\_\_inline char\* AllocFunc##Trace(size\_t size) \

{ \

return AllocWithAttributesFunc<(ObjectInfoBits)(attributes | TraceBit), /\* nothrow = \*/ false>(size); \

}

#else

#define DEFINE\_RECYCLER\_ALLOC\_TRACE(AllocFunc, AllocWithAttributeFunc, attributes)

#endif

#define DEFINE\_RECYCLER\_ALLOC\_BASE(AllocFunc, AllocWithAttributesFunc, attributes) \

\_\_inline char \* AllocFunc(size\_t size) \

{ \

return AllocWithAttributesFunc<attributes, /\* nothrow = \*/ false>(size); \

} \

\_\_forceinline char \* AllocFunc##Inlined(size\_t size) \

{ \

return AllocWithAttributesFunc##Inlined<attributes, /\* nothrow = \*/ false>(size); \

} \

DEFINE\_RECYCLER\_ALLOC\_TRACE(AllocFunc, AllocWithAttributesFunc, attributes);

#define DEFINE\_RECYCLER\_NOTHROW\_ALLOC\_BASE(AllocFunc, AllocWithAttributesFunc, attributes) \

\_\_inline char \* NoThrow##AllocFunc(size\_t size) \

{ \

return AllocWithAttributesFunc<attributes, /\* nothrow = \*/ true>(size); \

} \

\_\_inline char \* NoThrow##AllocFunc##Inlined(size\_t size) \

{ \

return AllocWithAttributesFunc##Inlined<attributes, /\* nothrow = \*/ true>(size); \

} \

DEFINE\_RECYCLER\_ALLOC\_TRACE(AllocFunc, AllocWithAttributesFunc, attributes);

#define DEFINE\_RECYCLER\_ALLOC(AllocFunc, attributes) DEFINE\_RECYCLER\_ALLOC\_BASE(AllocFunc, AllocWithAttributes, attributes)

#define DEFINE\_RECYCLER\_ALLOC\_ZERO(AllocFunc, attributes) DEFINE\_RECYCLER\_ALLOC\_BASE(AllocFunc, AllocZeroWithAttributes, attributes)

#define DEFINE\_RECYCLER\_NOTHROW\_ALLOC(AllocFunc, attributes) DEFINE\_RECYCLER\_NOTHROW\_ALLOC\_BASE(AllocFunc, AllocWithAttributes, attributes)

#define DEFINE\_RECYCLER\_NOTHROW\_ALLOC\_ZERO(AllocFunc, attributes) DEFINE\_RECYCLER\_NOTHROW\_ALLOC\_BASE(AllocFunc, AllocZeroWithAttributes, attributes)

DEFINE\_RECYCLER\_ALLOC(Alloc, NoBit);

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC

DEFINE\_RECYCLER\_ALLOC(AllocWithBarrier, WithBarrierBit);

DEFINE\_RECYCLER\_ALLOC(AllocFinalizedWithBarrier, FinalizableWithBarrierObjectBits);

#endif

DEFINE\_RECYCLER\_ALLOC(AllocFinalized, FinalizableObjectBits);

DEFINE\_RECYCLER\_ALLOC(AllocFinalizedClientTracked, ClientFinalizableObjectBits);

// All trackable object are client trackable

DEFINE\_RECYCLER\_ALLOC(AllocTracked, ClientTrackableObjectBits);

DEFINE\_RECYCLER\_ALLOC(AllocLeaf, LeafBit);

DEFINE\_RECYCLER\_ALLOC(AllocFinalizedLeaf, FinalizableLeafBits);

DEFINE\_RECYCLER\_ALLOC(AllocTrackedLeaf, ClientTrackableLeafBits);

DEFINE\_RECYCLER\_ALLOC\_ZERO(AllocZero, NoBit);

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC

DEFINE\_RECYCLER\_ALLOC\_ZERO(AllocZeroWithBarrier, WithBarrierBit);

#endif

DEFINE\_RECYCLER\_ALLOC\_ZERO(AllocLeafZero, LeafBit);

DEFINE\_RECYCLER\_ALLOC\_ZERO(AllocZeroTrackedLeaf, ClientTrackableLeafBits);

DEFINE\_RECYCLER\_NOTHROW\_ALLOC\_ZERO(AllocImplicitRootLeaf, ImplicitRootLeafBits);

DEFINE\_RECYCLER\_NOTHROW\_ALLOC\_ZERO(AllocImplicitRoot, ImplicitRootBit);

template <ObjectInfoBits enumClass>

char \* AllocEnumClass(size\_t size)

{

Assert((enumClass & EnumClassMask) != 0);

Assert((enumClass & ~EnumClassMask) == 0);

return AllocWithAttributes<(ObjectInfoBits)(enumClass), /\* nothrow = \*/ false>(size);

}

template <ObjectInfoBits infoBits>

char \* AllocWithInfoBits(size\_t size)

{

return AllocWithAttributes<infoBits, /\* nothrow = \*/ false>(size);

}

template<typename T>

RecyclerWeakReference<T>\* CreateWeakReferenceHandle(T\* pStrongReference);

uint GetWeakReferenceCleanupId() const { return weakReferenceCleanupId; }

template<typename T>

bool FindOrCreateWeakReferenceHandle(T\* pStrongReference, RecyclerWeakReference<T> \*\*ppWeakRef);

template<typename T>

bool TryGetWeakReferenceHandle(T\* pStrongReference, RecyclerWeakReference<T> \*\*weakReference);

template <ObjectInfoBits attributes>

char\* GetAddressOfAllocator(size\_t sizeCat)

{

Assert(HeapInfo::IsAlignedSmallObjectSize(sizeCat));

return (char\*)this->autoHeap.GetBucket<attributes>(sizeCat).GetAllocator();

}

template <ObjectInfoBits attributes>

uint32 GetEndAddressOffset(size\_t sizeCat)

{

Assert(HeapInfo::IsAlignedSmallObjectSize(sizeCat));

return this->autoHeap.GetBucket<attributes>(sizeCat).GetAllocator()->GetEndAddressOffset();

}

template <ObjectInfoBits attributes>

uint32 GetFreeObjectListOffset(size\_t sizeCat)

{

Assert(HeapInfo::IsAlignedSmallObjectSize(sizeCat));

return this->autoHeap.GetBucket<attributes>(sizeCat).GetAllocator()->GetFreeObjectListOffset();

}

void GetNormalHeapBlockAllocatorInfoForNativeAllocation(size\_t sizeCat, void\*& allocatorAddress, uint32& endAddressOffset, uint32& freeListOffset);

bool AllowNativeCodeBumpAllocation();

static void TrackNativeAllocatedMemoryBlock(Recycler \* recycler, void \* memBlock, size\_t sizeCat);

void Free(void\* buffer, size\_t size)

{

Assert(false);

}

bool ExplicitFreeLeaf(void\* buffer, size\_t size);

bool ExplicitFreeNonLeaf(void\* buffer, size\_t size);

template <ObjectInfoBits attributes>

bool ExplicitFreeInternalWrapper(void\* buffer, size\_t allocSize);

template <ObjectInfoBits attributes, typename TBlockAttributes>

bool ExplicitFreeInternal(void\* buffer, size\_t size, size\_t sizeCat);

size\_t GetAllocSize(size\_t size);

template <typename TBlockAttributes>

void SetExplicitFreeBitOnSmallBlock(HeapBlock\* heapBlock, size\_t sizeCat, void\* buffer, ObjectInfoBits attributes);

char\* HeapAllocR(HeapInfo\* eHeap, size\_t size)

{

return RealAlloc<LeafBit, /\* nothrow = \*/ false>(eHeap, size);

}

void HeapFree(HeapInfo\* eHeap,void\* candidate);

void EnumerateObjects(ObjectInfoBits infoBits, void (\*CallBackFunction)(void \* address, size\_t size));

void RootAddRef(void\* obj, uint \*count = nullptr);

void RootRelease(void\* obj, uint \*count = nullptr);

template <ObjectInfoBits attributes, bool nothrow>

\_\_inline char\* RealAlloc(HeapInfo\* heap, size\_t size);

template <ObjectInfoBits attributes, bool isSmallAlloc, bool nothrow>

\_\_inline char\* RealAllocFromBucket(HeapInfo\* heap, size\_t size);

void EnterIdleDecommit();

void LeaveIdleDecommit();

void DisposeObjects();

BOOL IsValidObject(void\* candidate, size\_t minimumSize = 0);

#if DBG

void SetDisableThreadAccessCheck();

void SetDisableConcurentThreadExitedCheck();

void CheckAllocExternalMark() const;

BOOL IsFreeObject(void \* candidate);

BOOL IsReentrantState() const;

#endif

#if DBG\_DUMP

void PrintMarkStack();

#endif

#ifdef PROFILE\_EXEC

Js::Profiler \* GetProfiler() const { return this->profiler; }

ArenaAllocator \* AddBackgroundProfilerArena();

void ReleaseBackgroundProfilerArena(ArenaAllocator \* arena);

void SetProfiler(Js::Profiler \* profiler, Js::Profiler \* backgroundProfiler);

#endif

#ifdef RECYCLER\_MEMORY\_VERIFY

BOOL VerifyEnabled() const { return verifyEnabled; }

void Verify(Js::Phase phase);

static void VerifyCheck(BOOL cond, wchar\_t const \* msg, void \* address, void \* corruptedAddress);

static void VerifyCheckFill(void \* address, size\_t size);

void FillCheckPad(void \* address, size\_t size, size\_t alignedAllocSize, bool objectAlreadyInitialized);

void FillCheckPad(void \* address, size\_t size, size\_t alignedAllocSize)

{

FillCheckPad(address, size, alignedAllocSize, false);

}

void VerifyCheckPad(void \* address, size\_t size);

void VerifyCheckPadExplicitFreeList(void \* address, size\_t size);

static const byte VerifyMemFill = 0xCA;

#endif

#ifdef RECYCLER\_ZERO\_MEM\_CHECK

void VerifyZeroFill(void \* address, size\_t size);

#endif

#ifdef RECYCLER\_DUMP\_OBJECT\_GRAPH

bool DumpObjectGraph(RecyclerObjectGraphDumper::Param \* param = nullptr);

void DumpObjectDescription(void \*object);

#endif

#ifdef LEAK\_REPORT

void ReportLeaks();

void ReportLeaksOnProcessDetach();

#endif

#ifdef CHECK\_MEMORY\_LEAK

void CheckLeaks(wchar\_t const \* header);

void CheckLeaksOnProcessDetach(wchar\_t const \* header);

#endif

#ifdef RECYCLER\_TRACE

void SetDomCollect(bool isDomCollect)

{

collectionParam.domCollect = isDomCollect;

}

void CaptureCollectionParam(CollectionFlags flags, bool repeat = false);

#endif

#ifdef ENABLE\_BASIC\_TELEMETRY

Js::GCPauseStats GetGCPauseStats()

{

return gcTel.GetGCPauseStats(); // returns the maxGCpause time in ms

}

void ResetGCPauseStats()

{

gcTel.Reset();

}

void SetIsScriptSiteCloseGC(bool val)

{

gcTel.SetIsScriptSiteCloseGC(val);

}

#endif

private:

// RecyclerRootPtr has implicit conversion to pointers, prevent it to be

// passed to RootAddRef/RootRelease directly

template <typename T>

void RootAddRef(RecyclerRootPtr<T>& ptr, uint \*count = nullptr);

template <typename T>

void RootRelease(RecyclerRootPtr<T>& ptr, uint \*count = nullptr);

template <CollectionFlags flags>

BOOL CollectInternal();

template <CollectionFlags flags>

BOOL Collect();

template <CollectionFlags flags>

BOOL CollectWithHeuristic();

template <CollectionFlags flags>

BOOL CollectWithExhaustiveCandidate();

template <CollectionFlags flags>

BOOL GetPartialFlag();

bool NeedExhaustiveRepeatCollect() const;

#if DBG

bool ExpectStackSkip() const;

#endif

static size\_t const InvalidScanRootBytes = (size\_t)-1;

// Small Allocator

template <typename SmallHeapBlockAllocatorType>

void AddSmallAllocator(SmallHeapBlockAllocatorType \* allocator, size\_t sizeCat);

template <typename SmallHeapBlockAllocatorType>

void RemoveSmallAllocator(SmallHeapBlockAllocatorType \* allocator, size\_t sizeCat);

template <ObjectInfoBits attributes, typename SmallHeapBlockAllocatorType>

char \* SmallAllocatorAlloc(SmallHeapBlockAllocatorType \* allocator, size\_t sizeCat);

// Allocation

template <ObjectInfoBits attributes, bool nothrow>

\_\_inline char \* AllocWithAttributesInlined(size\_t size);

template <ObjectInfoBits attributes, bool nothrow>

char \* AllocWithAttributes(size\_t size)

{

return AllocWithAttributesInlined<attributes, nothrow>(size);

}

template <ObjectInfoBits attributes, bool nothrow>

\_\_inline char\* AllocZeroWithAttributesInlined(size\_t size);

template <ObjectInfoBits attributes, bool nothrow>

char\* AllocZeroWithAttributes(size\_t size)

{

return AllocZeroWithAttributesInlined<attributes, nothrow>(size);

}

char\* AllocWeakReferenceEntry(size\_t size)

{

return AllocWithAttributes<WeakReferenceEntryBits, /\* nothrow = \*/ false>(size);

}

bool NeedDisposeTimed()

{

DWORD ticks = ::GetTickCount();

return (ticks > tickCountNextDispose && this->hasDisposableObject);

}

char\* TryLargeAlloc(HeapInfo\* heap, size\_t size, ObjectInfoBits attributes, bool nothrow);

template <bool nothrow>

char\* LargeAlloc(HeapInfo\* heap, size\_t size, ObjectInfoBits attributes);

void OutOfMemory();

// Collection

BOOL DoCollect(CollectionFlags flags);

BOOL DoCollectWrapped(CollectionFlags flags);

BOOL CollectOnAllocatorThread();

#if DBG

void ResetThreadId();

#endif

template <bool background>

size\_t ScanPinnedObjects();

size\_t ScanStack();

size\_t ScanArena(ArenaData \* alloc, bool background);

void ScanImplicitRoots();

void ScanInitialImplicitRoots();

void ScanNewImplicitRoots();

size\_t FindRoots();

size\_t TryMarkArenaMemoryBlockList(ArenaMemoryBlock \* memoryBlocks);

size\_t TryMarkBigBlockList(BigBlock \* memoryBlocks);

size\_t TryMarkBigBlockListWithWriteWatch(BigBlock \* memoryBlocks);

// Mark

void ResetMarks(ResetMarkFlags flags);

void Mark();

bool EndMark();

bool EndMarkCheckOOMRescan();

void EndMarkOnLowMemory();

void DoParallelMark();

void DoBackgroundParallelMark();

size\_t RootMark(CollectionState markState);

void ProcessMark(bool background);

void ProcessParallelMark(bool background, MarkContext \* markContext);

template <bool parallel, bool interior>

void ProcessMarkContext(MarkContext \* markContext);

public:

bool IsObjectMarked(void\* candidate) { return this->heapBlockMap.IsMarked(candidate); }

#ifdef RECYCLER\_STRESS

bool StressCollectNow();

#endif

private:

HeapBlock\* FindHeapBlock(void \* candidate);

struct FindBlockCache

{

FindBlockCache():

heapBlock(nullptr),

candidate(nullptr)

{

}

HeapBlock\* heapBlock;

void\* candidate;

} blockCache;

\_\_inline void ScanObjectInline(void \*\* obj, size\_t byteCount);

\_\_inline void ScanObjectInlineInterior(void \*\* obj, size\_t byteCount);

\_\_inline void ScanMemoryInline(void \*\* obj, size\_t byteCount);

void ScanMemory(void \*\* obj, size\_t byteCount) { if (byteCount != 0) { ScanMemoryInline(obj, byteCount); } }

bool AddMark(void \* candidate, size\_t byteCount);

// Sweep

bool Sweep(size\_t rescanRootBytes = (size\_t)-1, bool concurrent = false, bool adjustPartialHeuristics = false);

void SweepWeakReference();

void SweepHeap(bool concurrent, RecyclerSweep& recyclerSweep);

void FinishSweep(RecyclerSweep& recyclerSweep);

bool FinishDisposeObjects();

template <CollectionFlags flags>

bool FinishDisposeObjectsWrapped();

// end collection

void FinishCollection();

void FinishCollection(bool needConcurrentSweep);

void EndCollection();

void ResetCollectionState();

void ResetMarkCollectionState();

void ResetHeuristicCounters();

void ResetPartialHeuristicCounters();

BOOL IsMarkState() const;

BOOL IsFindRootsState() const;

BOOL IsInThreadFindRootsState() const;

template <Js::Phase phase>

void CollectionBegin();

template <Js::Phase phase>

void CollectionEnd();

#ifdef PARTIAL\_GC\_ENABLED

void ProcessClientTrackedObjects();

bool PartialCollect(bool concurrent);

void FinishPartialCollect(RecyclerSweep \* recyclerSweep = nullptr);

void ClearPartialCollect();

void BackgroundFinishPartialCollect(RecyclerSweep \* recyclerSweep);

#endif

#if defined(PARTIAL\_GC\_ENABLED) || defined(CONCURRENT\_GC\_ENABLED)

size\_t RescanMark(DWORD waitTime);

size\_t FinishMark(DWORD waitTime);

size\_t FinishMarkRescan(bool background);

void ProcessTrackedObjects();

#endif

#ifdef CONCURRENT\_GC\_ENABLED

// Concurrent GC

BOOL IsConcurrentEnabled() const { return this->enableConcurrentMark || this->enableParallelMark || this->enableConcurrentSweep; }

BOOL IsConcurrentMarkState() const;

BOOL IsConcurrentMarkExecutingState() const;

BOOL IsConcurrentResetMarksState() const;

BOOL IsConcurrentFindRootState() const;

BOOL IsConcurrentExecutingState() const;

BOOL IsConcurrentSweepExecutingState() const;

BOOL IsConcurrentState() const;

BOOL InConcurrentSweep()

{

return ((collectionState & Collection\_ConcurrentSweep) == Collection\_ConcurrentSweep);

}

#if DBG

BOOL IsConcurrentFinishedState() const;

#endif // DBG

bool InitializeConcurrent(JsUtil::ThreadService\* threadService);

bool AbortConcurrent(bool restoreState);

void FinalizeConcurrent(bool restoreState);

static unsigned int StaticThreadProc(LPVOID lpParameter);

static int ExceptFilter(LPEXCEPTION\_POINTERS pEP);

DWORD ThreadProc();

void DoBackgroundWork(bool forceForeground = false);

static void StaticBackgroundWorkCallback(void \* callbackData);

BOOL CollectOnConcurrentThread();

bool StartConcurrent(CollectionState const state);

BOOL StartBackgroundMarkCollect();

BOOL StartSynchronousBackgroundMark();

BOOL StartAsynchronousBackgroundMark();

BOOL StartBackgroundMark(bool foregroundResetMark, bool foregroundFindRoots);

BOOL StartConcurrentSweepCollect();

template <CollectionFlags flags>

BOOL TryFinishConcurrentCollect();

BOOL WaitForConcurrentThread(DWORD waitTime);

void FlushBackgroundPages();

BOOL FinishConcurrentCollect(CollectionFlags flags);

BOOL FinishConcurrentCollectWrapped(CollectionFlags flags);

void BackgroundMark();

void BackgroundResetMarks();

void PrepareBackgroundFindRoots();

void RevertPrepareBackgroundFindRoots();

size\_t BackgroundFindRoots();

size\_t BackgroundScanStack();

size\_t BackgroundRepeatMark();

size\_t BackgroundRescan(RescanFlags rescanFlags);

void BackgroundResetWriteWatchAll();

size\_t BackgroundFinishMark();

char\* GetScriptThreadStackTop();

void SweepPendingObjects(RecyclerSweep& recyclerSweep);

void ConcurrentTransferSweptObjects(RecyclerSweep& recyclerSweep);

#ifdef PARTIAL\_GC\_ENABLED

void ConcurrentPartialTransferSweptObjects(RecyclerSweep& recyclerSweep);

#endif // PARTIAL\_GC\_ENABLED

#endif // CONCURRENT\_GC\_ENABLED

bool ForceSweepObject();

void NotifyFree(\_\_in char \* address, size\_t size);

template <bool pageheap, typename T>

void NotifyFree(T \* heapBlock);

void CleanupPendingUnroot();

#ifdef ENABLE\_JS\_ETW

ULONG EventWriteFreeMemoryBlock(HeapBlock\* heapBlock);

void FlushFreeRecord();

void AppendFreeMemoryETWRecord(\_\_in char \*address, size\_t size);

static const uint BulkFreeMemoryCount = 400;

uint bulkFreeMemoryWrittenCount;

struct ETWFreeRecord {

char\* memoryAddress;

uint32 objectSize;

};

ETWFreeRecord etwFreeRecords[BulkFreeMemoryCount];

#endif

template <ObjectInfoBits attributes>

bool IntegrateBlock(char \* blockAddress, PageSegment \* segment, size\_t allocSize, size\_t objectSize);

template <class TBlockAttributes> friend class SmallHeapBlockT;

template <class TBlockAttributes> friend class SmallNormalHeapBlockT;

template <class TBlockAttributes> friend class SmallLeafHeapBlockT;

template <class TBlockAttributes> friend class SmallFinalizableHeapBlockT;

friend class LargeHeapBlock;

friend class HeapInfo;

friend class LargeHeapBucket;

template <typename TBlockType>

friend class HeapBucketT;

template <typename TBlockType>

friend class SmallNormalHeapBucketBase;

template <typename T, ObjectInfoBits attributes = LeafBit>

friend class RecyclerFastAllocator;

#ifdef RECYCLER\_TRACE

void PrintCollectTrace(Js::Phase phase, bool finish = false, bool noConcurrentWork = false);

#endif

#ifdef RECYCLER\_VERIFY\_MARK

void VerifyMark();

void VerifyMarkRoots();

void VerifyMarkStack();

void VerifyMarkArena(ArenaData \* arena);

void VerifyMarkBigBlockList(BigBlock \* memoryBlocks);

void VerifyMarkArenaMemoryBlockList(ArenaMemoryBlock \* memoryBlocks);

void VerifyMark(void \* address);

#endif

#if DBG\_DUMP

bool forceTraceMark;

#endif

bool isHeapEnumInProgress;

#if DBG

bool allowAllocationDuringHeapEnum;

bool allowAllocationDuringRenentrance;

#ifdef ENABLE\_PROJECTION

bool isInRefCountTrackingForProjection;

#endif

#endif

// There are two scenarios we allow limited allocation but disallow GC during those allocations:

// in heapenum when we allocate PropertyRecord, and

// in projection ExternalMark allowing allocating VarToDispEx. This is the common flag

// while we have debug only flag for each of the two scenarios.

bool isCollectionDisabled;

#ifdef TRACK\_ALLOC

public:

Recycler \* TrackAllocInfo(TrackAllocData const& data);

void ClearTrackAllocInfo(TrackAllocData\* data = NULL);

#ifdef PROFILE\_RECYCLER\_ALLOC

void PrintAllocStats();

private:

static bool DoProfileAllocTracker();

void InitializeProfileAllocTracker();

void TrackUnallocated(\_\_in char\* address, \_\_in char \*endAddress, size\_t sizeCat);

void TrackAllocCore(void \* object, size\_t size, const TrackAllocData& trackAllocData, bool traceLifetime = false);

void\* TrackAlloc(void \* object, size\_t size, const TrackAllocData& trackAllocData, bool traceLifetime = false);

void TrackIntegrate(\_\_in\_ecount(blockSize) char \* blockAddress, size\_t blockSize, size\_t allocSize, size\_t objectSize, const TrackAllocData& trackAllocData);

BOOL TrackFree(const char\* address, size\_t size);

void TrackAllocWeakRef(RecyclerWeakReferenceBase \* weakRef);

void TrackFreeWeakRef(RecyclerWeakReferenceBase \* weakRef);

struct TrackerData

{

TrackerData(type\_info const \* typeinfo, bool isArray) : typeinfo(typeinfo), isArray(isArray),

ItemSize(0), ItemCount(0), AllocCount(0), ReqSize(0), AllocSize(0), FreeCount(0), FreeSize(0), TraceLifetime(false)

#ifdef PERF\_COUNTERS

, counter(PerfCounter::RecyclerTrackerCounterSet::GetPerfCounter(typeinfo, isArray))

, sizeCounter(PerfCounter::RecyclerTrackerCounterSet::GetPerfSizeCounter(typeinfo, isArray))

#endif

{

}

type\_info const \* typeinfo;

bool isArray;

#ifdef TRACE\_OBJECT\_LIFETIME

bool TraceLifetime;

#endif

size\_t ItemSize;

size\_t ItemCount;

int AllocCount;

int64 ReqSize;

int64 AllocSize;

int FreeCount;

int64 FreeSize;

#ifdef PERF\_COUNTERS

PerfCounter::Counter& counter;

PerfCounter::Counter& sizeCounter;

#endif

static TrackerData EmptyData;

static TrackerData ExplicitFreeListObjectData;

};

TrackerData \* GetTrackerData(void \* address);

void SetTrackerData(void \* address, TrackerData \* data);

struct TrackerItem

{

TrackerItem(type\_info const \* typeinfo) : instanceData(typeinfo, false), arrayData(typeinfo, true)

#ifdef PERF\_COUNTERS

, weakRefCounter(PerfCounter::RecyclerTrackerCounterSet::GetWeakRefPerfCounter(typeinfo))

#endif

{}

TrackerData instanceData;

TrackerData arrayData;

#ifdef PERF\_COUNTERS

PerfCounter::Counter& weakRefCounter;

#endif

};

typedef JsUtil::BaseDictionary<type\_info const \*, TrackerItem \*, NoCheckHeapAllocator, PrimeSizePolicy, DefaultComparer, JsUtil::SimpleDictionaryEntry, JsUtil::NoResizeLock> TypeInfotoTrackerItemMap;

typedef JsUtil::BaseDictionary<void \*, TrackerData \*, NoCheckHeapAllocator, PrimeSizePolicy, RecyclerPointerComparer, JsUtil::SimpleDictionaryEntry, JsUtil::NoResizeLock> PointerToTrackerDataMap;

TypeInfotoTrackerItemMap \* trackerDictionary;

CRITICAL\_SECTION trackerCriticalSection;

#endif

TrackAllocData nextAllocData;

#endif

public:

// Enumeration

class AutoSetupRecyclerForNonCollectingMark

{

private:

Recycler& m\_recycler;

bool m\_setupDone;

CollectionState m\_previousCollectionState;

#ifdef RECYCLER\_STATS

RecyclerCollectionStats m\_previousCollectionStats;

#endif

public:

AutoSetupRecyclerForNonCollectingMark(Recycler& recycler, bool setupForHeapEnumeration = false);

~AutoSetupRecyclerForNonCollectingMark();

void DoCommonSetup();

void SetupForHeapEnumeration();

};

friend class RecyclerHeapObjectInfo;

bool FindImplicitRootObject(void\* candidate, RecyclerHeapObjectInfo& heapObject);

bool FindHeapObject(void\* candidate, FindHeapObjectFlags flags, RecyclerHeapObjectInfo& heapObject);

bool FindHeapObjectWithClearedAllocators(void\* candidate, RecyclerHeapObjectInfo& heapObject);

bool IsCollectionDisabled() const { return isCollectionDisabled; }

bool IsHeapEnumInProgress() const { Assert(isHeapEnumInProgress ? isCollectionDisabled : true); return isHeapEnumInProgress; }

#if DBG

// There are limited cases that we have to allow allocation during heap enumeration. GC is explicitly

// disabled during heap enumeration for these limited cases. (See DefaultRecyclerCollectionWrapper)

// The only case of allocation right now is allocating property record for string based type handler

// so we can use the propertyId as the relation Id.

// Allocation during enumeration is still frown upon and should still be avoid if possible.

bool AllowAllocationDuringHeapEnum() const { return allowAllocationDuringHeapEnum; }

class AutoAllowAllocationDuringHeapEnum : public AutoBooleanToggle

{

public:

AutoAllowAllocationDuringHeapEnum(Recycler \* recycler) : AutoBooleanToggle(&recycler->allowAllocationDuringHeapEnum) {};

};

#ifdef ENABLE\_PROJECTION

bool IsInRefCountTrackingForProjection() const { return isInRefCountTrackingForProjection;}

class AutoIsInRefCountTrackingForProjection : public AutoBooleanToggle

{

public:

AutoIsInRefCountTrackingForProjection(Recycler \* recycler) : AutoBooleanToggle(&recycler->isInRefCountTrackingForProjection) {};

};

#endif

#endif

class AutoAllowAllocationDuringReentrance : public AutoBooleanToggle

{

public:

AutoAllowAllocationDuringReentrance(Recycler \* recycler) :

AutoBooleanToggle(&recycler->isCollectionDisabled)

#if DBG

, allowAllocationDuringRenentrance(&recycler->allowAllocationDuringRenentrance)

#endif

{};

#if DBG

private:

AutoBooleanToggle allowAllocationDuringRenentrance;

#endif

};

#ifdef HEAP\_ENUMERATION\_VALIDATION

typedef void(\*PostHeapEnumScanCallback)(const HeapObject& heapObject, void \*data);

PostHeapEnumScanCallback pfPostHeapEnumScanCallback;

void \*postHeapEnunScanData;

void PostHeapEnumScan(PostHeapEnumScanCallback callback, void\*data);

bool IsPostEnumHeapValidationInProgress() const { return pfPostHeapEnumScanCallback != NULL; }

#endif

private:

void\* GetRealAddressFromInterior(void\* candidate);

void BeginNonCollectingMark();

void EndNonCollectingMark();

#if defined(RECYCLER\_DUMP\_OBJECT\_GRAPH) || defined(LEAK\_REPORT) || defined(CHECK\_MEMORY\_LEAK)

public:

bool IsInDllCanUnloadNow() const { return inDllCanUnloadNow; }

bool IsInDetachProcess() const { return inDetachProcess; }

void SetInDllCanUnloadNow();

void SetInDetachProcess();

private:

bool inDllCanUnloadNow;

bool inDetachProcess;

bool isPrimaryMarkContextInitialized;

#endif

#if defined(LEAK\_REPORT) || defined(CHECK\_MEMORY\_LEAK)

template <class Fn>

void ReportOnProcessDetach(Fn fn);

void PrintPinnedObjectStackTraces();

#endif

public:

typedef void (CALLBACK \*ObjectBeforeCollectCallback)(void\* object, void\* callbackState); // same as jsrt JsObjectBeforeCollectCallback

void SetObjectBeforeCollectCallback(void\* object, ObjectBeforeCollectCallback callback, void\* callbackState);

void ClearObjectBeforeCollectCallbacks();

bool IsInObjectBeforeCollectCallback() const { return objectBeforeCollectCallbackState != ObjectBeforeCollectCallback\_None; }

private:

struct ObjectBeforeCollectCallbackData

{

ObjectBeforeCollectCallback callback;

void\* callbackState;

ObjectBeforeCollectCallbackData() {}

ObjectBeforeCollectCallbackData(ObjectBeforeCollectCallback callback, void\* callbackState) : callback(callback), callbackState(callbackState) {}

};

typedef JsUtil::BaseDictionary<void\*, ObjectBeforeCollectCallbackData, HeapAllocator,

PrimeSizePolicy, RecyclerPointerComparer, JsUtil::SimpleDictionaryEntry, JsUtil::NoResizeLock> ObjectBeforeCollectCallbackMap;

ObjectBeforeCollectCallbackMap\* objectBeforeCollectCallbackMap;

enum ObjectBeforeCollectCallbackState

{

ObjectBeforeCollectCallback\_None,

ObjectBeforeCollectCallback\_Normal, // Normal GC BeforeCollect callback

ObjectBeforeCollectCallback\_Shutdown, // At shutdown invoke all BeforeCollect callback

} objectBeforeCollectCallbackState;

bool ProcessObjectBeforeCollectCallbacks(bool atShutdown = false);

};

class RecyclerHeapObjectInfo

{

void\* m\_address;

Recycler \* m\_recycler;

HeapBlock\* m\_heapBlock;

#if LARGEHEAPBLOCK\_ENCODING

union

{

byte \* m\_attributes;

LargeObjectHeader \* m\_largeHeapBlockHeader;

};

bool isUsingLargeHeapBlock = false;

#else

byte \* m\_attributes;

#endif

public:

RecyclerHeapObjectInfo() : m\_address(NULL), m\_recycler(NULL), m\_heapBlock(NULL), m\_attributes(NULL) {}

RecyclerHeapObjectInfo(void\* address, Recycler \* recycler, HeapBlock\* heapBlock, byte \* attributes) :

m\_address(address), m\_recycler(recycler), m\_heapBlock(heapBlock), m\_attributes(attributes) { }

void\* GetObjectAddress() const { return m\_address; }

bool IsLeaf() const

{

#if LARGEHEAPBLOCK\_ENCODING

if (isUsingLargeHeapBlock)

{

return (m\_largeHeapBlockHeader->GetAttributes(m\_recycler->Cookie) & LeafBit) != 0;

}

#endif

return ((\*m\_attributes & LeafBit) != 0 || this->m\_heapBlock->IsLeafBlock());

}

bool IsImplicitRoot() const

{

#if LARGEHEAPBLOCK\_ENCODING

if (isUsingLargeHeapBlock)

{

return (m\_largeHeapBlockHeader->GetAttributes(m\_recycler->Cookie) & ImplicitRootBit) != 0;

}

#endif

return (\*m\_attributes & ImplicitRootBit) != 0;

}

bool IsObjectMarked() const { Assert(m\_recycler); return m\_recycler->heapBlockMap.IsMarked(m\_address); }

void SetObjectMarked() { Assert(m\_recycler); m\_recycler->heapBlockMap.SetMark(m\_address); }

ObjectInfoBits GetAttributes() const

{

#if LARGEHEAPBLOCK\_ENCODING

if (isUsingLargeHeapBlock)

{

return (ObjectInfoBits)m\_largeHeapBlockHeader->GetAttributes(m\_recycler->Cookie);

}

#endif

return (ObjectInfoBits)\*m\_attributes;

}

size\_t GetSize() const;

#if LARGEHEAPBLOCK\_ENCODING

void SetLargeHeapBlockHeader(LargeObjectHeader \* largeHeapBlockHeader)

{

m\_largeHeapBlockHeader = largeHeapBlockHeader;

isUsingLargeHeapBlock = true;

}

#endif

bool SetMemoryProfilerHasEnumerated()

{

Assert(m\_heapBlock);

#if LARGEHEAPBLOCK\_ENCODING

if (isUsingLargeHeapBlock)

{

return SetMemoryProfilerHasEnumeratedForLargeHeapBlock();

}

#endif

bool wasMemoryProfilerOldObject = (\*m\_attributes & MemoryProfilerOldObjectBit) != 0;

\*m\_attributes |= MemoryProfilerOldObjectBit;

return wasMemoryProfilerOldObject;

}

bool ClearImplicitRootBit()

{

// This can only be called on the main thread for non-finalizable block

// As finalizable block requires that the bit not be change during concurrent mark

// since the background thread change the NewTrackBit

Assert(!m\_heapBlock->IsAnyFinalizableBlock());

#ifdef RECYCLER\_PAGE\_HEAP

Recycler\* recycler = this->m\_recycler;

if (recycler->ShouldCapturePageHeapFreeStack())

{

Assert(recycler->IsPageHeapEnabled());

this->m\_heapBlock->CapturePageHeapFreeStack();

}

#endif

#if LARGEHEAPBLOCK\_ENCODING

if (isUsingLargeHeapBlock)

{

return ClearImplicitRootBitsForLargeHeapBlock();

}

#endif

Assert(m\_attributes);

bool wasImplicitRoot = (\*m\_attributes & ImplicitRootBit) != 0;

\*m\_attributes &= ~ImplicitRootBit;

return wasImplicitRoot;

}

void ExplicitFree()

{

if (\*m\_attributes == ObjectInfoBits::LeafBit)

{

m\_recycler->ExplicitFreeLeaf(m\_address, GetSize());

}

else

{

Assert(\*m\_attributes == ObjectInfoBits::NoBit);

m\_recycler->ExplicitFreeNonLeaf(m\_address, GetSize());

}

}

#if LARGEHEAPBLOCK\_ENCODING

bool ClearImplicitRootBitsForLargeHeapBlock()

{

Assert(m\_largeHeapBlockHeader);

byte attributes = m\_largeHeapBlockHeader->GetAttributes(m\_recycler->Cookie);

bool wasImplicitRoot = (attributes & ImplicitRootBit) != 0;

m\_largeHeapBlockHeader->SetAttributes(m\_recycler->Cookie, attributes & ~ImplicitRootBit);

return wasImplicitRoot;

}

bool SetMemoryProfilerHasEnumeratedForLargeHeapBlock()

{

Assert(m\_largeHeapBlockHeader);

byte attributes = m\_largeHeapBlockHeader->GetAttributes(m\_recycler->Cookie);

bool wasMemoryProfilerOldObject = (attributes & MemoryProfilerOldObjectBit) != 0;

m\_largeHeapBlockHeader->SetAttributes(m\_recycler->Cookie, attributes | MemoryProfilerOldObjectBit);

return wasMemoryProfilerOldObject;

}

#endif

};

// A fake heap block to replace the original heap block where the strong ref is when it has been collected

// as the original heap block may have been freed

class CollectedRecyclerWeakRefHeapBlock : public HeapBlock

{

public:

#if DBG

virtual BOOL IsFreeObject(void\* objectAddress) override { Assert(false); return false; }

#endif

virtual BOOL IsValidObject(void\* objectAddress) override { Assert(false); return false; }

virtual byte\* GetRealAddressFromInterior(void\* interiorAddress) override { Assert(false); return nullptr; }

virtual size\_t GetObjectSize(void\* object) override { Assert(false); return 0; }

virtual bool FindHeapObject(void\* objectAddress, Recycler \* recycler, FindHeapObjectFlags flags, RecyclerHeapObjectInfo& heapObject) override { Assert(false); return false; }

virtual bool TestObjectMarkedBit(void\* objectAddress) override { Assert(false); return false; }

virtual void SetObjectMarkedBit(void\* objectAddress) override { Assert(false); }

#ifdef RECYCLER\_VERIFY\_MARK

virtual void VerifyMark(void \* objectAddress) override { Assert(false); }

#endif

#ifdef RECYCLER\_PERF\_COUNTERS

virtual void UpdatePerfCountersOnFree() override { Assert(false); }

#endif

#ifdef PROFILE\_RECYCLER\_ALLOC

virtual void \* GetTrackerData(void \* address) override { Assert(false); return nullptr; }

virtual void SetTrackerData(void \* address, void \* data) override { Assert(false); }

#endif

static CollectedRecyclerWeakRefHeapBlock Instance;

private:

CollectedRecyclerWeakRefHeapBlock() : HeapBlock(BlockTypeCount) { isPendingConcurrentSweep = false; }

};

class AutoIdleDecommit

{

public:

AutoIdleDecommit(Recycler \* recycler) : recycler(recycler) { recycler->EnterIdleDecommit(); }

~AutoIdleDecommit() { recycler->LeaveIdleDecommit(); }

private:

Recycler \* recycler;

};

template <typename SmallHeapBlockAllocatorType>

void

Recycler::AddSmallAllocator(SmallHeapBlockAllocatorType \* allocator, size\_t sizeCat)

{

autoHeap.AddSmallAllocator(allocator, sizeCat);

}

template <typename SmallHeapBlockAllocatorType>

void

Recycler::RemoveSmallAllocator(SmallHeapBlockAllocatorType \* allocator, size\_t sizeCat)

{

autoHeap.RemoveSmallAllocator(allocator, sizeCat);

}

template <ObjectInfoBits attributes, typename SmallHeapBlockAllocatorType>

char \*

Recycler::SmallAllocatorAlloc(SmallHeapBlockAllocatorType \* allocator, size\_t sizeCat)

{

return autoHeap.SmallAllocatorAlloc<attributes>(this, allocator, sizeCat);

}

// Dummy recycler allocator policy classes to choose the allocation function

class \_RecyclerLeafPolicy;

class \_RecyclerNonLeafPolicy;

template <typename Policy>

class \_RecyclerAllocatorFunc

{};

template <>

class \_RecyclerAllocatorFunc<\_RecyclerLeafPolicy>

{

public:

typedef char \* (Recycler::\*AllocFuncType)(size\_t);

typedef bool (Recycler::\*FreeFuncType)(void\*, size\_t);

static AllocFuncType GetAllocFunc()

{

return &Recycler::AllocLeaf;

}

static AllocFuncType GetAllocZeroFunc()

{

return &Recycler::AllocLeafZero;

}

static FreeFuncType GetFreeFunc()

{

return &Recycler::ExplicitFreeLeaf;

}

};

template <>

class \_RecyclerAllocatorFunc<\_RecyclerNonLeafPolicy>

{

public:

typedef char \* (Recycler::\*AllocFuncType)(size\_t);

typedef bool (Recycler::\*FreeFuncType)(void\*, size\_t);

static AllocFuncType GetAllocFunc()

{

return &Recycler::Alloc;

}

static AllocFuncType GetAllocZeroFunc()

{

return &Recycler::AllocZero;

}

static FreeFuncType GetFreeFunc()

{

return &Recycler::ExplicitFreeNonLeaf;

}

};

// This is used by the compiler; when T is NOT a pointer i.e. a value type - it causes leaf allocation

template <typename T>

class TypeAllocatorFunc<Recycler, T> : public \_RecyclerAllocatorFunc<\_RecyclerLeafPolicy>

{

};

// Partial template specialization; applies to T when it is a pointer

template <typename T>

class TypeAllocatorFunc<Recycler, T \*> : public \_RecyclerAllocatorFunc<\_RecyclerNonLeafPolicy>

{

};

template <bool isLeaf>

class ListTypeAllocatorFunc<Recycler, isLeaf>

{

public:

typedef bool (Recycler::\*FreeFuncType)(void\*, size\_t);

static FreeFuncType GetFreeFunc()

{

if (isLeaf)

{

return &Recycler::ExplicitFreeLeaf;

}

else

{

return &Recycler::ExplicitFreeNonLeaf;

}

}

};

// Dummy class to choose the allocation function

class RecyclerLeafAllocator;

class RecyclerNonLeafAllocator;

// Partial template specialization to allocate as non leaf

template <typename T>

class TypeAllocatorFunc<RecyclerNonLeafAllocator, T> : public \_RecyclerAllocatorFunc<\_RecyclerNonLeafPolicy>

