**CHENEY’S ALGORITHM:**

#include "gc\_cheney.h"

#include "garbage\_collector.h"

#include <unistd.h>

#include <string.h>

#include "gc\_util.h"

#include <syslog.h>

#include "gc\_cheney\_base.h"

int gc\_cheney\_init()

{

void \*chunk;

chunk = get\_memory\_primitive(2\*SEMISPACE\_SIZE);

if(chunk == NULL)

{

return 1;

}

gc\_cheney\_base\_to\_space = (block\_t\*)((uint64\_t)chunk + SEMISPACE\_SIZE);

if(gc\_cheney\_base\_to\_space == NULL)

{

return 2;

}

gc\_cheney\_base\_from\_space = chunk;

gc\_cheney\_base\_semispace\_middle = gc\_cheney\_base\_to\_space;

if(gc\_cheney\_base\_from\_space == NULL)

{

return 3;

}

gc\_cheney\_base\_remaining\_block = gc\_cheney\_base\_from\_space;

gc\_cheney\_base\_roots\_count = 0;

return 0;

}

int gc\_cheney\_cleanup()

{

void \*ptr = gc\_cheney\_base\_from\_space > gc\_cheney\_base\_to\_space ? gc\_cheney\_base\_to\_space : gc\_cheney\_base\_from\_space;

release\_memory\_primitive(ptr);

}

void \*gc\_cheney\_malloc(int type)

{

block\_t \*block;

if(type\_table[type].size <= sizeof(uint64\_t))

{

block = gc\_cheney\_base\_alloc\_block\_of\_size(0);

}

else

{

block = gc\_cheney\_base\_alloc\_block\_of\_size(type\_table[type].size - sizeof(uint64\_t));

}

if(block == NULL)

{

return block;

}

block\_set\_is\_array(block, 0);

block\_set\_type(block, type);

block\_set\_array\_size(block, 0);

return get\_data\_start(block);

}

void \*gc\_cheney\_malloc\_array(int type, size\_t size)

{

block\_t \*block;

block = gc\_cheney\_base\_alloc\_block\_of\_size(type\_table[type].size \* size);

if(block == NULL)

{

return block;

}

block\_set\_is\_array(block, 1);

block\_set\_type(block, type);

block\_set\_array\_size(block, size);

return get\_data\_start(block);

}

int gc\_cheney\_collect()

{

return gc\_cheney\_collect\_from\_roots(gc\_cheney\_base\_roots, gc\_cheney\_base\_roots\_count);

}

int gc\_cheney\_collect\_from\_roots(root\_ptr roots[], size\_t size)

{

block\_t \*todo\_ptr;

int i;

gc\_cheney\_base\_remaining\_to\_space = gc\_cheney\_base\_to\_space;

todo\_ptr = gc\_cheney\_base\_to\_space;

for(i = 0; i < size; i++)

{

roots[i].ptr = gc\_cheney\_scan\_ptr(roots[i].ptr, TYPE\_PTR, roots[i].is\_array);

}

while(todo\_ptr < gc\_cheney\_base\_remaining\_to\_space)

{

gc\_cheney\_walk\_block(todo\_ptr);

todo\_ptr = next\_block(todo\_ptr);

}

gc\_cheney\_base\_swich\_semispaces();

return 0;

}

void \*gc\_cheney\_scan\_ptr(void \*ptr, uint64\_t type, int is\_array\_ptr)

{

block\_t \*block;

block = (block\_t\*)((art\_ptr\_t)ptr - (is\_array\_ptr ? sizeof(block\_t) : sizeof(uint64\_t)));

if(gc\_cheney\_base\_is\_old\_mem(block))

{

if(!block\_has\_forward(block))

{

block\_t \*dst;

size\_t block\_size = block\_get\_size(block);

dst = gc\_cheney\_base\_get\_mem((void\*\*)&gc\_cheney\_base\_remaining\_to\_space, block\_size - sizeof(block\_t));

memcpy(dst, block, block\_size);

block\_set\_forward(block, dst);

}

return gc\_cheney\_base\_get\_forwarding\_addr(ptr, block, block\_get\_forward(block));

}

return NULL;

}

int gc\_cheney\_scan\_struct(void \*ptr, type\_info\_t \*info)

{

int i;

for(i = 0; i < info->number\_of\_references; i++)

{

void \*\*slot\_ptr, \*fwd;

slot\_ptr = (void\*\*)(ptr + info->references[i].offset);

fwd = gc\_cheney\_scan\_ptr(\*slot\_ptr, ptr\_info\_get\_type(&info->references[i]), ptr\_info\_is\_array(&info->references[i]));

if(fwd != NULL)

