Real-Time Memory Management and Compaction with Bounded Response Times

Hannes Payer

University of Salzburg

July 11, 2007

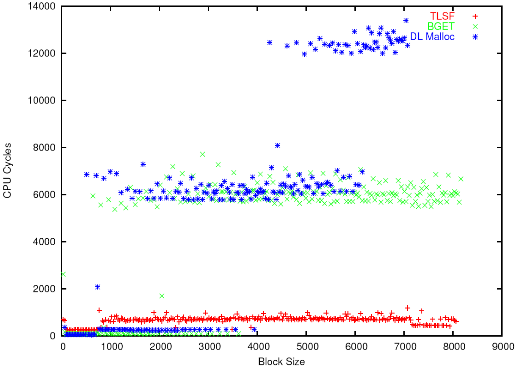
Motivation

- traditional memory management is typically non-deterministic:
  - unpredictable response times
  - worst-case response times are usually unbounded
  - memory fragmentation
- for WCET analysis it is important that all the running code (application, libraries, and operating system) is predictable

Motivation

- response time and memory compaction should be independent of the memory context
- integration in a cooperative system:
  - no preemption
  - incremental behaviour of each operation

Motivation



Source: *M. Masmano, I. Ripoll, and A. Crespo. Dynamic storage allocation for real-time embedded systems, 2003.*

Goals and Requirements

memory operations malloc, free, dereference

- bounded response time (predictability)
- fast response time

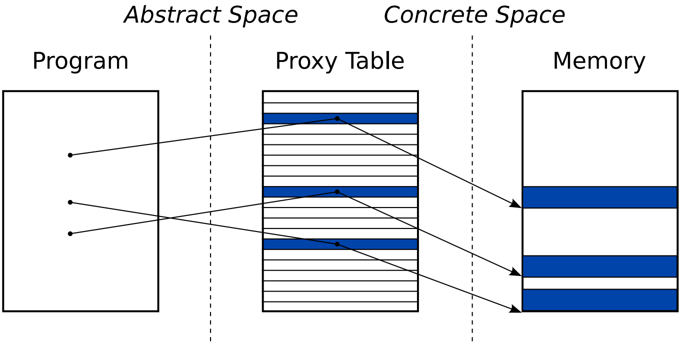
fragmentation

- eliminated by compaction
- distribute compaction workload (deterministic timing behaviour)
- predictable timing solution for reference updates required

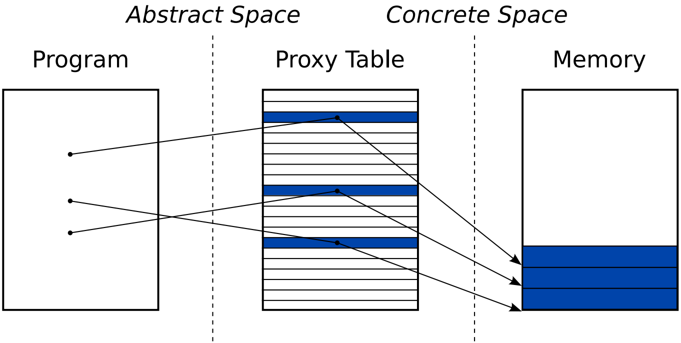
Compaction & Abstract vs. Concrete Address Space

- reference updates degrades predictability
- compaction with reference updates can not be done fast and bounded?!
- $\Rightarrow$ memory operations operate on abstract addresses to be independent of compaction operations
- abstract address refers to concrete addresses
- abstract address never changes
- concrete addresses may change due to compaction

Proxy - abstract to concrete address mapping



Proxy - abstract to concrete address mapping



Memory Operations API

- `void **malloc(size)`: allocates memory and returns an abstract address that points to the concrete address space
- `void free(abstract address)`: frees the concrete address space that belongs to the abstract address
- `void *dereference(abstract address, offset)`: returns an address of the concrete address space that corresponds to the abstract address + offset