{

};

template <typename T>

class TypeAllocatorFunc<RecyclerLeafAllocator, T> : public \_RecyclerAllocatorFunc<\_RecyclerLeafPolicy>

{

};

template <typename TAllocType>

struct AllocatorInfo<Recycler, TAllocType>

{

typedef Recycler AllocatorType;

typedef TypeAllocatorFunc<Recycler, TAllocType> AllocatorFunc;

typedef \_RecyclerAllocatorFunc<\_RecyclerNonLeafPolicy> InstAllocatorFunc; // By default any instance considered non-leaf

};

template <typename TAllocType>

struct AllocatorInfo<RecyclerNonLeafAllocator, TAllocType>

{

typedef Recycler AllocatorType;

typedef TypeAllocatorFunc<RecyclerNonLeafAllocator, TAllocType> AllocatorFunc;

typedef TypeAllocatorFunc<RecyclerNonLeafAllocator, TAllocType> InstAllocatorFunc; // Same as TypeAllocatorFunc

};

template <typename TAllocType>

struct AllocatorInfo<RecyclerLeafAllocator, TAllocType>

{

typedef Recycler AllocatorType;

typedef TypeAllocatorFunc<RecyclerLeafAllocator, TAllocType> AllocatorFunc;

typedef TypeAllocatorFunc<RecyclerLeafAllocator, TAllocType> InstAllocatorFunc; // Same as TypeAllocatorFunc

};

template <>

struct ForceNonLeafAllocator<Recycler>

{

typedef RecyclerNonLeafAllocator AllocatorType;

};

template <>

struct ForceNonLeafAllocator<RecyclerLeafAllocator>

{

typedef RecyclerNonLeafAllocator AllocatorType;

};

template <>

struct ForceLeafAllocator<Recycler>

{

typedef RecyclerLeafAllocator AllocatorType;

};

template <>

struct ForceLeafAllocator<RecyclerNonLeafAllocator>

{

typedef RecyclerLeafAllocator AllocatorType;

};

#ifdef PROFILE\_EXEC

#define RECYCLER\_PROFILE\_EXEC\_BEGIN(recycler, phase) if (recycler->profiler != nullptr) { recycler->profiler->Begin(phase); }

#define RECYCLER\_PROFILE\_EXEC\_END(recycler, phase) if (recycler->profiler != nullptr) { recycler->profiler->End(phase); }

#define RECYCLER\_PROFILE\_EXEC\_BEGIN2(recycler, phase1, phase2) if (recycler->profiler != nullptr) { recycler->profiler->Begin(phase1); recycler->profiler->Begin(phase2);}

#define RECYCLER\_PROFILE\_EXEC\_END2(recycler, phase1, phase2) if (recycler->profiler != nullptr) { recycler->profiler->End(phase1); recycler->profiler->End(phase2);}

#define RECYCLER\_PROFILE\_EXEC\_CHANGE(recydler, phase1, phase2) if (recycler->profiler != nullptr) { recycler->profiler->End(phase1); recycler->profiler->Begin(phase2); }

#define RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(recycler, phase) if (recycler->backgroundProfiler != nullptr) { recycler->backgroundProfiler->Begin(phase); }

#define RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(recycler, phase) if (recycler->backgroundProfiler != nullptr) { recycler->backgroundProfiler->End(phase); }

#define RECYCLER\_PROFILE\_EXEC\_THREAD\_BEGIN(background, recycler, phase) if (background) { RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(recycler, phase); } else { RECYCLER\_PROFILE\_EXEC\_BEGIN(recycler, phase); }

#define RECYCLER\_PROFILE\_EXEC\_THREAD\_END(background, recycler, phase) if (background) { RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(recycler, phase); } else { RECYCLER\_PROFILE\_EXEC\_END(recycler, phase); }

#else

#define RECYCLER\_PROFILE\_EXEC\_BEGIN(recycler, phase)

#define RECYCLER\_PROFILE\_EXEC\_END(recycler, phase)

#define RECYCLER\_PROFILE\_EXEC\_BEGIN2(recycler, phase1, phase2)

#define RECYCLER\_PROFILE\_EXEC\_END2(recycler, phase1, phase2)

#define RECYCLER\_PROFILE\_EXEC\_CHANGE(recydler, phase1, phase2)

#define RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(recycler, phase)

#define RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(recycler, phase)

#define RECYCLER\_PROFILE\_EXEC\_THREAD\_BEGIN(background, recycler, phase)

#define RECYCLER\_PROFILE\_EXEC\_THREAD\_END(background, recycler, phase)

#endif

}

\_Ret\_notnull\_ inline void \* \_\_cdecl

operator new(size\_t byteSize, Recycler \* alloc, HeapInfo \* heapInfo)

{

return alloc->HeapAllocR(heapInfo, byteSize);

}

inline void \_\_cdecl

operator delete(void \* obj, Recycler \* alloc, HeapInfo \* heapInfo)

{

alloc->HeapFree(heapInfo, obj);

}

\_Ret\_notnull\_ inline void \* \_\_cdecl

operator new(size\_t byteSize, Recycler \* recycler, ObjectInfoBits enumClassBits)

{

AssertCanHandleOutOfMemory();

Assert(byteSize != 0);

Assert(enumClassBits == EnumClass\_1\_Bit);

void \* buffer = recycler->AllocEnumClass<EnumClass\_1\_Bit>(byteSize);

// All of our allocation should throw on out of memory

Assume(buffer != nullptr);

return buffer;

}

template<ObjectInfoBits infoBits>

\_Ret\_notnull\_ inline void \* \_\_cdecl

operator new(size\_t byteSize, Recycler \* recycler, const InfoBitsWrapper<infoBits>&)

{

AssertCanHandleOutOfMemory();

Assert(byteSize != 0);

void \* buffer = recycler->AllocWithInfoBits<infoBits>(byteSize);

// All of our allocation should throw on out of memory

Assume(buffer != nullptr);

return buffer;

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

template <ObjectInfoBits attributes>

bool

Recycler::IntegrateBlock(char \* blockAddress, PageSegment \* segment, size\_t allocSize, size\_t objectSize)

{

// We only support no bit and leaf bit right now, where we don't need to set the object info in either case

CompileAssert(attributes == NoBit || attributes == LeafBit);

// Currently only small object is supported

Assert(HeapInfo::IsSmallObject(allocSize));

Assert(HeapInfo::GetAlignedSizeNoCheck(allocSize) == allocSize);

// This should only happen during a pre-collection callback.

Assert(this->collectionState == Collection\_PreCollection);

bool success = autoHeap.IntegrateBlock<attributes>(blockAddress, segment, this, allocSize);

#ifdef PROFILE\_RECYCLER\_ALLOC

if (success)

{

TrackAllocData trackAllocData;

ClearTrackAllocInfo(&trackAllocData);

TrackIntegrate(blockAddress, SmallAllocationBlockAttributes::PageCount \* AutoSystemInfo::PageSize, allocSize, objectSize, trackAllocData);

}

#endif

return success;

}

namespace Memory

{

class DummyVTableObject : public FinalizableObject

{

public:

virtual void Finalize(bool isShutdown) {}

virtual void Dispose(bool isShutdown) {}

virtual void Mark(Recycler \* recycler) {}

};

}

template <ObjectInfoBits attributes, bool nothrow>

\_\_inline char \*

Recycler::AllocWithAttributesInlined(size\_t size)

{

// All tracked objects are client tracked objects

CompileAssert((attributes & TrackBit) == 0 || (attributes & ClientTrackedBit) != 0);

Assert(this->enableScanImplicitRoots || (attributes & ImplicitRootBit) == 0);

AssertMsg(this->disableThreadAccessCheck || this->mainThreadId == GetCurrentThreadContextId(),

"Allocating from the recycler can only be done on the main thread");

Assert(size != 0);

AssertMsg(collectionState != Collection\_PreCollection, "we cannot have allocation in precollection callback");

// We shouldn't be allocating memory when we are running GC in thread, including finalizers

Assert(this->IsConcurrentState() || !this->CollectionInProgress() || this->collectionState == CollectionStatePostCollectionCallback);

// There are some cases where we allow allocation during heap enum that doesn't affect the enumeration

// Those should be really rare and not rely upon.

Assert(!isHeapEnumInProgress || allowAllocationDuringHeapEnum);

#ifdef PROFILE\_RECYCLER\_ALLOC

TrackAllocData trackAllocData;

ClearTrackAllocInfo(&trackAllocData);

#endif

size\_t allocSize = size;

#ifdef RECYCLER\_MEMORY\_VERIFY

if (this->VerifyEnabled())

{

allocSize += verifyPad + sizeof(size\_t);

if (allocSize < size)

{

// An overflow occurred- if nothrow is false, we can throw here

// Otherwise, return null

if (nothrow == false)

{

this->OutOfMemory();

}

else

{

return nullptr;

}

}

}

#endif

char\* memBlock = RealAlloc<(ObjectInfoBits)(attributes & InternalObjectInfoBitMask), nothrow>(&autoHeap, allocSize);

if (nothrow)

{

// If we aren't allowed to throw, then the memblock returned could be null

// so we should check for that and bail out early here

if (memBlock == nullptr)

{

return nullptr;

}

}

#ifdef PROFILE\_RECYCLER\_ALLOC

TrackAlloc(memBlock, size, trackAllocData, (CUSTOM\_CONFIG\_ISENABLED(GetRecyclerFlagsTable(), Js::TraceObjectAllocationFlag) && (attributes & TraceBit) == TraceBit));

#endif

RecyclerMemoryTracking::ReportAllocation(this, memBlock, size);

RECYCLER\_PERF\_COUNTER\_INC(LiveObject);

RECYCLER\_PERF\_COUNTER\_ADD(LiveObjectSize, HeapInfo::GetAlignedSizeNoCheck(allocSize));

RECYCLER\_PERF\_COUNTER\_SUB(FreeObjectSize, HeapInfo::GetAlignedSizeNoCheck(allocSize));

if (HeapInfo::IsSmallBlockAllocation(HeapInfo::GetAlignedSizeNoCheck(allocSize)))

{

RECYCLER\_PERF\_COUNTER\_INC(SmallHeapBlockLiveObject);

RECYCLER\_PERF\_COUNTER\_ADD(SmallHeapBlockLiveObjectSize, HeapInfo::GetAlignedSizeNoCheck(allocSize));

RECYCLER\_PERF\_COUNTER\_SUB(SmallHeapBlockFreeObjectSize, HeapInfo::GetAlignedSizeNoCheck(allocSize));

}

else

{

RECYCLER\_PERF\_COUNTER\_INC(LargeHeapBlockLiveObject);

RECYCLER\_PERF\_COUNTER\_ADD(LargeHeapBlockLiveObjectSize, HeapInfo::GetAlignedSizeNoCheck(allocSize));

RECYCLER\_PERF\_COUNTER\_SUB(LargeHeapBlockFreeObjectSize, HeapInfo::GetAlignedSizeNoCheck(allocSize));

}

#ifdef RECYCLER\_MEMORY\_VERIFY

size\_t alignedSize = HeapInfo::GetAlignedSizeNoCheck(allocSize);

if (HeapInfo::IsMediumObject(allocSize))

{

#if SMALLBLOCK\_MEDIUM\_ALLOC

alignedSize = HeapInfo::GetMediumObjectAlignedSizeNoCheck(allocSize);

#else

HeapBlock\* heapBlock = this->FindHeapBlock(memBlock);

Assert(heapBlock->IsLargeHeapBlock());

LargeHeapBlock\* largeHeapBlock = (LargeHeapBlock\*) heapBlock;

LargeObjectHeader\* header = nullptr;

if (largeHeapBlock->GetObjectHeader(memBlock, &header))

{

size = header->objectSize - (verifyPad + sizeof(size\_t));

alignedSize = HeapInfo::GetAlignedSizeNoCheck(header->objectSize);

}

#endif

}

this->FillCheckPad(memBlock, size, alignedSize);

#endif

#pragma prefast(suppress:6313, "attributes is a template parameter and can be 0")

if (attributes & (FinalizeBit | TrackBit))

{

// Make sure a valid vtable is installed in case of OOM before the real vtable is set

memBlock = (char \*)new (memBlock) DummyVTableObject();

}

#ifdef RECYCLER\_WRITE\_BARRIER

SwbVerboseTrace(this->GetRecyclerFlagsTable(), L"Allocated SWB memory: 0x%p\n", memBlock);

#pragma prefast(suppress:6313, "attributes is a template parameter and can be 0")

if (attributes & (NewTrackBit))

{

// For objects allocated with NewTrackBit, we need to trigger the write barrier since

// there could be a GC triggered by an allocation in the constructor, and we'd miss

// calling track on the partially constructed object. To deal with this, we set the write

// barrier on all the pages of objects allocated with the NewTrackBit

RecyclerWriteBarrierManager::WriteBarrier(memBlock, size / sizeof(void\*));

}

#endif

#ifdef PARTIAL\_GC\_ENABLED

#pragma prefast(suppress:6313, "attributes is a template parameter and can be 0")

if (attributes & ClientTrackedBit)

{

if (this->inPartialCollectMode)

{

// with partial GC, we don't traverse ITrackable

// So we have to mark all objects that could be in the ITrackable graph

// This includes JavascriptDispatch and HostVariant

this->clientTrackedObjectList.Prepend(&this->clientTrackedObjectAllocator, memBlock);

}

else

{

Assert(this->hasBackgroundFinishPartial || this->clientTrackedObjectList.Empty());

}

}

#endif

return memBlock;

}

template <ObjectInfoBits attributes, bool nothrow>

\_\_inline char \*

Recycler::AllocZeroWithAttributesInlined(size\_t size)

{

char\* obj = AllocWithAttributesInlined<attributes, nothrow>(size);

if (nothrow)

{

// If we aren't allowed to throw, then the obj returned could be null

// so we should check for that and bail out early here

if (obj == nullptr)

{

return nullptr;

}

}

#ifdef RECYCLER\_MEMORY\_VERIFY

if (this->VerifyEnabled())

{

memset(obj, 0, min(size, sizeof(FreeObject \*)));

}

else

#endif

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_THREAD\_PAGE

if ((((attributes & LeafBit) == LeafBit) || ((attributes & WithBarrierBit) == WithBarrierBit)) && HeapInfo::IsSmallObjectAllocation(size))

#else

if (((attributes & LeafBit) == LeafBit) && HeapInfo::IsSmallBlockAllocation(size))

#endif

{

// If this was allocated from the small heap block, it's not

// guaranteed to be zero so we should zero out here.

memset((void\*) obj, 0, size);

}

else

{

// All recycler memory are allocated with zero except for the first word,

// which store the next pointer for the free list. Just zero that one out

((FreeObject \*)obj)->ZeroNext();

}

return obj;

}

template <ObjectInfoBits attributes, bool isSmallAlloc, bool nothrow>

\_\_inline char\*

Recycler::RealAllocFromBucket(HeapInfo\* heap, size\_t size)

{

// Align the size

Assert(HeapInfo::GetAlignedSizeNoCheck(size) <= UINT\_MAX);

uint sizeCat;

char \* memBlock;

if (isSmallAlloc)

{

sizeCat = (uint)HeapInfo::GetAlignedSizeNoCheck(size);

memBlock = heap->RealAlloc<attributes, nothrow>(this, sizeCat);

}

#ifdef BUCKETIZE\_MEDIUM\_ALLOCATIONS

else

{

sizeCat = (uint)HeapInfo::GetMediumObjectAlignedSizeNoCheck(size);

memBlock = heap->MediumAlloc<attributes, nothrow>(this, sizeCat);

}

#endif

// If we are not allowed to throw, then the memory returned here could be null so check for that

// If we are allowed to throw, then memBlock is not allowed to null so assert that

if (nothrow)

{

if (memBlock == nullptr)

{

return nullptr;

}

}

else

{

Assert(memBlock != nullptr);

}

#ifdef RECYCLER\_ZERO\_MEM\_CHECK

// Don't bother checking leaf allocations for zeroing out- they're not guaranteed to be so

if ((attributes & LeafBit) == 0

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_THREAD\_PAGE

&& (attributes & WithBarrierBit) == 0

#endif

)

{

VerifyZeroFill(memBlock + sizeof(FreeObject), sizeCat - (2 \* sizeof(FreeObject)));

}

#endif

#ifdef PROFILE\_MEM

if (this->memoryData)

{

this->memoryData->requestCount++;

this->memoryData->requestBytes += size;

this->memoryData->alignmentBytes += sizeCat - size;

}

#endif

return memBlock;

}

template <ObjectInfoBits attributes, bool nothrow>

\_\_inline char\*

Recycler::RealAlloc(HeapInfo\* heap, size\_t size)

{

#ifdef RECYCLER\_STRESS

this->StressCollectNow();

#endif

if (nothrow)

{

FAULTINJECT\_MEMORY\_NOTHROW(L"Recycler", size);

}

else

{

FAULTINJECT\_MEMORY\_THROW(L"Recycler", size);

}

if (HeapInfo::IsSmallObject(size))

{

return RealAllocFromBucket<attributes, /\* isSmallAlloc = \*/ true, nothrow>(heap, size);

}

#ifdef BUCKETIZE\_MEDIUM\_ALLOCATIONS

if (HeapInfo::IsMediumObject(size))

{

return RealAllocFromBucket<attributes, /\* isSmallAlloc = \*/ false, nothrow>(heap, size);

}

#endif

return LargeAlloc<nothrow>(heap, size, attributes);

}

template<typename T>

\_\_inline RecyclerWeakReference<T>\* Recycler::CreateWeakReferenceHandle(T\* pStrongReference)

{

// Return the weak reference that calling Add on the WR map returns

// The entry returned is recycler-allocated memory

RecyclerWeakReference<T>\* weakRef = (RecyclerWeakReference<T>\*) this->weakReferenceMap.Add((char\*) pStrongReference, this);

#if DBG

if (weakRef->typeInfo == nullptr)

{

weakRef->typeInfo = &typeid(T);

#ifdef TRACK\_ALLOC

TrackAllocWeakRef(weakRef);

#endif

}

#endif

return weakRef;

}

template<typename T>

\_\_inline bool Recycler::FindOrCreateWeakReferenceHandle(T\* pStrongReference, RecyclerWeakReference<T> \*\*ppWeakRef)

{

// Ensure that the given strong ref has a weak ref in the map.

// Return a result to indicate whether a new weak ref was created.

bool ret = this->weakReferenceMap.FindOrAdd((char\*) pStrongReference, this, (RecyclerWeakReferenceBase\*\*)ppWeakRef);

#if DBG

if (!ret)

{

(\*ppWeakRef)->typeInfo = &typeid(T);

#ifdef TRACK\_ALLOC

TrackAllocWeakRef(\*ppWeakRef);

#endif

}

#endif

return ret;

}

template<typename T>

\_\_inline bool Recycler::TryGetWeakReferenceHandle(T\* pStrongReference, RecyclerWeakReference<T> \*\*weakReference)

{

return this->weakReferenceMap.TryGetValue((char\*) pStrongReference, (RecyclerWeakReferenceBase\*\*)weakReference);

}

\_\_inline HeapBlock\*

Recycler::FindHeapBlock(void\* candidate)

{

if ((size\_t)candidate < 0x10000)

{

return nullptr;

}

if (!HeapInfo::IsAlignedAddress(candidate))

{

return nullptr;

}

return heapBlockMap.GetHeapBlock(candidate);

}

\_\_inline void

Recycler::ScanObjectInline(void \*\* obj, size\_t byteCount)

{

// This is never called during parallel marking

Assert(this->collectionState != CollectionStateParallelMark);

if (this->enableScanInteriorPointers)

{

ScanObjectInlineInterior(obj, byteCount);

}

else

{

markContext.ScanObject<false, false>(obj, byteCount);

}

}

\_\_inline void

Recycler::ScanObjectInlineInterior(void \*\* obj, size\_t byteCount)

{

// This is never called during parallel marking

Assert(this->collectionState != CollectionStateParallelMark);

Assert(this->enableScanInteriorPointers);

markContext.ScanObject<false, true>(obj, byteCount);

}

\_\_inline void

Recycler::ScanMemoryInline(void \*\* obj, size\_t byteCount)

{

// This is never called during parallel marking

Assert(this->collectionState != CollectionStateParallelMark);

if (this->enableScanInteriorPointers)

{

markContext.ScanMemory<false, true>(obj, byteCount);

}

else

{

markContext.ScanMemory<false, false>(obj, byteCount);

}

}

\_\_inline bool

Recycler::AddMark(void \* candidate, size\_t byteCount) throw()

{

// This is never called during parallel marking

Assert(this->collectionState != CollectionStateParallelMark);

return markContext.AddMarkedObject(candidate, byteCount);

}

template <bool pageheap, typename T>

void

Recycler::NotifyFree(T \* heapBlock)

{

bool forceSweepObject = ForceSweepObject();

if (forceSweepObject)

{

#if DBG || defined(RECYCLER\_STATS)

this->isForceSweeping = true;

heapBlock->isForceSweeping = true;

#endif

heapBlock->SweepObjects<pageheap, SweepMode\_InThread>(this);

#if DBG || defined(RECYCLER\_STATS)

heapBlock->isForceSweeping = false;

this->isForceSweeping = false;

#endif

RECYCLER\_STATS\_INC(this, heapBlockFreeCount[heapBlock->GetHeapBlockType()]);

}

JS\_ETW(EventWriteFreeMemoryBlock(heapBlock));

#ifdef RECYCLER\_PERF\_COUNTERS

if (forceSweepObject)

{

RECYCLER\_PERF\_COUNTER\_SUB(FreeObjectSize, heapBlock->GetPageCount() \* AutoSystemInfo::PageSize);

if (heapBlock->IsLargeHeapBlock())

{

RECYCLER\_PERF\_COUNTER\_SUB(LargeHeapBlockFreeObjectSize, heapBlock->GetPageCount() \* AutoSystemInfo::PageSize);

}

else

{

RECYCLER\_PERF\_COUNTER\_SUB(SmallHeapBlockFreeObjectSize, heapBlock->GetPageCount() \* AutoSystemInfo::PageSize);

}

}

else

{

heapBlock->UpdatePerfCountersOnFree();

}

#endif

}

template <class TBlockAttributes>

\_\_inline ushort

SmallHeapBlockT<TBlockAttributes>::GetObjectBitDelta()

{

return this->objectSize / HeapConstants::ObjectGranularity;

}

// Map any object address to it's bit index in the heap block bit vectors.

// static

template <class TBlockAttributes>

\_\_forceinline ushort

SmallHeapBlockT<TBlockAttributes>::GetAddressBitIndex(void \* objectAddress)

{

Assert(HeapInfo::IsAlignedAddress(objectAddress));

ushort offset = (ushort)(::Math::PointerCastToIntegralTruncate<uint>(objectAddress) % (TBlockAttributes::PageCount \* AutoSystemInfo::PageSize));

offset = offset >> HeapConstants::ObjectAllocationShift;

Assert(offset <= USHRT\_MAX);

Assert(offset <= TBlockAttributes::MaxAddressBit);

return (ushort) offset;

}

template <class TBlockAttributes>

\_\_forceinline ushort

SmallHeapBlockT<TBlockAttributes>::GetObjectIndexFromBitIndex(ushort bitIndex)

{

Assert(bitIndex <= TBlockAttributes::MaxAddressBit);

ushort objectIndex = validPointers.GetAddressIndex(bitIndex);

Assert(objectIndex == SmallHeapBlockT<TBlockAttributes>::InvalidAddressBit ||

objectIndex <= TBlockAttributes::MaxAddressBit);

return objectIndex;

}

template <class TBlockAttributes>

\_\_forceinline void \*

SmallHeapBlockT<TBlockAttributes>::GetRealAddressFromInterior(void \* interiorAddress, uint objectSize, byte bucketIndex)

{

const ValidPointers<TBlockAttributes> validPointers = HeapInfo::GetValidPointersMapForBucket<TBlockAttributes>(bucketIndex);

size\_t rawInteriorAddress = reinterpret\_cast<size\_t>(interiorAddress);

size\_t baseAddress = rawInteriorAddress & ~(TBlockAttributes::PageCount \* AutoSystemInfo::PageSize - 1);

ushort offset = (ushort)(rawInteriorAddress - baseAddress);

offset = validPointers.GetInteriorAddressIndex(offset >> HeapConstants::ObjectAllocationShift);

if (offset == SmallHeapBlockT<TBlockAttributes>::InvalidAddressBit)

{

return nullptr;

}

return reinterpret\_cast<void\*>(baseAddress + offset \* objectSize);

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "CommonMemoryPch.h"

#ifdef CONCURRENT\_GC\_ENABLED

#include <process.h>

#endif

#ifdef \_M\_AMD64

#include "amd64.h"

#endif

#ifdef \_M\_ARM

#include "arm.h"

#endif

#ifdef \_M\_ARM64

#include "arm64.h"

#endif

#include "core\BinaryFeatureControl.h"

#include "Common\ThreadService.h"

#include "Memory\AutoAllocatorObjectPtr.h"

DEFINE\_RECYCLER\_TRACKER\_PERF\_COUNTER(RecyclerWeakReferenceBase);

#ifdef PROFILE\_RECYCLER\_ALLOC

struct UnallocatedPortionOfBumpAllocatedBlock

{

};

struct ExplicitFreeListedObject

{

};

Recycler::TrackerData Recycler::TrackerData::EmptyData(&typeid(UnallocatedPortionOfBumpAllocatedBlock), false);

Recycler::TrackerData Recycler::TrackerData::ExplicitFreeListObjectData(&typeid(ExplicitFreeListedObject), false);

#endif

enum ETWEventGCActivationKind : unsigned

{

ETWEvent\_GarbageCollect = 0, // force in-thread GC

ETWEvent\_ThreadCollect = 1, // thread GC with wait

ETWEvent\_ConcurrentCollect = 2,

ETWEvent\_PartialCollect = 3,

ETWEvent\_ConcurrentMark = 11,

ETWEvent\_ConcurrentRescan = 12,

ETWEvent\_ConcurrentSweep = 13,

ETWEvent\_ConcurrentTransferSwept = 14,

ETWEvent\_ConcurrentFinishMark = 15,

};

DefaultRecyclerCollectionWrapper DefaultRecyclerCollectionWrapper::Instance;

\_\_inline bool

DefaultRecyclerCollectionWrapper::IsCollectionDisabled(Recycler \* recycler)

{

// GC shouldn't be triggered during heap enum, unless we missed a case where it allocate memory (which

// shouldn't happen during heap enum) or for the case we explicitly allow allocation

// REVIEW: isHeapEnumInProgress should have been a collection state and checked before to avoid a check here.

// Collection will be disabled in VarDispEx because it could be called from projection re-entrance as ASTA allows

// QI/AddRef/Release to come back.

bool collectionDisabled = recycler->IsCollectionDisabled();

#if DBG

if (collectionDisabled)

{

// disabled collection should only happen if we allowed allocation during heap enum

if (recycler->IsHeapEnumInProgress())

{

Assert(recycler->AllowAllocationDuringHeapEnum());

}

else

{

#ifdef ENABLE\_PROJECTION

Assert(recycler->IsInRefCountTrackingForProjection());

#else

Assert(false);

#endif

}

}

#endif

return collectionDisabled;

}

BOOL DefaultRecyclerCollectionWrapper::ExecuteRecyclerCollectionFunction(Recycler \* recycler, CollectionFunction function, CollectionFlags flags)

{

if (IsCollectionDisabled(recycler))

{

return FALSE;

}

BOOL ret = FALSE;

BEGIN\_NO\_EXCEPTION

{

ret = (recycler->\*(function))(flags);

}

END\_NO\_EXCEPTION;

return ret;

}

void

DefaultRecyclerCollectionWrapper::DisposeObjects(Recycler \* recycler)

{

if (IsCollectionDisabled(recycler))

{

return;

}

BEGIN\_NO\_EXCEPTION

{

recycler->DisposeObjects();

}

END\_NO\_EXCEPTION;

}

static void\* GetStackBase();

template \_\_forceinline char \* Recycler::AllocWithAttributesInlined<NoBit, false>(size\_t size);

template \_\_forceinline char\* Recycler::RealAlloc<NoBit, false>(HeapInfo\* heap, size\_t size);

template \_\_forceinline \_Ret\_notnull\_ void \* \_\_cdecl operator new<Recycler>(size\_t byteSize, Recycler \* alloc, char \* (Recycler::\*AllocFunc)(size\_t));

Recycler::Recycler(AllocationPolicyManager \* policyManager, IdleDecommitPageAllocator \* pageAllocator, void (\*outOfMemoryFunc)(), Js::ConfigFlagsTable& configFlagsTable) :

collectionState(CollectionStateNotCollecting),

recyclerFlagsTable(configFlagsTable),

recyclerPageAllocator(this, policyManager, configFlagsTable, RecyclerHeuristic::Instance.DefaultMaxFreePageCount, RecyclerHeuristic::Instance.DefaultMaxAllocPageCount),

recyclerLargeBlockPageAllocator(this, policyManager, configFlagsTable, RecyclerHeuristic::Instance.DefaultMaxFreePageCount),

threadService(nullptr),

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator(this, policyManager, configFlagsTable, RecyclerHeuristic::Instance.DefaultMaxFreePageCount),

#endif

threadPageAllocator(pageAllocator),

markPagePool(configFlagsTable),

parallelMarkPagePool1(configFlagsTable),

parallelMarkPagePool2(configFlagsTable),

parallelMarkPagePool3(configFlagsTable),

markContext(this, &this->markPagePool),

parallelMarkContext1(this, &this->parallelMarkPagePool1),

parallelMarkContext2(this, &this->parallelMarkPagePool2),

parallelMarkContext3(this, &this->parallelMarkPagePool3),

clientTrackedObjectAllocator(L"CTO-List", GetPageAllocator(), Js::Throw::OutOfMemory),

outOfMemoryFunc(outOfMemoryFunc),

#ifdef RECYCLER\_TEST\_SUPPORT

checkFn(NULL),

#endif

externalRootMarker(NULL),

externalRootMarkerContext(NULL),

recyclerSweep(nullptr),

inEndMarkOnLowMemory(false),

enableScanInteriorPointers(CUSTOM\_CONFIG\_FLAG(configFlagsTable, RecyclerForceMarkInterior)),

enableScanImplicitRoots(false),

disableCollectOnAllocationHeuristics(false),

#ifdef CONCURRENT\_GC\_ENABLED

backgroundFinishMarkCount(0),

hasPendingUnpinnedObject(false),

hasPendingConcurrentFindRoot(false),

queueTrackedObject(false),

enableConcurrentMark(false), // Default to non-concurrent

enableParallelMark(false),

enableConcurrentSweep(false),

concurrentThread(NULL),

concurrentWorkReadyEvent(NULL),

concurrentWorkDoneEvent(NULL),

mainThreadHandle(NULL),

parallelThread1(this, &Recycler::ParallelWorkFunc<0>),

parallelThread2(this, &Recycler::ParallelWorkFunc<1>),

priorityBoost(false),

skipStack(false),

isAborting(false),

#if DBG

concurrentThreadExited(true),

isProcessingTrackedObjects(false),

hasIncompletedDoCollect(false),

isConcurrentGCOnIdle(false),

isFinishGCOnIdle(false),

isExternalStackSkippingGC(false),

isProcessingRescan(false),

#endif

#ifdef IDLE\_DECOMMIT\_ENABLED

concurrentIdleDecommitEvent(nullptr),

#endif

#endif

#ifdef PARTIAL\_GC\_ENABLED

inPartialCollectMode(false),

scanPinnedObjectMap(false),

partialUncollectedAllocBytes(0),

uncollectedNewPageCountPartialCollect((size\_t)-1),

partialConcurrentNextCollection(false),

#ifdef RECYCLER\_STRESS

forcePartialScanStack(false),

#endif

#endif

#if defined(RECYCLER\_DUMP\_OBJECT\_GRAPH) || defined(LEAK\_REPORT) || defined(CHECK\_MEMORY\_LEAK)

isPrimaryMarkContextInitialized(false),

#endif

allowDispose(false),

hasDisposableObject(false),

tickCountNextDispose(0),

hasPendingTransferDisposedObjects(false),

transientPinnedObject(nullptr),

pinnedObjectMap(1024, HeapAllocator::GetNoMemProtectInstance()),

weakReferenceMap(1024, HeapAllocator::GetNoMemProtectInstance()),

weakReferenceCleanupId(0),

collectionWrapper(&DefaultRecyclerCollectionWrapper::Instance),

isScriptActive(false),

isInScript(false),

isShuttingDown(false),

inExhaustiveCollection(false),

hasExhaustiveCandidate(false),

inDecommitNowCollection(false),

inCacheCleanupCollection(false),

hasPendingDeleteGuestArena(false),

needOOMRescan(false),

hasBackgroundFinishPartial(false),

decommitOnFinish(false)

#ifdef PROFILE\_EXEC

, profiler(nullptr)

, backgroundProfiler(nullptr)

, backgroundProfilerPageAllocator(nullptr, configFlagsTable, PageAllocatorType\_GCThread)

, backgroundProfilerArena()

#endif

#ifdef PROFILE\_MEM

, memoryData(nullptr)

#endif

#ifdef RECYCLER\_DUMP\_OBJECT\_GRAPH

, objectGraphDumper(nullptr)

, dumpObjectOnceOnCollect(false)

#endif

#ifdef PROFILE\_RECYCLER\_ALLOC

, trackerDictionary(nullptr)

#endif

#ifdef HEAP\_ENUMERATION\_VALIDATION

,pfPostHeapEnumScanCallback(nullptr)

#endif

, telemetryBlock(&localTelemetryBlock)

#ifdef ENABLE\_JS\_ETW

,bulkFreeMemoryWrittenCount(0)

#endif

#ifdef RECYCLER\_PAGE\_HEAP

, isPageHeapEnabled(false)

, capturePageHeapAllocStack(false)

, capturePageHeapFreeStack(false)

#endif

, objectBeforeCollectCallbackMap(nullptr)

, objectBeforeCollectCallbackState(ObjectBeforeCollectCallback\_None)

{

#ifdef RECYCLER\_MARK\_TRACK

this->markMap = NoCheckHeapNew(MarkMap, &NoCheckHeapAllocator::Instance, 163, &markMapCriticalSection);

markContext.SetMarkMap(markMap);

parallelMarkContext1.SetMarkMap(markMap);

parallelMarkContext2.SetMarkMap(markMap);

parallelMarkContext3.SetMarkMap(markMap);

#endif

#ifdef RECYCLER\_MEMORY\_VERIFY

verifyPad = GetRecyclerFlagsTable().RecyclerVerifyPadSize;

verifyEnabled = GetRecyclerFlagsTable().IsEnabled(Js::RecyclerVerifyFlag);

if (verifyEnabled)

{

ForRecyclerPageAllocator(EnableVerify());

}

#endif

#ifdef RECYCLER\_NO\_PAGE\_REUSE

if (GetRecyclerFlagsTable().IsEnabled(Js::RecyclerNoPageReuseFlag))

{

ForRecyclerPageAllocator(DisablePageReuse());

}

#endif

this->inDispose = false;

#if DBG

this->heapBlockCount = 0;

this->collectionCount = 0;

this->disableThreadAccessCheck = false;

this->disableConcurentThreadExitedCheck = false;

#endif

#if DBG || defined RECYCLER\_TRACE

this->inResolveExternalWeakReferences = false;

#endif

#if DBG || defined(RECYCLER\_STATS)

isForceSweeping = false;

#endif

#ifdef RECYCLER\_FINALIZE\_CHECK

collectionStats.finalizeCount = 0;

#endif

RecyclerMemoryTracking::ReportRecyclerCreate(this);

#if DBG\_DUMP

forceTraceMark = false;

recyclerPageAllocator.debugName = L"Recycler";

recyclerLargeBlockPageAllocator.debugName = L"RecyclerLargeBlock";

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.debugName = L"RecyclerWithBarrier";

#endif

#endif

isHeapEnumInProgress = false;

isCollectionDisabled = false;

#if DBG

allowAllocationDuringRenentrance = false;

allowAllocationDuringHeapEnum = false;

#ifdef ENABLE\_PROJECTION

isInRefCountTrackingForProjection = false;

#endif

#endif

ScheduleNextCollection();

#if defined(RECYCLER\_DUMP\_OBJECT\_GRAPH) || defined(LEAK\_REPORT) || defined(CHECK\_MEMORY\_LEAK)

this->inDllCanUnloadNow = false;

this->inDetachProcess = false;

#endif

memset(&localTelemetryBlock, 0, sizeof(localTelemetryBlock));

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

// recycler requires at least Recycler::PrimaryMarkStackReservedPageCount to function properly for the main mark context

this->markContext.SetMaxPageCount(max<size\_t>(GetRecyclerFlagsTable().MaxMarkStackPageCount, Recycler::PrimaryMarkStackReservedPageCount));

this->parallelMarkContext1.SetMaxPageCount(GetRecyclerFlagsTable().MaxMarkStackPageCount);

this->parallelMarkContext2.SetMaxPageCount(GetRecyclerFlagsTable().MaxMarkStackPageCount);

this->parallelMarkContext3.SetMaxPageCount(GetRecyclerFlagsTable().MaxMarkStackPageCount);

if (GetRecyclerFlagsTable().IsEnabled(Js::GCMemoryThresholdFlag))

{

// Note, we can't do this in the constructor for RecyclerHeuristic::Instance because it runs before config is processed

RecyclerHeuristic::Instance.ConfigureBaseFactor(GetRecyclerFlagsTable().GCMemoryThreshold);

}

#endif

}

#if DBG

void

Recycler::SetDisableThreadAccessCheck()

{

recyclerPageAllocator.SetDisableThreadAccessCheck();

recyclerLargeBlockPageAllocator.SetDisableThreadAccessCheck();

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.SetDisableThreadAccessCheck();

#endif

disableThreadAccessCheck = true;

}

#endif

void

Recycler::SetMemProtectMode()

{

this->enableScanInteriorPointers = true;

this->enableScanImplicitRoots = true;

this->disableCollectOnAllocationHeuristics = true;

#ifdef RECYCLER\_STRESS

this->recyclerStress = GetRecyclerFlagsTable().MemProtectHeapStress;

#ifdef CONCURRENT\_GC\_ENABLED

this->recyclerBackgroundStress = GetRecyclerFlagsTable().MemProtectHeapBackgroundStress;

this->recyclerConcurrentStress = GetRecyclerFlagsTable().MemProtectHeapConcurrentStress;

this->recyclerConcurrentRepeatStress = GetRecyclerFlagsTable().MemProtectHeapConcurrentRepeatStress;

#endif

#ifdef PARTIAL\_GC\_ENABLED

this->recyclerPartialStress = GetRecyclerFlagsTable().MemProtectHeapPartialStress;

#endif

#endif

}

void

Recycler::LogMemProtectHeapSize(bool fromGC)

{

Assert(IsMemProtectMode());

#ifdef ENABLE\_JS\_ETW

if (IS\_JS\_ETW(EventEnabledMEMPROTECT\_GC\_HEAP\_SIZE()))

{

IdleDecommitPageAllocator\* recyclerPageAllocator = GetRecyclerPageAllocator();

IdleDecommitPageAllocator\* recyclerLeafPageAllocator = GetRecyclerLeafPageAllocator();

IdleDecommitPageAllocator\* recyclerLargeBlockPageAllocator = GetRecyclerLargeBlockPageAllocator();

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

IdleDecommitPageAllocator\* recyclerWithBarrierPageAllocator = GetRecyclerWithBarrierPageAllocator();

#endif

size\_t usedBytes = (recyclerPageAllocator->usedBytes + recyclerLeafPageAllocator->usedBytes +

recyclerLargeBlockPageAllocator->usedBytes);

size\_t reservedBytes = (recyclerPageAllocator->reservedBytes + recyclerLeafPageAllocator->reservedBytes +

recyclerLargeBlockPageAllocator->reservedBytes);

size\_t committedBytes = (recyclerPageAllocator->committedBytes + recyclerLeafPageAllocator->committedBytes +

recyclerLargeBlockPageAllocator->committedBytes);

size\_t numberOfSegments = (recyclerPageAllocator->numberOfSegments +

recyclerLeafPageAllocator->numberOfSegments +

recyclerLargeBlockPageAllocator->numberOfSegments);

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

usedBytes += recyclerWithBarrierPageAllocator->usedBytes;

reservedBytes += recyclerWithBarrierPageAllocator->reservedBytes;

committedBytes += recyclerWithBarrierPageAllocator->committedBytes;

numberOfSegments += recyclerWithBarrierPageAllocator->numberOfSegments;

#endif

JS\_ETW(EventWriteMEMPROTECT\_GC\_HEAP\_SIZE(this, usedBytes, reservedBytes, committedBytes, numberOfSegments, fromGC));

}

#endif

}

#if DBG

void

Recycler::SetDisableConcurentThreadExitedCheck()

{

disableConcurentThreadExitedCheck = true;

#ifdef RECYCLER\_STRESS

this->recyclerStress = false;

#ifdef CONCURRENT\_GC\_ENABLED

this->recyclerBackgroundStress = false;

this->recyclerConcurrentStress = false;

this->recyclerConcurrentRepeatStress = false;

#endif

#ifdef PARTIAL\_GC\_ENABLED

this->recyclerPartialStress = false;

#endif

#endif

}

#endif

#if DBG

void

Recycler::ResetThreadId()

{

// Transfer all the page allocator to the current thread id

ForRecyclerPageAllocator(ClearConcurrentThreadId());

#ifdef CONCURRENT\_GC\_ENABLED

if (this->IsConcurrentEnabled())

{

markContext.GetPageAllocator()->ClearConcurrentThreadId();

}

#endif

#if defined(DBG) && defined(PROFILE\_EXEC)

this->backgroundProfilerPageAllocator.ClearConcurrentThreadId();

#endif

}

#endif

Recycler::~Recycler()

{

Assert(!this->isAborting);

// Stop any further collection

this->isShuttingDown = true;

#if DBG

this->ResetThreadId();

#endif

#ifdef ENABLE\_JS\_ETW

FlushFreeRecord();

#endif

ClearObjectBeforeCollectCallbacks();

#ifdef RECYCLER\_DUMP\_OBJECT\_GRAPH

if (GetRecyclerFlagsTable().DumpObjectGraphOnExit)

{

// Always skip stack here, as we may be running the dtor on another thread.