{

\*slot\_ptr = fwd;

}

}

return 0;

}

int gc\_cheney\_walk\_block(block\_t \*block)

{

if(block\_is\_array(block))

{

return gc\_cheney\_walk\_array(block);

}

switch(block\_get\_type(block))

{

case TYPE\_UNDEFINED:

case TYPE\_INT:

case TYPE\_DOUBLE:

return 0;

default:

return gc\_cheney\_walk\_struct(block);

}

}

int gc\_cheney\_walk\_struct(block\_t \*block)

{

gc\_cheney\_scan\_struct(get\_data\_start(block), block\_get\_info(block));

}

int gc\_cheney\_walk\_array(block\_t \*block)

{

if(block\_is\_struct\_block(block))

{

void \*ptr;

type\_info\_t \*info;

info = block\_get\_info(block);

for(ptr = get\_data\_start(block); ptr < get\_data\_end(block); ptr += info->size)

{

gc\_cheney\_scan\_struct(ptr, info);

}

}

}

int64\_t gc\_cheney\_remaining\_space()

{

return gc\_cheney\_base\_remaining\_space();

}

**MARK AND COMPACT ALGORITHM:**

#ifndef V8\_MARK\_COMPACT\_INL\_H\_

#define V8\_MARK\_COMPACT\_INL\_H\_

#include "isolate.h"

#include "memory.h"

#include "mark-compact.h"

namespace v8 {

namespace internal {

MarkBit Marking::MarkBitFrom(Address addr) {

MemoryChunk\* p = MemoryChunk::FromAddress(addr);

return p->markbits()->MarkBitFromIndex(p->AddressToMarkbitIndex(addr), p->ContainsOnlyData());

}

void MarkCompactCollector::SetFlags(int flags) {

sweep\_precisely\_ = ((flags & Heap::kSweepPreciselyMask) != 0);

reduce\_memory\_footprint\_ = ((flags & Heap::kReduceMemoryFootprintMask) != 0);

abort\_incremental\_marking\_ =

((flags & Heap::kAbortIncrementalMarkingMask) != 0);

}

void MarkCompactCollector::MarkObject(HeapObject\* obj, MarkBit mark\_bit) {

ASSERT(Marking::MarkBitFrom(obj) == mark\_bit);

if (!mark\_bit.Get()) {

mark\_bit.Set();

MemoryChunk::IncrementLiveBytesFromGC(obj->address(), obj->Size());

ProcessNewlyMarkedObject(obj);

}

}

bool MarkCompactCollector::MarkObjectWithoutPush(HeapObject\* object) {

MarkBit mark = Marking::MarkBitFrom(object);

bool old\_mark = mark.Get();

if (!old\_mark) SetMark(object, mark);

return old\_mark;

}

void MarkCompactCollector::MarkObjectAndPush(HeapObject\* object) {

if (!MarkObjectWithoutPush(object)) marking\_deque\_.PushBlack(object);

}

void MarkCompactCollector::SetMark(HeapObject\* obj, MarkBit mark\_bit) {

ASSERT(!mark\_bit.Get());

ASSERT(Marking::MarkBitFrom(obj) == mark\_bit);

mark\_bit.Set();

MemoryChunk::IncrementLiveBytesFromGC(obj->address(), obj->Size());

if (obj->IsMap()) {

heap\_->ClearCacheOnMap(Map::cast(obj));

}

}

bool MarkCompactCollector::IsMarked(Object\* obj) {

ASSERT(obj->IsHeapObject());

HeapObject\* heap\_object = HeapObject::cast(obj);

return Marking::MarkBitFrom(heap\_object).Get();

}

void MarkCompactCollector::RecordSlot(Object\*\* anchor\_slot,

Object\*\* slot,

Object\* object) {

Page\* object\_page = Page::FromAddress(reinterpret\_cast<Address>(object));

if (object\_page->IsEvacuationCandidate() &&

!ShouldSkipEvacuationSlotRecording(anchor\_slot)) {

if (!SlotsBuffer::AddTo(&slots\_buffer\_allocator\_,

object\_page->slots\_buffer\_address(),

slot,

SlotsBuffer::FAIL\_ON\_OVERFLOW)) {

EvictEvacuationCandidate(object\_page);

}

}

}

} }

#endif // V8\_MARK\_COMPACT\_INL\_H\_