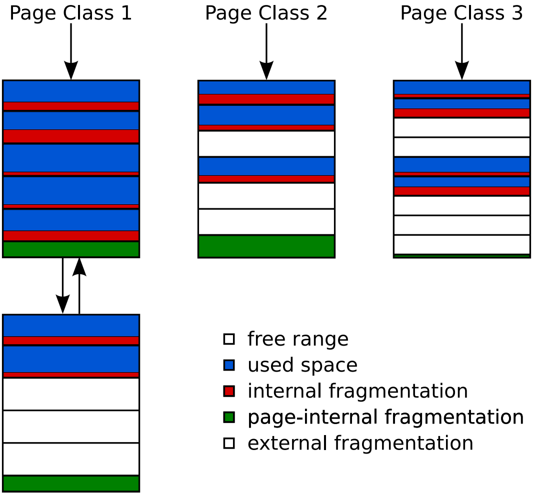
Pages (Metronome)

- concrete address space is divided into pages of equal size
- each page itself is divided into fixed-sized page blocks
- there exist n predefined page block sizes, which results in n different size classes
- allocation operations are handled by the smallest size class that fits the allocation request
- segregated free lists
- redistribute unused pages to other size classes

Fragmentation (Metronome)

- internal fragmentation (unused space at the end of a page block)
- page-internal fragmentation (unused space at the and of a page)
- external fragmentation (unused page blocks)

Page Classes & Fragmentation (Metronome)



Compaction

Invariant 1: There exists at most one page in a page class, which is not full.
Invariant 2: The not-full-page has to be the last element of its page class list.
Rule 1: If an element of a full page gets freed and there exists no not-full-page, then this page is the new not-full-page of the page class and it is moved to the end of the page class list.
Rule 2: If an element of a full page gets freed and there exists a not-full-page, then one element of the not-full-page has to be moved to this page.

Compaction

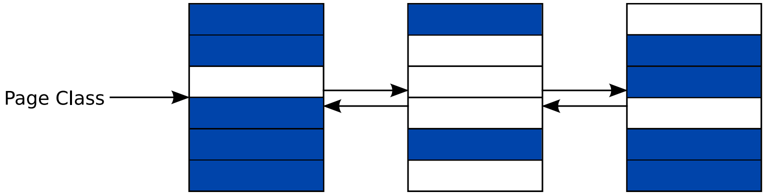


Figure: fragmented pages

Compaction

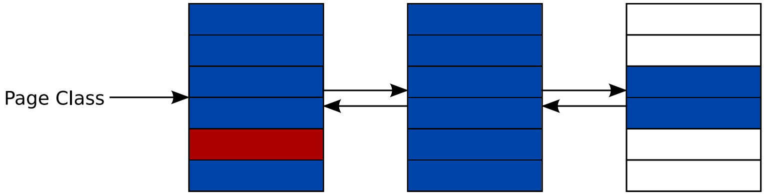


Figure: free one memory range

Compaction

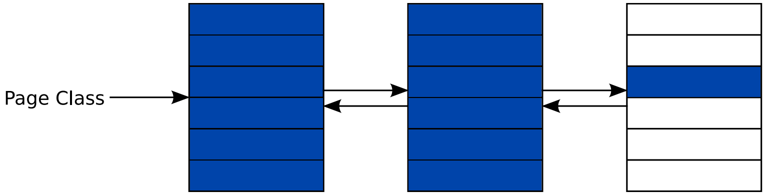


Figure: move memory range of the last page to the affected page

Page Management Internals

administration information

- concrete address space
- free entries counter
- next/previous pointer to the next/previous page of the page class (doubly-circularly-linked list)
- free page block list
- used page block list
- reference to the page class

Page Management Internals

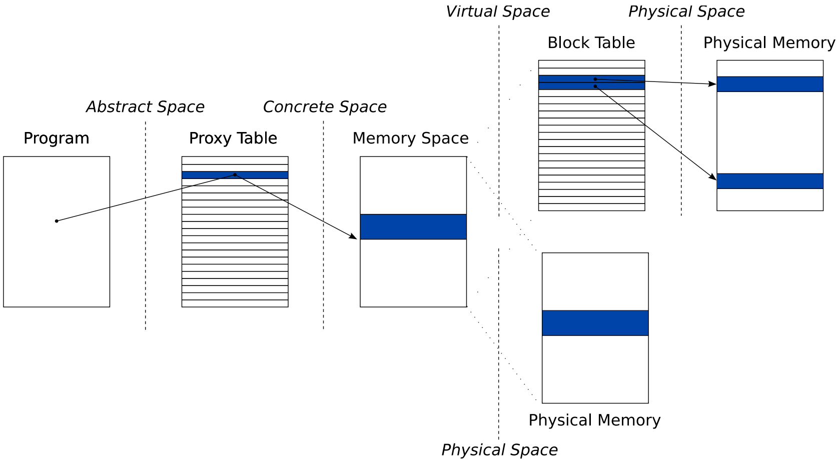
free page block list

- singly-linked list (stack) for free page blocks of a page
- special list construct
- no list initialization (no free element concatenation)
- build up free list incrementally
- 16 bits needed for link pointer

used page block list

- needed for compaction
- doubly-circularly-linked list (32 bits for link pointers)
- 2-dimensional bitmap (1 bit for each page block)