RecyclerObjectGraphDumper::Param param = { 0 };

param.skipStack = true;

this->DumpObjectGraph(&param);

}

#endif

AUTO\_LEAK\_REPORT\_SECTION(this->GetRecyclerFlagsTable(), L"Recycler (%p): %s", this, this->IsInDllCanUnloadNow()? L"DllCanUnloadNow" :

this->IsInDetachProcess()? L"DetachProcess" : L"Destructor");

#ifdef LEAK\_REPORT

ReportLeaks();

#endif

#ifdef CHECK\_MEMORY\_LEAK

CheckLeaks(this->IsInDllCanUnloadNow()? L"DllCanUnloadNow" : this->IsInDetachProcess()? L"DetachProcess" : L"Destructor");

#endif

AUTO\_LEAK\_REPORT\_SECTION(this->GetRecyclerFlagsTable(), L"Skipped finalizers");

#ifdef CONCURRENT\_GC\_ENABLED

Assert(concurrentThread == nullptr);

// We only sometime clean up the state after abort concurrent to not collection

// Still need to delete heap block that is held by the recyclerSweep

if (recyclerSweep != nullptr)

{

recyclerSweep->ShutdownCleanup();

recyclerSweep = nullptr;

}

if (mainThreadHandle != nullptr)

{

CloseHandle(mainThreadHandle);

}

#endif

recyclerPageAllocator.Close();

recyclerLargeBlockPageAllocator.Close();

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.Close();

#endif

markContext.Release();

parallelMarkContext1.Release();

parallelMarkContext2.Release();

parallelMarkContext3.Release();

// Clean up the weak reference map so that

// objects being finalized can safely refer to weak references

// (this could otherwise become a problem for weak references held

// to large objects since their block would be destroyed before

// the finalizer was run)

// When the recycler is shutting down, all objects are going to be reclaimed

// so null out the weak references so that anyone relying on weak

// references simply thinks the object has been reclaimed

weakReferenceMap.Map([](RecyclerWeakReferenceBase \* weakRef) -> bool

{

weakRef->strongRef = nullptr;

// Put in a dummy heap block so that we can still do the isPendingConcurrentSweep check first.

weakRef->strongRefHeapBlock = &CollectedRecyclerWeakRefHeapBlock::Instance;

// Remove

return false;

});

#ifdef PARTIAL\_GC\_ENABLED

clientTrackedObjectList.Clear(&this->clientTrackedObjectAllocator);

#endif

#ifdef PROFILE\_RECYCLER\_ALLOC

if (trackerDictionary != nullptr)

{

this->trackerDictionary->Map([](type\_info const \*, TrackerItem \* item)

{

NoCheckHeapDelete(item);

});

NoCheckHeapDelete(this->trackerDictionary);

this->trackerDictionary = nullptr;

::DeleteCriticalSection(&trackerCriticalSection);

}

#endif

#ifdef RECYCLER\_MARK\_TRACK

NoCheckHeapDelete(this->markMap);

this->markMap = nullptr;

#endif

#if DBG

// Disable idle decommit asserts

ForRecyclerPageAllocator(ShutdownIdleDecommit());

#endif

Assert(this->collectionState == CollectionStateExit || this->collectionState == CollectionStateNotCollecting);

#ifdef CONCURRENT\_GC\_ENABLED

Assert(this->disableConcurentThreadExitedCheck || this->concurrentThreadExited == true);

#endif

}

void

Recycler::SetIsThreadBound()

{

Assert(mainThreadHandle == nullptr);

::DuplicateHandle(::GetCurrentProcess(), ::GetCurrentThread(), ::GetCurrentProcess(), &mainThreadHandle,

0, FALSE, DUPLICATE\_SAME\_ACCESS);

stackBase = GetStackBase();

}

void

Recycler::RootAddRef(void\* obj, uint \*count)

{

Assert(this->IsValidObject(obj));

if (transientPinnedObject)

{

PinRecord& refCount = pinnedObjectMap.GetReference(transientPinnedObject);

++refCount;

if (refCount == 1)

{

this->scanPinnedObjectMap = true;

RECYCLER\_PERF\_COUNTER\_INC(PinnedObject);

}

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

if (GetRecyclerFlagsTable().LeakStackTrace)

{

StackBackTraceNode::Prepend(&NoCheckHeapAllocator::Instance, refCount.stackBackTraces,

transientPinnedObjectStackBackTrace);

}

#endif

}

if (count != nullptr)

{

PinRecord\* refCount = pinnedObjectMap.TryGetReference(obj);

\*count = (refCount != nullptr) ? (\*refCount + 1) : 1;

}

transientPinnedObject = obj;

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

if (GetRecyclerFlagsTable().LeakStackTrace)

{

transientPinnedObjectStackBackTrace = StackBackTrace::Capture(&NoCheckHeapAllocator::Instance);

}

#endif

}

void

Recycler::RootRelease(void\* obj, uint \*count)

{

Assert(this->IsValidObject(obj));

if (transientPinnedObject == obj)

{

transientPinnedObject = nullptr;

if (count != nullptr)

{

PinRecord \*refCount = pinnedObjectMap.TryGetReference(obj);

\*count = (refCount != nullptr) ? \*refCount : 0;

}

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

if (GetRecyclerFlagsTable().LeakStackTrace)

{

transientPinnedObjectStackBackTrace->Delete(&NoCheckHeapAllocator::Instance);

}

#endif

}

else

{

PinRecord \*refCount = pinnedObjectMap.TryGetReference(obj);

if (refCount == nullptr)

{

if (count != nullptr)

{

\*count = (uint)-1;

}

// REVIEW: throw if not found

Assert(false);

return;

}

uint newRefCount = (--(\*refCount));

if (count != nullptr)

{

\*count = newRefCount;

}

if (newRefCount != 0)

{

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

if (GetRecyclerFlagsTable().LeakStackTrace)

{

StackBackTraceNode::Prepend(&NoCheckHeapAllocator::Instance, refCount->stackBackTraces,

StackBackTrace::Capture(&NoCheckHeapAllocator::Instance));

}

#endif

return;

}

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

StackBackTraceNode::DeleteAll(&NoCheckHeapAllocator::Instance, refCount->stackBackTraces);

refCount->stackBackTraces = nullptr;

#endif

// Don't delete the entry if we are in concurrent find root state

// We will delete it later on in-thread find root

if (this->hasPendingConcurrentFindRoot)

{

this->hasPendingUnpinnedObject = true;

}

else

{

pinnedObjectMap.Remove(obj);

}

RECYCLER\_PERF\_COUNTER\_DEC(PinnedObject);

}

// Not a real collection. This doesn't activate GC.

// This tell the GC that we have an exhaustive candidate, and should trigger

// another GC if there is an exhaustive GC going on.

this->CollectNow<CollectExhaustiveCandidate>();

}

void

Recycler::Initialize(const bool forceInThread, JsUtil::ThreadService \*threadService, const bool deferThreadStartup

#ifdef RECYCLER\_PAGE\_HEAP

, PageHeapMode pageheapmode

, bool captureAllocCallStack

, bool captureFreeCallStack

#endif

)

{

#ifdef PROFILE\_RECYCLER\_ALLOC

this->InitializeProfileAllocTracker();

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

this->disableCollection = CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::RecyclerPhase);

#endif

#ifdef CONCURRENT\_GC\_ENABLED

this->skipStack = false;

#endif

#ifdef PARTIAL\_GC\_ENABLED

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

this->enablePartialCollect = !CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::PartialCollectPhase);

#else

this->enablePartialCollect = true;

#endif

#endif

#ifdef PROFILE\_MEM

this->memoryData = MemoryProfiler::GetRecyclerMemoryData();

#endif

#if DBG || DBG\_DUMP || defined(RECYCLER\_TRACE)

mainThreadId = GetCurrentThreadContextId();

#endif

#ifdef RECYCLER\_TRACE

collectionParam.domCollect = false;

#endif

#if defined(PROFILE\_RECYCLER\_ALLOC) || defined(RECYCLER\_MEMORY\_VERIFY) || defined(MEMSPECT\_TRACKING) || defined(ETW\_MEMORY\_TRACKING)

bool dontNeedDetailedTracking = false;

#if defined(PROFILE\_RECYCLER\_ALLOC)

dontNeedDetailedTracking = dontNeedDetailedTracking || this->trackerDictionary == nullptr;

#endif

#if defined(RECYCLER\_MEMORY\_VERIFY)

dontNeedDetailedTracking = dontNeedDetailedTracking || !this->verifyEnabled;

#endif

// If we need detailed tracking we force allocation fast path in the JIT to fail and go to the helper, so there is no

// need for the TrackNativeAllocatedMemoryBlock callback.

if (dontNeedDetailedTracking)

{

autoHeap.Initialize(this, TrackNativeAllocatedMemoryBlock

#ifdef RECYCLER\_PAGE\_HEAP

, pageheapmode

, captureAllocCallStack

, captureFreeCallStack

#endif

);

}

else

{

autoHeap.Initialize(this

#ifdef RECYCLER\_PAGE\_HEAP

, pageheapmode

, captureAllocCallStack

, captureFreeCallStack

#endif

);

}

#else

autoHeap.Initialize(this

#ifdef RECYCLER\_PAGE\_HEAP

, pageheapmode

, captureAllocCallStack

, captureFreeCallStack

#endif

);

#endif

markContext.Init(Recycler::PrimaryMarkStackReservedPageCount);

#if defined(RECYCLER\_DUMP\_OBJECT\_GRAPH) || defined(LEAK\_REPORT) || defined(CHECK\_MEMORY\_LEAK)

isPrimaryMarkContextInitialized = true;

#endif

#ifdef RECYCLER\_PAGE\_HEAP

isPageHeapEnabled = autoHeap.IsPageHeapEnabled();

if (IsPageHeapEnabled())

{

capturePageHeapAllocStack = autoHeap.captureAllocCallStack;

capturePageHeapFreeStack = autoHeap.captureFreeCallStack;

}

#endif

#ifdef RECYCLER\_STRESS

if (GetRecyclerFlagsTable().RecyclerTrackStress)

{

// Disable partial if we are doing track stress, since partial relies on ClientTracked processing

// and track stress doesn't support this.

this->enablePartialCollect = false;

}

this->recyclerStress = GetRecyclerFlagsTable().RecyclerStress;

#ifdef CONCURRENT\_GC\_ENABLED

this->recyclerBackgroundStress = GetRecyclerFlagsTable().RecyclerBackgroundStress;

this->recyclerConcurrentStress = GetRecyclerFlagsTable().RecyclerConcurrentStress;

this->recyclerConcurrentRepeatStress = GetRecyclerFlagsTable().RecyclerConcurrentRepeatStress;

#endif

#ifdef PARTIAL\_GC\_ENABLED

this->recyclerPartialStress = GetRecyclerFlagsTable().RecyclerPartialStress;

#endif

#endif

bool needWriteWatch = false;

#ifdef CONCURRENT\_GC\_ENABLED

// Default to non-concurrent

uint numProcs = (uint)AutoSystemInfo::Data.GetNumberOfPhysicalProcessors();

this->maxParallelism = (numProcs > 4) || CUSTOM\_PHASE\_FORCE1(GetRecyclerFlagsTable(), Js::ParallelMarkPhase) ? 4 : numProcs;

if (forceInThread)

{

// Requested a non-concurrent recycler

this->disableConcurrent = true;

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

else if (CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::ConcurrentCollectPhase))

{

// Concurrent collection disabled

this->disableConcurrent = true;

}

else if (CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::ConcurrentMarkPhase) &&

CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::ParallelMarkPhase) &&

CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::ConcurrentSweepPhase))

{

// All concurrent collection phases disabled

this->disableConcurrent = true;

}

#endif

else

{

this->disableConcurrent = false;

if (deferThreadStartup || EnableConcurrent(threadService, false))

{

needWriteWatch = true;

}

}

#endif

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

needWriteWatch = true;

}

#endif

if (needWriteWatch)

{

// need write watch to support concurrent and/or partial collection

recyclerPageAllocator.EnableWriteWatch();

recyclerLargeBlockPageAllocator.EnableWriteWatch();

}

}

#if DBG

BOOL

Recycler::IsFreeObject(void \* candidate)

{

HeapBlock \* heapBlock = this->FindHeapBlock(candidate);

if (heapBlock != NULL)

{

return heapBlock->IsFreeObject(candidate);

}

return false;

}

#endif

BOOL

Recycler::IsValidObject(void\* candidate, size\_t minimumSize)

{

HeapBlock \* heapBlock = this->FindHeapBlock(candidate);

if (heapBlock != NULL)

{

return heapBlock->IsValidObject(candidate) && (minimumSize == 0 || heapBlock->GetObjectSize(candidate) >= minimumSize);

}

return false;

}

void

Recycler::Prime()

{

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (GetRecyclerFlagsTable().IsEnabled(Js::ForceFragmentAddressSpaceFlag))

{

// Never prime the recycler if we are forced to fragment address space

return;

}

#endif

ForRecyclerPageAllocator(Prime(RecyclerPageAllocator::DefaultPrimePageCount));

}

void

Recycler::AddExternalMemoryUsage(size\_t size)

{

this->autoHeap.uncollectedAllocBytes += size;

this->autoHeap.uncollectedExternalBytes += size;

// Generally normal GC can cleanup the uncollectedAllocBytes. But if external components

// do fast large allocations in a row, normal GC might not kick in. Let's force the GC

// here if we need to collect anyhow.

CollectNow<CollectOnAllocation>();

}

BOOL Recycler::ReportExternalMemoryAllocation(size\_t size)

{

return recyclerPageAllocator.RequestAlloc(size);

}

void Recycler::ReportExternalMemoryFailure(size\_t size)

{

recyclerPageAllocator.ReportFailure(size);

}

void Recycler::ReportExternalMemoryFree(size\_t size)

{

recyclerPageAllocator.ReportFree(size);

}

/\*------------------------------------------------------------------------------------------------

\* Idle Decommit

\*------------------------------------------------------------------------------------------------\*/

void

Recycler::EnterIdleDecommit()

{

ForRecyclerPageAllocator(EnterIdleDecommit());

#ifdef IDLE\_DECOMMIT\_ENABLED

::InterlockedCompareExchange(&needIdleDecommitSignal, IdleDecommitSignal\_None, IdleDecommitSignal\_NeedTimer);

#endif

}

void

Recycler::LeaveIdleDecommit()

{

#ifdef IDLE\_DECOMMIT\_ENABLED

bool allowTimer = (this->concurrentIdleDecommitEvent != nullptr);

IdleDecommitSignal idleDecommitSignalRecycler = recyclerPageAllocator.LeaveIdleDecommit(allowTimer);

IdleDecommitSignal idleDecommitSignalRecyclerLargeBlock = recyclerLargeBlockPageAllocator.LeaveIdleDecommit(allowTimer);

IdleDecommitSignal idleDecommitSignal = max(idleDecommitSignalRecycler, idleDecommitSignalRecyclerLargeBlock);

IdleDecommitSignal idleDecommitSignalThread = threadPageAllocator->LeaveIdleDecommit(allowTimer);

idleDecommitSignal = max(idleDecommitSignal, idleDecommitSignalThread);

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

IdleDecommitSignal idleDecommitSignalRecyclerWithBarrier = recyclerWithBarrierPageAllocator.LeaveIdleDecommit(allowTimer);

idleDecommitSignal = max(idleDecommitSignal, idleDecommitSignalRecyclerWithBarrier);

#endif

if (idleDecommitSignal != IdleDecommitSignal\_None)

{

Assert(allowTimer);

// Reduce the number of times we need to signal the background thread

// by detecting whether the thread is waiting on a time out or not

if (idleDecommitSignal == IdleDecommitSignal\_NeedSignal ||

::InterlockedCompareExchange(&needIdleDecommitSignal, IdleDecommitSignal\_NeedTimer, IdleDecommitSignal\_None) == IdleDecommitSignal\_NeedSignal)

{

#if DBG

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::IdleDecommitPhase))

{

Output::Print(L"Recycler Thread IdleDecommit Need Signal\n");

Output::Flush();

}

#endif

SetEvent(this->concurrentIdleDecommitEvent);

}

}

#else

ForEachRecyclerPageAllocatorIn(this, LeaveIdleDecommit(false));

#endif

}

/\*------------------------------------------------------------------------------------------------

\* Freeing

\*------------------------------------------------------------------------------------------------\*/

bool Recycler::ExplicitFreeLeaf(void\* buffer, size\_t size)

{

return ExplicitFreeInternalWrapper<ObjectInfoBits::LeafBit>(buffer, size);

}

bool Recycler::ExplicitFreeNonLeaf(void\* buffer, size\_t size)

{

return ExplicitFreeInternalWrapper<ObjectInfoBits::NoBit>(buffer, size);

}

size\_t Recycler::GetAllocSize(size\_t size)

{

size\_t allocSize = size;

#ifdef RECYCLER\_MEMORY\_VERIFY

if (this->VerifyEnabled())

{

allocSize += verifyPad + sizeof(size\_t);

Assert(allocSize > size);

}

#endif

return allocSize;

}

template <typename TBlockAttributes>

void Recycler::SetExplicitFreeBitOnSmallBlock(HeapBlock\* heapBlock, size\_t sizeCat, void\* buffer, ObjectInfoBits attributes)

{

Assert(!heapBlock->IsLargeHeapBlock());

Assert(heapBlock->GetObjectSize(buffer) == sizeCat);

SmallHeapBlockT<TBlockAttributes>\* smallBlock = (SmallHeapBlockT<TBlockAttributes>\*)heapBlock;

if ((attributes & ObjectInfoBits::LeafBit) == LeafBit)

{

Assert(smallBlock->IsLeafBlock());

}

else

{

Assert(smallBlock->IsAnyNormalBlock());

}

#ifdef RECYCLER\_MEMORY\_VERIFY

smallBlock->SetExplicitFreeBitForObject(buffer);

#endif

}

template <ObjectInfoBits attributes>

bool Recycler::ExplicitFreeInternalWrapper(void\* buffer, size\_t size)

{

Assert(buffer != nullptr);

Assert(size > 0);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::ExplicitFreePhase))

{

return false;

}

#endif

size\_t allocSize = GetAllocSize(size);

if (HeapInfo::IsSmallObject(allocSize))

{

return ExplicitFreeInternal<attributes, SmallAllocationBlockAttributes>(buffer, size, HeapInfo::GetAlignedSizeNoCheck(allocSize));

}

if (HeapInfo::IsMediumObject(allocSize))

{

return ExplicitFreeInternal<attributes, MediumAllocationBlockAttributes>(buffer, size, HeapInfo::GetMediumObjectAlignedSizeNoCheck(allocSize));

}

return false;

}

template <ObjectInfoBits attributes, typename TBlockAttributes>

bool Recycler::ExplicitFreeInternal(void\* buffer, size\_t size, size\_t sizeCat)

{

// If the GC is in sweep state while FreeInternal is called, we might be executing a finalizer

// which called Free, which would cause a "sweepable" buffer to be free-listed. Don't allow this.

// Also don't allow freeing while we're shutting down the recycler since finalizers get executed

// at this stage too

if (this->IsSweeping() || this->IsExiting())

{

return false;

}

// We shouldn't be freeing object when we are running GC in thread

Assert(this->IsConcurrentState() || !this->CollectionInProgress() || this->collectionState == CollectionStatePostCollectionCallback);

DebugOnly(RecyclerHeapObjectInfo info);

Assert(this->FindHeapObject(buffer, FindHeapObjectFlags\_NoFreeBitVerify, info));

Assert((info.GetAttributes() & ~ObjectInfoBits::LeafBit) == 0); // Only NoBit or LeafBit

#if DBG || defined(RECYCLER\_MEMORY\_VERIFY) || defined(RECYCLER\_PAGE\_HEAP)

// Either the mainThreadHandle is null (we're not thread bound)

// or we should be calling this function on the main script thread

Assert(this->mainThreadHandle == NULL ||

::GetCurrentThreadId() == ::GetThreadId(this->mainThreadHandle));

HeapBlock\* heapBlock = this->FindHeapBlock(buffer);

Assert(heapBlock != nullptr);

#ifdef RECYCLER\_PAGE\_HEAP

if (this->IsPageHeapEnabled() && this->ShouldCapturePageHeapFreeStack())

{

heapBlock->CapturePageHeapFreeStack();

// Don't do actual explicit free in page heap mode

return false;

}

#endif

SetExplicitFreeBitOnSmallBlock<TBlockAttributes>(heapBlock, sizeCat, buffer, attributes);

#endif

if (TBlockAttributes::IsMediumBlock)

{

autoHeap.FreeMediumObject<attributes>(buffer, sizeCat);

}

else

{

autoHeap.FreeSmallObject<attributes>(buffer, sizeCat);

}

if (size > sizeof(FreeObject) || TBlockAttributes::IsMediumBlock)

{

// Do this on the background somehow?

byte expectedFill = 0;

size\_t fillSize = size - sizeof(FreeObject);

#ifdef RECYCLER\_MEMORY\_VERIFY

if (this->VerifyEnabled())

{

expectedFill = Recycler::VerifyMemFill;

}

#endif

memset(((char\*)buffer) + sizeof(FreeObject), expectedFill, fillSize);

}

#ifdef PROFILE\_RECYCLER\_ALLOC

if (this->trackerDictionary != nullptr)

{

this->SetTrackerData(buffer, &TrackerData::ExplicitFreeListObjectData);

}

#endif

return true;

}

/\*------------------------------------------------------------------------------------------------

\* Allocation

\*------------------------------------------------------------------------------------------------\*/

char \*

Recycler::TryLargeAlloc(HeapInfo \* heap, size\_t size, ObjectInfoBits attributes, bool nothrow)

{

Assert((attributes & InternalObjectInfoBitMask) == attributes);

Assert(size != 0);

size\_t sizeCat = HeapInfo::GetAlignedSizeNoCheck(size);

if (sizeCat == 0)

{

// overflow scenario

// if onthrow is false, throw out of memory

// otherwise, return null

if (nothrow == false)

{

this->OutOfMemory();

}

return nullptr;

}

char \* memBlock;

if (heap->largeObjectBucket.largeBlockList != nullptr)

{

memBlock = heap->largeObjectBucket.largeBlockList->Alloc(sizeCat, attributes);

if (memBlock != nullptr)

{

#ifdef RECYCLER\_ZERO\_MEM\_CHECK

VerifyZeroFill(memBlock, sizeCat);

#endif

return memBlock;

}

}

// We don't care whether a GC happened here or not, because we are not reusing freed

// large objects. We might try to allocate from existing block if we implement

// large object reuse.

if (!this->disableCollectOnAllocationHeuristics)

{

CollectNow<CollectOnAllocation>();

}

#ifdef RECYCLER\_PAGE\_HEAP

if (IsPageHeapEnabled())

{

if (heap->largeObjectBucket.IsPageHeapEnabled())

{

memBlock = heap->largeObjectBucket.PageHeapAlloc(this, size, (ObjectInfoBits)attributes, autoHeap.pageHeapMode, nothrow);

if (memBlock != nullptr)

{

#ifdef RECYCLER\_ZERO\_MEM\_CHECK

VerifyZeroFill(memBlock, sizeCat);

#endif

return memBlock;

}

}

}

#endif

LargeHeapBlock \* heapBlock = heap->AddLargeHeapBlock(sizeCat);

if (heapBlock == nullptr)

{

return nullptr;

}

memBlock = heapBlock->Alloc(sizeCat, attributes);

Assert(memBlock != nullptr);

#ifdef RECYCLER\_ZERO\_MEM\_CHECK

VerifyZeroFill(memBlock, sizeCat);

#endif

return memBlock;

}

template <bool nothrow>

char\*

Recycler::LargeAlloc(HeapInfo\* heap, size\_t size, ObjectInfoBits attributes)

{

Assert((attributes & InternalObjectInfoBitMask) == attributes);

char \* addr = TryLargeAlloc(heap, size, attributes, nothrow);

if (addr == nullptr)

{

// Force a collection and try to allocate again.

this->CollectNow<CollectNowForceInThread>();

addr = TryLargeAlloc(heap, size, attributes, nothrow);

if (addr == nullptr)

{

if (nothrow == false)

{

// Still fails, we are out of memory

// Since nothrow is false, it's okay to throw here

this->OutOfMemory();

}

else

{

return nullptr;

}

}

}

autoHeap.uncollectedAllocBytes += size;

return addr;

}

// Explicitly instantiate both versions of LargeAlloc

template char\* Recycler::LargeAlloc<true>(HeapInfo\* heap, size\_t size, ObjectInfoBits attributes);

template char\* Recycler::LargeAlloc<false>(HeapInfo\* heap, size\_t size, ObjectInfoBits attributes);

void

Recycler::OutOfMemory()

{

outOfMemoryFunc();

}

void Recycler::GetNormalHeapBlockAllocatorInfoForNativeAllocation(size\_t allocSize, void\*& allocatorAddress, uint32& endAddressOffset, uint32& freeListOffset)

{

Assert(HeapInfo::IsAlignedSize(allocSize));

Assert(HeapInfo::IsSmallObject(allocSize));

allocatorAddress = GetAddressOfAllocator<NoBit>(allocSize);

endAddressOffset = GetEndAddressOffset<NoBit>(allocSize);

freeListOffset = GetFreeObjectListOffset<NoBit>(allocSize);

if (!AllowNativeCodeBumpAllocation())

{

freeListOffset = endAddressOffset;

}

}

bool Recycler::AllowNativeCodeBumpAllocation()

{

// In debug builds, if we need to track allocation info, we pretend there is no pointer-bump-allocation space

// on this page, so that we always fail the check in native code and go to helper, which does the tracking.

#ifdef PROFILE\_RECYCLER\_ALLOC

if (this->trackerDictionary != nullptr)

{

return false;

}

#endif

#ifdef RECYCLER\_MEMORY\_VERIFY

if (this->verifyEnabled)

{

return false;

}

#endif

#ifdef RECYCLER\_PAGE\_HEAP

// Don't allow bump allocation in the JIT when page heap is turned on

if (this->IsPageHeapEnabled())

{

return false;

}

#endif

return true;

}

void Recycler::TrackNativeAllocatedMemoryBlock(Recycler \* recycler, void \* memBlock, size\_t sizeCat)

{

Assert(HeapInfo::IsAlignedSize(sizeCat));

Assert(HeapInfo::IsSmallObject(sizeCat));

#ifdef PROFILE\_RECYCLER\_ALLOC

AssertMsg(!Recycler::DoProfileAllocTracker(), "Why did we register allocation tracking callback if all allocations are forced to slow path?");

#endif

RecyclerMemoryTracking::ReportAllocation(recycler, memBlock, sizeCat);

RECYCLER\_PERF\_COUNTER\_INC(LiveObject);

RECYCLER\_PERF\_COUNTER\_ADD(LiveObjectSize, sizeCat);

RECYCLER\_PERF\_COUNTER\_SUB(FreeObjectSize, sizeCat);

#ifdef RECYCLER\_MEMORY\_VERIFY

AssertMsg(!recycler->VerifyEnabled(), "Why did we register allocation tracking callback if all allocations are forced to slow path?");

#endif

}

/\*------------------------------------------------------------------------------------------------

\* FindRoots

\*------------------------------------------------------------------------------------------------\*/

static void\* GetStackBase()

{

return ((NT\_TIB \*)NtCurrentTeb())->StackBase;

}

#if \_M\_IX86

// REVIEW: For x86, do we care about scanning esp/ebp?

// At GC time, they shouldn't be pointing to GC memory.

#define SAVE\_THREAD\_CONTEXT() \

void\*\* targetBuffer = this->savedThreadContext.GetRegisters(); \

\_\_asm { push eax } \

\_\_asm { mov eax, targetBuffer } \

\_\_asm { mov [eax], esp} \

\_\_asm { mov [eax+0x4], eax} \

\_\_asm { mov [eax+0x8], ebx} \

\_\_asm { mov [eax+0xc], ecx} \

\_\_asm { mov [eax+0x10], edx} \

\_\_asm { mov [eax+0x14], ebp} \

\_\_asm { mov [eax+0x18], esi} \

\_\_asm { mov [eax+0x1c], edi} \

\_\_asm { pop eax }

#elif \_M\_ARM

#define SAVE\_THREAD\_CONTEXT() arm\_SAVE\_REGISTERS(this->savedThreadContext.GetRegisters());

#elif \_M\_ARM64

#define SAVE\_THREAD\_CONTEXT() arm64\_SAVE\_REGISTERS(this->savedThreadContext.GetRegisters());

#elif \_M\_AMD64

#define SAVE\_THREAD\_CONTEXT() amd64\_SAVE\_REGISTERS(this->savedThreadContext.GetRegisters());

#else

#error Unexpected architecture

#endif

size\_t

Recycler::ScanArena(ArenaData \* alloc, bool background)

{

#if DBG\_DUMP

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::MarkPhase)

|| GetRecyclerFlagsTable().Trace.IsEnabled(Js::FindRootPhase))

{

this->forceTraceMark = true;

Output::Print(L"Scanning Guest Arena %p: ", alloc);

}

#endif

size\_t scanRootBytes = 0;

BEGIN\_DUMP\_OBJECT\_ADDRESS(L"Guest Arena", alloc);

#if defined(PARTIAL\_GC\_ENABLED) || defined(CONCURRENT\_GC\_ENABLED)

// The new write watch batching logic broke the write watch handling here.

// For now, just disable write watch for guest arenas.

// Re-enable this in the future.

#if FALSE

// Note, guest arenas are allocated out of the large block page allocator.

bool writeWatch = alloc->GetPageAllocator() == &this->recyclerLargeBlockPageAllocator;

// Only use write watch when we are doing rescan (Partial collect or finish concurrent)

if (writeWatch && this->collectionState == CollectionStateRescanFindRoots)

{

scanRootBytes += TryMarkBigBlockListWithWriteWatch(alloc->GetBigBlocks(background));

scanRootBytes += TryMarkBigBlockListWithWriteWatch(alloc->GetFullBlocks());

}

else

#endif

#endif

{

scanRootBytes += TryMarkBigBlockList(alloc->GetBigBlocks(background));

scanRootBytes += TryMarkBigBlockList(alloc->GetFullBlocks());

}

scanRootBytes += TryMarkArenaMemoryBlockList(alloc->GetMemoryBlocks());

END\_DUMP\_OBJECT(this);

#if DBG\_DUMP

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::MarkPhase)

|| GetRecyclerFlagsTable().Trace.IsEnabled(Js::FindRootPhase))

{

this->forceTraceMark = false;

Output::Print(L"\n");

Output::Flush();

}

#endif

// The arena has been scanned so the full blocks can be rearranged at this point

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (background || !GetRecyclerFlagsTable().RecyclerProtectPagesOnRescan)

#endif

{

alloc->SetLockBlockList(false);

}

return scanRootBytes;

}

#if DBG

bool

Recycler::ExpectStackSkip() const

{

// Okay to skip the stack scan if we're in leak check mode

bool expectStackSkip = false;

#ifdef LEAK\_REPORT

expectStackSkip = expectStackSkip || GetRecyclerFlagsTable().IsEnabled(Js::LeakReportFlag);

#endif

#ifdef CHECK\_MEMORY\_LEAK

expectStackSkip = expectStackSkip || GetRecyclerFlagsTable().CheckMemoryLeak;

#endif

#ifdef RECYCLER\_DUMP\_OBJECT\_GRAPH

expectStackSkip = expectStackSkip || (this->objectGraphDumper != nullptr);

#endif

#if defined(INTERNAL\_MEM\_PROTECT\_HEAP\_ALLOC)

expectStackSkip = expectStackSkip || GetRecyclerFlagsTable().MemProtectHeap;

#endif

return expectStackSkip || isExternalStackSkippingGC;

}

#endif

#pragma warning(push)

#pragma warning(disable:4731) // 'pointer' : frame pointer register 'register' modified by inline assembly code

size\_t

Recycler::ScanStack()

{

#ifdef CONCURRENT\_GC\_ENABLED

if (this->skipStack)

{

#ifdef RECYCLER\_TRACE

CUSTOM\_PHASE\_PRINT\_VERBOSE\_TRACE1(GetRecyclerFlagsTable(), Js::ScanStackPhase, L"[%04X] Skipping the stack scan\n", ::GetCurrentThreadId());

#endif

Assert(this->isFinishGCOnIdle || this->isConcurrentGCOnIdle || this->ExpectStackSkip());

return 0;

}

#endif

#ifdef RECYCLER\_STATS

size\_t lastMarkCount = this->collectionStats.markData.markCount;

#endif

GCETW(GC\_SCANSTACK\_START, (this));

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::ScanStackPhase);

SAVE\_THREAD\_CONTEXT();

void \* stackTop = this->savedThreadContext.GetStackTop();

void \* stackStart = GetStackBase();

Assert(stackStart > stackTop);

size\_t stackScanned = (size\_t)((char \*)stackStart - (char \*)stackTop);

#if DBG\_DUMP

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::MarkPhase)

|| GetRecyclerFlagsTable().Trace.IsEnabled(Js::ScanStackPhase))

{

this->forceTraceMark = true;

Output::Print(L"Scanning Stack %p(%8d): ", stackTop, (char \*)stackStart - (char \*)stackTop);

}

#endif

BEGIN\_DUMP\_OBJECT(this, L"Registers");

ScanMemoryInline(this->savedThreadContext.GetRegisters(), sizeof(void\*) \* SavedRegisterState::NumRegistersToSave);

END\_DUMP\_OBJECT(this);

BEGIN\_DUMP\_OBJECT(this, L"Stack");

ScanMemoryInline((void\*\*) stackTop, stackScanned);

END\_DUMP\_OBJECT(this);

#if DBG\_DUMP

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::MarkPhase)

|| GetRecyclerFlagsTable().Trace.IsEnabled(Js::ScanStackPhase))

{

this->forceTraceMark = false;

Output::Print(L"\n");

Output::Flush();

}

#endif

RECYCLER\_PROFILE\_EXEC\_END(this, Js::ScanStackPhase);

RECYCLER\_STATS\_ADD(this, stackCount, this->collectionStats.markData.markCount - lastMarkCount);

GCETW(GC\_SCANSTACK\_STOP, (this));

return stackScanned;

}

#pragma warning(pop)

template <bool background>

size\_t Recycler::ScanPinnedObjects()

{

size\_t scanRootBytes = 0;

BEGIN\_DUMP\_OBJECT(this, L"Pinned");

{

this->TryMarkNonInterior(transientPinnedObject, &transientPinnedObject /\* parentReference \*/);

if (this->scanPinnedObjectMap)

{

// We are scanning the pinned object map now, we don't need to rescan unless

// we reset mark or we add stuff to the map in Recycler::AddRef

this->scanPinnedObjectMap = false;

pinnedObjectMap.MapAndRemoveIf([this, &scanRootBytes](void \* obj, PinRecord const& refCount)

{

if (refCount == 0)

{

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

Assert(refCount.stackBackTraces == nullptr);

#endif

// Only remove if we are not doing this in the background.

return !background;

}

this->TryMarkNonInterior(obj, static\_cast<void\*>(const\_cast<PinRecord\*>(&refCount)) /\* parentReference \*/);

scanRootBytes += sizeof(void \*);

return false;

});

if (!background)

{

this->hasPendingUnpinnedObject = false;

}

}

}

END\_DUMP\_OBJECT(this);

if (background)

{

// Re-enable resize now that we are done

pinnedObjectMap.EnableResize();

}

return scanRootBytes;

}

void

RecyclerScanMemoryCallback::operator()(void\*\* obj, size\_t byteCount)

{

this->recycler->ScanMemoryInline(obj, byteCount);

}

size\_t

Recycler::FindRoots()

{

size\_t scanRootBytes = 0;

#ifdef RECYCLER\_STATS

size\_t lastMarkCount = this->collectionStats.markData.markCount;

#endif

GCETW(GC\_SCANROOTS\_START, (this));

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::FindRootPhase);

#ifdef ENABLE\_PROJECTION

{

AUTO\_TIMESTAMP(externalWeakReferenceObjectResolve);

BEGIN\_DUMP\_OBJECT(this, L"External Weak Referenced Roots");

Assert(!this->IsInRefCountTrackingForProjection());

#if DBG

AutoIsInRefCountTrackingForProjection autoIsInRefCountTrackingForProjection(this);

#endif

collectionWrapper->MarkExternalWeakReferencedObjects(this->inPartialCollectMode);

END\_DUMP\_OBJECT(this);

}

#endif

// go through ITracker\* stuff. Don't need to do it if we are doing a partial collection

// as we keep track and mark all trackable objects.

// Do this first because the host might unpin stuff in the process

if (externalRootMarker != NULL)

{

#ifdef PARTIAL\_GC\_ENABLED

if (!this->inPartialCollectMode)

#endif

{

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::FindRootExtPhase);

#if DBG\_DUMP

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::MarkPhase)

|| GetRecyclerFlagsTable().Trace.IsEnabled(Js::FindRootPhase))

{

this->forceTraceMark = true;

Output::Print(L"Scanning External Roots: ");

}

#endif

BEGIN\_DUMP\_OBJECT(this, L"External Roots");

// PARTIALGC-TODO: How do we count external roots?

externalRootMarker(externalRootMarkerContext);

END\_DUMP\_OBJECT(this);

#if DBG\_DUMP

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::MarkPhase)

|| GetRecyclerFlagsTable().Trace.IsEnabled(Js::FindRootPhase))

{

this->forceTraceMark = false;

Output::Print(L"\n");

Output::Flush();

}

#endif

RECYCLER\_PROFILE\_EXEC\_END(this, Js::FindRootExtPhase);

}

}

#if DBG\_DUMP

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::MarkPhase)

|| GetRecyclerFlagsTable().Trace.IsEnabled(Js::FindRootPhase))

{

this->forceTraceMark = true;

Output::Print(L"Scanning Pinned Objects: ");

}

#endif

scanRootBytes += this->ScanPinnedObjects</\*background = \*/false>();

#if DBG\_DUMP

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::MarkPhase)

|| GetRecyclerFlagsTable().Trace.IsEnabled(Js::FindRootPhase))

{

this->forceTraceMark = false;

Output::Print(L"\n");

Output::Flush();

}

#endif

Assert(!this->hasPendingConcurrentFindRoot);

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::FindRootArenaPhase);

DListBase<GuestArenaAllocator>::EditingIterator guestArenaIter(&guestArenaList);

while (guestArenaIter.Next())

{

GuestArenaAllocator& allocator = guestArenaIter.Data();

if (allocator.pendingDelete)

{

Assert(this->hasPendingDeleteGuestArena);

allocator.SetLockBlockList(false);

guestArenaIter.RemoveCurrent(&HeapAllocator::Instance);

}

else if (this->backgroundFinishMarkCount == 0)

{

// Only scan arena if we haven't finished mark in the background

scanRootBytes += ScanArena(&allocator, false);

}

}

this->hasPendingDeleteGuestArena = false;

DList<ArenaData \*, HeapAllocator>::Iterator externalGuestArenaIter(&externalGuestArenaList);

while (externalGuestArenaIter.Next())

{

scanRootBytes += ScanArena(externalGuestArenaIter.Data(), false);

}

RECYCLER\_PROFILE\_EXEC\_END(this, Js::FindRootArenaPhase);

this->ScanImplicitRoots();

RECYCLER\_PROFILE\_EXEC\_END(this, Js::FindRootPhase);

GCETW(GC\_SCANROOTS\_STOP, (this));

RECYCLER\_STATS\_ADD(this, rootCount, this->collectionStats.markData.markCount - lastMarkCount);

return scanRootBytes;

}

void

Recycler::ScanImplicitRoots()

{

if (this->enableScanImplicitRoots)

{

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::FindImplicitRootPhase);

if (!this->hasScannedInitialImplicitRoots)

{

this->ScanInitialImplicitRoots();

this->hasScannedInitialImplicitRoots = true;

}

else

{

this->ScanNewImplicitRoots();

}

RECYCLER\_PROFILE\_EXEC\_END(this, Js::FindImplicitRootPhase);

}

}

size\_t

Recycler::TryMarkArenaMemoryBlockList(ArenaMemoryBlock \* memoryBlocks)

{

size\_t scanRootBytes = 0;

ArenaMemoryBlock \*blockp = memoryBlocks;

while (blockp != NULL)

{

void\*\* base=(void\*\*)blockp->GetBytes();

size\_t byteCount = blockp->nbytes;

scanRootBytes += byteCount;

this->ScanMemory(base, byteCount);

blockp = blockp->next;

}

return scanRootBytes;

}

size\_t

Recycler::TryMarkBigBlockListWithWriteWatch(BigBlock \* memoryBlocks)

{

DWORD pageSize = AutoSystemInfo::PageSize;

size\_t scanRootBytes = 0;

BigBlock \*blockp = memoryBlocks;

// Reset the write watch bit if we are scanning this in the background thread

DWORD const writeWatchFlags = this->IsConcurrentFindRootState()? WRITE\_WATCH\_FLAG\_RESET : 0;

while (blockp != NULL)

{

char \* currentAddress = (char \*)blockp->GetBytes();

char \* endAddress = currentAddress + blockp->currentByte;

char \* currentPageStart = (char \*)blockp->allocation;

while (currentAddress < endAddress)

{

void \* written;

ULONG\_PTR count = 1;

if (::GetWriteWatch(writeWatchFlags, currentPageStart, AutoSystemInfo::PageSize, &written, &count, &pageSize) != 0 || count == 1)

{

char \* currentEnd = min(currentPageStart + pageSize, endAddress);

size\_t byteCount = (size\_t)(currentEnd - currentAddress);

scanRootBytes += byteCount;

this->ScanMemory((void \*\*)currentAddress, byteCount);

}

currentPageStart += pageSize;

currentAddress = currentPageStart;

}

blockp = blockp->nextBigBlock;

}

return scanRootBytes;

}

size\_t

Recycler::TryMarkBigBlockList(BigBlock \* memoryBlocks)

{

size\_t scanRootBytes = 0;

BigBlock \*blockp = memoryBlocks;

while (blockp != NULL)

{

void\*\* base = (void\*\*)blockp->GetBytes();

size\_t byteCount = blockp->currentByte;

scanRootBytes += byteCount;

this->ScanMemory(base, byteCount);

blockp = blockp->nextBigBlock;

}

return scanRootBytes;

}

void

Recycler::ScanInitialImplicitRoots()

{

autoHeap.ScanInitialImplicitRoots();

}

void

Recycler::ScanNewImplicitRoots()

{

autoHeap.ScanNewImplicitRoots();

}

/\*------------------------------------------------------------------------------------------------

\* Mark

\*------------------------------------------------------------------------------------------------\*/

void

Recycler::ResetMarks(ResetMarkFlags flags)

{

Assert(!this->CollectionInProgress());

collectionState = CollectionStateResetMarks;

RecyclerVerboseTrace(GetRecyclerFlagsTable(), L"Reset marks\n");

GCETW(GC\_RESETMARKS\_START, (this));

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::ResetMarksPhase);

Assert(IsMarkStackEmpty());

this->scanPinnedObjectMap = true;

this->hasScannedInitialImplicitRoots = false;

heapBlockMap.ResetMarks();

autoHeap.ResetMarks(flags);

RECYCLER\_PROFILE\_EXEC\_END(this, Js::ResetMarksPhase);

GCETW(GC\_RESETMARKS\_STOP, (this));

#ifdef RECYCLER\_MARK\_TRACK

this->ClearMarkMap();

#endif

}

#ifdef RECYCLER\_MARK\_TRACK

void Recycler::ClearMarkMap()

{

this->markMap->Clear();

}

void Recycler::PrintMarkMap()

{

this->markMap->Map([](void\* key, void\* value)

{

Output::Print(L"0x%P => 0x%P\n", key, value);

});

}

#endif

#if DBG

void

Recycler::CheckAllocExternalMark() const

{

Assert(!disableThreadAccessCheck);

Assert(GetCurrentThreadContextId() == mainThreadId);

#ifdef CONCURRENT\_GC\_ENABLED

#ifdef HEAP\_ENUMERATION\_VALIDATION

Assert((this->IsMarkState() || this->IsPostEnumHeapValidationInProgress()) && collectionState != CollectionStateConcurrentMark);

#else

Assert(this->IsMarkState() && collectionState != CollectionStateConcurrentMark);

#endif

#else

Assert(this->IsMarkState());

#endif

}

#endif

void

Recycler::TryMarkNonInterior(void\* candidate, void\* parentReference)

{

#ifdef HEAP\_ENUMERATION\_VALIDATION

Assert(!isHeapEnumInProgress || this->IsPostEnumHeapValidationInProgress());

#else

Assert(!isHeapEnumInProgress);

#endif

Assert(this->collectionState != CollectionStateParallelMark);

markContext.Mark</\*parallel \*/ false, /\* interior \*/ false>(candidate, parentReference);

}

void

Recycler::TryMarkInterior(void\* candidate, void\* parentReference)

{

#ifdef HEAP\_ENUMERATION\_VALIDATION

Assert(!isHeapEnumInProgress || this->IsPostEnumHeapValidationInProgress());

#else

Assert(!isHeapEnumInProgress);

#endif

Assert(this->collectionState != CollectionStateParallelMark);

markContext.Mark</\*parallel \*/ false, /\* interior \*/ true>(candidate, parentReference);

}

template <bool parallel, bool interior>

void

Recycler::ProcessMarkContext(MarkContext \* markContext)

{

// Copying the markContext onto the stack messes up tracked object handling, because

// the tracked object will call TryMark[Non]Interior to report its references.