Overview - 2 different implementations

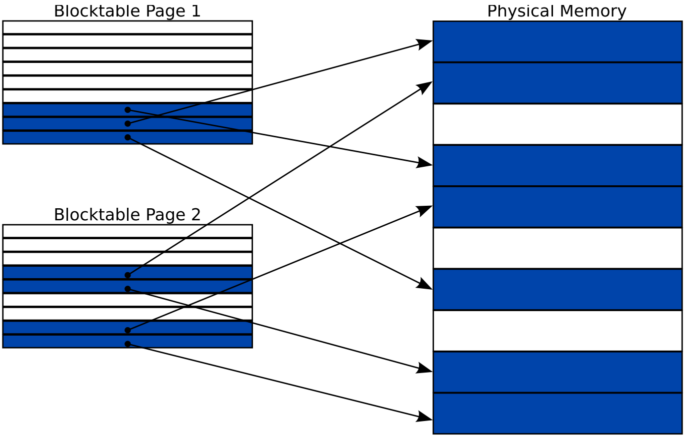


Virtual Memory

- the virtual memory is divided into blocks of equal size
- the physical memory is divided into block frames of equal size
- allocated memory objects are contiguous in virtual space, but arbitrarily distributed in physical space
- compaction happens in virtual space - block frames are never moved in physical space
- constant compaction speedup

- every page contains a block table that implements the virtual-block to physical-block-frame mapping

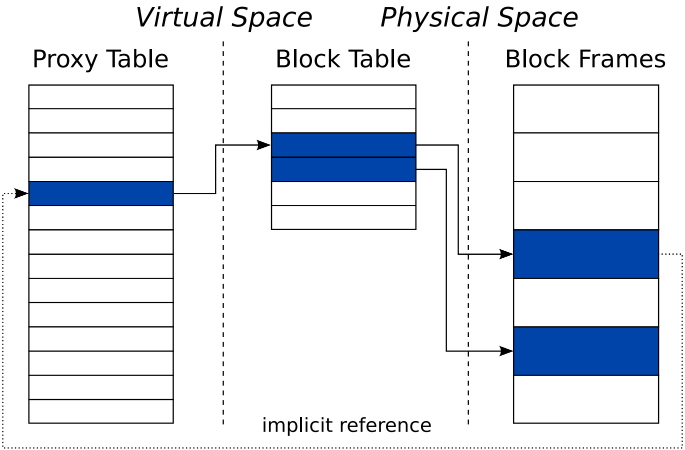
Block Table



Proxy Name-Generation

- allocated memory is never moved physically
- # block frames = # proxy entries
- proxy entry assignment in constant time
- given the concrete address of a concrete memory object, the abstract address can be determined implicitly

Proxy Implicit Reference



void **malloc(size)

```
page = getPageOfPageClass(size);
freeIndexInPage = getFreeIndexOfPage(page);
for(i=0; i<numberOfBlockFrames; i++){
    blockframe = getBlockFrameFromFreeList();
    page->bf[index+i] = blockframe;
}
addUsedIndexToPage(page, index);
proxyIndex = getIndexOfBlockFrame(page->bf[index]);
return getProxyEntry(proxyIndex, &page->bf[index]);
```

void free(abstractAddress)

```
concreteAddress = dereference(abstractAddress);
page = getPageToConcreteAddress(concreteAddress);
index = indexOfRangeInPage(page, concreteAddress);
removeUsedIndexFromPage(page, index);
for(i=0; i<numberOfBlockFrames; i++){
    addBlockFrameToFreeList(page->bf[indexOfBf+i]);
}
addFreeIndexToPage(page, index);

if(page->entries == 0){
    removePageFromPageClass(page);
}
else{
    doCompaction(page);
}
```