// These functions implicitly use the main markContext on the Recycler, but this will

// be overridden if we're processing the main markContext here.

// So, don't do this if we are going to process tracked objects.

// (This will be the case if we're not queuing and we're not in partial mode, which ignores tracked objects.)

// In this case we shouldn't be parallel anyway, so we don't need to worry about cache behavior.

// We should revisit how we manage markContexts in general in the future, and clean this up

// by passing the MarkContext through to the tracked object's Mark method.

if (this->inPartialCollectMode || DoQueueTrackedObject())

{

// The markContext as passed is one of the markContexts that lives on the Recycler.

// Copy it locally for processing.

// This serves two purposes:

// (1) Allow for better codegen because the markContext is local and we don't need to track the this pointer separately

// (because all the key processing is inlined into this function).

// (2) Ensure we don't have weird cache behavior because we're accidentally writing to the same cache line from

// multiple threads during parallel marking.

MarkContext localMarkContext = \*markContext;

// Do the actual marking.

localMarkContext.ProcessMark<parallel, interior>();

// Copy back to the original location.

\*markContext = localMarkContext;

// Clear the local mark context.

localMarkContext.Clear();

}

else

{

Assert(!parallel);

markContext->ProcessMark<parallel, interior>();

}

}

void

Recycler::ProcessMark(bool background)

{

if (background)

{

GCETW(GC\_BACKGROUNDMARK\_START, (this, backgroundRescanCount));

}

else

{

GCETW(GC\_MARK\_START, (this));

}

RECYCLER\_PROFILE\_EXEC\_THREAD\_BEGIN(background, this, Js::MarkPhase);

if (this->enableScanInteriorPointers)

{

this->ProcessMarkContext</\* parallel \*/ false, /\* interior \*/ true>(&markContext);

}

else

{

this->ProcessMarkContext</\* parallel \*/ false, /\* interior \*/ false>(&markContext);

}

RECYCLER\_PROFILE\_EXEC\_THREAD\_END(background, this, Js::MarkPhase);

if (background)

{

GCETW(GC\_BACKGROUNDMARK\_STOP, (this, backgroundRescanCount));

}

else

{

GCETW(GC\_MARK\_STOP, (this));

}

DebugOnly(this->markContext.VerifyPostMarkState());

}

void

Recycler::ProcessParallelMark(bool background, MarkContext \* markContext)

{

if (background)

{

GCETW(GC\_BACKGROUNDPARALLELMARK\_START, (this, backgroundRescanCount));

}

else

{

GCETW(GC\_PARALLELMARK\_START, (this));

}

RECYCLER\_PROFILE\_EXEC\_THREAD\_BEGIN(background, this, Js::MarkPhase);

if (this->enableScanInteriorPointers)

{

this->ProcessMarkContext</\* parallel \*/ true, /\* interior \*/ true>(markContext);

}

else

{

this->ProcessMarkContext</\* parallel \*/ true, /\* interior \*/ false>(markContext);

}

RECYCLER\_PROFILE\_EXEC\_THREAD\_END(background, this, Js::MarkPhase);

if (background)

{

GCETW(GC\_BACKGROUNDPARALLELMARK\_STOP, (this, backgroundRescanCount));

}

else

{

GCETW(GC\_PARALLELMARK\_STOP, (this));

}

}

void

Recycler::Mark()

{

// Marking in thread, we can just pre-mark them

ResetMarks(this->enableScanImplicitRoots ? ResetMarkFlags\_InThreadImplicitRoots : ResetMarkFlags\_InThread);

collectionState = CollectionStateFindRoots;

RootMark(CollectionStateMark);

}

#ifdef CONCURRENT\_GC\_ENABLED

void

Recycler::StartQueueTrackedObject()

{

Assert(!this->queueTrackedObject);

Assert(!this->HasPendingTrackObjects());

Assert(this->clientTrackedObjectList.Empty());

Assert(!this->inPartialCollectMode);

this->queueTrackedObject = true;

}

bool

Recycler::DoQueueTrackedObject() const

{

Assert(this->queueTrackedObject || !this->IsConcurrentMarkState());

Assert(this->queueTrackedObject || this->inPartialCollectMode || !(this->collectionState == CollectionStateParallelMark));

Assert(this->queueTrackedObject || this->isProcessingTrackedObjects || !this->HasPendingTrackObjects());

#ifdef PARTIAL\_GC\_ENABLED

Assert(!this->queueTrackedObject || (this->clientTrackedObjectList.Empty() && !this->inPartialCollectMode));

#endif

return this->queueTrackedObject;

}

#endif

void

Recycler::ResetCollectionState()

{

Assert(IsMarkStackEmpty());

this->collectionState = CollectionStateNotCollecting;

this->backgroundFinishMarkCount = 0;

this->inExhaustiveCollection = false;

this->inDecommitNowCollection = false;

CleanupPendingUnroot();

#ifdef PARTIAL\_GC\_ENABLED

if (inPartialCollectMode)

{

FinishPartialCollect();

}

#endif

#ifdef CONCURRENT\_GC\_ENABLED

Assert(!this->DoQueueTrackedObject());

#endif

#ifdef RECYCLER\_FINALIZE\_CHECK

// Reset the collection stats.

this->collectionStats.finalizeCount = this->autoHeap.liveFinalizableObjectCount - this->autoHeap.newFinalizableObjectCount - this->autoHeap.pendingDisposableObjectCount;

#endif

}

void

Recycler::ResetMarkCollectionState()

{

// If we aborted after doing a background Rescan, there will be entries in the markContext.

// Abort these entries and reset the markContext state.

markContext.Abort();

// If we aborted after doing a background parallel Mark, we wouldn't have cleaned up the

// parallel markContexts yet. Clean these up now.

// Note parallelMarkContext1 is not used in background parallel (see DoBackgroundParallelMark)

parallelMarkContext2.Cleanup();

parallelMarkContext3.Cleanup();

this->ClearNeedOOMRescan();

DebugOnly(this->isProcessingRescan = false);

#ifdef CONCURRENT\_GC\_ENABLED

// If we're reseting the mark collection state, we need to unlock the block list

DListBase<GuestArenaAllocator>::EditingIterator guestArenaIter(&guestArenaList);

while (guestArenaIter.Next())

{

GuestArenaAllocator& allocator = guestArenaIter.Data();

allocator.SetLockBlockList(false);

}

this->queueTrackedObject = false;

#endif

ResetCollectionState();

}

void

Recycler::ResetHeuristicCounters()

{

autoHeap.lastUncollectedAllocBytes = autoHeap.uncollectedAllocBytes;

autoHeap.uncollectedAllocBytes = 0;

autoHeap.uncollectedExternalBytes = 0;

ResetPartialHeuristicCounters();

}

void Recycler::ResetPartialHeuristicCounters()

{

#ifdef PARTIAL\_GC\_ENABLED

autoHeap.uncollectedNewPageCount = 0;

#endif

}

void

Recycler::ScheduleNextCollection()

{

this->tickCountNextCollection = ::GetTickCount() + RecyclerHeuristic::TickCountCollection;

this->tickCountNextFinishCollection = ::GetTickCount() + RecyclerHeuristic::TickCountFinishCollection;

}

#ifdef CONCURRENT\_GC\_ENABLED

void

Recycler::PrepareSweep()

{

autoHeap.PrepareSweep();

}

#endif

size\_t

Recycler::RescanMark(DWORD waitTime)

{

bool const onLowMemory = this->NeedOOMRescan();

Assert(this->inPartialCollectMode || DoQueueTrackedObject());

{

// We are about to do a rescan mark, which for consistency requires the runtime to stop any additional mutator threads

AUTO\_NO\_EXCEPTION\_REGION;

collectionWrapper->PreRescanMarkCallback();

}

// Always called in-thread

Assert(collectionState == CollectionStateRescanFindRoots);

#ifdef CONCURRENT\_GC\_ENABLED

if (!onLowMemory && // Don't do background finish mark if we are low on memory

// Only do background finish mark if we have a time limit or it is forced

(CUSTOM\_PHASE\_FORCE1(GetRecyclerFlagsTable(), Js::BackgroundFinishMarkPhase) || waitTime != INFINITE) &&

// Don't do background finish mark if we failed to finish mark too many times

(this->backgroundFinishMarkCount < RecyclerHeuristic::MaxBackgroundFinishMarkCount(this->GetRecyclerFlagsTable())))

{

this->PrepareBackgroundFindRoots();

if (StartConcurrent(CollectionStateConcurrentFinishMark))

{

this->backgroundFinishMarkCount++;

this->PrepareSweep();

GCETW(GC\_RESCANMARKWAIT\_START, (this, waitTime));

const BOOL waited = WaitForConcurrentThread(waitTime);

GCETW(GC\_RESCANMARKWAIT\_STOP, (this, !waited));

if (!waited)

{

CUSTOM\_PHASE\_PRINT\_TRACE1(GetRecyclerFlagsTable(), Js::BackgroundFinishMarkPhase, L"Finish mark timed out\n");

{

// We timed out doing the finish mark, notify the runtime

AUTO\_NO\_EXCEPTION\_REGION;

collectionWrapper->RescanMarkTimeoutCallback();

}

return Recycler::InvalidScanRootBytes;

}

Assert(collectionState == CollectionStateRescanWait);

collectionState = CollectionStateRescanFindRoots;

Assert(recyclerPageAllocator.GetWriteWatchPageCount() == 0);

Assert(recyclerLargeBlockPageAllocator.GetWriteWatchPageCount() == 0);

return this->backgroundRescanRootBytes;

}

this->RevertPrepareBackgroundFindRoots();

}

#endif

this->backgroundFinishMarkCount = 0;

return FinishMarkRescan(false) \* AutoSystemInfo::PageSize;

}

#if defined(PARTIAL\_GC\_ENABLED) || defined(CONCURRENT\_GC\_ENABLED)

size\_t

Recycler::FinishMark(DWORD waitTime)

{

size\_t scannedRootBytes = RescanMark(waitTime);

Assert(waitTime != INFINITE || scannedRootBytes != Recycler::InvalidScanRootBytes);

if (scannedRootBytes != Recycler::InvalidScanRootBytes)

{

#if DBG

RecyclerVerboseTrace(GetRecyclerFlagsTable(), L"CTO: %d\n", this->clientTrackedObjectList.Count());

#endif

#ifdef PARTIAL\_GC\_ENABLED

if (this->inPartialCollectMode)

{

RecyclerVerboseTrace(GetRecyclerFlagsTable(), L"Processing client tracked objects\n");

ProcessClientTrackedObjects();

}

else

#endif

if (DoQueueTrackedObject())

{

RecyclerVerboseTrace(GetRecyclerFlagsTable(), L"Processing regular tracked objects\n");

ProcessTrackedObjects();

Assert(this->backgroundFinishMarkCount == 0 ||

(this->recyclerPageAllocator.GetWriteWatchPageCount() == 0 &&

this->recyclerLargeBlockPageAllocator.GetWriteWatchPageCount() == 0));

}

// Continue to mark from root one more time

scannedRootBytes += RootMark(CollectionStateRescanMark);

}

return scannedRootBytes;

}

#endif

void

Recycler::DoParallelMark()

{

Assert(this->enableParallelMark);

Assert(this->maxParallelism > 1 && this->maxParallelism <= 4);

// Split the mark stack into [this->maxParallelism] equal pieces.

// The actual # of splits is returned, in case the stack was too small to split that many ways.

MarkContext \* splitContexts[3] = { &parallelMarkContext1, &parallelMarkContext2, &parallelMarkContext3 };

uint actualSplitCount = markContext.Split(this->maxParallelism - 1, splitContexts);

Assert(actualSplitCount <= 3);

// If we failed to split at all, just mark in thread with no parallelism.

if (actualSplitCount == 0)

{

this->ProcessMark(false);

return;

}

// We need to queue tracked objects while we mark in parallel.

// (Unless it's a partial collect, in which case we don't process tracked objects at all)

if (!this->inPartialCollectMode)

{

StartQueueTrackedObject();

}

// Kick off marking on the background thread

bool concurrentSuccess = StartConcurrent(CollectionStateParallelMark);

// If there's enough work to split, then kick off marking on parallel threads too.

// If the threads haven't been created yet, this will create them (or fail).

bool parallelSuccess1 = false;

bool parallelSuccess2 = false;

if (concurrentSuccess && actualSplitCount >= 2)

{

parallelSuccess1 = parallelThread1.StartConcurrent();

if (parallelSuccess1 && actualSplitCount == 3)

{

parallelSuccess2 = parallelThread2.StartConcurrent();

}

}

// Process our portion of the split.

this->ProcessParallelMark(false, &parallelMarkContext1);

// If we successfully launched parallel work, wait for it to complete.

// If we failed, then process the work in-thread now.

if (concurrentSuccess)

{

WaitForConcurrentThread(INFINITE);

}

else

{

this->ProcessParallelMark(false, &markContext);

}

if (actualSplitCount >= 2)

{

if (parallelSuccess1)

{

parallelThread1.WaitForConcurrent();

}

else

{

this->ProcessParallelMark(false, &parallelMarkContext2);

}

if (actualSplitCount == 3)

{

if (parallelSuccess2)

{

parallelThread2.WaitForConcurrent();

}

else

{

this->ProcessParallelMark(false, &parallelMarkContext3);

}

}

}

this->collectionState = CollectionStateMark;

// Process tracked objects, if any, then do one final mark phase in case they marked any new objects.

// (Unless it's a partial collect, in which case we don't process tracked objects at all)

if (!this->inPartialCollectMode)

{

this->ProcessTrackedObjects();

this->ProcessMark(false);

}

else

{

Assert(!this->HasPendingTrackObjects());

}

}

void

Recycler::DoBackgroundParallelMark()

{

// Split the mark stack into [this->maxParallelism - 1] equal pieces (thus, "- 2" below).

// The actual # of splits is returned, in case the stack was too small to split that many ways.

// The parallel threads are hardwired to use parallelMarkContext2/3, so we split using those.

uint actualSplitCount = 0;

MarkContext \* splitContexts[2] = { &parallelMarkContext2, &parallelMarkContext3 };

if (this->enableParallelMark)

{

Assert(this->maxParallelism > 1 && this->maxParallelism <= 4);

if (this->maxParallelism > 2)

{

actualSplitCount = markContext.Split(this->maxParallelism - 2, splitContexts);

}

}

Assert(actualSplitCount <= 2);

// If we failed to split at all, just mark in thread with no parallelism.

if (actualSplitCount == 0)

{

this->ProcessMark(true);

return;

}

// We should already be set up to queue tracked objects, unless this is a partial collect

Assert(this->DoQueueTrackedObject() || this->inPartialCollectMode);

this->collectionState = CollectionStateBackgroundParallelMark;

// Kick off marking on parallel threads too, if there is work for them

// If the threads haven't been created yet, this will create them (or fail).

bool parallelSuccess1 = false;

bool parallelSuccess2 = false;

parallelSuccess1 = parallelThread1.StartConcurrent();

if (parallelSuccess1 && actualSplitCount == 2)

{

parallelSuccess2 = parallelThread2.StartConcurrent();

}

// Process our portion of the split.

this->ProcessParallelMark(true, &markContext);

// If we successfully launched parallel work, wait for it to complete.

// If we failed, then process the work in-thread now.

if (parallelSuccess1)

{

parallelThread1.WaitForConcurrent();

}

else

{

this->ProcessParallelMark(true, &parallelMarkContext2);

}

if (actualSplitCount == 2)

{

if (parallelSuccess2)

{

parallelThread2.WaitForConcurrent();

}

else

{

this->ProcessParallelMark(true, &parallelMarkContext3);

}

}

this->collectionState = CollectionStateConcurrentMark;

}

size\_t

Recycler::RootMark(CollectionState markState)

{

size\_t scannedRootBytes = 0;

Assert(!this->NeedOOMRescan() || markState == CollectionStateRescanMark);

RecyclerVerboseTrace(GetRecyclerFlagsTable(), L"PreMark done, partial collect: %d\n", this->inPartialCollectMode);

Assert(collectionState == (markState == CollectionStateMark? CollectionStateFindRoots : CollectionStateRescanFindRoots));

BOOL stacksScannedByRuntime = FALSE;

{

// We are about to scan roots in thread, notify the runtime first so it can stop threads if necessary and also provide additional roots

AUTO\_NO\_EXCEPTION\_REGION;

RecyclerScanMemoryCallback scanMemory(this);

scannedRootBytes += collectionWrapper->RootMarkCallback(scanMemory, &stacksScannedByRuntime);

}

scannedRootBytes += FindRoots();

if (!stacksScannedByRuntime)

{

// The runtime did not scan the stack(s) for us, so we use the normal Recycler code.

scannedRootBytes += ScanStack();

}

this->collectionState = markState;

if (this->enableParallelMark)

{

this->DoParallelMark();

}

else

{

this->ProcessMark(false);

}

if (this->EndMark())

{

// return large root scanned byte to not get into partial mode if we are low on memory

scannedRootBytes = RecyclerSweep::MaxPartialCollectRescanRootBytes + 1;

}

return scannedRootBytes;

}

bool

Recycler::EndMarkCheckOOMRescan()

{

bool oomRescan = false;

if (this->NeedOOMRescan())

{

#ifdef RECYCLER\_DUMP\_OBJECT\_GRAPH

if (this->objectGraphDumper)

{

// Do not complete the mark if we are just dumping the object graph

// Just report out of memory

this->objectGraphDumper->isOutOfMemory = true;

this->ClearNeedOOMRescan();

}

else

#endif

{

EndMarkOnLowMemory();

oomRescan = true;

}

}

// Done with the mark stack, it should be empty.

// Release pages it is holding.

Assert(!HasPendingMarkObjects());

Assert(!HasPendingTrackObjects());

return oomRescan;

}

bool

Recycler::EndMark()

{

Assert(!this->DoQueueTrackedObject());

Assert(this->clientTrackedObjectList.Empty());

{

// We have finished marking

AUTO\_NO\_EXCEPTION\_REGION;

collectionWrapper->EndMarkCallback();

}

bool oomRescan = EndMarkCheckOOMRescan();

if (ProcessObjectBeforeCollectCallbacks())

{

// callbacks may trigger additional marking, need to check OOMRescan again

oomRescan |= EndMarkCheckOOMRescan();

}

// GC-CONSIDER: Consider keeping some page around

GCETW(GC\_DECOMMIT\_CONCURRENT\_COLLECT\_PAGE\_ALLOCATOR\_START, (this));

// Clean up mark contexts, which will release held free pages

// Do this for all contexts before we decommit, to make sure all pages are freed

markContext.Cleanup();

parallelMarkContext1.Cleanup();

parallelMarkContext2.Cleanup();

parallelMarkContext3.Cleanup();

// Decommit all pages

markContext.DecommitPages();

parallelMarkContext1.DecommitPages();

parallelMarkContext2.DecommitPages();

parallelMarkContext3.DecommitPages();

GCETW(GC\_DECOMMIT\_CONCURRENT\_COLLECT\_PAGE\_ALLOCATOR\_STOP, (this));

return oomRescan;

}

void

Recycler::EndMarkOnLowMemory()

{

GCETW(GC\_ENDMARKONLOWMEMORY\_START, (this));

Assert(this->NeedOOMRescan());

this->inEndMarkOnLowMemory = true;

// Treat this as a concurrent mark reset so that we don't invalidate the allocators

RecyclerVerboseTrace(GetRecyclerFlagsTable(), L"OOM during mark- rerunning mark\n");

// Try to release as much memory as possible

ForRecyclerPageAllocator(DecommitNow());

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

uint iterations = 0;

#endif

do

{

Assert(this->clientTrackedObjectList.Empty());

// Always queue tracked objects during rescan, to avoid changes to mark state.

// (Unless we're in a partial, in which case we ignore tracked objects)

Assert(!this->DoQueueTrackedObject());

#ifdef PARTIAL\_GC\_ENABLED

if (!this->inPartialCollectMode)

#endif

{

this->StartQueueTrackedObject();

}

this->collectionState = CollectionStateRescanFindRoots;

this->ClearNeedOOMRescan();

#if DBG

Assert(!this->isProcessingRescan);

this->isProcessingRescan = true;

#endif

if (!heapBlockMap.OOMRescan(this))

{

// Kill the process- we couldn't even rescan a single block

// We are in pretty low memory state at this point

// The fail-fast is present for two reasons:

// 1) Defense-in-depth for cases we hadn't thought about

// 2) Deal with cases like -MaxMarkStackPageCount:1 which can still hang without the fail-fast

MarkStack\_OOM\_fatal\_error();

}

autoHeap.Rescan(RescanFlags\_None);

DebugOnly(this->isProcessingRescan = false);

this->ProcessMark(false);

// Process any tracked objects we found

#ifdef PARTIAL\_GC\_ENABLED

if (!this->inPartialCollectMode)

#endif

{

ProcessTrackedObjects();

}

// Drain the mark stack

ProcessMark(false);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

iterations++;

#endif

}

while (this->NeedOOMRescan());

Assert(!markContext.GetPageAllocator()->DisableAllocationOutOfMemory());

Assert(!parallelMarkContext1.GetPageAllocator()->DisableAllocationOutOfMemory());

Assert(!parallelMarkContext2.GetPageAllocator()->DisableAllocationOutOfMemory());

Assert(!parallelMarkContext3.GetPageAllocator()->DisableAllocationOutOfMemory());

CUSTOM\_PHASE\_PRINT\_TRACE1(GetRecyclerFlagsTable(), Js::RecyclerPhase, L"EndMarkOnLowMemory iterations: %d\n", iterations);

Assert(this->clientTrackedObjectList.Empty());

Assert(!this->DoQueueTrackedObject());

this->inEndMarkOnLowMemory = false;

#ifdef PARTIAL\_GC\_ENABLED

if (this->inPartialCollectMode)

{

this->FinishPartialCollect();

}

#endif

GCETW(GC\_ENDMARKONLOWMEMORY\_STOP, (this));

}

#if DBG

bool

Recycler::IsMarkStackEmpty()

{

return (markContext.IsEmpty() && parallelMarkContext1.IsEmpty() && parallelMarkContext2.IsEmpty() && parallelMarkContext3.IsEmpty());

}

#endif

#ifdef HEAP\_ENUMERATION\_VALIDATION

void

Recycler::PostHeapEnumScan(PostHeapEnumScanCallback callback, void \*data)

{

this->pfPostHeapEnumScanCallback = callback;

this->postHeapEnunScanData = data;

FindRoots();

ProcessMark(false);

this->pfPostHeapEnumScanCallback = NULL;

this->postHeapEnunScanData = NULL;

}

#endif

bool

Recycler::QueueTrackedObject(FinalizableObject \* trackableObject)

{

return markContext.AddTrackedObject(trackableObject);

}

bool

Recycler::FindImplicitRootObject(void\* candidate, RecyclerHeapObjectInfo& heapObject)

{

HeapBlock\* heapBlock = FindHeapBlock(candidate);

if (heapBlock == nullptr)

{

return false;

}

if (heapBlock->GetHeapBlockType() < HeapBlock::HeapBlockType::SmallAllocBlockTypeCount)

{

return ((SmallHeapBlock\*)heapBlock)->FindImplicitRootObject(candidate, this, heapObject);

}

else if (!heapBlock->IsLargeHeapBlock())

{

return ((MediumHeapBlock\*)heapBlock)->FindImplicitRootObject(candidate, this, heapObject);

}

else

{

return ((LargeHeapBlock\*)heapBlock)->FindImplicitRootObject(candidate, this, heapObject);

}

}

bool

Recycler::FindHeapObject(void\* candidate, FindHeapObjectFlags flags, RecyclerHeapObjectInfo& heapObject)

{

HeapBlock\* heapBlock = FindHeapBlock(candidate);

return heapBlock && heapBlock->FindHeapObject(candidate, this, flags, heapObject);

}

bool

Recycler::FindHeapObjectWithClearedAllocators(void\* candidate, RecyclerHeapObjectInfo& heapObject)

{

// Heap enum has some case where it allocates, so we can't assert

Assert(autoHeap.AllocatorsAreEmpty() || this->isHeapEnumInProgress);

return FindHeapObject(candidate, FindHeapObjectFlags\_ClearedAllocators, heapObject);

}

void\*

Recycler::GetRealAddressFromInterior(void\* candidate)

{

HeapBlock \* heapBlock = heapBlockMap.GetHeapBlock(candidate);

if (heapBlock == NULL)

{

return NULL;

}

return heapBlock->GetRealAddressFromInterior(candidate);

}

/\*------------------------------------------------------------------------------------------------

\* Sweep

\*------------------------------------------------------------------------------------------------\*/

bool

Recycler::Sweep(size\_t rescanRootBytes, bool concurrent, bool adjustPartialHeuristics)

{

Assert(!this->hasBackgroundFinishPartial);

#ifdef CONCURRENT\_GC\_ENABLED

if (!this->enableConcurrentSweep)

#endif

{

concurrent = false;

}

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, concurrent? Js::ConcurrentSweepPhase : Js::SweepPhase);

recyclerSweepInstance.BeginSweep(this, rescanRootBytes, adjustPartialHeuristics);

this->SweepHeap(concurrent, \*recyclerSweep);

if (concurrent)

{

// If we finished mark in the background, all the relevant write watches should already be reset

// Only reset write watch if we didn't finish mark in the background

if (this->backgroundFinishMarkCount == 0)

{

if (this->inPartialCollectMode)

{

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::ResetWriteWatchPhase);

if (!recyclerPageAllocator.ResetWriteWatch() || !recyclerLargeBlockPageAllocator.ResetWriteWatch())

{

// Shouldn't happen

Assert(false);

// Disable partial collect

this->enablePartialCollect = false;

// We haven't done any partial collection yet, just get out of partial collect mode

this->inPartialCollectMode = false;

}

RECYCLER\_PROFILE\_EXEC\_END(this, Js::ResetWriteWatchPhase);

}

}

}

else

{

recyclerSweep->FinishSweep();

recyclerSweep->EndSweep();

}

RECYCLER\_PROFILE\_EXEC\_END(this, concurrent? Js::ConcurrentSweepPhase : Js::SweepPhase);

#ifdef CONCURRENT\_GC\_ENABLED

if (concurrent)

{

if (!StartConcurrent(CollectionStateConcurrentSweep))

{

// Failed to spawn the concurrent sweep.

// Instead, force the concurrent sweep to happen right here in thread.

this->collectionState = CollectionStateConcurrentSweep;

DoBackgroundWork(true);

// Continue as if the concurrent sweep were executing

// Next time we check for completion, we will finish the sweep just as if it had happened out of thread.

}

return true;

}

#endif

return false;

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void Recycler::DisplayMemStats()

{

#ifdef PERF\_COUNTERS

#if DBG\_DUMP

printf("Recycler Live Object Count %u\n", PerfCounter::RecyclerCounterSet::GetLiveObjectCounter().GetValue());

printf("Recycler Live Object Size %u\n", PerfCounter::RecyclerCounterSet::GetLiveObjectSizeCounter().GetValue());

#endif

printf("Recycler Used Page Size %u\n", PerfCounter::PageAllocatorCounterSet::GetUsedSizeCounter(PageAllocatorType::PageAllocatorType\_Recycler).GetValue());

#endif

}

#endif

CollectedRecyclerWeakRefHeapBlock CollectedRecyclerWeakRefHeapBlock::Instance;

void

Recycler::SweepWeakReference()

{

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::SweepWeakPhase);

GCETW(GC\_SWEEP\_WEAKREF\_START, (this));

// REVIEW: Clean up the weak reference map concurrently?

bool hasCleanup = false;

weakReferenceMap.Map([&hasCleanup](RecyclerWeakReferenceBase \* weakRef) -> bool

{

if (!weakRef->weakRefHeapBlock->TestObjectMarkedBit(weakRef))

{

hasCleanup = true;

// Remove

return false;

}

if (!weakRef->strongRefHeapBlock->TestObjectMarkedBit(weakRef->strongRef))

{

hasCleanup = true;

weakRef->strongRef = nullptr;

// Put in a dummy heap block so that we can still do the isPendingConcurrentSweep check first.

weakRef->strongRefHeapBlock = &CollectedRecyclerWeakRefHeapBlock::Instance;

// Remove

return false;

}

// Keep

return true;

});

this->weakReferenceCleanupId += hasCleanup;

GCETW(GC\_SWEEP\_WEAKREF\_STOP, (this));

RECYCLER\_PROFILE\_EXEC\_END(this, Js::SweepWeakPhase);

}

void

Recycler::SweepHeap(bool concurrent, RecyclerSweep& recyclerSweep)

{

Assert(!this->hasPendingDeleteGuestArena);

Assert(!this->isHeapEnumInProgress);

#ifdef CONCURRENT\_GC\_ENABLED

Assert(!this->DoQueueTrackedObject());

if (concurrent)

{

collectionState = CollectionStateSetupConcurrentSweep;

// Only queue up non-leaf pages- leaf pages don't need to be zeroed out

recyclerPageAllocator.StartQueueZeroPage();

recyclerLargeBlockPageAllocator.StartQueueZeroPage();

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.StartQueueZeroPage();

#endif

}

else

#endif

{

Assert(!concurrent);

collectionState = CollectionStateSweep;

}

this->SweepWeakReference();

#ifdef CONCURRENT\_GC\_ENABLED

if (concurrent)

{

GCETW(GC\_SETUPBACKGROUNDSWEEP\_START, (this));

}

else

#endif

{

GCETW(GC\_SWEEP\_START, (this));

}

recyclerPageAllocator.SuspendIdleDecommit();

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.SuspendIdleDecommit();

#endif

recyclerLargeBlockPageAllocator.SuspendIdleDecommit();

autoHeap.Sweep(recyclerSweep, concurrent);

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.ResumeIdleDecommit();

#endif

recyclerPageAllocator.ResumeIdleDecommit();

recyclerLargeBlockPageAllocator.ResumeIdleDecommit();

#ifdef CONCURRENT\_GC\_ENABLED

if (concurrent)

{

recyclerPageAllocator.StopQueueZeroPage();

recyclerLargeBlockPageAllocator.StopQueueZeroPage();

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.StopQueueZeroPage();

#endif

GCETW(GC\_SETUPBACKGROUNDSWEEP\_STOP, (this));

}

else

{

Assert(!recyclerPageAllocator.HasZeroQueuedPages());

Assert(!recyclerLargeBlockPageAllocator.HasZeroQueuedPages());

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

Assert(!recyclerWithBarrierPageAllocator.HasZeroQueuedPages());

#endif

uint sweptBytes = 0;

#ifdef RECYCLER\_STATS

sweptBytes = (uint)collectionStats.objectSweptBytes;

#endif

GCETW(GC\_SWEEP\_STOP, (this, sweptBytes));

}

#endif

}

void

Recycler::BackgroundFinishPartialCollect(RecyclerSweep \* recyclerSweep)

{

Assert(this->inPartialCollectMode);

Assert(recyclerSweep != nullptr && recyclerSweep->IsBackground());

this->hasBackgroundFinishPartial = true;

this->autoHeap.FinishPartialCollect(recyclerSweep);

this->inPartialCollectMode = false;

}

void

Recycler::DisposeObjects()

{

Assert(this->allowDispose && this->hasDisposableObject && !this->inDispose);

Assert(!isHeapEnumInProgress);

GCETW(GC\_DISPOSE\_START, (this));

ASYNC\_HOST\_OPERATION\_START(collectionWrapper);

this->inDispose = true;

#ifdef PROFILE\_RECYCLER\_ALLOC

// finalizer may allocate memory and dispose object can happen in the middle of allocation

// save and restore the tracked object info

TrackAllocData oldAllocData = { 0 };

if (trackerDictionary != nullptr)

{

oldAllocData = nextAllocData;

nextAllocData.Clear();

}

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::RecyclerPhase))

{

Output::Print(L"Disposing objects\n");

}

#endif

// Disable dispose within this method, restore it when we're done

AutoRestoreValue<bool> disableDispose(&this->allowDispose, false);

#ifdef FAULT\_INJECTION

this->collectionWrapper->DisposeScriptContextByFaultInjectionCallBack();

#endif

// Scope timestamp to just dispose

{

AUTO\_TIMESTAMP(dispose);

autoHeap.DisposeObjects();

}

#ifdef PROFILE\_RECYCLER\_ALLOC

if (trackerDictionary != nullptr)

{

Assert(nextAllocData.IsEmpty());

nextAllocData = oldAllocData;

}

#endif

#ifdef ENABLE\_PROJECTION

{

Assert(!this->inResolveExternalWeakReferences);

Assert(!this->allowDispose);

#if DBG || defined RECYCLER\_TRACE

AutoRestoreValue<bool> inResolveExternalWeakReferencedObjects(&this->inResolveExternalWeakReferences, true);

#endif

AUTO\_TIMESTAMP(externalWeakReferenceObjectResolve);

// This is where it is safe to resolve external weak references as they can lead to new script entry

collectionWrapper->ResolveExternalWeakReferencedObjects();

}

#endif

Assert(!this->inResolveExternalWeakReferences);

Assert(this->inDispose);

this->inDispose = false;

ASYNC\_HOST\_OPERATION\_END(collectionWrapper);

uint sweptBytes = 0;

#ifdef RECYCLER\_STATS

sweptBytes = (uint)collectionStats.objectSweptBytes;

#endif

GCETW(GC\_DISPOSE\_STOP, (this, sweptBytes));

}

bool

Recycler::FinishDisposeObjects()

{

CUSTOM\_PHASE\_PRINT\_TRACE1(GetRecyclerFlagsTable(), Js::DisposePhase, L"[Dispose] AllowDispose in FinishDisposeObject: %d\n", this->allowDispose);

if (this->hasDisposableObject && this->allowDispose)

{

CUSTOM\_PHASE\_PRINT\_TRACE1(GetRecyclerFlagsTable(), Js::DisposePhase, L"[Dispose] FinishDisposeObject, calling Dispose: %d\n", this->allowDispose);

#ifdef RECYCLER\_TRACE

CollectionParam savedCollectionParam = collectionParam;

#endif

DisposeObjects();

#ifdef RECYCLER\_TRACE

collectionParam = savedCollectionParam;

#endif

// FinishDisposeObjects is always called either during a collection,

// or we will check the NeedExhaustiveRepeatCollect(), so no need to check it here

return true;

}

#ifdef RECYCLER\_TRACE

if (!this->inDispose && this->hasDisposableObject

&& GetRecyclerFlagsTable().Trace.IsEnabled(Js::RecyclerPhase))

{

Output::Print(L"%04X> RC(%p): %s\n", this->mainThreadId, this, L"Dispose object delayed");

}

#endif

return false;

}

template bool Recycler::FinishDisposeObjectsNow<FinishDispose>();

template bool Recycler::FinishDisposeObjectsNow<FinishDisposeTimed>();

template <CollectionFlags flags>

bool

Recycler::FinishDisposeObjectsNow()

{

if (inDisposeWrapper)

{

return false;

}

return FinishDisposeObjectsWrapped<flags>();

}

template <CollectionFlags flags>

\_\_inline

bool

Recycler::FinishDisposeObjectsWrapped()

{

const BOOL allowDisposeFlag = flags & CollectOverride\_AllowDispose;

if (allowDisposeFlag && this->NeedDispose())

{

if ((flags & CollectHeuristic\_TimeIfScriptActive) == CollectHeuristic\_TimeIfScriptActive)

{

if (!this->NeedDisposeTimed())

{

return false;

}

}

this->allowDispose = true;

this->inDisposeWrapper = true;

#ifdef RECYCLER\_TRACE

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::RecyclerPhase))

{

Output::Print(L"%04X> RC(%p): %s\n", this->mainThreadId, this, L"Process delayed dispose object");

}

#endif

collectionWrapper->DisposeObjects(this);

// Dispose may get into message loop and cause a reentrant GC. If those don't allow reentrant

// it will get added to a pending collect request.

// FinishDipsosedObjectsWrapped/DisposeObjectsWrapped is called at a place that might not be during a collection

// and won't check NeedExhaustiveRepeatCollect(), need to check it here to honor those requests

if (!this->CollectionInProgress() && NeedExhaustiveRepeatCollect() && ((flags & CollectOverride\_NoExhaustiveCollect) != CollectOverride\_NoExhaustiveCollect))

{

#ifdef RECYCLER\_TRACE

CaptureCollectionParam((CollectionFlags)(flags & ~CollectMode\_Partial), true);

#endif

DoCollectWrapped((CollectionFlags)(flags & ~CollectMode\_Partial));

}

this->inDisposeWrapper = false;

return true;

}

return false;

}

/\*------------------------------------------------------------------------------------------------

\* Collect

\*------------------------------------------------------------------------------------------------\*/

BOOL

Recycler::CollectOnAllocatorThread()

{

#ifdef PARTIAL\_GC\_ENABLED

Assert(!inPartialCollectMode);

#endif

#ifdef RECYCLER\_TRACE

PrintCollectTrace(Js::GarbageCollectPhase);

#endif

this->CollectionBegin<Js::GarbageCollectPhase>();

this->Mark();

// Partial collect mode is not re-enabled after a non-partial in-thread GC because partial GC heuristics are not adjusted

// after a full in-thread GC. Enabling partial collect mode causes partial GC heuristics to be reset before the next full

// in-thread GC, thereby allowing partial GC to kick in more easily without being able to adjust heuristics after the full

// GCs. Until we have a way of adjusting partial GC heuristics after a full in-thread GC, once partial collect mode is

// turned off, it will remain off until a concurrent GC happens

this->Sweep();

this->CollectionEnd<Js::GarbageCollectPhase>();

FinishCollection();

return true;

}

// Explicitly instantiate all possible modes

template BOOL Recycler::CollectNow<CollectOnScriptIdle>();

template BOOL Recycler::CollectNow<CollectOnScriptExit>();

template BOOL Recycler::CollectNow<CollectOnAllocation>();

template BOOL Recycler::CollectNow<CollectOnTypedArrayAllocation>();

template BOOL Recycler::CollectNow<CollectOnScriptCloseNonPrimary>();

template BOOL Recycler::CollectNow<CollectExhaustiveCandidate>();

template BOOL Recycler::CollectNow<CollectNowConcurrent>();

template BOOL Recycler::CollectNow<CollectNowExhaustive>();

template BOOL Recycler::CollectNow<CollectNowDecommitNowExplicit>();

template BOOL Recycler::CollectNow<CollectNowPartial>();

template BOOL Recycler::CollectNow<CollectNowConcurrentPartial>();

template BOOL Recycler::CollectNow<CollectNowForceInThread>();

template BOOL Recycler::CollectNow<CollectNowForceInThreadExternal>();

template BOOL Recycler::CollectNow<CollectNowForceInThreadExternalNoStack>();

template BOOL Recycler::CollectNow<CollectOnRecoverFromOutOfMemory>();

template BOOL Recycler::CollectNow<CollectNowDefault>();

template BOOL Recycler::CollectNow<CollectOnSuspendCleanup>();

template BOOL Recycler::CollectNow<CollectNowDefaultLSCleanup>();

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

template BOOL Recycler::CollectNow<CollectNowFinalGC>();

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

template BOOL Recycler::CollectNow<CollectNowExhaustiveSkipStack>();

#endif

template <CollectionFlags flags>

BOOL

Recycler::CollectNow()

{

// Force-in-thread cannot be concurrent or partial

CompileAssert((flags & CollectOverride\_ForceInThread) == 0 || (flags & (CollectMode\_Concurrent | CollectMode\_Partial)) == 0);

// Collections not allowed when the recycler is currently executing the PostCollectionCallback

if (collectionState == CollectionStatePostCollectionCallback)

{

return false;

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

if ((disableCollection && (flags & CollectOverride\_Explicit) == 0) || isShuttingDown)

#else

if (isShuttingDown)

#endif

{

Assert(collectionState == CollectionStateNotCollecting

|| collectionState == CollectionStateExit

|| this->isShuttingDown);

return false;

}

if (flags & CollectOverride\_ExhaustiveCandidate)

{

return CollectWithExhaustiveCandidate<flags>();

}

return CollectInternal<flags>();

}

template <CollectionFlags flags>

BOOL

Recycler::GetPartialFlag()

{

#ifdef PARTIAL\_GC\_ENABLED

#pragma prefast(suppress:6313, "flags is a template parameter and can be 0")

return(flags & CollectMode\_Partial) && inPartialCollectMode;

#else

return false;

#endif

}

template <CollectionFlags flags>

BOOL

Recycler::CollectWithExhaustiveCandidate()

{

Assert(flags & CollectOverride\_ExhaustiveCandidate);

// Currently we don't have any exhaustive candidate that has heuristic.

Assert((flags & CollectHeuristic\_Mask & ~CollectHeuristic\_Never) == 0);

this->hasExhaustiveCandidate = true;

if (flags & CollectHeuristic\_Never)

{

// This is just an exhaustive candidate notification. Don't trigger a GC.

return false;

}

// Continue with the GC heuristic

return CollectInternal<flags>();

}

template <CollectionFlags flags>

BOOL

Recycler::CollectInternal()

{

// CollectHeuristic\_Never flag should only be used with exhaustive candidate

Assert((flags & CollectHeuristic\_Never) == 0);

// If we're in a re-entrant state, we want to allow GC to be triggered only

// from allocation (or trigger points with AllowReentrant). This is to minimize

// the number of reentrant GCs

if ((flags & CollectOverride\_AllowReentrant) == 0 && this->inDispose)

{

return false;

}

#ifdef RECYCLER\_TRACE

CaptureCollectionParam(flags);

#endif

#ifdef CONCURRENT\_GC\_ENABLED

const BOOL concurrent = flags & CollectMode\_Concurrent;

const BOOL finishConcurrent = flags & CollectOverride\_FinishConcurrent;

// If we priority boosted, we should try to finish it every chance we get

// Otherwise, we should finishing it if we are not doing a concurrent GC,

// or the flags tell us to always try to finish a concurrent GC (CollectOverride\_FinishConcurrent)

if ((!concurrent || finishConcurrent || priorityBoost) && this->CollectionInProgress())

{

return TryFinishConcurrentCollect<flags>();

}

#endif

if (flags & CollectHeuristic\_Mask)

{

// Check some heuristics first before starting a collection

return CollectWithHeuristic<flags>();

}

// Start a collection now.

return Collect<flags>();

}

template <CollectionFlags flags>

BOOL

Recycler::CollectWithHeuristic()

{

// CollectHeuristic\_Never flag should only be used with exhaustive candidate

Assert((flags & CollectHeuristic\_Never) == 0);

const BOOL allocSize = flags & CollectHeuristic\_AllocSize;

const BOOL timedIfScriptActive = flags & CollectHeuristic\_TimeIfScriptActive;

const BOOL timedIfInScript = flags & CollectHeuristic\_TimeIfInScript;

const BOOL timed = (timedIfScriptActive && isScriptActive) || (timedIfInScript && isInScript) || (flags & CollectHeuristic\_Time);

#ifdef PARTIAL\_GC\_ENABLED

if (GetPartialFlag<flags>())

{

Assert(enablePartialCollect);

Assert(allocSize);

Assert(this->uncollectedNewPageCountPartialCollect >= RecyclerSweep::MinPartialUncollectedNewPageCount

&& this->uncollectedNewPageCountPartialCollect <= RecyclerHeuristic::Instance.MaxPartialUncollectedNewPageCount);

// PARTIAL-GC-REVIEW: For now, we have only alloc size heuristic

// Maybe improve this heuristic by looking at how many free pages are in the page allocator.

if (autoHeap.uncollectedNewPageCount > this->uncollectedNewPageCountPartialCollect)

{

return Collect<flags>();

}

}

#endif

// allocation byte count heuristic, collect every 1 MB allocated

if (allocSize && (autoHeap.uncollectedAllocBytes < RecyclerHeuristic::UncollectedAllocBytesCollection()))

{

return FinishDisposeObjectsWrapped<flags>();

}

// time heuristic, allocate every 1000 clock tick, or 64 MB is allocated in a short time

if (timed && (autoHeap.uncollectedAllocBytes < RecyclerHeuristic::Instance.MaxUncollectedAllocBytes))

{

uint currentTickCount = GetTickCount();

#ifdef RECYCLER\_TRACE

collectionParam.timeDiff = currentTickCount - tickCountNextCollection;

#endif

if ((int)(tickCountNextCollection - currentTickCount) >= 0)

{

return FinishDisposeObjectsWrapped<flags>();

}

}

#ifdef RECYCLER\_TRACE

else

{

uint currentTickCount = GetTickCount();

collectionParam.timeDiff = currentTickCount - tickCountNextCollection;

}

#endif

// Passed all the heuristic, do some GC work, maybe

return Collect<(CollectionFlags)(flags & ~CollectMode\_Partial)>();

}

template <CollectionFlags flags>

BOOL

Recycler::Collect()

{

#ifdef CONCURRENT\_GC\_ENABLED

if (this->CollectionInProgress())

{

// If we are forced in thread, we can't be concurrent

// If we are not concurrent we should have been handled before in CollectInternal and we shouldn't be here

Assert((flags & CollectOverride\_ForceInThread) == 0);

Assert((flags & CollectMode\_Concurrent) != 0);

return TryFinishConcurrentCollect<flags>();

}

#endif

SetupPostCollectionFlags<flags>();

const BOOL partial = GetPartialFlag<flags>();

CollectionFlags finalFlags = flags;

if (!partial)

{

finalFlags = (CollectionFlags)(flags & ~CollectMode\_Partial);

}

// ExecuteRecyclerCollectionFunction may cause exception. In which case, we may trigger the assert

// in SetupPostCollectionFlags because we didn't reset the inExhausitvECollection variable if

// an exception. Use this flag to disable it the assertion if exception occur

DebugOnly(this->hasIncompletedDoCollect = true);

{

RECORD\_TIMESTAMP(initialCollectionStartTime);

this->telemetryBlock->initialCollectionStartProcessUsedBytes = PageAllocator::GetProcessUsedBytes();

this->telemetryBlock->exhaustiveRepeatedCount = 0;

return DoCollectWrapped(finalFlags);

}

}

template <CollectionFlags flags>

void Recycler::SetupPostCollectionFlags()

{

// If we are not in a collection (collection in progress or in dispose), inExhastivecollection should not be set

// Otherwise, we have missed an exhaustive collection.

Assert(this->hasIncompletedDoCollect ||

this->CollectionInProgress() || this->inDispose || (!this->inExhaustiveCollection && !this->inDecommitNowCollection));

// Record whether we want to start exhaustive detection or do decommit now after GC

const BOOL exhaustive = flags & CollectMode\_Exhaustive;

const BOOL decommitNow = flags & CollectMode\_DecommitNow;

const BOOL cacheCleanup = flags & CollectMode\_CacheCleanup;

if (decommitNow)

{

this->inDecommitNowCollection = true;

}

if (exhaustive)

{

this->inExhaustiveCollection = true;

}

if (cacheCleanup)

{

this->inCacheCleanupCollection = true;

}

}

BOOL

Recycler::DoCollectWrapped(CollectionFlags flags)

{

#ifdef CONCURRENT\_GC\_ENABLED

this->skipStack = ((flags & CollectOverride\_SkipStack) != 0);

DebugOnly(this->isConcurrentGCOnIdle = (flags == CollectOnScriptIdle));

#endif

this->allowDispose = (flags & CollectOverride\_AllowDispose) == CollectOverride\_AllowDispose;

BOOL collected = collectionWrapper->ExecuteRecyclerCollectionFunction(this, &Recycler::DoCollect, flags);

#ifdef CONCURRENT\_GC\_ENABLED

Assert(IsConcurrentExecutingState() || IsConcurrentFinishedState() || !CollectionInProgress());

#else

Assert(!CollectionInProgress());

#endif

return collected;

}

bool

Recycler::NeedExhaustiveRepeatCollect() const

{

return this->inExhaustiveCollection && this->hasExhaustiveCandidate;

}

BOOL

Recycler::DoCollect(CollectionFlags flags)

{

// ExecuteRecyclerCollectionFunction may cause exception. In which case, we may trigger the assert

// in SetupPostCollectionFlags because we didn't reset the inExhausitvECollection variable if

// an exception. We are not in DoCollect, there shouldn't be any more exception. Reset the flag

DebugOnly(this->hasIncompletedDoCollect = false);

#ifdef RECYCLER\_MEMORY\_VERIFY

this->Verify(Js::RecyclerPhase);

#endif

#ifdef RECYCLER\_FINALIZE\_CHECK

autoHeap.VerifyFinalize();

#endif

#ifdef PARTIAL\_GC\_ENABLED

BOOL partial = flags & CollectMode\_Partial;

#if DBG && defined(RECYCLER\_DUMP\_OBJECT\_GRAPH)

// Can't pass in RecyclerPartialStress and DumpObjectGraphOnCollect or call CollectGarbage with DumpObjectGraph

if (GetRecyclerFlagsTable().RecyclerPartialStress) {

Assert(!GetRecyclerFlagsTable().DumpObjectGraphOnCollect && !this->dumpObjectOnceOnCollect);

} else if (GetRecyclerFlagsTable().DumpObjectGraphOnCollect || this->dumpObjectOnceOnCollect) {

Assert(!GetRecyclerFlagsTable().RecyclerPartialStress);

}

#endif

#ifdef RECYCLER\_STRESS

if (partial && GetRecyclerFlagsTable().RecyclerPartialStress)

{

this->inPartialCollectMode = true;

this->forcePartialScanStack = true;

}

#endif

#endif

#ifdef RECYCLER\_DUMP\_OBJECT\_GRAPH

if (dumpObjectOnceOnCollect || GetRecyclerFlagsTable().DumpObjectGraphOnCollect)

{

DumpObjectGraph();

dumpObjectOnceOnCollect = false;

// Can't do a partial collect if DumpObjectGraph is set since it'll call FinishPartial

// which will set inPartialCollectMode to false.

partial = false;

}

#endif

#ifdef CONCURRENT\_GC\_ENABLED

const bool concurrent = (flags & CollectMode\_Concurrent) != 0;

const BOOL forceInThread = flags & CollectOverride\_ForceInThread;

#endif

// Flush the pending dispose objects first if dispose is allowed

Assert(!this->CollectionInProgress());

Assert(this->backgroundFinishMarkCount == 0);

bool collected = FinishDisposeObjects();

do

{

INC\_TIMESTAMP\_FIELD(exhaustiveRepeatedCount);

RECORD\_TIMESTAMP(currentCollectionStartTime);

this->telemetryBlock->currentCollectionStartProcessUsedBytes = PageAllocator::GetProcessUsedBytes();

#ifdef CONCURRENT\_GC\_ENABLED

// DisposeObject may call script again and start another GC, so we may still be in concurrent GC state

if (this->CollectionInProgress())

{

Assert(this->IsConcurrentState());

Assert(collected);

if (forceInThread)

{

return this->FinishConcurrentCollect(flags);

}

return true;

}

#endif

Assert(this->backgroundFinishMarkCount == 0);

#if DBG

collectionCount++;

#endif

collectionState = Collection\_PreCollection;

collectionWrapper->PreCollectionCallBack(flags);

collectionState = CollectionStateNotCollecting;

hasExhaustiveCandidate = false; // reset the candidate detection

#ifdef RECYCLER\_STATS

#ifdef PARTIAL\_GC\_ENABLED

RecyclerCollectionStats oldCollectionStats = collectionStats;

#endif

memset(&collectionStats, 0, sizeof(RecyclerCollectionStats));

this->collectionStats.startCollectAllocBytes = autoHeap.uncollectedAllocBytes;

#ifdef PARTIAL\_GC\_ENABLED

this->collectionStats.startCollectNewPageCount = autoHeap.uncollectedNewPageCount;

this->collectionStats.uncollectedNewPageCountPartialCollect = this->uncollectedNewPageCountPartialCollect;

#endif

#endif

#ifdef PARTIAL\_GC\_ENABLED

if (partial)

{

#ifdef CONCURRENT\_GC\_ENABLED

Assert(!forceInThread);

#endif

#ifdef RECYCLER\_STATS

// We are only doing a partial GC, copy some old stats

collectionStats.finalizeCount = oldCollectionStats.finalizeCount;

memcpy(collectionStats.heapBlockCount, oldCollectionStats.smallNonLeafHeapBlockPartialUnusedCount,

sizeof(oldCollectionStats.smallNonLeafHeapBlockPartialUnusedCount));

memcpy(collectionStats.heapBlockFreeByteCount, oldCollectionStats.smallNonLeafHeapBlockPartialUnusedBytes,

sizeof(oldCollectionStats.smallNonLeafHeapBlockPartialUnusedBytes));

memcpy(collectionStats.smallNonLeafHeapBlockPartialUnusedCount, oldCollectionStats.smallNonLeafHeapBlockPartialUnusedCount,

sizeof(oldCollectionStats.smallNonLeafHeapBlockPartialUnusedCount));

memcpy(collectionStats.smallNonLeafHeapBlockPartialUnusedBytes, oldCollectionStats.smallNonLeafHeapBlockPartialUnusedBytes,

sizeof(oldCollectionStats.smallNonLeafHeapBlockPartialUnusedBytes));

#endif

Assert(enablePartialCollect && inPartialCollectMode);

if (!this->PartialCollect(concurrent))

{

return collected;

}

// This disable partial if we do a repeated exhaustive GC

partial = false;

collected = true;

continue;

}

// Not doing partial collect, we should decommit on finish collect

decommitOnFinish = true;

if (inPartialCollectMode)

{

// finish the partial collect first

FinishPartialCollect();

// Old heap block with free object is made available, count that as being collected

collected = true;

// PARTIAL-GC-CONSIDER: should we just pretend we did a GC, since we have made the free listed object

// available to be used, instead of starting off another GC?

}

#endif

#ifdef CONCURRENT\_GC\_ENABLED

bool skipConcurrent = false;

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

// If the below flag is passed in, skip doing a non-blocking concurrent collect. Instead,

// we will do a blocking concurrent collect, which is basically an in-thread GC

skipConcurrent = GetRecyclerFlagsTable().ForceBlockingConcurrentCollect;

#endif

// We are about to start a collection. Reset our heuristic counters now, so that

// any allocations that occur during concurrent collection count toward the next collection's threshold.

ResetHeuristicCounters();

if (concurrent && !skipConcurrent)

{

Assert(!forceInThread);

if (enableConcurrentMark)

{

if (StartBackgroundMarkCollect())

{

// Tell the caller whether we have finish a collection and there maybe free object to reuse

return collected;

}

// Either ResetWriteWatch failed or the thread service failed

// So concurrent mark is disabled, at least for now

}

if (enableConcurrentSweep)

{

if (StartConcurrentSweepCollect())

{

collected = true;

continue;

}

// out of memory during collection

return collected;

}

// concurrent collection failed, default back to non-concurrent collection

}

if (!forceInThread && enableConcurrentMark)

{

if (!CollectOnConcurrentThread())

{

// time out or out of memory during collection

return collected;

}

}

else

#endif

{

if (!CollectOnAllocatorThread())

{

// out of memory during collection

return collected;

}

}

collected = true;

#ifdef RECYCLER\_TRACE

collectionParam.repeat = true;

#endif

}

while (this->NeedExhaustiveRepeatCollect());

#ifdef CONCURRENT\_GC\_ENABLED

// DisposeObject may call script again and start another GC, so we may still be in concurrent GC state

if (this->CollectionInProgress())

{

Assert(this->IsConcurrentState());

Assert(collected);

return true;

}

#endif

EndCollection();

// Tell the caller whether we have finish a collection and there maybe free object to reuse

return collected;

}

void

Recycler::EndCollection()

{

Assert(this->backgroundFinishMarkCount == 0);

Assert(!this->CollectionInProgress());

// no more collection is requested, we can turn exhaustive back off

this->inExhaustiveCollection = false;

if (this->inDecommitNowCollection || CUSTOM\_CONFIG\_FLAG(GetRecyclerFlagsTable(), ForceDecommitOnCollect))

{

#ifdef RECYCLER\_TRACE

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::RecyclerPhase))

{

Output::Print(L"%04X> RC(%p): %s\n", this->mainThreadId, this, L"Decommit now");

}

#endif

ForRecyclerPageAllocator(DecommitNow());

this->inDecommitNowCollection = false;

}

RECORD\_TIMESTAMP(lastCollectionEndTime);

}

#ifdef PARTIAL\_GC\_ENABLED

bool

Recycler::PartialCollect(bool concurrent)

{

Assert(IsMarkStackEmpty());

Assert(this->inPartialCollectMode);

Assert(collectionState == CollectionStateNotCollecting);

// Rescan again

collectionState = CollectionStateRescanFindRoots;

if (concurrent && enableConcurrentMark && this->partialConcurrentNextCollection)

{

this->PrepareBackgroundFindRoots();

if (StartConcurrent(CollectionStateConcurrentFinishMark))

{

#ifdef RECYCLER\_TRACE

PrintCollectTrace(Js::ConcurrentPartialCollectPhase);

#endif

return false;

}

this->RevertPrepareBackgroundFindRoots();

}

#ifdef RECYCLER\_STRESS

if (forcePartialScanStack)

{

// Mark the roots since they need not have been marked

// in RecyclerPartialStress mode

this->RootMark(collectionState);

}

#endif

#ifdef RECYCLER\_TRACE

PrintCollectTrace(Js::PartialCollectPhase);

#endif

bool needConcurrentSweep = false;

this->CollectionBegin<Js::PartialCollectPhase>();

size\_t rescanRootBytes = FinishMark(INFINITE);

Assert(rescanRootBytes != Recycler::InvalidScanRootBytes);

needConcurrentSweep = this->Sweep(rescanRootBytes, concurrent, true);

this->CollectionEnd<Js::PartialCollectPhase>();

// Only reset the new page counter

autoHeap.uncollectedNewPageCount = 0;

// Finish collection

FinishCollection(needConcurrentSweep);

return true;

}

void

Recycler::ProcessClientTrackedObjects()

{

GCETW(GC\_PROCESS\_CLIENT\_TRACKED\_OBJECT\_START, (this));

Assert(this->inPartialCollectMode);

#ifdef CONCURRENT\_GC\_ENABLED

Assert(!this->DoQueueTrackedObject());

#endif

if (!this->clientTrackedObjectList.Empty())

{

SListBase<void \*>::Iterator iter(&this->clientTrackedObjectList);

while (iter.Next())

{

auto& reference = iter.Data();

this->TryMarkNonInterior(reference, &reference /\* parentReference \*/); // Reference to inside the node

RECYCLER\_STATS\_INC(this, clientTrackedObjectCount);

}

this->clientTrackedObjectList.Clear(&this->clientTrackedObjectAllocator);

}

GCETW(GC\_PROCESS\_CLIENT\_TRACKED\_OBJECT\_STOP, (this));

}

void

Recycler::ClearPartialCollect()

{

Assert(!this->DoQueueTrackedObject());

this->autoHeap.unusedPartialCollectFreeBytes = 0;

this->partialUncollectedAllocBytes = 0;

this->clientTrackedObjectList.Clear(&this->clientTrackedObjectAllocator);

this->uncollectedNewPageCountPartialCollect = (size\_t)-1;

}

void

Recycler::FinishPartialCollect(RecyclerSweep \* recyclerSweep)

{

Assert(recyclerSweep == nullptr || !recyclerSweep->IsBackground());

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::FinishPartialPhase);

Assert(inPartialCollectMode);

#ifdef CONCURRENT\_GC\_ENABLED

Assert(!this->DoQueueTrackedObject());

#endif

autoHeap.FinishPartialCollect(recyclerSweep);

this->inPartialCollectMode = false;

ClearPartialCollect();

RECYCLER\_PROFILE\_EXEC\_END(this, Js::FinishPartialPhase);

}

#endif

void

Recycler::EnsureNotCollecting()

{

#ifdef CONCURRENT\_GC\_ENABLED

FinishConcurrent<ForceFinishCollection>();

#endif

Assert(!this->CollectionInProgress());

}

void Recycler::EnumerateObjects(ObjectInfoBits infoBits, void (\*CallBackFunction)(void \* address, size\_t size))

{

// Make sure we are not collecting

EnsureNotCollecting();

#ifdef PARTIAL\_GC\_ENABLED

// We are updating the free bit vector, messing up the partial collection state.

// Just get out of partial collect mode

// GC-CONSIDER: consider adding an option in FinishConcurrent to not get into partial collect mode during sweep.

if (inPartialCollectMode)

{

FinishPartialCollect();

}

#endif

autoHeap.EnumerateObjects(infoBits, CallBackFunction);

// GC-TODO: Explicit heap?

}

BOOL

Recycler::IsMarkState() const

{

return (collectionState & Collection\_Mark);

}

BOOL

Recycler::IsFindRootsState() const

{

return (collectionState & Collection\_FindRoots);

}

#if DBG

BOOL

Recycler::IsReentrantState() const

{

#ifdef CONCURRENT\_GC\_ENABLED

return !this->CollectionInProgress() || this->IsConcurrentState();

#else

return !this->CollectionInProgress();

#endif

}

#endif

#ifdef ENABLE\_JS\_ETW

template <Js::Phase phase> static ETWEventGCActivationKind GetETWEventGCActivationKind();

template <> static ETWEventGCActivationKind GetETWEventGCActivationKind<Js::GarbageCollectPhase>() { return ETWEvent\_GarbageCollect; }

template <> static ETWEventGCActivationKind GetETWEventGCActivationKind<Js::ThreadCollectPhase>() { return ETWEvent\_ThreadCollect; }

template <> static ETWEventGCActivationKind GetETWEventGCActivationKind<Js::ConcurrentCollectPhase>() { return ETWEvent\_ConcurrentCollect; }

template <> static ETWEventGCActivationKind GetETWEventGCActivationKind<Js::PartialCollectPhase>() { return ETWEvent\_PartialCollect; }

#endif

template <Js::Phase phase>

void

Recycler::CollectionBegin()

{

RECYCLER\_PROFILE\_EXEC\_BEGIN2(this, Js::RecyclerPhase, phase);

GCETW(GC\_START, (this, GetETWEventGCActivationKind<phase>()));

#ifdef ENABLE\_BASIC\_TELEMETRY

if (this->IsMemProtectMode() == false)

{

gcTel.LogGCPauseStartTime();

}

#endif

}

template <Js::Phase phase>

void

Recycler::CollectionEnd()

{

GCETW(GC\_STOP, (this, GetETWEventGCActivationKind<phase>()));

#ifdef ENABLE\_BASIC\_TELEMETRY

if (GetCurrentThreadContextId() == mainThreadId && IsMemProtectMode() == false)

{

gcTel.LogGCPauseEndTime();

}

#endif

RECYCLER\_PROFILE\_EXEC\_END2(this, phase, Js::RecyclerPhase);

}

#ifdef CONCURRENT\_GC\_ENABLED

size\_t

Recycler::BackgroundRescan(RescanFlags rescanFlags)

{

Assert(!this->isProcessingRescan);

DebugOnly(this->isProcessingRescan = true);

GCETW(GC\_BACKGROUNDRESCAN\_START, (this, backgroundRescanCount));

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(this, Js::BackgroundRescanPhase);

size\_t rescannedPageCount = heapBlockMap.Rescan(this, ((rescanFlags & RescanFlags\_ResetWriteWatch) != 0));

rescannedPageCount += autoHeap.Rescan(rescanFlags);

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(this, Js::BackgroundRescanPhase);

GCETW(GC\_BACKGROUNDRESCAN\_STOP, (this, backgroundRescanCount));

this->backgroundRescanCount++;

if (!this->NeedOOMRescan())

{

if ((rescanFlags & RescanFlags\_ResetWriteWatch) != 0)

{

DebugOnly(this->isProcessingRescan = false);

}

return rescannedPageCount;

}

DebugOnly(this->isProcessingRescan = false);

return Recycler::InvalidScanRootBytes;

}

void

Recycler::BackgroundResetWriteWatchAll()

{

GCETW(GC\_BACKGROUNDRESETWRITEWATCH\_START, (this, -1));

heapBlockMap.ResetWriteWatch(this);

GCETW(GC\_BACKGROUNDRESETWRITEWATCH\_STOP, (this, -1));

}

#endif

#if defined(PARTIAL\_GC\_ENABLED) || defined(CONCURRENT\_GC\_ENABLED)

size\_t

Recycler::FinishMarkRescan(bool background)

{

if (background)

{

GCETW(GC\_BACKGROUNDRESCAN\_START, (this, 0));

}

else

{

GCETW(GC\_RESCAN\_START, (this));

}

RECYCLER\_PROFILE\_EXEC\_THREAD\_BEGIN(background, this, Js::RescanPhase);

#ifdef CONCURRENT\_GC\_ENABLED

RescanFlags const flags = (background ? RescanFlags\_ResetWriteWatch : RescanFlags\_None);

#else

Assert(!background);

RescanFlags const flags = RescanFlags\_None;

#endif

#if DBG

Assert(!this->isProcessingRescan);

this->isProcessingRescan = true;

#endif

size\_t scannedPageCount = heapBlockMap.Rescan(this, ((flags & RescanFlags\_ResetWriteWatch) != 0));

scannedPageCount += autoHeap.Rescan(flags);

DebugOnly(this->isProcessingRescan = false);

RECYCLER\_PROFILE\_EXEC\_THREAD\_END(background, this, Js::RescanPhase);

if (background)

{

GCETW(GC\_BACKGROUNDRESCAN\_STOP, (this, 0));

}

else

{

GCETW(GC\_RESCAN\_STOP, (this));

}

return scannedPageCount;

}

void

Recycler::ProcessTrackedObjects()

{

GCETW(GC\_PROCESS\_TRACKED\_OBJECT\_START, (this));

#ifdef PARTIAL\_GC\_ENABLED

Assert(this->clientTrackedObjectList.Empty());

Assert(!this->inPartialCollectMode);

#endif

#ifdef CONCURRENT\_GC\_ENABLED

Assert(this->DoQueueTrackedObject());

this->queueTrackedObject = false;

DebugOnly(this->isProcessingTrackedObjects = true);

markContext.ProcessTracked();

// If we did a parallel mark, we need to process any queued tracked objects from the parallel mark stack as well.

// If we didn't, this will do nothing.

parallelMarkContext1.ProcessTracked();

parallelMarkContext2.ProcessTracked();

parallelMarkContext3.ProcessTracked();

DebugOnly(this->isProcessingTrackedObjects = false);

#endif

GCETW(GC\_PROCESS\_TRACKED\_OBJECT\_STOP, (this));

}

#endif // defined(PARTIAL\_GC\_ENABLED) || defined(CONCURRENT\_GC\_ENABLED)

BOOL

Recycler::RequestConcurrentWrapperCallback()

{

#ifdef CONCURRENT\_GC\_ENABLED

Assert(!IsConcurrentExecutingState());

// Save the original collection state

CollectionState oldState = this->collectionState;

// Get the background thread to start the callback

if (StartConcurrent(CollectionStateConcurrentWrapperCallback))

{

// Wait for the callback to complete

WaitForConcurrentThread(INFINITE);

// The state must not change back until we restore the original state

Assert(collectionState == CollectionStateConcurrentWrapperCallback);

this->collectionState = oldState;

return true;

}

#endif

return false;

}

#ifdef CONCURRENT\_GC\_ENABLED

/\*------------------------------------------------------------------------------------------------

\* Concurrent

\*------------------------------------------------------------------------------------------------\*/

BOOL

Recycler::CollectOnConcurrentThread()

{

#ifdef PARTIAL\_GC\_ENABLED

Assert(!inPartialCollectMode);

#endif

#ifdef RECYCLER\_TRACE

PrintCollectTrace(Js::ThreadCollectPhase);

#endif

this->CollectionBegin<Js::ThreadCollectPhase>();

// Synchronous concurrent mark

if (!StartSynchronousBackgroundMark())

{

this->CollectionEnd<Js::ThreadCollectPhase>();

return false;

}

const DWORD waitTime = RecyclerHeuristic::FinishConcurrentCollectWaitTime(this->GetRecyclerFlagsTable());

GCETW(GC\_SYNCHRONOUSMARKWAIT\_START, (this, waitTime));

const BOOL waited = WaitForConcurrentThread(waitTime);

GCETW(GC\_SYNCHRONOUSMARKWAIT\_STOP, (this, !waited));

if (!waited)

{

#ifdef RECYCLER\_TRACE

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::RecyclerPhase)

|| GetRecyclerFlagsTable().Trace.IsEnabled(Js::ThreadCollectPhase))

{

Output::Print(L"%04X> RC(%p): %s: %s\n", this->mainThreadId, this, Js::PhaseNames[Js::ThreadCollectPhase], L"Timeout");

}

#endif

this->CollectionEnd<Js::ThreadCollectPhase>();

return false;

}

// If the concurrent thread was done within the time limit, there shouldn't be

// any object needs to be rescanned

// CONCURRENT-TODO: Optimize it so we don't rescan in the background if we are still waiting

// GC-TODO: Unfortunately we can't assert this, as the background code gen thread may still

// touch GC memory (e.g. FunctionBody), causing write watch and rescan

// in the background.

// Assert(markContext.Empty());

DebugOnly(this->isProcessingRescan = false);

this->collectionState = CollectionStateMark;

this->ProcessTrackedObjects();

this->ProcessMark(false);

this->EndMark();

// Partial collect mode is not re-enabled after a non-partial in-thread GC because partial GC heuristics are not adjusted

// after a full in-thread GC. Enabling partial collect mode causes partial GC heuristics to be reset before the next full

// in-thread GC, thereby allowing partial GC to kick in more easily without being able to adjust heuristics after the full

// GCs. Until we have a way of adjusting partial GC heuristics after a full in-thread GC, once partial collect mode is

// turned off, it will remain off until a concurrent GC happens

this->Sweep();

this->CollectionEnd<Js::ThreadCollectPhase>();

FinishCollection();

return true;

}

// explicit instantiation

template BOOL Recycler::FinishConcurrent<FinishConcurrentOnIdle>();

template BOOL Recycler::FinishConcurrent<FinishConcurrentOnIdleAtRoot>();

template BOOL Recycler::FinishConcurrent<FinishConcurrentOnExitScript>();

template BOOL Recycler::FinishConcurrent<FinishConcurrentOnEnterScript>();

template BOOL Recycler::FinishConcurrent<ForceFinishCollection>();

template <CollectionFlags flags>

BOOL

Recycler::FinishConcurrent()

{

CompileAssert((flags & ~(CollectOverride\_AllowDispose | CollectOverride\_ForceFinish | CollectOverride\_ForceInThread

| CollectMode\_Concurrent | CollectOverride\_DisableIdleFinish | CollectOverride\_BackgroundFinishMark

| CollectOverride\_SkipStack | CollectOverride\_FinishConcurrentTimeout)) == 0);

if (this->CollectionInProgress())

{

Assert(this->IsConcurrentEnabled());

Assert(IsConcurrentState());

const BOOL forceFinish = flags & CollectOverride\_ForceFinish;

if (forceFinish || !IsConcurrentExecutingState())

{

if (this->collectionState == CollectionStateConcurrentSweep)

{

// Help with the background thread to zero and flush zero pages

// if we are going to wait anyways.

recyclerPageAllocator.ZeroQueuedPages();

recyclerLargeBlockPageAllocator.ZeroQueuedPages();

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.ZeroQueuedPages();

#endif

this->FlushBackgroundPages();

}

#ifdef RECYCLER\_TRACE

collectionParam.finishOnly = true;

collectionParam.flags = flags;

#endif

#ifdef CONCURRENT\_GC\_ENABLED

// If SkipStack is provided, and we're not forcing the finish (i.e we're not in concurrent executing state)

// then, it's fine to set the skipStack flag to true, so that during the in-thread find-roots, we'll skip

// the stack scan

this->skipStack = ((flags & CollectOverride\_SkipStack) != 0) && !forceFinish;

#if DBG

this->isFinishGCOnIdle = (flags == FinishConcurrentOnIdleAtRoot);

#endif

#endif

return FinishConcurrentCollectWrapped(flags);

}

}

return false;

}

template <CollectionFlags flags>

BOOL

Recycler::TryFinishConcurrentCollect()

{

Assert(this->CollectionInProgress());

RECYCLER\_STATS\_INC(this, finishCollectTryCount);

SetupPostCollectionFlags<flags>();

const BOOL concurrent = flags & CollectMode\_Concurrent;

const BOOL forceInThread = flags & CollectOverride\_ForceInThread;

Assert(this->IsConcurrentEnabled());

Assert(IsConcurrentState() || IsCollectionDisabled());

Assert(!concurrent || !forceInThread);

if (concurrent && concurrentThread != NULL)

{

if (IsConcurrentExecutingState())

{

if (!this->priorityBoost)

{

uint tickCount = GetTickCount();

if ((autoHeap.uncollectedAllocBytes > RecyclerHeuristic::Instance.UncollectedAllocBytesConcurrentPriorityBoost)

|| (tickCount - this->tickCountStartConcurrent > RecyclerHeuristic::PriorityBoostTimeout(this->GetRecyclerFlagsTable())))

{

#ifdef RECYCLER\_TRACE

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::RecyclerPhase))

{

Output::Print(L"%04X> RC(%p): %s: ", this->mainThreadId, this, L"Set priority normal");

if (autoHeap.uncollectedAllocBytes > RecyclerHeuristic::Instance.UncollectedAllocBytesConcurrentPriorityBoost)

{

Output::Print(L"AllocBytes=%d (Time=%d)\n", autoHeap.uncollectedAllocBytes, tickCount - this->tickCountStartConcurrent);

}

else

{

Output::Print(L"Time=%d (AllocBytes=%d\n", tickCount - this->tickCountStartConcurrent, autoHeap.uncollectedAllocBytes);

}

}

#endif

// Set it to a large number so we don't set the thread priority again

this->priorityBoost = true;

// The recycler thread hasn't come back in 5 seconds

// It either has a large object graph, or it is starving.

// Set the priority back to normal

SetThreadPriority(this->concurrentThread, THREAD\_PRIORITY\_NORMAL);

}

}

return FinishDisposeObjectsWrapped<flags>();

}

else if ((flags & CollectOverride\_FinishConcurrentTimeout) != 0)

{

uint tickCount = GetTickCount();

// If we haven't gone past the time to call finish collection,

// simply call FinishDisposeObjects and return

// Otherwise, actually go ahead and call FinishConcurrentCollectWrapped

// We do this only if this is a collection that allows finish concurrent to timeout

// If not, by default, we finish the collection

if (tickCount <= this->tickCountNextFinishCollection)

{

return FinishDisposeObjectsWrapped<flags>();

}

}

}

return FinishConcurrentCollectWrapped(flags);

}

BOOL

Recycler::IsConcurrentMarkState() const

{

return (collectionState & Collection\_ConcurrentMark) == Collection\_ConcurrentMark;

}

BOOL

Recycler::IsConcurrentMarkExecutingState() const

{

return (collectionState & (Collection\_ConcurrentMark | Collection\_ExecutingConcurrent)) == (Collection\_ConcurrentMark | Collection\_ExecutingConcurrent);

}

BOOL

Recycler::IsConcurrentResetMarksState() const

{

return collectionState == CollectionStateConcurrentResetMarks;

}

BOOL

Recycler::IsInThreadFindRootsState() const

{

CollectionState currentCollectionState = collectionState;

return (currentCollectionState & Collection\_FindRoots) && (currentCollectionState != CollectionStateConcurrentFindRoots);

}

BOOL

Recycler::IsConcurrentFindRootState() const

{

return collectionState == CollectionStateConcurrentFindRoots;

}

BOOL

Recycler::IsConcurrentExecutingState() const

{

return (collectionState & Collection\_ExecutingConcurrent);

}

BOOL

Recycler::IsConcurrentSweepExecutingState() const

{

return (collectionState & (Collection\_ConcurrentSweep | Collection\_ExecutingConcurrent)) == (Collection\_ConcurrentSweep | Collection\_ExecutingConcurrent);

}

BOOL

Recycler::IsConcurrentState() const

{

return (collectionState & Collection\_Concurrent);

}

#if DBG

BOOL

Recycler::IsConcurrentFinishedState() const

{

return (collectionState & Collection\_FinishConcurrent);

}

#endif

bool

Recycler::InitializeConcurrent(JsUtil::ThreadService \*threadService)

{

try

{

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE(ExceptionType\_OutOfMemory);

concurrentWorkDoneEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

if (concurrentWorkDoneEvent == nullptr)

{

throw Js::OutOfMemoryException();

}

#if DBG\_DUMP

markContext.GetPageAllocator()->debugName = L"ConcurrentCollect";

#endif

if (!threadService->HasCallback())

{

#ifdef IDLE\_DECOMMIT\_ENABLED

concurrentIdleDecommitEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

if (concurrentIdleDecommitEvent == nullptr)

{

throw Js::OutOfMemoryException();

}

#endif

concurrentWorkReadyEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

if (concurrentWorkReadyEvent == nullptr)

{

throw Js::OutOfMemoryException();

}

}

}

catch (Js::OutOfMemoryException)

{

Assert(concurrentWorkReadyEvent == nullptr);

if (concurrentWorkDoneEvent)

{

CloseHandle(concurrentWorkDoneEvent);

concurrentWorkDoneEvent = nullptr;

}

#ifdef IDLE\_DECOMMIT\_ENABLED

if (concurrentIdleDecommitEvent)

{

CloseHandle(concurrentIdleDecommitEvent);

concurrentIdleDecommitEvent = nullptr;

}

#endif

return false;

}

return true;

}

#pragma prefast(suppress:6262, "Where this function is call should have ample of stack space")

bool Recycler::AbortConcurrent(bool restoreState)

{

Assert(!this->CollectionInProgress() || this->IsConcurrentState());

// In case the thread already died, wait for that too

HANDLE handle[2] = { concurrentWorkDoneEvent, concurrentThread };

// Note, concurrentThread will be null if we have a threadService.

Assert(concurrentThread != NULL || threadService->HasCallback());

DWORD handleCount = (concurrentThread == NULL ? 1 : 2);

DWORD ret = WAIT\_OBJECT\_0;

if (this->IsConcurrentState())

{

this->isAborting = true;

if (this->concurrentThread != NULL)

{

SetThreadPriority(this->concurrentThread, THREAD\_PRIORITY\_NORMAL);

}

ret = WaitForMultipleObjectsEx(handleCount, handle, FALSE, INFINITE, FALSE);

this->isAborting = false;

Assert(this->IsConcurrentFinishedState() || ret == WAIT\_OBJECT\_0 + 1);

if (ret == WAIT\_OBJECT\_0 && restoreState)

{

if (collectionState == CollectionStateRescanWait)

{

this->ResetMarkCollectionState();

}

else if (collectionState == CollectionStateTransferSweptWait)

{

// Make sure we don't do another GC after finishing this one.

this->inExhaustiveCollection = false;

// Let's just finish the sweep so that GC is in a consistent state, but don't run dispose

// AbortConcurrent already consumed the event from the concurrent thread, just signal it so

// FinishConcurrentCollect can wait for it again.