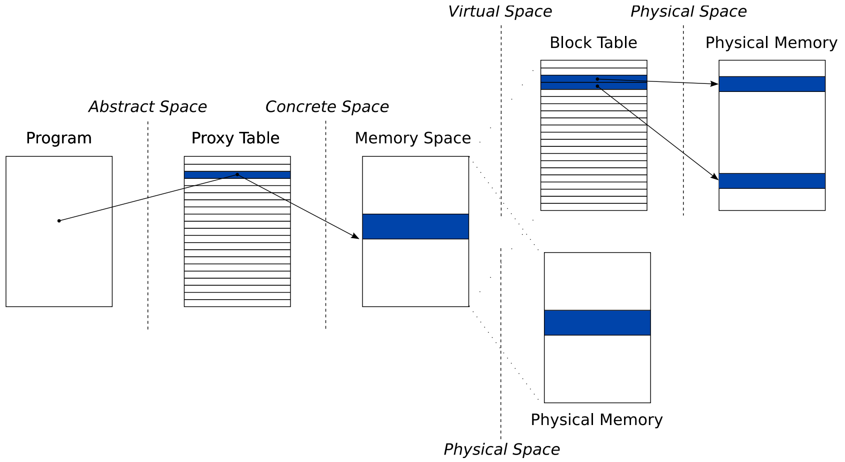
void *dereference(abstractAddress, index)

```
concreteAddress = dereference(abstractAddress);
return getWordReference(concreteAddress, index);
```

Conclusion I

- malloc: $\Theta(\frac{n}{\textit{Block Frame Size}})$ timing behaviour (constant in page class size)
- free: $\Theta(\frac{n}{\textit{Block Frame Size}})$ timing behaviour (constant in page class size)
  - $\frac{n}{\textit{Block Frame Size}}$ block table free operations
  - $\frac{n}{\textit{Block Frame Size}}$ compaction operations (if compaction occurs)
- dereference: $\Theta(1)$ timing behaviour
- the timing behaviour of all memory operations is independent of the global memory state

Overview - 2 different implementations



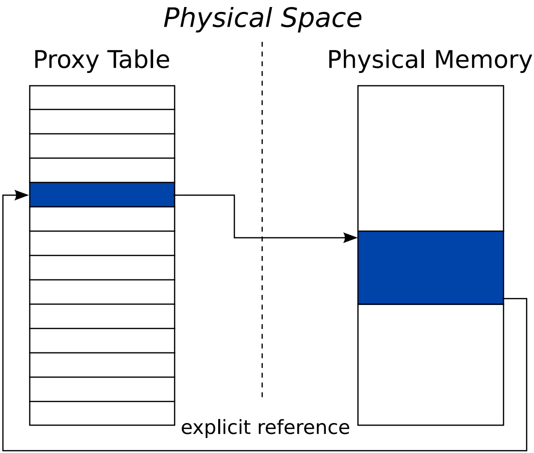
Physical Memory

- the concrete address space is the physical space $\Rightarrow$ physical space is divided into pages of equal size
- allocated memory objects are contiguous in physical space
- compaction leads to movements in physical space

Proxy Name-Generation

- physical space locations change due to compaction operations
- free list over the proxy table (constant time)
- beginning address of the memory object is not related to the proxy table entry position
- $\Rightarrow$ memory objects need an explicit reference back to the corresponding proxy entry

Proxy Explicit Reference



void **malloc(size)

```
page = getPageOfPageClass(size);
freeIndexInPage = getFreeIndexOfPage(page);
addUsedIndexToPage(page, index);
proxyIndex = getIndexOfBlockFrame(page->bf[index]);
return getProxyEntry(proxyIndex, &page->bf[index]);
```

void free(abstractAddress)

```
concreteAddress = dereference(abstractAddress);
page = getPageToConcreteAddress(concreteAddress);
index = indexOfRangeInPage(page, concreteAddress);
removeUsedIndexFromPage(page, index);
addFreeIndexToPage(page, index);

if(page->entries == 0){
    removePageFromPageClass(page);
}
else{
    doCompaction(page);
}
```

void *dereference(abstractAddress, index)

`just use the *(*abstractAddress + index)`
`explicit dereference method is not necessary`

Conclusion II

- malloc: $\Theta(1)$ timing behaviour
- free: $\mathcal{O}(n)$ timing behaviour (constant in range size)
  - $\Theta(1)$ free operation
  - n compaction operations $\Rightarrow \Theta(n)$
- dereference: $\Theta(1)$ timing behaviour
- the timing behaviour of all memory operations is independent of the global memory state