SetEvent(this->concurrentWorkDoneEvent);

EnsureNotCollecting();

}

else

{

Assert(UNREACHED);

}

Assert(collectionState == CollectionStateNotCollecting);

Assert(this->isProcessingRescan == false);

}

else

{

// Even if we weren't asked to restore states, we need to clean up the pending guest arena

CleanupPendingUnroot();

// Also need to release any pages held by the mark stack, if we abandoned it

markContext.Abort();

}

}

Assert(!this->hasPendingDeleteGuestArena);

return ret == WAIT\_OBJECT\_0;

}

void

Recycler::CleanupPendingUnroot()

{

Assert(!this->hasPendingConcurrentFindRoot);

if (hasPendingUnpinnedObject)

{

pinnedObjectMap.MapAndRemoveIf([](void \* obj, PinRecord const &refCount)

{

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

Assert(refCount != 0 || refCount.stackBackTraces == nullptr);

#endif

return refCount == 0;

});

hasPendingUnpinnedObject = false;

}

if (hasPendingDeleteGuestArena)

{

DebugOnly(bool foundPendingDelete = false);

DListBase<GuestArenaAllocator>::EditingIterator guestArenaIter(&guestArenaList);

while (guestArenaIter.Next())

{

GuestArenaAllocator& allocator = guestArenaIter.Data();

if (allocator.pendingDelete)

{

allocator.SetLockBlockList(false);

guestArenaIter.RemoveCurrent(&HeapAllocator::Instance);

DebugOnly(foundPendingDelete = true);

}

}

hasPendingDeleteGuestArena = false;

Assert(foundPendingDelete);

}

#if DBG

else

{

DListBase<GuestArenaAllocator>::Iterator guestArenaIter(&guestArenaList);

while (guestArenaIter.Next())

{

GuestArenaAllocator& allocator = guestArenaIter.Data();

Assert(!allocator.pendingDelete);

}

}

#endif

}

void

Recycler::FinalizeConcurrent(bool restoreState)

{

bool needCleanExitState = restoreState;

#if defined(RECYCLER\_DUMP\_OBJECT\_GRAPH)

needCleanExitState = needCleanExitState || GetRecyclerFlagsTable().DumpObjectGraphOnExit;

#endif

#ifdef LEAK\_REPORT

needCleanExitState = needCleanExitState || GetRecyclerFlagsTable().IsEnabled(Js::LeakReportFlag);

#endif

#ifdef CHECK\_MEMORY\_LEAK

needCleanExitState = needCleanExitState || GetRecyclerFlagsTable().CheckMemoryLeak;

#endif

bool aborted = AbortConcurrent(needCleanExitState);

collectionState = CollectionStateExit;

if (aborted && this->concurrentThread != NULL)

{

// In case the thread already died, wait for that too

HANDLE handle[2] = { concurrentWorkDoneEvent, concurrentThread };

SetEvent(concurrentWorkReadyEvent);

SetThreadPriority(this->concurrentThread, THREAD\_PRIORITY\_NORMAL);

// In case the thread already died, wait for that too

DWORD fRet = WaitForMultipleObjectsEx(2, handle, FALSE, INFINITE, FALSE);

AssertMsg(fRet != WAIT\_FAILED, "Check handles passed to WaitForMultipleObjectsEx.");

}

// Shutdown parallel threads and return the handle for them so the caller can

// close it.

parallelThread1.Shutdown();

parallelThread2.Shutdown();

#ifdef IDLE\_DECOMMIT\_ENABLED

if (concurrentIdleDecommitEvent != nullptr)

{

CloseHandle(concurrentIdleDecommitEvent);

concurrentIdleDecommitEvent = nullptr;

}

#endif

CloseHandle(concurrentWorkDoneEvent);

concurrentWorkDoneEvent = nullptr;

if (concurrentWorkReadyEvent != NULL)

{

CloseHandle(concurrentWorkReadyEvent);

concurrentWorkReadyEvent = nullptr;

}

if (needCleanExitState)

{

// We may do another marking pass to look for memory leaks;

// Since we have shut down the concurrent thread, don't do a parallel mark.

this->enableConcurrentMark = false;

this->enableParallelMark = false;

this->enableConcurrentSweep = false;

}

this->threadService = nullptr;

this->concurrentThread = nullptr;

}

bool

Recycler::EnableConcurrent(JsUtil::ThreadService \*threadService, bool startAllThreads)

{

if (this->disableConcurrent)

{

return false;

}

if (!this->InitializeConcurrent(threadService))

{

return false;

}

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

this->enableConcurrentMark = !CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::ConcurrentMarkPhase);

this->enableParallelMark = !CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::ParallelMarkPhase);

this->enableConcurrentSweep = !CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::ConcurrentSweepPhase);

#else

this->enableConcurrentMark = true;

this->enableParallelMark = true;

this->enableConcurrentSweep = true;

#endif

if (this->enableParallelMark && this->maxParallelism == 1)

{

// Disable parallel mark if only 1 CPU

this->enableParallelMark = false;

}

if (threadService->HasCallback())

{

this->threadService = threadService;

return true;

}

else

{

bool startConcurrentThread = true;

bool startedParallelThread1 = false;

bool startedParallelThread2 = false;

if (startAllThreads)

{

if (this->enableParallelMark && this->maxParallelism > 2)

{

if (!parallelThread1.EnableConcurrent(true))

{

startConcurrentThread = false;

}

else

{

startedParallelThread1 = true;

if (this->maxParallelism > 3)

{

if (!parallelThread2.EnableConcurrent(true))

{

startConcurrentThread = false;

}

else

{

startedParallelThread2 = true;

}

}

}

}

}

if (startConcurrentThread)

{

HANDLE concurrentThread = (HANDLE)\_beginthreadex(NULL, Recycler::ConcurrentThreadStackSize, &Recycler::StaticThreadProc, this, STACK\_SIZE\_PARAM\_IS\_A\_RESERVATION, NULL);

if (concurrentThread != nullptr)

{

// Wait for recycler thread to initialize

HANDLE handle[2] = { this->concurrentWorkDoneEvent, concurrentThread };

DWORD ret = WaitForMultipleObjectsEx(2, handle, FALSE, INFINITE, FALSE);

if (ret == WAIT\_OBJECT\_0)

{

this->threadService = threadService;

this->concurrentThread = concurrentThread;

return true;

}

CloseHandle(concurrentThread);

}

}

if (startedParallelThread1)

{

parallelThread1.Shutdown();

if (startedParallelThread2)

{

parallelThread2.Shutdown();

}

}

}

// We failed to start a concurrent thread so we set these back to false and clean up

this->enableConcurrentMark = false;

this->enableParallelMark = false;

this->enableConcurrentSweep = false;

if (concurrentWorkReadyEvent)

{

CloseHandle(concurrentWorkReadyEvent);

concurrentWorkReadyEvent = nullptr;

}

if (concurrentWorkDoneEvent)

{

CloseHandle(concurrentWorkDoneEvent);

concurrentWorkDoneEvent = nullptr;

}

#ifdef IDLE\_DECOMMIT\_ENABLED

if (concurrentIdleDecommitEvent)

{

CloseHandle(concurrentIdleDecommitEvent);

concurrentIdleDecommitEvent = nullptr;

}

#endif

return false;

}

void

Recycler::ShutdownThread()

{

if (this->IsConcurrentEnabled())

{

Assert(concurrentThread != NULL || threadService->HasCallback());

FinalizeConcurrent(false);

if (concurrentThread)

{

CloseHandle(concurrentThread);

}

}

}

void

Recycler::DisableConcurrent()

{

if (this->IsConcurrentEnabled())

{

Assert(concurrentThread != NULL || threadService->HasCallback());

FinalizeConcurrent(true);

if (concurrentThread)

{

CloseHandle(concurrentThread);

}

this->collectionState = CollectionStateNotCollecting;

}

}

bool

Recycler::StartConcurrent(CollectionState const state)

{

// Reset the tick count to detect if the concurrent thread is taking too long

tickCountStartConcurrent = GetTickCount();

CollectionState oldState = this->collectionState;

this->collectionState = state;

if (threadService->HasCallback())

{

Assert(concurrentThread == NULL);

Assert(concurrentWorkReadyEvent == NULL);

if (!threadService->Invoke(Recycler::StaticBackgroundWorkCallback, this))

{

this->collectionState = oldState;

return false;

}

return true;

}

else

{

Assert(concurrentThread != NULL);

Assert(concurrentWorkReadyEvent != NULL);

SetEvent(concurrentWorkReadyEvent);

return true;

}

}

BOOL

Recycler::StartBackgroundMarkCollect()

{

#ifdef RECYCLER\_TRACE

PrintCollectTrace(Js::ConcurrentMarkPhase);

#endif

this->CollectionBegin<Js::ConcurrentCollectPhase>();

// Asynchronous concurrent mark

BOOL success = StartAsynchronousBackgroundMark();

this->CollectionEnd<Js::ConcurrentCollectPhase>();

return success;

}

BOOL

Recycler::StartBackgroundMark(bool foregroundResetMark, bool foregroundFindRoots)

{

Assert(!this->CollectionInProgress());

CollectionState backgroundState = CollectionStateConcurrentResetMarks;

bool doBackgroundFindRoots = true;

if (foregroundResetMark || foregroundFindRoots)

{

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::ResetWriteWatchPhase);

bool hasWriteWatch = (recyclerPageAllocator.ResetWriteWatch() && recyclerLargeBlockPageAllocator.ResetWriteWatch());

RECYCLER\_PROFILE\_EXEC\_END(this, Js::ResetWriteWatchPhase);

if (!hasWriteWatch)

{

// Disable concurrent mark

this->enableConcurrentMark = false;

return false;

}

// In-thread synchronized GC on the concurrent thread

ResetMarks(this->enableScanImplicitRoots ? ResetMarkFlags\_SynchronizedImplicitRoots : ResetMarkFlags\_Synchronized);

if (foregroundFindRoots)

{

this->collectionState = CollectionStateFindRoots;

FindRoots();

ScanStack();

Assert(collectionState == CollectionStateFindRoots);

backgroundState = CollectionStateConcurrentMark;

doBackgroundFindRoots = false;

}

else

{

// Do find roots in the background

backgroundState = CollectionStateConcurrentFindRoots;

}

}

if (doBackgroundFindRoots)

{

this->PrepareBackgroundFindRoots();

}

if (!StartConcurrent(backgroundState))

{

if (doBackgroundFindRoots)

{

this->RevertPrepareBackgroundFindRoots();

}

this->collectionState = CollectionStateNotCollecting;

return false;

}

return true;

}

BOOL

Recycler::StartAsynchronousBackgroundMark()

{

// Debug flags to turn off background reset mark or background find roots, default to doing every concurrently

return StartBackgroundMark(CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::BackgroundResetMarksPhase), CUSTOM\_PHASE\_OFF1(GetRecyclerFlagsTable(), Js::BackgroundFindRootsPhase));

}

BOOL

Recycler::StartSynchronousBackgroundMark()

{

return StartBackgroundMark(true, true);

}

BOOL

Recycler::StartConcurrentSweepCollect()

{

Assert(collectionState == CollectionStateNotCollecting);

#ifdef RECYCLER\_TRACE

PrintCollectTrace(Js::ConcurrentSweepPhase);

#endif

this->CollectionBegin<Js::ConcurrentCollectPhase>();

this->Mark();

// We don't have rescan data if we disabled concurrent mark, assume the worst

// (which means it is harder to get into partial collect mode)

bool needConcurrentSweep = this->Sweep(RecyclerSweep::MaxPartialCollectRescanRootBytes, true, true);

this->CollectionEnd<Js::ConcurrentCollectPhase>();

FinishCollection(needConcurrentSweep);

return true;

}

size\_t

Recycler::BackgroundRepeatMark()

{

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(this, Js::BackgroundRepeatMarkPhase);

Assert(this->backgroundRescanCount <= RecyclerHeuristic::MaxBackgroundRepeatMarkCount - 1);

size\_t rescannedPageCount = this->BackgroundRescan(RescanFlags\_ResetWriteWatch);

if (this->NeedOOMRescan() || this->isAborting)

{

// OOM'ed. Let's not continue

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(this, Js::BackgroundRepeatMarkPhase);

return Recycler::InvalidScanRootBytes;

}

// Rescan the stack

this->BackgroundScanStack();

// Process mark stack

this->DoBackgroundParallelMark();

if (this->NeedOOMRescan())

{

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(this, Js::BackgroundRepeatMarkPhase);

return Recycler::InvalidScanRootBytes;

}

#ifdef RECYCLER\_STATS

Assert(this->backgroundRescanCount >= 1 && this->backgroundRescanCount <= RecyclerHeuristic::MaxBackgroundRepeatMarkCount);

this->collectionStats.backgroundMarkData[this->backgroundRescanCount - 1] = this->collectionStats.markData;

#endif

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(this, Js::BackgroundRepeatMarkPhase);

return rescannedPageCount;

}

char\* Recycler::GetScriptThreadStackTop()

{

// We should have already checked if the recycler is thread bound or not

Assert(mainThreadHandle != NULL);

return (char\*) savedThreadContext.GetStackTop();

}

size\_t

Recycler::BackgroundScanStack()

{

if (this->skipStack)

{

#ifdef RECYCLER\_TRACE

CUSTOM\_PHASE\_PRINT\_VERBOSE\_TRACE1(GetRecyclerFlagsTable(), Js::ScanStackPhase, L"[%04X] Skipping the stack scan\n", ::GetCurrentThreadId());

#endif

Output::Print(Js::ScanStackPhase, L"[%04X] Skipping the stack scan\n", ::GetCurrentThreadId());

return 0;

}

if (!this->isInScript || mainThreadHandle == nullptr)

{

// No point in scanning the main thread's stack if we are not in script

// We also can't scan the main thread's stack if we are not thread bounded, and didn't create the main thread's handle

return 0;

}

char\* stackTop = this->GetScriptThreadStackTop();

if (stackTop != nullptr)

{

size\_t size = (char \*)stackBase - stackTop;

ScanMemoryInline((void \*\*)stackTop, size);

return size;

}

return 0;

}

void

Recycler::BackgroundMark()

{

Assert(this->DoQueueTrackedObject());

this->backgroundRescanCount = 0;

this->DoBackgroundParallelMark();

if (this->NeedOOMRescan() || this->isAborting)

{

return;

}

#ifdef RECYCLER\_STATS

this->collectionStats.backgroundMarkData[0] = this->collectionStats.markData;

#endif

if (PHASE\_OFF1(Js::BackgroundRepeatMarkPhase))

{

return;

}

// We always do one repeat mark pass.

size\_t rescannedPageCount = this->BackgroundRepeatMark();

if (this->NeedOOMRescan() || this->isAborting)

{

// OOM'ed. Let's not continue

return;

}

Assert(rescannedPageCount != Recycler::InvalidScanRootBytes);

// If we rescanned enough pages in the previous repeat mark pass, then do one more

// to try to reduce the amount of work we need to do in-thread

if (rescannedPageCount >= RecyclerHeuristic::BackgroundSecondRepeatMarkThreshold)

{

this->BackgroundRepeatMark();

if (this->NeedOOMRescan() || this->isAborting)

{

// OOM'ed. Let's not continue

return;

}

}

}

void

Recycler::BackgroundResetMarks()

{

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(this, Js::BackgroundResetMarksPhase);

GCETW(GC\_BACKGROUNDRESETMARKS\_START, (this));

Assert(IsMarkStackEmpty());

this->scanPinnedObjectMap = true;

this->hasScannedInitialImplicitRoots = false;

heapBlockMap.ResetMarks();

autoHeap.ResetMarks(this->enableScanImplicitRoots ? ResetMarkFlags\_InBackgroundThreadImplicitRoots : ResetMarkFlags\_InBackgroundThread);

GCETW(GC\_BACKGROUNDRESETMARKS\_STOP, (this));

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(this, Js::BackgroundResetMarksPhase);

}

void

Recycler::PrepareBackgroundFindRoots()

{

Assert(!this->hasPendingConcurrentFindRoot);

this->hasPendingConcurrentFindRoot = true;

// Save the thread context here. The background thread

// will use this saved context for the marking instead of

// trying to get the live thread context of the thread

SAVE\_THREAD\_CONTEXT();

// Temporarily disable resize so the background can scan without

// the memory being freed from under it

pinnedObjectMap.DisableResize();

// Update the cached info for big blocks in the guest arena

DListBase<GuestArenaAllocator>::EditingIterator guestArenaIter(&guestArenaList);

while (guestArenaIter.Next())

{

GuestArenaAllocator& allocator = guestArenaIter.Data();

allocator.SetLockBlockList(true);

if (allocator.pendingDelete)

{

Assert(this->hasPendingDeleteGuestArena);

allocator.SetLockBlockList(false);

guestArenaIter.RemoveCurrent(&HeapAllocator::Instance);

}

else if (this->backgroundFinishMarkCount == 0)

{

// Update the cached info for big block

allocator.GetBigBlocks(false);

}

}

this->hasPendingDeleteGuestArena = false;

}

void

Recycler::RevertPrepareBackgroundFindRoots()

{

Assert(this->hasPendingConcurrentFindRoot);

this->hasPendingConcurrentFindRoot = false;

pinnedObjectMap.EnableResize();

}

size\_t

Recycler::BackgroundFindRoots()

{

#ifdef RECYCLER\_STATS

size\_t lastMarkCount = this->collectionStats.markData.markCount;

#endif

size\_t scanRootBytes = 0;

Assert(this->IsConcurrentFindRootState());

Assert(this->hasPendingConcurrentFindRoot);

Assert(this->inPartialCollectMode || this->DoQueueTrackedObject());

// Only mark pinned object and guest arenas, which is where most of the roots are.

// When we go back to the main thread to rescan, we will scan the rest of the root.

// NOTE: purposefully not marking the transientPinnedObject there. as it is transient :)

// background mark the pinned object. Since we are in concurrent find root state

// the main thread won't delete any entries from the map, so concurrent read

// to the map safe.

GCETW(GC\_BACKGROUNDSCANROOTS\_START, (this));

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(this, Js::BackgroundFindRootsPhase);

scanRootBytes += this->ScanPinnedObjects</\*background = \*/true>();

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(this, Js::FindRootArenaPhase);

// background mark the guest arenas. Since we are in concurrent find root state

// the main thread won't delete any arena, so concurrent reads to them are ok.

DListBase<GuestArenaAllocator>::EditingIterator guestArenaIter(&guestArenaList);

while (guestArenaIter.Next())

{

GuestArenaAllocator& allocator = guestArenaIter.Data();

if (allocator.pendingDelete)

{

// Skip guest arena that are already marked for delete

Assert(this->hasPendingDeleteGuestArena);

continue;

}

scanRootBytes += ScanArena(&allocator, true);

}

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(this, Js::FindRootArenaPhase);

this->ScanImplicitRoots();

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(this, Js::BackgroundFindRootsPhase);

this->hasPendingConcurrentFindRoot = false;

this->collectionState = CollectionStateConcurrentMark;

GCETW(GC\_BACKGROUNDSCANROOTS\_STOP, (this));

RECYCLER\_STATS\_ADD(this, rootCount, this->collectionStats.markData.markCount - lastMarkCount);

return scanRootBytes;

}

size\_t

Recycler::BackgroundFinishMark()

{

Assert(this->inPartialCollectMode || this->DoQueueTrackedObject());

Assert(collectionState == CollectionStateConcurrentFinishMark);

size\_t rescannedRootBytes = FinishMarkRescan(true) \* AutoSystemInfo::PageSize;

this->collectionState = CollectionStateConcurrentFindRoots;

rescannedRootBytes += this->BackgroundFindRoots();

this->collectionState = CollectionStateConcurrentFinishMark;

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(this, Js::MarkPhase);

ProcessMark(true);

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(this, Js::MarkPhase);

return rescannedRootBytes;

}

void

Recycler::SweepPendingObjects(RecyclerSweep& recyclerSweep)

{

autoHeap.SweepPendingObjects(recyclerSweep);

}

void

Recycler::ConcurrentTransferSweptObjects(RecyclerSweep& recyclerSweep)

{

Assert(!recyclerSweep.IsBackground());

Assert((this->collectionState & Collection\_TransferSwept) == Collection\_TransferSwept);

if (this->hasBackgroundFinishPartial)

{

this->hasBackgroundFinishPartial = false;

this->ClearPartialCollect();

}

autoHeap.ConcurrentTransferSweptObjects(recyclerSweep);

}

#ifdef PARTIAL\_GC\_ENABLED

void

Recycler::ConcurrentPartialTransferSweptObjects(RecyclerSweep& recyclerSweep)

{

Assert(!recyclerSweep.IsBackground());

Assert(!this->hasBackgroundFinishPartial);

autoHeap.ConcurrentPartialTransferSweptObjects(recyclerSweep);

}

#endif

BOOL

Recycler::FinishConcurrentCollectWrapped(CollectionFlags flags)

{

this->allowDispose = (flags & CollectOverride\_AllowDispose) == CollectOverride\_AllowDispose;

#ifdef CONCURRENT\_GC\_ENABLED

this->skipStack = ((flags & CollectOverride\_SkipStack) != 0);

DebugOnly(this->isConcurrentGCOnIdle = (flags == CollectOnScriptIdle));

#endif

BOOL collected = collectionWrapper->ExecuteRecyclerCollectionFunction(this, &Recycler::FinishConcurrentCollect, flags);

return collected;

}

BOOL

Recycler::WaitForConcurrentThread(DWORD waitTime)

{

Assert(this->IsConcurrentState() || this->collectionState == CollectionStateParallelMark);

RECYCLER\_PROFILE\_EXEC\_BEGIN(this, Js::ConcurrentWaitPhase);

if (concurrentThread != NULL)

{

// Set the priority back to normal before we wait to ensure it doesn't starve

SetThreadPriority(this->concurrentThread, THREAD\_PRIORITY\_NORMAL);

}

DWORD ret = WaitForSingleObject(concurrentWorkDoneEvent, waitTime);

if (concurrentThread != NULL)

{

if (ret == WAIT\_TIMEOUT)

{

// Keep the priority boost.

priorityBoost = true;

}

else

{

Assert(ret == WAIT\_OBJECT\_0);

// Back to below normal

SetThreadPriority(this->concurrentThread, THREAD\_PRIORITY\_BELOW\_NORMAL);

priorityBoost = false;

}

}

RECYCLER\_PROFILE\_EXEC\_END(this, Js::ConcurrentWaitPhase);

return (ret == WAIT\_OBJECT\_0);

}

void

Recycler::FlushBackgroundPages()

{

recyclerPageAllocator.SuspendIdleDecommit();

recyclerPageAllocator.FlushBackgroundPages();

recyclerPageAllocator.ResumeIdleDecommit();

recyclerLargeBlockPageAllocator.SuspendIdleDecommit();

recyclerLargeBlockPageAllocator.FlushBackgroundPages();

recyclerLargeBlockPageAllocator.ResumeIdleDecommit();

this->threadPageAllocator->SuspendIdleDecommit();

this->threadPageAllocator->FlushBackgroundPages();

this->threadPageAllocator->ResumeIdleDecommit();

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.SuspendIdleDecommit();

recyclerWithBarrierPageAllocator.FlushBackgroundPages();

recyclerWithBarrierPageAllocator.ResumeIdleDecommit();

#endif

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

AutoProtectPages::AutoProtectPages(Recycler\* recycler, bool protectEnabled) :

isReadOnly(false),

recycler(recycler)

{

if (protectEnabled)

{

recycler->heapBlockMap.MakeAllPagesReadOnly(recycler);

isReadOnly = true;

}

}

AutoProtectPages::~AutoProtectPages()

{

Unprotect();

}

void AutoProtectPages::Unprotect()

{

if (isReadOnly)

{

recycler->heapBlockMap.MakeAllPagesReadWrite(recycler);

isReadOnly = false;

}

}

#endif

BOOL

Recycler::FinishConcurrentCollect(CollectionFlags flags)

{

if (!this->IsConcurrentState())

{

Assert(false);

return false;

}

#ifdef PROFILE\_EXEC

Js::Phase concurrentPhase = Js::ConcurrentCollectPhase;

#endif

RECYCLER\_PROFILE\_EXEC\_BEGIN2(this, Js::RecyclerPhase,

(concurrentPhase = ((this->inPartialCollectMode && this->IsConcurrentMarkState())?

Js::ConcurrentPartialCollectPhase : Js::ConcurrentCollectPhase)));

// Don't do concurrent sweep if we have priority boosted.

const BOOL forceInThread = flags & CollectOverride\_ForceInThread;

bool concurrent = (flags & CollectMode\_Concurrent) != 0;

concurrent = concurrent && (!priorityBoost || this->backgroundRescanCount != 1);

#ifdef RECYCLER\_TRACE

collectionParam.priorityBoostConcurentSweepOverride = priorityBoost;

#endif

const DWORD waitTime = forceInThread? INFINITE : RecyclerHeuristic::FinishConcurrentCollectWaitTime(this->GetRecyclerFlagsTable());

GCETW(GC\_FINISHCONCURRENTWAIT\_START, (this, waitTime));

const BOOL waited = WaitForConcurrentThread(waitTime);

GCETW(GC\_FINISHCONCURRENTWAIT\_STOP, (this, !waited));

if (!waited)

{

RECYCLER\_PROFILE\_EXEC\_END2(this, concurrentPhase, Js::RecyclerPhase);

return false;

}

bool needConcurrentSweep = false;

if (collectionState == CollectionStateRescanWait)

{

GCETW(GC\_START, (this, ETWEvent\_ConcurrentRescan));

#ifdef ENABLE\_BASIC\_TELEMETRY

if(GetCurrentThreadContextId()==mainThreadId && IsMemProtectMode()==false)

{

gcTel.LogGCPauseStartTime();

}

#endif

#ifdef RECYCLER\_TRACE

PrintCollectTrace(this->inPartialCollectMode ? Js::ConcurrentPartialCollectPhase : Js::ConcurrentMarkPhase, true);

#endif

collectionState = CollectionStateRescanFindRoots;

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

// TODO: Change this behavior

// ProtectPagesOnRescan is not supported in PageHeap mode because the page protection is changed

// outside the PageAllocator in PageHeap mode and so pages are not in the state that the

// PageAllocator expects when it goes to change the page protection

// One viable fix is to move the guard page protection logic outside of the heap blocks

// and into the page allocator

AssertMsg(!(IsPageHeapEnabled() && GetRecyclerFlagsTable().RecyclerProtectPagesOnRescan), "ProtectPagesOnRescan not supported in page heap mode");

AutoProtectPages protectPages(this, GetRecyclerFlagsTable().RecyclerProtectPagesOnRescan);

#endif

const bool backgroundFinishMark = !forceInThread && concurrent && ((flags & CollectOverride\_BackgroundFinishMark) != 0);

const DWORD finishMarkWaitTime = RecyclerHeuristic::BackgroundFinishMarkWaitTime(backgroundFinishMark, GetRecyclerFlagsTable());

size\_t rescanRootBytes = FinishMark(finishMarkWaitTime);

if (rescanRootBytes == Recycler::InvalidScanRootBytes)

{

Assert(this->IsMarkState());

RECYCLER\_PROFILE\_EXEC\_END2(this, concurrentPhase, Js::RecyclerPhase);

GCETW(GC\_STOP, (this, ETWEvent\_ConcurrentRescan));

#ifdef ENABLE\_BASIC\_TELEMETRY

if (GetCurrentThreadContextId() == mainThreadId && IsMemProtectMode() == false)

{

gcTel.LogGCPauseEndTime();

}

#endif

// we timeout trying to mark.

return false;

}

#ifdef RECYCLER\_STATS

collectionStats.continueCollectAllocBytes = autoHeap.uncollectedAllocBytes;

#endif

#ifdef RECYCLER\_VERIFY\_MARK

if (GetRecyclerFlagsTable().RecyclerVerifyMark)

{

this->VerifyMark();

}

#endif

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

protectPages.Unprotect();

#endif

needConcurrentSweep = this->Sweep(rescanRootBytes, concurrent, true);

GCETW(GC\_STOP, (this, ETWEvent\_ConcurrentRescan));

}

else

{

GCETW(GC\_START, (this, ETWEvent\_ConcurrentTransferSwept));

#ifdef ENABLE\_BASIC\_TELEMETRY

if(GetCurrentThreadContextId()==mainThreadId && IsMemProtectMode()==false)

{

gcTel.LogGCPauseStartTime();

}

#endif

GCETW(GC\_FLUSHZEROPAGE\_START, (this));

Assert(collectionState == CollectionStateTransferSweptWait);

#ifdef RECYCLER\_TRACE

PrintCollectTrace(Js::ConcurrentSweepPhase, true);

#endif

collectionState = CollectionStateTransferSwept;

// We should have zeroed all the pages in the background thread

Assert(!recyclerPageAllocator.HasZeroQueuedPages());

Assert(!recyclerLargeBlockPageAllocator.HasZeroQueuedPages());

this->FlushBackgroundPages();

GCETW(GC\_FLUSHZEROPAGE\_STOP, (this));

GCETW(GC\_TRANSFERSWEPTOBJECTS\_START, (this));

Assert(this->recyclerSweep != nullptr);

Assert(!this->recyclerSweep->IsBackground());

#ifdef PARTIAL\_GC\_ENABLED

if (this->inPartialCollectMode)

{

ConcurrentPartialTransferSweptObjects(\*this->recyclerSweep);

}

else

#endif

{

ConcurrentTransferSweptObjects(\*this->recyclerSweep);

}

recyclerSweep->EndSweep();

GCETW(GC\_TRANSFERSWEPTOBJECTS\_STOP, (this));

GCETW(GC\_STOP, (this, ETWEvent\_ConcurrentTransferSwept));

#ifdef ENABLE\_BASIC\_TELEMETRY

if (GetCurrentThreadContextId() == mainThreadId && IsMemProtectMode() == false)

{

gcTel.LogGCPauseEndTime();

}

#endif

}

RECYCLER\_PROFILE\_EXEC\_END2(this, concurrentPhase, Js::RecyclerPhase);

FinishCollection(needConcurrentSweep);

if (!this->CollectionInProgress())

{

if (NeedExhaustiveRepeatCollect())

{

DoCollect((CollectionFlags)(flags & ~CollectMode\_Partial));

}

else

{

EndCollection();

}

}

return true;

}

int

Recycler::ExceptFilter(LPEXCEPTION\_POINTERS pEP)

{

#if DBG

// Assert exception code

if (pEP->ExceptionRecord->ExceptionCode == STATUS\_ASSERTION\_FAILURE)

{

return EXCEPTION\_CONTINUE\_SEARCH;

}

#endif

#ifdef GENERATE\_DUMP

if (Js::Configuration::Global.flags.IsEnabled(Js::DumpOnCrashFlag))

{

Js::Throw::GenerateDump(pEP, Js::Configuration::Global.flags.DumpOnCrash);

}

#endif

#if DBG && \_M\_IX86

int callerEBP = \*((int\*)pEP->ContextRecord->Ebp);

Output::Print(L"Recycler Concurrent Thread: Uncaught exception: EIP: 0x%X ExceptionCode: 0x%X EBP: 0x%X ReturnAddress: 0x%X ReturnAddress2: 0x%X\n",

pEP->ExceptionRecord->ExceptionAddress, pEP->ExceptionRecord->ExceptionCode, pEP->ContextRecord->Eip,

pEP->ContextRecord->Ebp, \*((int\*)pEP->ContextRecord->Ebp + 1), \*((int\*) callerEBP + 1));

#endif

Output::Flush();

return EXCEPTION\_CONTINUE\_SEARCH;

}

unsigned int

Recycler::StaticThreadProc(LPVOID lpParameter)

{

DWORD ret = (DWORD)-1;

\_\_try

{

Recycler \* recycler = (Recycler \*)lpParameter;

#if DBG

recycler->concurrentThreadExited = false;

#endif

ret = recycler->ThreadProc();

}

\_\_except(Recycler::ExceptFilter(GetExceptionInformation()))

{

Assert(false);

}

return ret;

}

void

Recycler::StaticBackgroundWorkCallback(void \* callbackData)

{

Recycler \* recycler = (Recycler \*) callbackData;

recycler->DoBackgroundWork(true);

}

#ifdef ENABLE\_JS\_ETW

static ETWEventGCActivationKind

BackgroundMarkETWEventGCActivationKind(CollectionState collectionState)

{

return collectionState == CollectionStateConcurrentFinishMark?

ETWEvent\_ConcurrentFinishMark : ETWEvent\_ConcurrentMark;

}

#endif

void

Recycler::DoBackgroundWork(bool forceForeground)

{

if (this->collectionState == CollectionStateConcurrentWrapperCallback)

{

this->collectionWrapper->ConcurrentCallback();

}

else if (this->collectionState == CollectionStateParallelMark)

{

this->ProcessParallelMark(false, &this->markContext);

}

else if (this->IsConcurrentMarkState())

{

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(this, this->collectionState == CollectionStateConcurrentFinishMark?

Js::BackgroundFinishMarkPhase : Js::ConcurrentMarkPhase);

GCETW(GC\_START, (this, BackgroundMarkETWEventGCActivationKind(this->collectionState)));

#ifdef ENABLE\_BASIC\_TELEMETRY

if (GetCurrentThreadContextId() == mainThreadId && IsMemProtectMode() == false)

{

gcTel.LogGCPauseStartTime();

}

#endif

DebugOnly(this->markContext.GetPageAllocator()->SetConcurrentThreadId(::GetCurrentThreadId()));

Assert(this->enableConcurrentMark);

if (this->collectionState != CollectionStateConcurrentFinishMark)

{

this->StartQueueTrackedObject();

}

switch (this->collectionState)

{

case CollectionStateConcurrentResetMarks:

this->BackgroundResetMarks();

this->BackgroundResetWriteWatchAll();

this->collectionState = CollectionStateConcurrentFindRoots;

// fall-through

case CollectionStateConcurrentFindRoots:

this->BackgroundFindRoots();

this->BackgroundScanStack();

this->collectionState = CollectionStateConcurrentMark;

// fall-through

case CollectionStateConcurrentMark:

this->BackgroundMark();

Assert(this->collectionState == CollectionStateConcurrentMark);

RECORD\_TIMESTAMP(concurrentMarkFinishTime);

break;

case CollectionStateConcurrentFinishMark:

this->backgroundRescanRootBytes = this->BackgroundFinishMark();

Assert(!HasPendingMarkObjects());

break;

default:

Assert(false);

break;

};

GCETW(GC\_STOP, (this, BackgroundMarkETWEventGCActivationKind(this->collectionState)));

#ifdef ENABLE\_BASIC\_TELEMETRY

if (GetCurrentThreadContextId() == mainThreadId && IsMemProtectMode() == false)

{

gcTel.LogGCPauseEndTime();

}

#endif

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(this, this->collectionState == CollectionStateConcurrentFinishMark?

Js::BackgroundFinishMarkPhase : Js::ConcurrentMarkPhase);

this->collectionState = CollectionStateRescanWait;

DebugOnly(this->markContext.GetPageAllocator()->ClearConcurrentThreadId());

}

else

{

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_BEGIN(this, Js::ConcurrentSweepPhase);

GCETW(GC\_START, (this, ETWEvent\_ConcurrentSweep));

#ifdef ENABLE\_BASIC\_TELEMETRY

if (GetCurrentThreadContextId() == mainThreadId && IsMemProtectMode() == false)

{

gcTel.LogGCPauseStartTime();

}

#endif

GCETW(GC\_BACKGROUNDZEROPAGE\_START, (this));

Assert(this->enableConcurrentSweep);

Assert(this->collectionState == CollectionStateConcurrentSweep);

// Zero the queued pages first so they are available to be allocated

recyclerPageAllocator.BackgroundZeroQueuedPages();

recyclerLargeBlockPageAllocator.BackgroundZeroQueuedPages();

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.BackgroundZeroQueuedPages();

#endif

GCETW(GC\_BACKGROUNDZEROPAGE\_STOP, (this));

GCETW(GC\_BACKGROUNDSWEEP\_START, (this));

Assert(this->recyclerSweep != nullptr);

this->recyclerSweep->BackgroundSweep();

uint sweptBytes = 0;

#ifdef RECYCLER\_STATS

sweptBytes = (uint)collectionStats.objectSweptBytes;

#endif

GCETW(GC\_BACKGROUNDSWEEP\_STOP, (this, sweptBytes));

// Drain the zero queue again as we might have free more during sweep

// in the background

GCETW(GC\_BACKGROUNDZEROPAGE\_START, (this));

recyclerPageAllocator.BackgroundZeroQueuedPages();

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

recyclerWithBarrierPageAllocator.BackgroundZeroQueuedPages();

#endif

recyclerLargeBlockPageAllocator.BackgroundZeroQueuedPages();

GCETW(GC\_BACKGROUNDZEROPAGE\_STOP, (this));

GCETW(GC\_STOP, (this, ETWEvent\_ConcurrentSweep));

#ifdef ENABLE\_BASIC\_TELEMETRY

if (GetCurrentThreadContextId() == mainThreadId && IsMemProtectMode() == false)

{

gcTel.LogGCPauseEndTime();

}

#endif

Assert(this->collectionState == CollectionStateConcurrentSweep);

this->collectionState = CollectionStateTransferSweptWait;

RECYCLER\_PROFILE\_EXEC\_BACKGROUND\_END(this, Js::ConcurrentSweepPhase);

}

SetEvent(this->concurrentWorkDoneEvent);

collectionWrapper->WaitCollectionCallBack();

}

DWORD

Recycler::ThreadProc()

{

Assert(this->IsConcurrentEnabled());

#if !defined(\_UCRT)

// We do this before we set the concurrentWorkDoneEvent because GetModuleHandleEx requires

// getting the loader lock. We could have the following case:

// Thread A => Initialize Concurrent Thread (C)

// C signals Signal Done

// C yields since its lower priority

// Thread A starts running- and is told to shut down.

// Thread A grabs loader lock as part of the shutdown sequence

// Thread A waits for C to be done

// C wakes up now- and tries to grab loader lock.

// To prevent this deadlock, we call GetModuleHandleEx first and then set the concurrentWorkDoneEvent

HMODULE dllHandle = NULL;

if (!GetModuleHandleEx(GET\_MODULE\_HANDLE\_EX\_FLAG\_FROM\_ADDRESS, (LPCTSTR)&Recycler::StaticThreadProc, &dllHandle))

{

dllHandle = NULL;

}

#endif

#ifdef ENABLE\_JS\_ETW

// Create an ETW ActivityId for this thread, to help tools correlate ETW events we generate

GUID activityId = { 0 };

auto result = EventActivityIdControl(EVENT\_ACTIVITY\_CTRL\_CREATE\_SET\_ID, &activityId);

Assert(result == ERROR\_SUCCESS);

#endif

// Signal that the thread has started

SetEvent(this->concurrentWorkDoneEvent);

SetThreadPriority(::GetCurrentThread(), THREAD\_PRIORITY\_BELOW\_NORMAL);

#if defined(DBG) && defined(PROFILE\_EXEC)

this->backgroundProfilerPageAllocator.SetConcurrentThreadId(::GetCurrentThreadId());

#endif

#ifdef IDLE\_DECOMMIT\_ENABLED

DWORD handleCount = this->concurrentIdleDecommitEvent? 2 : 1;

HANDLE handles[2] = { this->concurrentWorkReadyEvent, this->concurrentIdleDecommitEvent };

#endif

do

{

#ifdef IDLE\_DECOMMIT\_ENABLED

needIdleDecommitSignal = IdleDecommitSignal\_None;

DWORD threadPageAllocatorWaitTime = threadPageAllocator->IdleDecommit();

DWORD recyclerPageAllocatorWaitTime = recyclerPageAllocator.IdleDecommit();

DWORD waitTime = min(threadPageAllocatorWaitTime, recyclerPageAllocatorWaitTime);

DWORD recyclerLargeBlockPageAllocatorWaitTime = recyclerLargeBlockPageAllocator.IdleDecommit();

waitTime = min(waitTime, recyclerLargeBlockPageAllocatorWaitTime);

#ifdef RECYCLER\_WRITE\_BARRIER\_ALLOC\_SEPARATE\_PAGE

DWORD recyclerWithBarrierPageAllocatorWaitTime = recyclerWithBarrierPageAllocator.IdleDecommit();

waitTime = min(waitTime, recyclerWithBarrierPageAllocatorWaitTime);

#endif

if (waitTime == INFINITE)

{

DWORD ret = ::InterlockedCompareExchange(&needIdleDecommitSignal, IdleDecommitSignal\_NeedSignal, IdleDecommitSignal\_None);

if (ret == IdleDecommitSignal\_NeedTimer)

{

#if DBG

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::IdleDecommitPhase))

{

Output::Print(L"Recycler Thread IdleDecommit Need Timer\n");

Output::Flush();

}

#endif

continue;

}

}

#if DBG

else

{

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::IdleDecommitPhase))

{

Output::Print(L"Recycler Thread IdleDecommit Wait %d\n", waitTime);

Output::Flush();

}

}

#endif

DWORD result = WaitForMultipleObjectsEx(handleCount, handles, FALSE, waitTime, FALSE);

if (result != WAIT\_OBJECT\_0)

{

Assert((handleCount == 2 && result == WAIT\_OBJECT\_0 + 1) || (waitTime != INFINITE && result == WAIT\_TIMEOUT));

#if DBG

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::IdleDecommitPhase))

{

if (result == WAIT\_TIMEOUT)

{

Output::Print(L"Recycler Thread IdleDecommit Timeout: %d\n", waitTime);

}

else

{

Output::Print(L"Recycler Thread IdleDecommit Signaled\n");

}

Output::Flush();

}

#endif

continue;

}

#else

DWORD result = WaitForSingleObject(this->concurrentWorkReadyEvent, INFINITE);

Assert(result == WAIT\_OBJECT\_0);

#endif

if (this->collectionState == CollectionStateExit)

{

#if DBG

this->concurrentThreadExited = true;

#endif

break;

}

DoBackgroundWork();

}

while (true);

SetEvent(this->concurrentWorkDoneEvent);

#if !defined(\_UCRT)

if (dllHandle)

{

FreeLibraryAndExitThread(dllHandle, 0);

}

else

#endif

{

return 0;

}

}

#endif //defined(CONCURRENT\_GC\_ENABLED)

void

Recycler::FinishCollection(bool needConcurrentSweep)

{

Assert(!!this->InConcurrentSweep() == needConcurrentSweep);

if (!needConcurrentSweep)

{

FinishCollection();

}

else

{

FinishDisposeObjects();

}

}

void

Recycler::FinishCollection()

{

Assert(!this->hasBackgroundFinishPartial);

Assert(!this->hasPendingDeleteGuestArena);

// Reset the time heuristics

ScheduleNextCollection();

{

AutoSwitchCollectionStates collectionState(this,

/\* entry state \*/ CollectionStatePostCollectionCallback,

/\* exit state \*/ CollectionStateNotCollecting);

collectionWrapper->PostCollectionCallBack();

}

this->backgroundFinishMarkCount = 0;

// Do a partial page decommit now

if (decommitOnFinish)

{

ForRecyclerPageAllocator(DecommitNow(false));

this->decommitOnFinish = false;

}

RECYCLER\_SLOW\_CHECK(autoHeap.Check());

#ifdef RECYCLER\_MEMORY\_VERIFY

this->Verify(Js::RecyclerPhase);

#endif

#ifdef RECYCLER\_FINALIZE\_CHECK

autoHeap.VerifyFinalize();

#endif

#ifdef ENABLE\_JS\_ETW

FlushFreeRecord();

#endif

FinishDisposeObjects();

#ifdef RECYCLER\_FINALIZE\_CHECK

if (!this->IsMarkState())

{

autoHeap.VerifyFinalize();

}

#endif

#ifdef RECYCLER\_STATS

if (CUSTOM\_PHASE\_STATS1(this->GetRecyclerFlagsTable(), Js::RecyclerPhase))

{

PrintCollectStats();

}

#endif

#ifdef PROFILE\_RECYCLER\_ALLOC

if (MemoryProfiler::IsTraceEnabled(true))

{

PrintAllocStats();

}

#endif

#ifdef DUMP\_FRAGMENTATION\_STATS

if (GetRecyclerFlagsTable().DumpFragmentationStats)

{

autoHeap.DumpFragmentationStats();

}

#endif

RECORD\_TIMESTAMP(currentCollectionEndTime);

}

char \*

Recycler::Realloc(void\* buffer, size\_t existingBytes, size\_t requestedBytes, bool truncate)

{

Assert(requestedBytes > 0);

if (existingBytes == 0)

{

Assert(buffer == nullptr);

return Alloc(requestedBytes);

}

Assert(buffer != nullptr);

size\_t nbytes = AllocSizeMath::Align(requestedBytes, HeapConstants::ObjectGranularity);

// Since we successfully allocated, we shouldn't have integer overflow here

size\_t nbytesExisting = AllocSizeMath::Align(existingBytes, HeapConstants::ObjectGranularity);

Assert(nbytesExisting >= existingBytes);

if (nbytes == nbytesExisting)

{

return (char \*)buffer;

}

char\* replacementBuf = this->Alloc(requestedBytes);

if (replacementBuf != nullptr)

{

// Truncate

if (existingBytes > requestedBytes && truncate)

{

js\_memcpy\_s(replacementBuf, requestedBytes, buffer, requestedBytes);

}

else

{

js\_memcpy\_s(replacementBuf, requestedBytes, buffer, existingBytes);

}

}

if (nbytesExisting > 0)

{

this->Free(buffer, nbytesExisting);

}

return replacementBuf;

}

bool

Recycler::ForceSweepObject()

{

#ifdef RECYCLER\_TEST\_SUPPORT

if (BinaryFeatureControl::RecyclerTest())

{

if (checkFn != nullptr)

{

return true;

}

}

#endif

#ifdef PROFILE\_RECYCLER\_ALLOC

if (trackerDictionary != nullptr)

{

// Need to sweep object if we are tracing recycler allocs

return true;

}

#endif

#ifdef RECYCLER\_STATS

if (CUSTOM\_PHASE\_STATS1(this->GetRecyclerFlagsTable(), Js::RecyclerPhase))

{

return true;

}

#endif

#if DBG

// Force sweeping the object so we can assert that we are not sweeping objects that are still implicit roots

if (this->enableScanImplicitRoots)

{

return true;

}

#endif

return false;

}

bool

Recycler::ShouldIdleCollectOnExit()

{

// Always reset partial heuristics even if we are not doing idle collecting

// So we don't carry the heuristics to the next script activation

this->ResetPartialHeuristicCounters();

if (this->CollectionInProgress())

{

#ifdef RECYCLER\_TRACE

CUSTOM\_PHASE\_PRINT\_VERBOSE\_TRACE1(GetRecyclerFlagsTable(), Js::IdleCollectPhase, L"%04X> Skipping scheduling Idle Collect. Reason: Collection in progress\n", ::GetCurrentThreadId());

#endif

// Don't schedule a idle collect if there is a collection going on already

// IDLE-GC-TODO: Fix ResetHeuristics in the GC so we can detect memory allocation during

// the concurrent collect and still schedule an idle collect

return false;

}

if (CUSTOM\_PHASE\_FORCE1(GetRecyclerFlagsTable(), Js::IdleCollectPhase))

{

return true;

}

ulong nextTime = tickCountNextCollection - tickDiffToNextCollect;

// We will try to start a concurrent collect if we are within .9 ms to next scheduled collection, AND,

// the size of allocation is larger than 32M. This is similar to CollectionAllocation logic, just

// earlier in both time heuristic and size heuristic, so we can do some concurrent GC while we are

// not in script.

if (autoHeap.uncollectedAllocBytes >= RecyclerHeuristic::Instance.MaxUncollectedAllocBytesOnExit

&& GetTickCount() > nextTime)

{

#ifdef RECYCLER\_TRACE

if (CUSTOM\_PHASE\_TRACE1(GetRecyclerFlagsTable(), Js::IdleCollectPhase))

{

if (autoHeap.uncollectedAllocBytes >= RecyclerHeuristic::Instance.MaxUncollectedAllocBytesOnExit)

{

Output::Print(L"%04X> Idle collect on exit: alloc %d\n", ::GetCurrentThreadId(), autoHeap.uncollectedAllocBytes);

}

else

{

Output::Print(L"%04X> Idle collect on exit: time %d\n", ::GetCurrentThreadId(), tickCountNextCollection - GetTickCount());

}

Output::Flush();

}

#endif

this->CollectNow<CollectNowConcurrent>();

return false;

}

Assert(!this->CollectionInProgress());

// Idle GC use the size heuristic. Only need to schedule on if we passed it.

return (autoHeap.uncollectedAllocBytes >= RecyclerHeuristic::IdleUncollectedAllocBytesCollection);

}

bool

RecyclerParallelThread::StartConcurrent()

{

if (this->recycler->threadService->HasCallback())

{

// This may be the first time. If so, initialize by creating the doneEvent.

if (this->concurrentWorkDoneEvent == NULL)

{

this->concurrentWorkDoneEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

if (this->concurrentWorkDoneEvent == nullptr)

{

return false;

}

}

Assert(concurrentThread == NULL);

Assert(concurrentWorkReadyEvent == NULL);

// Invoke thread service to process work

if (!this->recycler->threadService->Invoke(RecyclerParallelThread::StaticBackgroundWorkCallback, this))

{

return false;

}

}

else

{

// This may be the first time. If so, initialize and create thread.

if (this->concurrentWorkDoneEvent == NULL)

{

return this->EnableConcurrent(false);

}

else

{

Assert(this->concurrentThread != NULL);

Assert(this->concurrentWorkReadyEvent != NULL);

// signal that thread has been initialized

SetEvent(this->concurrentWorkReadyEvent);

}

}

return true;

}

bool

RecyclerParallelThread::EnableConcurrent(bool waitForThread)

{

this->synchronizeOnStartup = waitForThread;

Assert(this->concurrentWorkDoneEvent == NULL);

Assert(this->concurrentWorkReadyEvent == NULL);

Assert(this->concurrentThread == NULL);

this->concurrentWorkDoneEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

if (this->concurrentWorkDoneEvent == nullptr)

{

return false;

}

this->concurrentWorkReadyEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

if (this->concurrentWorkReadyEvent == nullptr)

{

CloseHandle(this->concurrentWorkDoneEvent);

this->concurrentWorkDoneEvent = NULL;

return false;

}

this->concurrentThread = (HANDLE)\_beginthreadex(NULL, Recycler::ConcurrentThreadStackSize, &RecyclerParallelThread::StaticThreadProc, this, STACK\_SIZE\_PARAM\_IS\_A\_RESERVATION, NULL);

if (this->concurrentThread != nullptr && waitForThread)

{

// Wait for thread to initialize

HANDLE handle[2] = { this->concurrentWorkDoneEvent, this->concurrentThread };

DWORD ret = WaitForMultipleObjectsEx(2, handle, FALSE, INFINITE, FALSE);

if (ret == WAIT\_OBJECT\_0)

{

return true;

}

CloseHandle(concurrentThread);

concurrentThread = nullptr;

}

if (this->concurrentThread == nullptr)

{

CloseHandle(this->concurrentWorkDoneEvent);

this->concurrentWorkDoneEvent = NULL;

CloseHandle(this->concurrentWorkReadyEvent);

this->concurrentWorkReadyEvent = NULL;

return false;

}

return true;

}

template <uint parallelId>

void

Recycler::ParallelWorkFunc()

{

Assert(parallelId == 0 || parallelId == 1);

MarkContext \* markContext = (parallelId == 0 ? &this->parallelMarkContext2 : &this->parallelMarkContext3);

switch (this->collectionState)

{

case CollectionStateParallelMark:

this->ProcessParallelMark(false, markContext);

break;

case CollectionStateBackgroundParallelMark:

this->ProcessParallelMark(true, markContext);

break;

default:

Assert(false);

}

}

void

RecyclerParallelThread::WaitForConcurrent()

{

Assert(this->concurrentThread != NULL || this->recycler->threadService->HasCallback());

Assert(this->concurrentWorkDoneEvent != NULL);

DWORD ret = WaitForSingleObject(concurrentWorkDoneEvent, INFINITE);

Assert(ret == WAIT\_OBJECT\_0);

}

void

RecyclerParallelThread::Shutdown()

{

Assert(this->recycler->collectionState == CollectionStateExit);

if (this->recycler->threadService->HasCallback())

{

if (this->concurrentWorkDoneEvent != NULL)

{

CloseHandle(this->concurrentWorkDoneEvent);

this->concurrentWorkDoneEvent = NULL;

}

}

else

{

if (this->concurrentThread != NULL)

{

HANDLE handles[2] = { concurrentWorkDoneEvent, concurrentThread };

SetEvent(concurrentWorkReadyEvent);

// During process shutdown, OS might kill this (recycler parallel i.e. concurrent) thread and it will not get chance to signal concurrentWorkDoneEvent.

// When we are performing shutdown of main (recycler) thread here, if we wait on concurrentWorkDoneEvent, WaitForObject() will never return.

// Hence wait for concurrentWorkDoneEvent + concurrentThread so if concurrentThread got killed, WaitForObject() will return and we will

// proceed further.

DWORD fRet = WaitForMultipleObjectsEx(2, handles, FALSE, INFINITE, FALSE);

AssertMsg(fRet != WAIT\_FAILED, "Check handles passed to WaitForMultipleObjectsEx.");

CloseHandle(this->concurrentWorkDoneEvent);

this->concurrentWorkDoneEvent = NULL;

CloseHandle(this->concurrentWorkReadyEvent);

this->concurrentWorkReadyEvent = NULL;

CloseHandle(this->concurrentThread);

this->concurrentThread = NULL;

}

}

Assert(this->concurrentThread == NULL);

Assert(this->concurrentWorkReadyEvent == NULL);

Assert(this->concurrentWorkDoneEvent == NULL);

}

// static

unsigned int

RecyclerParallelThread::StaticThreadProc(LPVOID lpParameter)

{

DWORD ret = (DWORD)-1;

\_\_try

{

RecyclerParallelThread \* parallelThread = (RecyclerParallelThread \*)lpParameter;

Recycler \* recycler = parallelThread->recycler;

RecyclerParallelThread::WorkFunc workFunc = parallelThread->workFunc;

Assert(recycler->IsConcurrentEnabled());

#if !defined(\_UCRT)

HMODULE dllHandle = NULL;

if (!GetModuleHandleEx(GET\_MODULE\_HANDLE\_EX\_FLAG\_FROM\_ADDRESS, (LPCTSTR)&RecyclerParallelThread::StaticThreadProc, &dllHandle))

{

dllHandle = NULL;

}

#endif

#ifdef ENABLE\_JS\_ETW

// Create an ETW ActivityId for this thread, to help tools correlate ETW events we generate

GUID activityId = { 0 };

auto result = EventActivityIdControl(EVENT\_ACTIVITY\_CTRL\_CREATE\_SET\_ID, &activityId);

Assert(result == ERROR\_SUCCESS);

#endif

// If this thread is created on demand we already have work to process and do not need to wait

bool mustWait = parallelThread->synchronizeOnStartup;

do

{

if (mustWait)

{

// Signal completion and wait for next work

SetEvent(parallelThread->concurrentWorkDoneEvent);

DWORD result = WaitForSingleObject(parallelThread->concurrentWorkReadyEvent, INFINITE);

Assert(result == WAIT\_OBJECT\_0);

}

if (recycler->collectionState == CollectionStateExit)

{

// Exit thread

break;

}

// Invoke the workFunc to do real work

(recycler->\*workFunc)();

// We always wait after the first time

mustWait = true;

}

while (true);

// Signal to main thread that we have stopped processing and will shut down.

// Note that after this point, we cannot access anything on the Recycler instance

// because the main thread may have torn it down already.

SetEvent(parallelThread->concurrentWorkDoneEvent);

#if !defined(\_UCRT)

if (dllHandle)

{

FreeLibraryAndExitThread(dllHandle, 0);

}

#endif

ret = 0;

}

\_\_except(Recycler::ExceptFilter(GetExceptionInformation()))

{

Assert(false);

}

return ret;

}

// static

void

RecyclerParallelThread::StaticBackgroundWorkCallback(void \* callbackData)

{

RecyclerParallelThread \* parallelThread = (RecyclerParallelThread \*)callbackData;

Recycler \* recycler = parallelThread->recycler;

RecyclerParallelThread::WorkFunc workFunc = parallelThread->workFunc;

(recycler->\*workFunc)();

SetEvent(parallelThread->concurrentWorkDoneEvent);

}

#ifdef RECYCLER\_TRACE

void

Recycler::CaptureCollectionParam(CollectionFlags flags, bool repeat)

{

collectionParam.priorityBoostConcurentSweepOverride = false;

collectionParam.repeat = repeat;

collectionParam.finishOnly = false;

collectionParam.flags = flags;

collectionParam.uncollectedAllocBytes = autoHeap.uncollectedAllocBytes;

#ifdef PARTIAL\_GC\_ENABLED

collectionParam.uncollectedNewPageCountPartialCollect = this->uncollectedNewPageCountPartialCollect;

collectionParam.inPartialCollectMode = inPartialCollectMode;

collectionParam.uncollectedNewPageCount = autoHeap.uncollectedNewPageCount;

collectionParam.unusedPartialCollectFreeBytes = autoHeap.unusedPartialCollectFreeBytes;

#endif

}

void

Recycler::PrintCollectTrace(Js::Phase phase, bool finish, bool noConcurrentWork)

{

if (GetRecyclerFlagsTable().Trace.IsEnabled(Js::RecyclerPhase) ||

GetRecyclerFlagsTable().Trace.IsEnabled(phase))

{

const BOOL allocSize = collectionParam.flags & CollectHeuristic\_AllocSize;

const BOOL timedIfScriptActive = collectionParam.flags & CollectHeuristic\_TimeIfScriptActive;

const BOOL timedIfInScript = collectionParam.flags & CollectHeuristic\_TimeIfInScript;

const BOOL timed = (timedIfScriptActive && isScriptActive) || (timedIfInScript && isInScript) || (collectionParam.flags & CollectHeuristic\_Time);

const BOOL concurrent = collectionParam.flags & CollectMode\_Concurrent;

const BOOL finishConcurrent = collectionParam.flags & CollectOverride\_FinishConcurrent;

const BOOL exhaustive = collectionParam.flags & CollectMode\_Exhaustive;

const BOOL forceInThread = collectionParam.flags & CollectOverride\_ForceInThread;

const BOOL forceFinish = collectionParam.flags & CollectOverride\_ForceFinish;

#ifdef PARTIAL\_GC\_ENABLED

BOOL partial = collectionParam.flags & CollectMode\_Partial ;

#endif

Output::Print(L"%04X> RC(%p): %s%s%s%s%s%s%s:", this->mainThreadId, this,

collectionParam.domCollect? L"[DOM] " : L"",

collectionParam.repeat? L"[Repeat] ": L"",

this->inDispose? L"[Nested]" : L"",

forceInThread? L"Force In thread " : L"",

finish? L"Finish " : L"",

exhaustive? L"Exhaustive " : L"",

Js::PhaseNames[phase]);

if (noConcurrentWork)

{

Assert(finish);

Output::Print(L" No concurrent work");

}

else if (collectionParam.finishOnly)

{

Assert(!collectionParam.repeat);

Assert(finish);

#ifdef CONCURRENT\_GC\_ENABLED

if (collectionState == CollectionStateRescanWait)

{

if (forceFinish)

{

Output::Print(L" Force finish mark and sweep");

}

else if (concurrent && this->enableConcurrentSweep)

{

if (!collectionParam.priorityBoostConcurentSweepOverride)

{

Output::Print(L" Finish mark and start concurrent sweep");

}

else

{

Output::Print(L" Finish mark and sweep (priority boost overrided concurrent sweep)");

}

}

else

{

Output::Print(L" Finish mark and sweep");

}

}

else

{

Assert(collectionState == CollectionStateTransferSweptWait);

if (forceFinish)

{

Output::Print(L" Force finish sweep");

}

else

{

Output::Print(L" Finish sweep");

}

}

#endif // CONCURRENT\_GC\_ENABLED

}

else

{

if (finish && !concurrent)

{

Output::Print(L" Not concurrent collect");

}

if ((finish && finishConcurrent))

{

Output::Print(L" No heuristic");

}

#ifdef CONCURRENT\_GC\_ENABLED

else if (finish && priorityBoost)

{

Output::Print(L" Priority boost no heuristic");

}

#endif

else

{

Output::SkipToColumn(50);

bool byteCountUsed = false;

bool timeUsed = false;

#ifdef PARTIAL\_GC\_ENABLED

bool newPageUsed = false;

if (phase == Js::PartialCollectPhase || phase == Js::ConcurrentPartialCollectPhase)

{

Assert(collectionParam.flags & CollectMode\_Partial);

newPageUsed = !!allocSize;

}

else if (partial && collectionParam.inPartialCollectMode && collectionParam.uncollectedNewPageCount > collectionParam.uncollectedNewPageCountPartialCollect)

{

newPageUsed = true;

}

else

#endif // PARTIAL\_GC\_ENABLED

{

byteCountUsed = !!allocSize;

timeUsed = !!timed;

}

Output::Print(byteCountUsed? L"\*" : (allocSize? L" " : L"~"));

Output::Print(L"B:%8d ", collectionParam.uncollectedAllocBytes);

Output::Print(timeUsed? L"\*" : (timed? L" " : L"~"));

Output::Print(L"T:%4d ", -collectionParam.timeDiff);

#ifdef PARTIAL\_GC\_ENABLED

if (collectionParam.inPartialCollectMode)

{

Output::Print(L"L:%5d ", collectionParam.uncollectedNewPageCountPartialCollect);

}

else

{

Output::Print(L"L:----- ");

}

Output::Print(newPageUsed? L"\*" : (partial? L" " : L"~"));

Output::Print(L"P:%5d(%9d) ", collectionParam.uncollectedNewPageCount, collectionParam.uncollectedNewPageCount \* AutoSystemInfo::PageSize);

Output::Print(L"U:%8d", collectionParam.unusedPartialCollectFreeBytes);

#endif // PARTIAL\_GC\_ENABLED

}

}

Output::Print(L"\n");

Output::Flush();

}

}

#endif

#ifdef RECYCLER\_STATS

void

Recycler::PrintHeapBlockStats(wchar\_t const \* name, HeapBlock::HeapBlockType type)

{

size\_t liveCount = collectionStats.heapBlockCount[type] - collectionStats.heapBlockFreeCount[type];

Output::Print(L" %6s : %5d %5d %5d %5.1f", name,

liveCount, collectionStats.heapBlockFreeCount[type], collectionStats.heapBlockCount[type],

(double)collectionStats.heapBlockFreeCount[type] / (double)collectionStats.heapBlockCount[type] \* 100);

if (type < HeapBlock::SmallBlockTypeCount)

{

Output::Print(L" : %5d %6.1f : %5d %6.1f",

collectionStats.heapBlockSweptCount[type],

(double)collectionStats.heapBlockSweptCount[type] / (double)liveCount \* 100,

collectionStats.heapBlockConcurrentSweptCount[type],

(double)collectionStats.heapBlockConcurrentSweptCount[type] / (double)collectionStats.heapBlockSweptCount[type] \* 100);

}

}

void

Recycler::PrintHeapBlockMemoryStats(wchar\_t const \* name, HeapBlock::HeapBlockType type)

{

size\_t allocableFreeByteCount = collectionStats.heapBlockFreeByteCount[type];

size\_t partialUnusedBytes = 0;

if (this->enablePartialCollect)

{

partialUnusedBytes = allocableFreeByteCount

- collectionStats.smallNonLeafHeapBlockPartialReuseBytes[type];

allocableFreeByteCount -= partialUnusedBytes;

}

size\_t totalByteCount = (collectionStats.heapBlockCount[type] - collectionStats.heapBlockFreeCount[type]) \* AutoSystemInfo::PageSize;

size\_t liveByteCount = totalByteCount - collectionStats.heapBlockFreeByteCount[type];

Output::Print(L" %6s: %10d %10d", name, liveByteCount, allocableFreeByteCount);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect &&

(type == HeapBlock::HeapBlockType::SmallNormalBlockType

|| type == HeapBlock::HeapBlockType::SmallFinalizableBlockType

#ifdef RECYCLER\_WRITE\_BARRIER

|| type == HeapBlock::HeapBlockType::SmallNormalBlockWithBarrierType

|| type == HeapBlock::HeapBlockType::SmallFinalizableBlockWithBarrierType

#endif

|| type == HeapBlock::HeapBlockType::MediumNormalBlockType

|| type == HeapBlock::HeapBlockType::MediumFinalizableBlockType

#ifdef RECYCLER\_WRITE\_BARRIER

|| type == HeapBlock::HeapBlockType::MediumNormalBlockWithBarrierType

|| type == HeapBlock::HeapBlockType::MediumFinalizableBlockWithBarrierType

#endif

))

{

Output::Print(L" %10d", partialUnusedBytes);

}

else

#endif

{

Output::Print(L" ");

}

Output::Print(L" %10d %6.1f", totalByteCount,

(double)allocableFreeByteCount / (double)totalByteCount \* 100);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect &&

(type == HeapBlock::HeapBlockType::SmallNormalBlockType

|| type == HeapBlock::HeapBlockType::SmallFinalizableBlockType

#ifdef RECYCLER\_WRITE\_BARRIER

|| type == HeapBlock::HeapBlockType::SmallNormalBlockWithBarrierType

|| type == HeapBlock::HeapBlockType::SmallFinalizableBlockWithBarrierType

#endif

|| type == HeapBlock::HeapBlockType::MediumNormalBlockType

|| type == HeapBlock::HeapBlockType::MediumFinalizableBlockType

#ifdef RECYCLER\_WRITE\_BARRIER

|| type == HeapBlock::HeapBlockType::MediumNormalBlockWithBarrierType

|| type == HeapBlock::HeapBlockType::MediumFinalizableBlockWithBarrierType

#endif

))

{

Output::Print(L" %6.1f", (double)partialUnusedBytes / (double)totalByteCount \* 100);

}

#endif

}

void

Recycler::PrintHeuristicCollectionStats()

{

Output::Print(L"---------------------------------------------------------------------------------------------------------------\n");

Output::Print(L"GC Trigger : %10s %10s %10s", L"Start", L"Continue", L"Finish");

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Heuristics : %10s %10s %5s", L"", L"", L"%");

}

#endif

Output::Print(L"\n");

Output::Print(L"---------------------------------------------------------------------------------------------------------------\n");

Output::Print(L" Alloc bytes : %10d %10d %10d", collectionStats.startCollectAllocBytes, collectionStats.continueCollectAllocBytes, this->autoHeap.uncollectedAllocBytes);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Cost : %10d %10d %5.1f", collectionStats.rescanRootBytes, collectionStats.estimatedPartialReuseBytes, collectionStats.collectCost \* 100);

}

#endif

Output::Print(L"\n");

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Efficacy : %10s %10s %5.1f\n", L"", L"", collectionStats.collectEfficacy \* 100);

}

#endif

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" New page : %10d %10s %10d", collectionStats.startCollectNewPageCount, L"", autoHeap.uncollectedNewPageCount);

Output::Print(L" | Partial Uncollect New Page : %10d %10d", collectionStats.uncollectedNewPageCountPartialCollect \* AutoSystemInfo::PageSize, this->uncollectedNewPageCountPartialCollect \* AutoSystemInfo::PageSize);

Output::Print(L"\n");

}

#endif

Output::Print(L" Finish try : %10d %10s %10s", collectionStats.finishCollectTryCount, L"", L"");

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Partial Reuse Min Free Bytes : %10d", collectionStats.partialCollectSmallHeapBlockReuseMinFreeBytes \* AutoSystemInfo::PageSize);

}

#endif

Output::Print(L"\n");

}

void

Recycler::PrintMarkCollectionStats()

{

size\_t nonMark = collectionStats.tryMarkCount + collectionStats.tryMarkInteriorCount - collectionStats.remarkCount - collectionStats.markData.markCount;

size\_t invalidCount = nonMark - collectionStats.tryMarkNullCount - collectionStats.tryMarkUnalignedCount

- collectionStats.tryMarkNonRecyclerMemoryCount

- collectionStats.tryMarkInteriorNonRecyclerMemoryCount

- collectionStats.tryMarkInteriorNullCount;

size\_t leafCount = collectionStats.markData.markCount - collectionStats.scanCount;

Output::Print(L"---------------------------------------------------------------------------------------------------------------\n");

Output::Print(L"Try Mark :%9s %5s %10s | Non-Mark : %9s %5s | Mark :%9s %5s \n", L"Count", L"%", L"Bytes", L"Count", L"%", L"Count", L"%");

Output::Print(L"---------------------------------------------------------------------------------------------------------------\n");

Output::Print(L" TryMark :%9d %10d | Null : %9d %5.1f | Scan :%9d %5.1f\n",

collectionStats.tryMarkCount, collectionStats.tryMarkCount \* sizeof(void \*),

collectionStats.tryMarkNullCount, (double)collectionStats.tryMarkNullCount / (double)nonMark \* 100,

collectionStats.scanCount, (double)collectionStats.scanCount / (double)collectionStats.markData.markCount \* 100);

Output::Print(L" Non-Mark :%9d %5.1f | Unaligned : %9d %5.1f | Leaf :%9d %5.1f\n",

nonMark, (double)nonMark / (double)collectionStats.tryMarkCount \* 100,

collectionStats.tryMarkUnalignedCount, (double)collectionStats.tryMarkUnalignedCount / (double)nonMark \* 100,

leafCount, (double)leafCount / (double)collectionStats.markData.markCount \* 100);

Output::Print(L" Mark :%9d %5.1f %10d | Non GC : %9d %5.1f | Track :%9d\n",

collectionStats.markData.markCount, (double)collectionStats.markData.markCount / (double)collectionStats.tryMarkCount \* 100, collectionStats.markData.markBytes,

collectionStats.tryMarkNonRecyclerMemoryCount, (double)collectionStats.tryMarkNonRecyclerMemoryCount / (double)nonMark \* 100,

collectionStats.trackCount);

Output::Print(L" Remark :%9d %5.1f | Invalid : %9d %5.1f \n",

collectionStats.remarkCount, (double)collectionStats.remarkCount / (double)collectionStats.tryMarkCount \* 100,

invalidCount, (double)invalidCount / (double)nonMark \* 100);

Output::Print(L" TryMark Int:%9d %10d | Null Int : %9d %5.1f | Root :%9d | New :%9d\n",

collectionStats.tryMarkInteriorCount, collectionStats.tryMarkInteriorCount \* sizeof(void \*),

collectionStats.tryMarkInteriorNullCount, (double)collectionStats.tryMarkInteriorNullCount / (double)nonMark \* 100,

collectionStats.rootCount, collectionStats.markThruNewObjCount);

Output::Print(L" | Non GC Int: %9d %5.1f | Stack :%9d | NewFalse:%9d\n",

collectionStats.tryMarkInteriorNonRecyclerMemoryCount, (double)collectionStats.tryMarkInteriorNonRecyclerMemoryCount / (double)nonMark \* 100,

collectionStats.stackCount, collectionStats.markThruFalseNewObjCount);

}

void

Recycler::PrintBackgroundCollectionStat(RecyclerCollectionStats::MarkData const& markData)

{

Output::Print(L"BgSmall : %5d %6d %10d | BgLarge : %5d %6d %10d | BgMark :%9d ",

markData.rescanPageCount,

markData.rescanObjectCount,

markData.rescanObjectByteCount,

markData.rescanLargePageCount,

markData.rescanLargeObjectCount,

markData.rescanLargeByteCount,

markData.markCount);

double markRatio = (double)markData.markCount / (double)collectionStats.markData.markCount \* 100;

if (markRatio == 100.0)

{

Output::Print(L" 100");

}

else

{

Output::Print(L"%4.1f", markRatio);

}

Output::Print(L"\n");

}

void

Recycler::PrintBackgroundCollectionStats()

{

#ifdef CONCURRENT\_GC\_ENABLED

Output::Print(L"---------------------------------------------------------------------------------------------------------------\n");

Output::Print(L"BgSmall : %5s %6s %10s | BgLarge : %5s %6s %10s | BgMark :%9s %4s %s\n",

L"Pages", L"Count", L"Bytes", L"Pages", L"Count", L"Bytes", L"Count", L"%", L"NonLeafBytes %");

Output::Print(L"---------------------------------------------------------------------------------------------------------------\n");

this->PrintBackgroundCollectionStat(collectionStats.backgroundMarkData[0]);

for (uint repeatCount = 1; repeatCount < RecyclerHeuristic::MaxBackgroundRepeatMarkCount; repeatCount++)

{

if (collectionStats.backgroundMarkData[repeatCount].markCount == 0)

{

break;

}

collectionStats.backgroundMarkData[repeatCount].rescanPageCount -= collectionStats.backgroundMarkData[repeatCount - 1].rescanPageCount;

collectionStats.backgroundMarkData[repeatCount].rescanObjectCount -= collectionStats.backgroundMarkData[repeatCount - 1].rescanObjectCount;

collectionStats.backgroundMarkData[repeatCount].rescanObjectByteCount -= collectionStats.backgroundMarkData[repeatCount - 1].rescanObjectByteCount;

collectionStats.backgroundMarkData[repeatCount].rescanLargePageCount -= collectionStats.backgroundMarkData[repeatCount - 1].rescanLargePageCount;

collectionStats.backgroundMarkData[repeatCount].rescanLargeObjectCount -= collectionStats.backgroundMarkData[repeatCount - 1].rescanLargeObjectCount;

collectionStats.backgroundMarkData[repeatCount].rescanLargeByteCount -= collectionStats.backgroundMarkData[repeatCount - 1].rescanLargeByteCount;

this->PrintBackgroundCollectionStat(collectionStats.backgroundMarkData[repeatCount]);

}

#endif

}

void

Recycler::PrintMemoryStats()

{

Output::Print(L"----------------------------------------------------------------------------------------------------------------\n");

Output::Print(L"Memory (Bytes) %4s %10s %10s %10s %6s %6s\n", L"Live", L"Free", L"Unused", L"Total", L"Free%", L"Unused%");

Output::Print(L"----------------------------------------------------------------------------------------------------------------\n");

PrintHeapBlockMemoryStats(L"Small", HeapBlock::SmallNormalBlockType);

Output::Print(L"\n");

PrintHeapBlockMemoryStats(L"SmFin", HeapBlock::SmallFinalizableBlockType);

Output::Print(L"\n");

#ifdef RECYCLER\_WRITE\_BARRIER

PrintHeapBlockMemoryStats(L"SmSWB", HeapBlock::SmallNormalBlockWithBarrierType);

Output::Print(L"\n");

PrintHeapBlockMemoryStats(L"SmFinSWB", HeapBlock::SmallFinalizableBlockWithBarrierType);

Output::Print(L"\n");

#endif

PrintHeapBlockMemoryStats(L"SmLeaf", HeapBlock::SmallLeafBlockType);

Output::Print(L"\n");

PrintHeapBlockMemoryStats(L"Medium", HeapBlock::MediumNormalBlockType);

Output::Print(L"\n");

PrintHeapBlockMemoryStats(L"MdFin", HeapBlock::MediumFinalizableBlockType);

Output::Print(L"\n");

#ifdef RECYCLER\_WRITE\_BARRIER

PrintHeapBlockMemoryStats(L"MdSWB", HeapBlock::MediumNormalBlockWithBarrierType);

Output::Print(L"\n");

PrintHeapBlockMemoryStats(L"MdFinSWB", HeapBlock::MediumFinalizableBlockWithBarrierType);

Output::Print(L"\n");

#endif

PrintHeapBlockMemoryStats(L"MdLeaf", HeapBlock::MediumLeafBlockType);

Output::Print(L"\n");

size\_t largeHeapBlockUnusedByteCount = collectionStats.largeHeapBlockTotalByteCount - collectionStats.largeHeapBlockUsedByteCount

- collectionStats.heapBlockFreeByteCount[HeapBlock::LargeBlockType];

Output::Print(L" Large: %10d %10d %10d %10d %6.1f %6.1f\n",

collectionStats.largeHeapBlockUsedByteCount,

collectionStats.heapBlockFreeByteCount[HeapBlock::LargeBlockType],

largeHeapBlockUnusedByteCount,

collectionStats.largeHeapBlockTotalByteCount,

(double)collectionStats.heapBlockFreeByteCount[HeapBlock::LargeBlockType] / (double)collectionStats.largeHeapBlockTotalByteCount \* 100,

(double)largeHeapBlockUnusedByteCount / (double)collectionStats.largeHeapBlockTotalByteCount \* 100);

Output::Print(L"\nSmall heap block zeroing stats since last GC\n");

Output::Print(L"Number of blocks with sweep state empty: normal=%d finalizable=%d leaf=%d\nNumber of blocks zeroed: %d\n",

collectionStats.numEmptySmallBlocks[HeapBlock::SmallNormalBlockType]

#ifdef RECYCLER\_WRITE\_BARRIER

+ collectionStats.numEmptySmallBlocks[HeapBlock::SmallNormalBlockWithBarrierType]

#endif

, collectionStats.numEmptySmallBlocks[HeapBlock::SmallFinalizableBlockType]

#ifdef RECYCLER\_WRITE\_BARRIER

+ collectionStats.numEmptySmallBlocks[HeapBlock::SmallFinalizableBlockWithBarrierType]

#endif

+ collectionStats.numEmptySmallBlocks[HeapBlock::MediumNormalBlockType]

#ifdef RECYCLER\_WRITE\_BARRIER

+ collectionStats.numEmptySmallBlocks[HeapBlock::MediumNormalBlockWithBarrierType]

#endif

, collectionStats.numEmptySmallBlocks[HeapBlock::MediumFinalizableBlockType]

#ifdef RECYCLER\_WRITE\_BARRIER

+ collectionStats.numEmptySmallBlocks[HeapBlock::MediumFinalizableBlockWithBarrierType]

#endif

, collectionStats.numEmptySmallBlocks[HeapBlock::SmallLeafBlockType]

+ collectionStats.numEmptySmallBlocks[HeapBlock::MediumLeafBlockType],

collectionStats.numZeroedOutSmallBlocks);

}

void

Recycler::PrintCollectStats()

{

Output::Print(L"Collection Stats:\n");

PrintHeuristicCollectionStats();

PrintMarkCollectionStats();

PrintBackgroundCollectionStats();

size\_t freeCount = collectionStats.objectSweptCount - collectionStats.objectSweptFreeListCount;

size\_t freeBytes = collectionStats.objectSweptBytes - collectionStats.objectSweptFreeListBytes;

Output::Print(L"---------------------------------------------------------------------------------------------------------------\n");

#if defined(PARTIAL\_GC\_ENABLED) || defined(CONCURRENT\_GC\_ENABLED)

Output::Print(L"Rescan : %5s %6s %10s | Track : %5s | ", L"Pages", L"Count", L"Bytes", L"Count");

#endif

Output::Print(L"Sweep : %7s | SweptObj : %5s %5s %10s\n", L"Count", L"Count", L"%%", L"Bytes");

Output::Print(L"---------------------------------------------------------------------------------------------------------------\n");

Output::Print(L" Small : ");

#if defined(PARTIAL\_GC\_ENABLED) || defined(CONCURRENT\_GC\_ENABLED)

Output::Print(L"%5d %6d %10d | ", collectionStats.markData.rescanPageCount, collectionStats.markData.rescanObjectCount, collectionStats.markData.rescanObjectByteCount);

#endif

#ifdef CONCURRENT\_GC\_ENABLED

Output::Print(L"Process : %5d | ", collectionStats.trackedObjectCount);

#else

Output::Print(L" | ");

#endif

Output::Print(L" Scan : %7d | Free : %6d %5.1f %10d\n",

collectionStats.objectSweepScanCount,

freeCount, (double)freeCount / (double) collectionStats.objectSweptCount \* 100, freeBytes);

Output::Print(L" Large : ");

#if defined(PARTIAL\_GC\_ENABLED) || defined(CONCURRENT\_GC\_ENABLED)

Output::Print(L"%5d %6d %10d | ",

collectionStats.markData.rescanLargePageCount, collectionStats.markData.rescanLargeObjectCount, collectionStats.markData.rescanLargeByteCount);

#endif

#ifdef PARTIAL\_GC\_ENABLED

Output::Print(L"Client : %5d | ", collectionStats.clientTrackedObjectCount);

#else

Output::Print(L" | ");

#endif

Output::Print(L" Finalize : %7d | Free List: %6d %5.1f %10d\n",

collectionStats.finalizeSweepCount,

collectionStats.objectSweptFreeListCount, (double)collectionStats.objectSweptFreeListCount / (double) collectionStats.objectSweptCount \* 100, collectionStats.objectSweptFreeListBytes);

Output::Print(L"---------------------------------------------------------------------------------------------------------------\n");

Output::Print(L"SweptBlk: Live Free Total Free%% : Swept Swept%% : CSwpt CSwpt%%");

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Partial : Count Bytes Existing");

}

#endif

Output::Print(L"\n");

Output::Print(L"---------------------------------------------------------------------------------------------------------------\n");

PrintHeapBlockStats(L"Small", HeapBlock::SmallNormalBlockType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Reuse : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialReuseCount[HeapBlock::SmallNormalBlockType],

collectionStats.smallNonLeafHeapBlockPartialReuseBytes[HeapBlock::MediumNormalBlockType],

collectionStats.smallNonLeafHeapBlockPartialReuseCount[HeapBlock::SmallNormalBlockType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialReuseBytes[HeapBlock::SmallNormalBlockType]);

}

#endif

Output::Print(L"\n");

PrintHeapBlockStats(L"SmFin", HeapBlock::SmallFinalizableBlockType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Unused : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::SmallFinalizableBlockType],

collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::SmallFinalizableBlockType],

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::SmallFinalizableBlockType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::SmallFinalizableBlockType]);

}

#endif

Output::Print(L"\n");

#ifdef RECYCLER\_WRITE\_BARRIER

PrintHeapBlockStats(L"SmSWB", HeapBlock::SmallNormalBlockWithBarrierType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Unused : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::SmallNormalBlockWithBarrierType],

collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::SmallNormalBlockWithBarrierType],

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::SmallNormalBlockWithBarrierType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::SmallNormalBlockWithBarrierType]);

}

#endif

Output::Print(L"\n");

PrintHeapBlockStats(L"SmFin", HeapBlock::SmallFinalizableBlockWithBarrierType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Unused : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::SmallFinalizableBlockWithBarrierType],

collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::SmallFinalizableBlockWithBarrierType],

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::SmallFinalizableBlockWithBarrierType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::SmallFinalizableBlockWithBarrierType]);

}

#endif

Output::Print(L"\n");

#endif

// TODO: This seems suspicious- why are we looking at smallNonLeaf while print out leaf...

PrintHeapBlockStats(L"SmLeaf", HeapBlock::SmallLeafBlockType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | ReuseFin : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialReuseCount[HeapBlock::SmallFinalizableBlockType],

collectionStats.smallNonLeafHeapBlockPartialReuseBytes[HeapBlock::SmallFinalizableBlockType],

collectionStats.smallNonLeafHeapBlockPartialReuseCount[HeapBlock::SmallFinalizableBlockType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialReuseBytes[HeapBlock::SmallFinalizableBlockType]);

}

#endif

Output::Print(L"\n");

PrintHeapBlockStats(L"Medium", HeapBlock::MediumNormalBlockType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Reuse : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialReuseCount[HeapBlock::MediumNormalBlockType],

collectionStats.smallNonLeafHeapBlockPartialReuseBytes[HeapBlock::MediumNormalBlockType],

collectionStats.smallNonLeafHeapBlockPartialReuseCount[HeapBlock::MediumNormalBlockType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialReuseBytes[HeapBlock::MediumNormalBlockType]);

}

#endif

Output::Print(L"\n");

PrintHeapBlockStats(L"MdFin", HeapBlock::MediumFinalizableBlockType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Unused : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::MediumFinalizableBlockType],

collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::MediumFinalizableBlockType],

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::MediumFinalizableBlockType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::MediumFinalizableBlockType]);

}

#endif

Output::Print(L"\n");

#ifdef RECYCLER\_WRITE\_BARRIER

PrintHeapBlockStats(L"MdSWB", HeapBlock::MediumNormalBlockWithBarrierType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Unused : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::MediumNormalBlockWithBarrierType],

collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::MediumNormalBlockWithBarrierType],

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::MediumNormalBlockWithBarrierType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::MediumNormalBlockWithBarrierType]);

}

#endif

Output::Print(L"\n");

PrintHeapBlockStats(L"MdFin", HeapBlock::MediumFinalizableBlockWithBarrierType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | Unused : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::MediumFinalizableBlockWithBarrierType],

collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::MediumFinalizableBlockWithBarrierType],

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::MediumFinalizableBlockWithBarrierType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::MediumFinalizableBlockWithBarrierType]);

}

#endif

Output::Print(L"\n");

#endif

// TODO: This seems suspicious- why are we looking at smallNonLeaf while print out leaf...

PrintHeapBlockStats(L"MdLeaf", HeapBlock::MediumNormalBlockType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | ReuseFin : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialReuseCount[HeapBlock::MediumFinalizableBlockType],

collectionStats.smallNonLeafHeapBlockPartialReuseBytes[HeapBlock::MediumFinalizableBlockType],

collectionStats.smallNonLeafHeapBlockPartialReuseCount[HeapBlock::MediumFinalizableBlockType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialReuseBytes[HeapBlock::MediumFinalizableBlockType]);

}

#endif

Output::Print(L"\n");

// TODO: This can't possibly be correct...check on this later

PrintHeapBlockStats(L"Large", HeapBlock::LargeBlockType);

#ifdef PARTIAL\_GC\_ENABLED

if (this->enablePartialCollect)

{

Output::Print(L" | UnusedFin : %5d %10d %10d",

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::SmallFinalizableBlockType],

collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::SmallFinalizableBlockType],

collectionStats.smallNonLeafHeapBlockPartialUnusedCount[HeapBlock::SmallFinalizableBlockType] \* AutoSystemInfo::PageSize

- collectionStats.smallNonLeafHeapBlockPartialUnusedBytes[HeapBlock::SmallFinalizableBlockType]);

}

#endif

Output::Print(L"\n");

PrintMemoryStats();

Output::Flush();

}

#endif

#ifdef RECYCLER\_ZERO\_MEM\_CHECK

void

Recycler::VerifyZeroFill(void \* address, size\_t size)

{

byte expectedFill = 0;

#ifdef RECYCLER\_MEMORY\_VERIFY

if (this->VerifyEnabled())

{

expectedFill = Recycler::VerifyMemFill;

}

#endif

for (uint i = 0; i < size; i++)

{

Assert(((byte \*)address)[i] == expectedFill);

}

}

#endif

#ifdef RECYCLER\_MEMORY\_VERIFY

void

Recycler::FillCheckPad(void \* address, size\_t size, size\_t alignedAllocSize, bool objectAlreadyInitialized)

{

if (this->VerifyEnabled())

{

void\* addressToVerify = address;

size\_t sizeToVerify = alignedAllocSize;

if (objectAlreadyInitialized)

{

addressToVerify = ((char\*) address + size);

sizeToVerify = (alignedAllocSize - size);

}

// Actually this is filling the non-pad to zero

VerifyCheckFill(addressToVerify, sizeToVerify - sizeof(size\_t));

// Ignore the first word

if (!objectAlreadyInitialized && size > sizeof(FreeObject))

{

memset((char \*)address + sizeof(FreeObject), 0, size - sizeof(FreeObject));

}

// write the pad size at the end;

\*(size\_t \*)((char \*)address + alignedAllocSize - sizeof(size\_t)) = alignedAllocSize - size;

}

}

void Recycler::Verify(Js::Phase phase)

{

if (verifyEnabled && (!this->CollectionInProgress()))

{

if (GetRecyclerFlagsTable().RecyclerVerify.IsEnabled(phase))

{

autoHeap.Verify();

}

}

}

void Recycler::VerifyCheck(BOOL cond, wchar\_t const \* msg, void \* address, void \* corruptedAddress)

{

if (!(cond))

{

fwprintf(stderr, L"RECYCLER CORRUPTION: StartAddress=%p CorruptedAddress=%p: %s", address, corruptedAddress, msg);

Js::Throw::FatalInternalError();

}

}

void Recycler::VerifyCheckFill(void \* address, size\_t size)

{

for (byte \* i = (byte \*)address; i < (byte \*)address + size; i++)

{

Recycler::VerifyCheck(\*i == Recycler::VerifyMemFill, L"memory written after freed", address, i);

}

}

void Recycler::VerifyCheckPadExplicitFreeList(void \* address, size\_t size)

{

size\_t \* paddingAddress = (size\_t \*)((byte \*)address + size - sizeof(size\_t));

size\_t padding = \*paddingAddress;

#pragma warning(suppress:4310)

Assert(padding != (size\_t)0xCACACACACACACACA); // Explicit free objects have to have been initialized at some point before they were freed

Recycler::VerifyCheck(padding >= verifyPad + sizeof(size\_t) && padding < size, L"Invalid padding size", address, paddingAddress);

for (byte \* i = (byte \*)address + size - padding; i < (byte \*)paddingAddress; i++)

{

Recycler::VerifyCheck(\*i == Recycler::VerifyMemFill, L"buffer overflow", address, i);

}

}

void Recycler::VerifyCheckPad(void \* address, size\_t size)

{

size\_t \* paddingAddress = (size\_t \*)((byte \*)address + size - sizeof(size\_t));

size\_t padding = \*paddingAddress;

#pragma warning(suppress:4310)

if (padding == (size\_t)0xCACACACACACACACA)

{

// Nascent block have objects that are not initialized with pad size

Recycler::VerifyCheckFill(address, size);

return;

}

Recycler::VerifyCheck(padding >= verifyPad + sizeof(size\_t) && padding < size, L"Invalid padding size", address, paddingAddress);

for (byte \* i = (byte \*)address + size - padding; i < (byte \*)paddingAddress; i++)

{

Recycler::VerifyCheck(\*i == Recycler::VerifyMemFill, L"buffer overflow", address, i);

}

}

#endif

Recycler::AutoSetupRecyclerForNonCollectingMark::AutoSetupRecyclerForNonCollectingMark(Recycler& recycler, bool setupForHeapEnumeration)

: m\_recycler(recycler), m\_setupDone(false)

{

if (! setupForHeapEnumeration)

{

DoCommonSetup();

}

}

void Recycler::AutoSetupRecyclerForNonCollectingMark::DoCommonSetup()

{

Assert(m\_recycler.collectionState == CollectionStateNotCollecting || m\_recycler.collectionState == CollectionStateExit);

Assert(!m\_recycler.DoQueueTrackedObject());

#ifdef PARTIAL\_GC\_ENABLED

// We need to get out of partial collect before we do the mark because we

// will mess with the free bit vector state

// GC-CONSIDER: don't mess with the free bit vector?

if (m\_recycler.inPartialCollectMode)

{

m\_recycler.FinishPartialCollect();

}

#endif

m\_previousCollectionState = m\_recycler.collectionState;

#ifdef RECYCLER\_STATS

m\_previousCollectionStats = m\_recycler.collectionStats;

memset(&m\_recycler.collectionStats, 0, sizeof(RecyclerCollectionStats));

#endif

m\_setupDone = true;

}

void Recycler::AutoSetupRecyclerForNonCollectingMark::SetupForHeapEnumeration()

{

Assert(!m\_recycler.isHeapEnumInProgress);

Assert(!m\_recycler.allowAllocationDuringHeapEnum);

m\_recycler.EnsureNotCollecting();

DoCommonSetup();

m\_recycler.ResetMarks(ResetMarkFlags\_HeapEnumeration);

m\_recycler.collectionState = CollectionStateNotCollecting;

m\_recycler.isHeapEnumInProgress = true;

m\_recycler.isCollectionDisabled = true;

}

Recycler::AutoSetupRecyclerForNonCollectingMark::~AutoSetupRecyclerForNonCollectingMark()

{

Assert(m\_setupDone);

Assert(!m\_recycler.allowAllocationDuringHeapEnum);

#ifdef RECYCLER\_STATS

m\_recycler.collectionStats = m\_previousCollectionStats;

#endif

m\_recycler.collectionState = m\_previousCollectionState;

m\_recycler.isHeapEnumInProgress = false;

m\_recycler.isCollectionDisabled = false;

}

#ifdef RECYCLER\_DUMP\_OBJECT\_GRAPH

bool Recycler::DumpObjectGraph(RecyclerObjectGraphDumper::Param \* param)

{

bool succeeded = false;

bool isExited = (this->collectionState == CollectionStateExit);

if (isExited)

{

this->collectionState = CollectionStateNotCollecting;

}

if (this->collectionState != CollectionStateNotCollecting)

{

Output::Print(L"Can't dump object graph when collecting\n");

Output::Flush();

return succeeded;

}

BEGIN\_NO\_EXCEPTION

{

RecyclerObjectGraphDumper objectGraphDumper(this, param);

Recycler::AutoSetupRecyclerForNonCollectingMark AutoSetupRecyclerForNonCollectingMark(\*this);

AutoRestoreValue<bool> skipStackToggle(&this->skipStack, this->skipStack || (param && param->skipStack));

this->Mark();

this->objectGraphDumper = nullptr;

#ifdef RECYCLER\_STATS

if (param)

{

param->stats = this->collectionStats;

}

#endif

succeeded = !objectGraphDumper.isOutOfMemory;

}

END\_NO\_EXCEPTION

if (isExited)

{

this->collectionState = CollectionStateExit;

}

if (!succeeded)

{

Output::Print(L"Out of memory dumping object graph\n");

}

Output::Flush();

return succeeded;

}

void

Recycler::DumpObjectDescription(void \*objectAddress)

{

#ifdef PROFILE\_RECYCLER\_ALLOC

type\_info const \* typeinfo = nullptr;

bool isArray = false;

if (this->trackerDictionary)

{

TrackerData \* trackerData = GetTrackerData(objectAddress);

if (trackerData != nullptr)

{

typeinfo = trackerData->typeinfo;

isArray = trackerData->isArray;

}

else

{

Assert(false);

}

}

RecyclerObjectDumper::DumpObject(typeinfo, isArray, objectAddress);

#else

Output::Print(L"Address %p", objectAddress);

#endif

}

#endif

#ifdef RECYCLER\_STRESS

// All stress mode collect art implicitly instantiate here

bool

Recycler::StressCollectNow()

{

if (this->recyclerStress)

{

this->CollectNow<CollectStress>();

return true;

}

#ifdef CONCURRENT\_GC\_ENABLED

else if (this->recyclerBackgroundStress)

{

this->CollectNow<CollectBackgroundStress>();

return true;

}

else if ((this->enableConcurrentMark || this->enableConcurrentSweep)

&& (this->recyclerConcurrentStress

|| this->recyclerConcurrentRepeatStress))

{

#ifdef PARTIAL\_GC\_ENABLED

if (this->recyclerPartialStress)

{

this->CollectNow<CollectConcurrentPartialStress>();

return true;

}

else

#endif // PARTIAL\_GC\_ENABLED

{

this->CollectNow<CollectConcurrentStress>();

return true;

}

}

#endif // CONCURRENT\_GC\_ENABLED

#ifdef PARTIAL\_GC\_ENABLED

else if (this->recyclerPartialStress)

{

this->CollectNow<CollectPartialStress>();

return true;

}

#endif // PARTIAL\_GC\_ENABLED

return false;

}

#endif // RECYCLER\_STRESS

#ifdef TRACK\_ALLOC

Recycler \*

Recycler::TrackAllocInfo(TrackAllocData const& data)

{

#ifdef PROFILE\_RECYCLER\_ALLOC

if (this->trackerDictionary != nullptr)

{

Assert(nextAllocData.IsEmpty());

nextAllocData = data;

}

#endif

return this;

}

void

Recycler::ClearTrackAllocInfo(TrackAllocData\* data/\* = NULL\*/)

{

#ifdef PROFILE\_RECYCLER\_ALLOC

if (this->trackerDictionary != nullptr)

{

AssertMsg(!nextAllocData.IsEmpty(), "Missing tracking information for this allocation, are you not using the macros?");

if (data)

{

\*data = nextAllocData;

}

nextAllocData.Clear();

}

#endif

}

#ifdef PROFILE\_RECYCLER\_ALLOC

bool

Recycler::DoProfileAllocTracker()

{

bool doTracker = false;

#ifdef RECYCLER\_DUMP\_OBJECT\_GRAPH

doTracker = Js::Configuration::Global.flags.DumpObjectGraphOnExit

|| Js::Configuration::Global.flags.DumpObjectGraphOnCollect

|| Js::Configuration::Global.flags.DumpObjectGraphOnEnum;

#endif

#ifdef LEAK\_REPORT

if (Js::Configuration::Global.flags.IsEnabled(Js::LeakReportFlag))

{

doTracker = true;

}

#endif

#ifdef CHECK\_MEMORY\_LEAK

if (Js::Configuration::Global.flags.CheckMemoryLeak)

{

doTracker = true;

}

#endif

return doTracker || MemoryProfiler::DoTrackRecyclerAllocation();

}

void

Recycler::InitializeProfileAllocTracker()

{

if (DoProfileAllocTracker())

{

trackerDictionary = NoCheckHeapNew(TypeInfotoTrackerItemMap, &NoCheckHeapAllocator::Instance, 163);

#pragma prefast(suppress:6031, "InitializeCriticalSectionAndSpinCount always succeed since Vista. No need to check return value");

InitializeCriticalSectionAndSpinCount(&trackerCriticalSection, 1000);

}

nextAllocData.Clear();

}

void

Recycler::TrackAllocCore(void \* object, size\_t size, const TrackAllocData& trackAllocData, bool traceLifetime)

{

Assert(GetTrackerData(object) == nullptr || GetTrackerData(object) == &TrackerData::ExplicitFreeListObjectData);

Assert(trackAllocData.GetTypeInfo() != nullptr);

TrackerItem \* item;

size\_t allocCount = trackAllocData.GetCount();

size\_t itemSize = (size - trackAllocData.GetPlusSize());

bool isArray;

if (allocCount != (size\_t)-1)

{

isArray = true;

itemSize = itemSize / allocCount;

}

else

{

isArray = false;

allocCount = 1;

}

if (!trackerDictionary->TryGetValue(trackAllocData.GetTypeInfo(), &item))

{

item = NoCheckHeapNew(TrackerItem, trackAllocData.GetTypeInfo());

item->instanceData.ItemSize = itemSize;

item->arrayData.ItemSize = itemSize;

trackerDictionary->Item(trackAllocData.GetTypeInfo(), item);

}

else

{

Assert(item->instanceData.typeinfo == trackAllocData.GetTypeInfo());

Assert(item->instanceData.ItemSize == itemSize);

Assert(item->arrayData.ItemSize == itemSize);

}

TrackerData& data = (isArray)? item->arrayData : item->instanceData;

data.ItemCount += allocCount;

data.AllocCount++;

data.ReqSize += size;

data.AllocSize += HeapInfo::GetAlignedSizeNoCheck(size);

#ifdef TRACE\_OBJECT\_LIFETIME

data.TraceLifetime = traceLifetime;

if (traceLifetime)

{

Output::Print(data.isArray ? L"Allocated %S[] %p\n" : L"Allocated %S %p\n", data.typeinfo->name(), object);

}

#endif

#ifdef PERF\_COUNTERS

++data.counter;

data.sizeCounter += HeapInfo::GetAlignedSizeNoCheck(size);

#endif

SetTrackerData(object, &data);

}

void\* Recycler::TrackAlloc(void\* object, size\_t size, const TrackAllocData& trackAllocData, bool traceLifetime)

{

if (this->trackerDictionary != nullptr)

{

Assert(nextAllocData.IsEmpty()); // should have been cleared

EnterCriticalSection(&trackerCriticalSection);

TrackAllocCore(object, size, trackAllocData);

LeaveCriticalSection(&trackerCriticalSection);

}

return object;

}

void

Recycler::TrackIntegrate(\_\_in\_ecount(blockSize) char \* blockAddress, size\_t blockSize, size\_t allocSize, size\_t objectSize, const TrackAllocData& trackAllocData)

{

if (this->trackerDictionary != nullptr)

{

Assert(nextAllocData.IsEmpty()); // should have been cleared

EnterCriticalSection(&trackerCriticalSection);

char \* address = blockAddress;

char \* blockEnd = blockAddress + blockSize;

while (address + allocSize <= blockEnd)

{

TrackAllocCore(address, objectSize, trackAllocData);

address += allocSize;

}

LeaveCriticalSection(&trackerCriticalSection);

}

}

BOOL Recycler::TrackFree(const char\* address, size\_t size)

{

if (this->trackerDictionary != nullptr)

{

EnterCriticalSection(&trackerCriticalSection);

TrackerData \* data = GetTrackerData((char \*)address);

if (data != nullptr)

{

if (data != &TrackerData::EmptyData)

{

#ifdef PERF\_COUNTERS

--data->counter;

data->sizeCounter -= size;

#endif

if (data->typeinfo == &typeid(RecyclerWeakReferenceBase))

{

TrackFreeWeakRef((RecyclerWeakReferenceBase \*)address);

}

data->FreeSize += size;

data->FreeCount++;

#ifdef TRACE\_OBJECT\_LIFETIME

if (data->TraceLifetime)

{

Output::Print(data->isArray ? L"Freed %S[] %p\n" : L"Freed %S %p\n", data->typeinfo->name(), address);

}

#endif

}

SetTrackerData((char \*)address, nullptr);

}

else

{

Assert(false);

}

LeaveCriticalSection(&trackerCriticalSection);

}

return true;

}

Recycler::TrackerData \*

Recycler::GetTrackerData(void \* address)

{

HeapBlock \* heapBlock = this->FindHeapBlock(address);

Assert(heapBlock != nullptr);

return (Recycler::TrackerData \*)heapBlock->GetTrackerData(address);

}

void

Recycler::SetTrackerData(void \* address, TrackerData \* data)

{

HeapBlock \* heapBlock = this->FindHeapBlock(address);

Assert(heapBlock != nullptr);

heapBlock->SetTrackerData(address, data);

}

void

Recycler::TrackUnallocated(\_\_in char\* address, \_\_in char \*endAddress, size\_t sizeCat)

{

if (this->trackerDictionary != nullptr)

{

EnterCriticalSection(&trackerCriticalSection);

while (address + sizeCat <= endAddress)

{

Assert(GetTrackerData(address) == nullptr);

SetTrackerData(address, &TrackerData::EmptyData);

address += sizeCat;

}

LeaveCriticalSection(&trackerCriticalSection);

}

}

void

Recycler::TrackAllocWeakRef(RecyclerWeakReferenceBase \* weakRef)

{

Assert(weakRef->typeInfo != nullptr);

#if DBG && defined(PERF\_COUNTERS)

if (this->trackerDictionary != nullptr)

{

TrackerItem \* item;

if (trackerDictionary->TryGetValue(weakRef->typeInfo, &item))

{

weakRef->counter = &item->weakRefCounter;

}

else

{

weakRef->counter = &PerfCounter::RecyclerTrackerCounterSet::GetWeakRefPerfCounter(weakRef->typeInfo);

}

++(\*weakRef->counter);

}

#endif

}

void

Recycler::TrackFreeWeakRef(RecyclerWeakReferenceBase \* weakRef)

{

#if DBG && defined(PERF\_COUNTERS)

if (weakRef->counter != nullptr)

{

--(\*weakRef->counter);

}

#endif

}

void

Recycler::PrintAllocStats()

{

if (this->trackerDictionary == nullptr)

{

return;

}

size\_t itemCount = 0;

int allocCount = 0;

int64 reqSize = 0;

int64 allocSize = 0;

int freeCount = 0;

int64 freeSize = 0;

Output::Print(L"=================================================================================================================\n");

Output::Print(L"Recycler Allocations\n");

Output::Print(L"=================================================================================================================\n");

Output::Print(L"ItemSize ItemCount AllocCount RequestSize AllocSize FreeCount FreeSize DiffCount DiffSize \n");

Output::Print(L"-------- ---------- ---------- --------------- --------------- ---------- --------------- ---------- ---------------\n");

for (int i = 0; i < trackerDictionary->Count(); i++)

{

TrackerItem \* item = trackerDictionary->GetValueAt(i);

type\_info const \* typeinfo = trackerDictionary->GetKeyAt(i);

if (item->instanceData.AllocCount != 0)

{

Output::Print(L"%8d %10d %10d %15I64d %15I64d %10d %15I64d %10d %15I64d %S\n",

item->instanceData.ItemSize, item->instanceData.ItemCount, item->instanceData.AllocCount, item->instanceData.ReqSize,

item->instanceData.AllocSize, item->instanceData.FreeCount, item->instanceData.FreeSize,

item->instanceData.AllocCount - item->instanceData.FreeCount, item->instanceData.AllocSize - item->instanceData.FreeSize, typeinfo->name());

itemCount += item->instanceData.ItemCount;

allocCount += item->instanceData.AllocCount;

reqSize += item->instanceData.ReqSize;

allocSize += item->instanceData.AllocSize;

freeCount += item->instanceData.FreeCount;

freeSize += item->instanceData.FreeSize;

}

if (item->arrayData.AllocCount != 0)

{

Output::Print(L"%8d %10d %10d %15I64d %15I64d %10d %15I64d %10d %15I64d %S[]\n",

item->arrayData.ItemSize, item->arrayData.ItemCount, item->arrayData.AllocCount, item->arrayData.ReqSize,

item->arrayData.AllocSize, item->arrayData.FreeCount, item->arrayData.FreeSize,

item->instanceData.AllocCount - item->instanceData.FreeCount, item->arrayData.AllocSize - item->arrayData.FreeSize, typeinfo->name());

itemCount += item->arrayData.ItemCount;

allocCount += item->arrayData.AllocCount;

reqSize += item->arrayData.ReqSize;

allocSize += item->arrayData.AllocSize;

freeCount += item->arrayData.FreeCount;

freeSize += item->arrayData.FreeSize;

}

}

Output::Print(L"-------- ---------- ---------- --------------- --------------- ---------- --------------- ---------- ---------------\n");

Output::Print(L" %8d %10d %15I64d %15I64d %10d %15I64d %10d %15I64d \*\*Total\*\*\n",

itemCount, allocCount, reqSize, allocSize, freeCount, freeSize, allocCount - freeCount, allocSize - freeSize);

#ifdef EXCEL\_FRIENDLY\_DUMP

Output::Print(L"\nExcel friendly version\nItemSize\tItemCount\tAllocCount\tRequestSize\tAllocSize\tFreeCount\tFreeSize\tDiffCount\tDiffSize\tType\n");

for (int i = 0; i < trackerDictionary->Count(); i++)

{

TrackerItem \* item = trackerDictionary->GetValueAt(i);

type\_info const \* typeinfo = trackerDictionary->GetKeyAt(i);

if (item->instanceData.AllocCount != 0)

{

Output::Print(L"%d\t%d\t%d\t%I64d\t%I64d\t%d\t%I64d\t%d\t%I64d\t%S\n",

item->instanceData.ItemSize, item->instanceData.ItemCount, item->instanceData.AllocCount, item->instanceData.ReqSize,

item->instanceData.AllocSize, item->instanceData.FreeCount, item->instanceData.FreeSize,

item->instanceData.AllocCount - item->instanceData.FreeCount, item->instanceData.AllocSize - item->instanceData.FreeSize, typeinfo->name());

}

if (item->arrayData.AllocCount != 0)

{

Output::Print(L"%d\t%d\t%d\t%I64d\t%I64d\t%d\t%I64d\t%d\t%I64d\t%S[]\n",

item->arrayData.ItemSize, item->arrayData.ItemCount, item->arrayData.AllocCount, item->arrayData.ReqSize,

item->arrayData.AllocSize, item->arrayData.FreeCount, item->arrayData.FreeSize,

item->instanceData.AllocCount - item->instanceData.FreeCount, item->arrayData.AllocSize - item->arrayData.FreeSize, typeinfo->name());

}

}

#endif // EXCEL\_FRIENDLY\_DUMP

Output::Flush();

}

#endif // PROFILE\_RECYCLER\_ALLOC

#endif // TRACK\_ALLOC

#ifdef RECYCLER\_VERIFY\_MARK

void

Recycler::VerifyMark()

{

VerifyMarkRoots();

// Can't really verify stack since the recycler code between ScanStack to now may have introduce false references.

// VerifyMarkStack();

autoHeap.VerifyMark();

}

void

Recycler::VerifyMarkRoots()

{

{

this->VerifyMark(transientPinnedObject);

pinnedObjectMap.Map([this](void \* obj, PinRecord const &refCount)

{

if (refCount == 0)

{

Assert(this->hasPendingUnpinnedObject);

}

else

{

// Use the pinrecord as the source reference

this->VerifyMark(obj);

}

});

}

DList<GuestArenaAllocator, HeapAllocator>::Iterator guestArenaIter(&guestArenaList);

while (guestArenaIter.Next())

{

if (guestArenaIter.Data().pendingDelete)

{

Assert(this->hasPendingDeleteGuestArena);

}

else

{

VerifyMarkArena(&guestArenaIter.Data());

}

}

DList<ArenaData \*, HeapAllocator>::Iterator externalGuestArenaIter(&externalGuestArenaList);

while (externalGuestArenaIter.Next())

{

VerifyMarkArena(externalGuestArenaIter.Data());

}

// We can't check external roots here

}

void

Recycler::VerifyMarkArena(ArenaData \* alloc)

{

VerifyMarkBigBlockList(alloc->GetBigBlocks(false));

VerifyMarkBigBlockList(alloc->GetFullBlocks());

VerifyMarkArenaMemoryBlockList(alloc->GetMemoryBlocks());

}

void

Recycler::VerifyMarkBigBlockList(BigBlock \* memoryBlocks)

{

size\_t scanRootBytes = 0;

BigBlock \*blockp = memoryBlocks;

while (blockp != NULL)

{

void\*\* base=(void\*\*)blockp->GetBytes();

size\_t slotCount = blockp->currentByte / sizeof(void\*);

scanRootBytes += blockp->currentByte;

for (size\_t i=0; i < slotCount; i++)

{

VerifyMark(base[i]);

}

blockp = blockp->nextBigBlock;

}

}

void

Recycler::VerifyMarkArenaMemoryBlockList(ArenaMemoryBlock \* memoryBlocks)

{

size\_t scanRootBytes = 0;

ArenaMemoryBlock \*blockp = memoryBlocks;

while (blockp != NULL)

{

void\*\* base=(void\*\*)blockp->GetBytes();

size\_t slotCount = blockp->nbytes / sizeof(void\*);

scanRootBytes += blockp->nbytes;

for (size\_t i=0; i< slotCount; i++)

{

VerifyMark(base[i]);

}

blockp = blockp->next;

}

}

void

Recycler::VerifyMarkStack()

{

SAVE\_THREAD\_CONTEXT();

void \*\* stackTop = (void\*\*) this->savedThreadContext.GetStackTop();

void \* stackStart = GetStackBase();

Assert(stackStart > stackTop);

for (;stackTop < stackStart; stackTop++)

{

void\* candidate = \*stackTop;

VerifyMark(candidate);

}

void\*\* registers = this->savedThreadContext.GetRegisters();

for (int i = 0; i < SavedRegisterState::NumRegistersToSave; i++)

{

VerifyMark(registers[i]);

}

}

void

Recycler::VerifyMark(void \* candidate)

{

void \* realAddress;

HeapBlock \* heapBlock;

if (this->enableScanInteriorPointers)

{

heapBlock = heapBlockMap.GetHeapBlock(candidate);

if (heapBlock == nullptr)

{

return;

}

realAddress = heapBlock->GetRealAddressFromInterior(candidate);

if (realAddress == nullptr)

{

return;

}

}

else

{

heapBlock = this->FindHeapBlock(candidate);

if (heapBlock == nullptr)

{

return;

}

realAddress = candidate;

}

heapBlock->VerifyMark(realAddress);

}

#endif

ArenaAllocator \*

Recycler::CreateGuestArena(wchar\_t const \* name, void (\*outOfMemoryFunc)())

{

// Note, guest arenas use the large block allocator.

return guestArenaList.PrependNode(&HeapAllocator::Instance, name, &recyclerLargeBlockPageAllocator, outOfMemoryFunc);

}

void

Recycler::DeleteGuestArena(ArenaAllocator \* arenaAllocator)

{

GuestArenaAllocator \* guestArenaAllocator = static\_cast<GuestArenaAllocator \*>(arenaAllocator);

if (this->hasPendingConcurrentFindRoot)

{

// We are doing concurrent find root, don't modify the list and mark the arena to be delete

// later when we do find root in thread.

Assert(guestArenaList.HasElement(guestArenaAllocator));

this->hasPendingDeleteGuestArena = true;

guestArenaAllocator->pendingDelete = true;

}

else

{

guestArenaList.RemoveElement(&HeapAllocator::Instance, guestArenaAllocator);

}

}

#ifdef LEAK\_REPORT

void

Recycler::ReportLeaks()

{

if (GetRecyclerFlagsTable().IsEnabled(Js::LeakReportFlag))

{

if (GetRecyclerFlagsTable().ForceMemoryLeak)

{

AUTO\_HANDLED\_EXCEPTION\_TYPE(ExceptionType\_DisableCheck);

struct FakeMemory { int f; };

FakeMemory \* f = RecyclerNewStruct(this, FakeMemory);

this->RootAddRef(f);

}

LeakReport::StartSection(L"Object Graph");

LeakReport::StartRedirectOutput();

RecyclerObjectGraphDumper::Param param = { 0 };

param.skipStack = true;

if (!this->DumpObjectGraph(&param))

{

LeakReport::Print(L"--------------------------------------------------------------------------------\n");

LeakReport::Print(L"ERROR: Out of memory generating leak report\n");

param.stats.markData.markCount = 0;

}

LeakReport::EndRedirectOutput();

if (param.stats.markData.markCount != 0)

{

LeakReport::Print(L"--------------------------------------------------------------------------------\n");

LeakReport::Print(L"Recycler Leaked Object: %d bytes (%d objects)\n",

param.stats.markData.markBytes, param.stats.markData.markCount);

if (GetRecyclerFlagsTable().LeakStackTrace)

{

LeakReport::StartSection(L"Pinned object stack traces");

LeakReport::StartRedirectOutput();

this->PrintPinnedObjectStackTraces();

LeakReport::EndRedirectOutput();

LeakReport::EndSection();

}

}

LeakReport::EndSection();

}

}

void

Recycler::ReportLeaksOnProcessDetach()

{

if (GetRecyclerFlagsTable().IsEnabled(Js::LeakReportFlag))

{

AUTO\_LEAK\_REPORT\_SECTION(this->GetRecyclerFlagsTable(), L"Recycler (%p): Process Termination", this);

LeakReport::StartRedirectOutput();

ReportOnProcessDetach([=]() { this->ReportLeaks(); });

LeakReport::EndRedirectOutput();

}

}

#endif

#ifdef CHECK\_MEMORY\_LEAK

void

Recycler::CheckLeaks(wchar\_t const \* header)

{

if (GetRecyclerFlagsTable().CheckMemoryLeak && this->isPrimaryMarkContextInitialized)

{

if (GetRecyclerFlagsTable().ForceMemoryLeak)

{

AUTO\_HANDLED\_EXCEPTION\_TYPE(ExceptionType\_DisableCheck);

struct FakeMemory { int f; };

FakeMemory \* f = RecyclerNewStruct(this, FakeMemory);

this->RootAddRef(f);

}

Output::CaptureStart();

Output::Print(L"-------------------------------------------------------------------------------------\n");

Output::Print(L"Recycler (%p): %s Leaked Roots\n", this, header);

Output::Print(L"-------------------------------------------------------------------------------------\n");

RecyclerObjectGraphDumper::Param param = { 0 };

param.dumpRootOnly = true;

param.skipStack = true;

if (!this->DumpObjectGraph(&param))

{

free(Output::CaptureEnd());

Output::Print(L"ERROR: Out of memory generating leak report\n");

return;

}

if (param.stats.markData.markCount != 0)

{

if (GetRecyclerFlagsTable().LeakStackTrace)

{

Output::Print(L"-------------------------------------------------------------------------------------\n");

Output::Print(L"Pinned object stack traces");

Output::Print(L"-------------------------------------------------------------------------------------\n");

this->PrintPinnedObjectStackTraces();

}

Output::Print(L"-------------------------------------------------------------------------------------\n");

Output::Print(L"Recycler Leaked Object: %d bytes (%d objects)\n",

param.stats.markData.markBytes, param.stats.markData.markCount);

wchar\_t \* buffer = Output::CaptureEnd();

MemoryLeakCheck::AddLeakDump(buffer, param.stats.markData.markBytes, param.stats.markData.markCount);

#ifdef GENERATE\_DUMP

if (GetRecyclerFlagsTable().IsEnabled(Js::DumpOnLeakFlag))

{

Js::Throw::GenerateDump(GetRecyclerFlagsTable().DumpOnLeak);

}

#endif

}

else

{

free(Output::CaptureEnd());

}

}

}

void

Recycler::CheckLeaksOnProcessDetach(wchar\_t const \* header)

{

if (GetRecyclerFlagsTable().CheckMemoryLeak)

{

ReportOnProcessDetach([=]() { this->CheckLeaks(header); });

}

}

#endif

#if defined(LEAK\_REPORT) || defined(CHECK\_MEMORY\_LEAK)

template <class Fn>

void

Recycler::ReportOnProcessDetach(Fn fn)

{

#if DBG

// Process detach can be done on any thread, just disable the thread check

this->markContext.GetPageAllocator()->SetDisableThreadAccessCheck();

#endif

if (this->IsConcurrentState())

{

this->AbortConcurrent(true);

}

if (this->CollectionInProgress())

{

Output::Print(L"WARNING: Thread terminated during GC. Can't dump object graph\n");

return;

}

// Don't mark external roots on another thread

this->SetExternalRootMarker(NULL, NULL);

#if DBG

this->ResetThreadId();

#endif

fn();

}

void

Recycler::PrintPinnedObjectStackTraces()

{

pinnedObjectMap.Map([this](void \* object, PinRecord const& pinRecord)

{

this->DumpObjectDescription(object);

Output::Print(L"\n");

StackBackTraceNode::PrintAll(pinRecord.stackBackTraces);

}

);

}

#endif

#if defined(RECYCLER\_DUMP\_OBJECT\_GRAPH) || defined(LEAK\_REPORT) || defined(CHECK\_MEMORY\_LEAK)

void

Recycler::SetInDllCanUnloadNow()

{

inDllCanUnloadNow = true;

// Just clear out the root marker for the dump graph and report leaks

SetExternalRootMarker(NULL, NULL);

}

void

Recycler::SetInDetachProcess()

{

inDetachProcess = true;

// Just clear out the root marker for the dump graph and report leaks

SetExternalRootMarker(NULL, NULL);

}

#endif

#ifdef ENABLE\_JS\_ETW

ULONG Recycler::EventWriteFreeMemoryBlock(HeapBlock\* heapBlock)

{

if (EventEnabledJSCRIPT\_RECYCLER\_FREE\_MEMORY\_BLOCK())

{

char\* memoryAddress = NULL;

ULONG objectSize = 0;

ULONG blockSize = 0;

switch (heapBlock->GetHeapBlockType())

{

case HeapBlock::HeapBlockType::SmallFinalizableBlockType:

case HeapBlock::HeapBlockType::SmallNormalBlockType:

#ifdef RECYCLER\_WRITE\_BARRIER

case HeapBlock::HeapBlockType::SmallFinalizableBlockWithBarrierType:

case HeapBlock::HeapBlockType::SmallNormalBlockWithBarrierType:

#endif

case HeapBlock::HeapBlockType::SmallLeafBlockType:

{

SmallHeapBlock\* smallHeapBlock = static\_cast<SmallHeapBlock\*>(heapBlock);

memoryAddress = smallHeapBlock->GetAddress();

blockSize = (ULONG)(smallHeapBlock->GetEndAddress() - memoryAddress);

objectSize = smallHeapBlock->GetObjectSize();

}

break;

case HeapBlock::HeapBlockType::MediumFinalizableBlockType:

case HeapBlock::HeapBlockType::MediumNormalBlockType:

#ifdef RECYCLER\_WRITE\_BARRIER

case HeapBlock::HeapBlockType::MediumFinalizableBlockWithBarrierType:

case HeapBlock::HeapBlockType::MediumNormalBlockWithBarrierType:

#endif

case HeapBlock::HeapBlockType::MediumLeafBlockType:

{

MediumHeapBlock\* mediumHeapBlock = static\_cast<MediumHeapBlock\*>(heapBlock);

memoryAddress = mediumHeapBlock->GetAddress();

blockSize = (ULONG)(mediumHeapBlock->GetEndAddress() - memoryAddress);

objectSize = mediumHeapBlock->GetObjectSize();

}

case HeapBlock::HeapBlockType::LargeBlockType:

{

LargeHeapBlock\* largeHeapBlock = static\_cast<LargeHeapBlock\*>(heapBlock);

memoryAddress = largeHeapBlock->GetBeginAddress();

blockSize = (ULONG)(largeHeapBlock->GetEndAddress() - memoryAddress);

objectSize = blockSize;

}

break;

default:

AssertMsg(FALSE, "invalid heapblock type");

}

EventWriteJSCRIPT\_RECYCLER\_FREE\_MEMORY\_BLOCK(memoryAddress, blockSize, objectSize);

}

return S\_OK;

}

void Recycler::FlushFreeRecord()

{

Assert(bulkFreeMemoryWrittenCount <= Recycler::BulkFreeMemoryCount);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_FREE\_MEMORY(bulkFreeMemoryWrittenCount, sizeof(Recycler::ETWFreeRecord), etwFreeRecords));

bulkFreeMemoryWrittenCount = 0;

}

void Recycler::AppendFreeMemoryETWRecord(\_\_in char \*address, size\_t size)

{

Assert(bulkFreeMemoryWrittenCount < Recycler::BulkFreeMemoryCount);

\_\_analysis\_assume(bulkFreeMemoryWrittenCount < Recycler::BulkFreeMemoryCount);

etwFreeRecords[bulkFreeMemoryWrittenCount].memoryAddress = address;

// TODO: change to size\_t or uint64?

etwFreeRecords[bulkFreeMemoryWrittenCount].objectSize = (uint)size;

bulkFreeMemoryWrittenCount++;

if (bulkFreeMemoryWrittenCount == Recycler::BulkFreeMemoryCount)

{

FlushFreeRecord();

Assert(bulkFreeMemoryWrittenCount == 0);

}

}

#endif

#ifdef PROFILE\_EXEC

ArenaAllocator \*

Recycler::AddBackgroundProfilerArena()

{

return this->backgroundProfilerArena.PrependNode(&HeapAllocator::Instance,

L"BgGCProfiler", &this->backgroundProfilerPageAllocator, Js::Throw::OutOfMemory);

}

void

Recycler::ReleaseBackgroundProfilerArena(ArenaAllocator \* arena)

{

this->backgroundProfilerArena.RemoveElement(&HeapAllocator::Instance, arena);

}

void

Recycler::SetProfiler(Js::Profiler \* profiler, Js::Profiler \* backgroundProfiler)

{

this->profiler = profiler;

this->backgroundProfiler = backgroundProfiler;

}

#endif

void Recycler::SetObjectBeforeCollectCallback(void\* object, ObjectBeforeCollectCallback callback, void\* callbackState)

{

if (objectBeforeCollectCallbackState == ObjectBeforeCollectCallback\_Shutdown)

{

return; // NOP at shutdown

}

if (objectBeforeCollectCallbackMap == nullptr)

{

if (callback == nullptr) return;

objectBeforeCollectCallbackMap = HeapNew(ObjectBeforeCollectCallbackMap, &HeapAllocator::Instance);

}

// only allow 1 callback per object

objectBeforeCollectCallbackMap->Item(object, ObjectBeforeCollectCallbackData(callback, callbackState));

if (callback != nullptr && this->IsInObjectBeforeCollectCallback()) // revive

{

this->ScanMemory(&object, sizeof(object));

this->ProcessMark(/\*background\*/false);

}

}

bool Recycler::ProcessObjectBeforeCollectCallbacks(bool atShutdown/\*= false\*/)

{

if (this->objectBeforeCollectCallbackMap == nullptr)

{

return false; // no callbacks

}

Assert(atShutdown || this->IsMarkState());

Assert(!this->IsInObjectBeforeCollectCallback());

AutoRestoreValue<ObjectBeforeCollectCallbackState> autoInObjectBeforeCollectCallback(&objectBeforeCollectCallbackState,

atShutdown ? ObjectBeforeCollectCallback\_Shutdown: ObjectBeforeCollectCallback\_Normal);

// The callbacks may register/unregister callbacks while we are enumerating the current map. To avoid

// conflicting usage of the callback map, we swap it out. New registration will go to a new map.

AutoAllocatorObjectPtr<ObjectBeforeCollectCallbackMap, HeapAllocator> oldCallbackMap(

this->objectBeforeCollectCallbackMap, &HeapAllocator::Instance);

this->objectBeforeCollectCallbackMap = nullptr;

bool hasRemainingCallbacks = false;

oldCallbackMap->MapAndRemoveIf([&](const ObjectBeforeCollectCallbackMap::EntryType& entry)

{

const ObjectBeforeCollectCallbackData& data = entry.Value();

if (data.callback != nullptr)

{

void\* object = entry.Key();

if (atShutdown || !this->IsObjectMarked(object))

{

data.callback(object, data.callbackState);

}

else

{

hasRemainingCallbacks = true;

return false; // Do not remove this entry, remaining callback for future

}

}

return true; // Remove this entry

});

// Merge back remaining callbacks if any

if (hasRemainingCallbacks)

{

if (this->objectBeforeCollectCallbackMap == nullptr)

{

this->objectBeforeCollectCallbackMap = oldCallbackMap.Detach();

}

else

{

if (oldCallbackMap->Count() > this->objectBeforeCollectCallbackMap->Count())

{

// Swap so that oldCallbackMap is the smaller one

ObjectBeforeCollectCallbackMap\* tmp = oldCallbackMap.Detach();

\*&oldCallbackMap = this->objectBeforeCollectCallbackMap;

this->objectBeforeCollectCallbackMap = tmp;

}

oldCallbackMap->Map([&](void\* object, const ObjectBeforeCollectCallbackData& data)

{

this->objectBeforeCollectCallbackMap->Item(object, data);

});

}

}

return true; // maybe called callbacks

}

void Recycler::ClearObjectBeforeCollectCallbacks()

{

// This is called at shutting down. All objects will be gone. Invoke each registered callback if any.

ProcessObjectBeforeCollectCallbacks(/\*atShutdown\*/true);

Assert(objectBeforeCollectCallbackMap == nullptr);

}

#ifdef RECYCLER\_TEST\_SUPPORT

void Recycler::SetCheckFn(BOOL(\*checkFn)(char\* addr, size\_t size))

{

Assert(BinaryFeatureControl::RecyclerTest());

this->EnsureNotCollecting();

this->checkFn = checkFn;

}

#endif

void

Recycler::NotifyFree(\_\_in char \*address, size\_t size)

{

RecyclerVerboseTrace(GetRecyclerFlagsTable(), L"Sweeping object %p\n", address);

#ifdef RECYCLER\_TEST\_SUPPORT

if (BinaryFeatureControl::RecyclerTest())

{

if (checkFn != NULL)

checkFn(address, size);

}

#endif

#ifdef ENABLE\_JS\_ETW

if (EventEnabledJSCRIPT\_RECYCLER\_FREE\_MEMORY())

{

AppendFreeMemoryETWRecord(address, (UINT)size);

}

#endif

RecyclerMemoryTracking::ReportFree(this, address, size);

RECYCLER\_PERF\_COUNTER\_DEC(LiveObject);

RECYCLER\_PERF\_COUNTER\_SUB(LiveObjectSize, size);

RECYCLER\_PERF\_COUNTER\_ADD(FreeObjectSize, size);

if (HeapInfo::IsSmallBlockAllocation(HeapInfo::GetAlignedSizeNoCheck(size)))

{

RECYCLER\_PERF\_COUNTER\_DEC(SmallHeapBlockLiveObject);

RECYCLER\_PERF\_COUNTER\_SUB(SmallHeapBlockLiveObjectSize, size);

RECYCLER\_PERF\_COUNTER\_ADD(SmallHeapBlockFreeObjectSize, size);

}

else

{

RECYCLER\_PERF\_COUNTER\_DEC(LargeHeapBlockLiveObject);

RECYCLER\_PERF\_COUNTER\_SUB(LargeHeapBlockLiveObjectSize, size);

RECYCLER\_PERF\_COUNTER\_ADD(LargeHeapBlockFreeObjectSize, size);

}

#ifdef RECYCLER\_MEMORY\_VERIFY

if (this->VerifyEnabled())

{

VerifyCheckPad(address, size);

}

#endif

#ifdef PROFILE\_RECYCLER\_ALLOC

TrackFree(address, size);

#endif

#ifdef RECYCLER\_STATS

collectionStats.objectSweptCount++;

collectionStats.objectSweptBytes += size;

if (!isForceSweeping)

{

collectionStats.objectSweptFreeListCount++;

collectionStats.objectSweptFreeListBytes += size;

}

#endif

}

size\_t

RecyclerHeapObjectInfo::GetSize() const

{

Assert(m\_heapBlock);

size\_t size;

#if LARGEHEAPBLOCK\_ENCODING

if (isUsingLargeHeapBlock)

{

size = m\_largeHeapBlockHeader->objectSize;

}

#else

if (m\_heapBlock->IsLargeHeapBlock())

{

size = ((LargeHeapBlock\*)m\_heapBlock)->GetObjectSize(m\_address);

}

#endif

else

{

// All small heap block types have the same layout for the object size field.

size = ((SmallHeapBlock\*)m\_heapBlock)->GetObjectSize();

}

#ifdef RECYCLER\_MEMORY\_VERIFY

if (m\_recycler->VerifyEnabled())

{

size -= \*(size\_t \*)(((char \*)m\_address) + size - sizeof(size\_t));

}

#endif

return size;

}