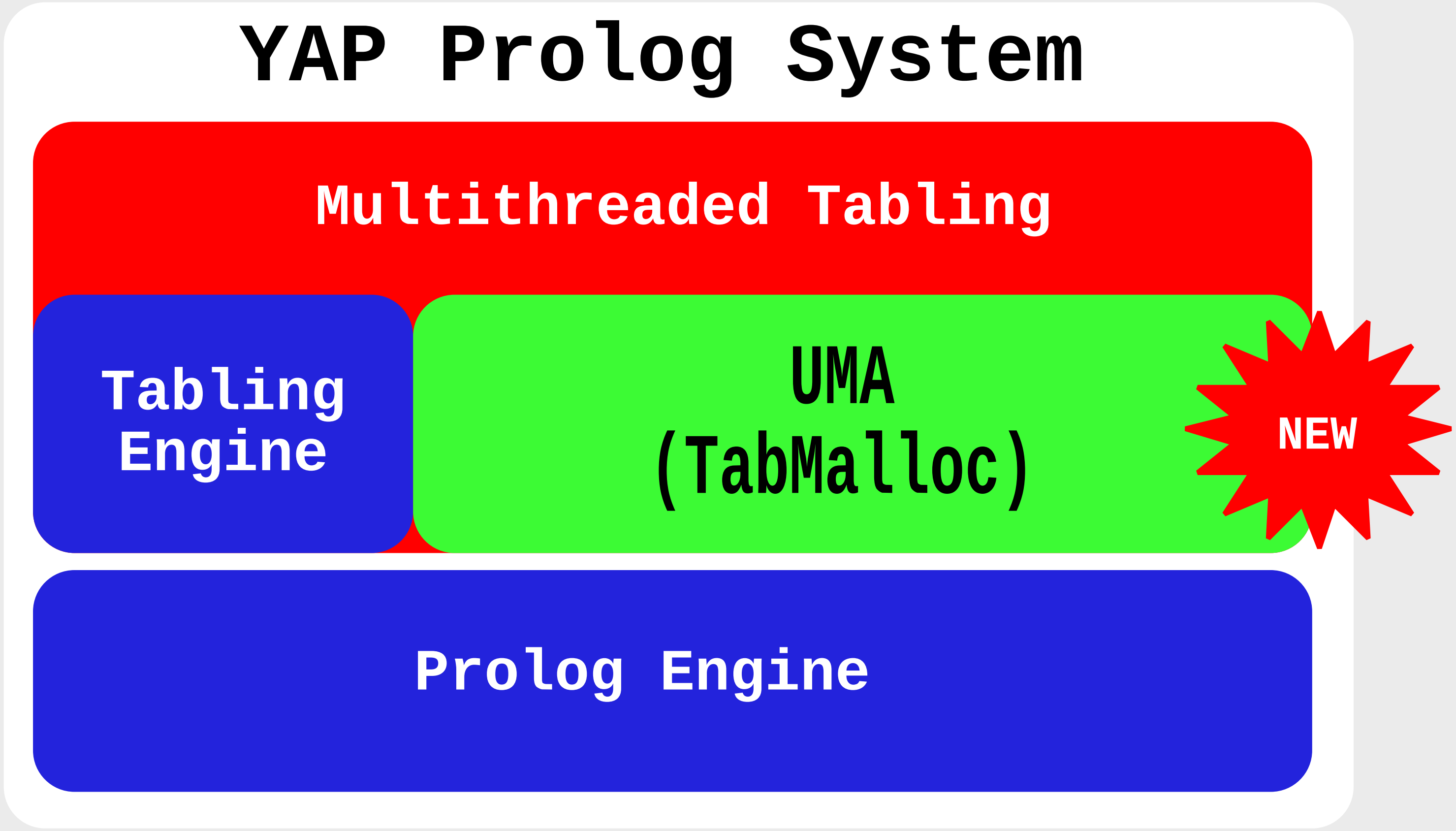
An Efficient and Scalable Memory Allocator for Multithreaded Tabled Evaluation of Logic Programs

Miguel Areias and Ricardo Rocha
CRACS & INESC-TEC LA
Faculty of Sciences, University of Porto, Portugal
miguel-areias@dcc.fc.up.pt *ricroc@dcc.fc.up.pt*

ICPADS, Singapore, December 2012

Motivation of This Work

- A New **User Level Memory Allocator**(UMA) aimed to improve the **scalability** of our **Multithreaded Tabling** engine.



Prolog Resolution

- Prolog systems are known to have good performances and flexibility, but their standard resolution, limits the potential of the Logic Programing paradigm.
- Prolog resolution cannot deal properly with the following situations:
  - **Positive Infinite Cycles** (insufficient expressiveness)
  - **Negative Infinite Cycles** (inconsistency)
  - **Redundant Computations** (inefficiency)

Prolog Resolution: Infinite Cycles

```
path(X,Z) :- path(X,Y), edge(Y,Z).
path(X,Z) :- edge(X,Z).

edge(1,2).
edge(2,1).
```

```
? - path(1,Z)
```

Prolog Resolution: Infinite Cycles

```
path(X,Z) :- path(X,Y), edge(Y,Z).
path(X,Z) :- edge(X,Z).

edge(1,2).
edge(2,1).
```

```
? - path(1,Z)
path(1,Y), edge(Y,Z)
```

Infinite Cycle

Prolog Resolution: Infinite Cycles

```
path(X,Z) :- edge(X,Y), path(Y,Z).
path(X,Z) :- edge(X,Z).

edge(1,2).
edge(2,1).
```

```
? - path(1,Z)
```

Prolog Resolution: Infinite Cycles

```
path(X,Z) :- edge(X,Y), path(Y,Z).
path(X,Z) :- edge(X,Z).

edge(1,2).
edge(2,1).
```

```
? - path(1,Z)
       |
edge(1,Y), path(Y,Z)
       |
   path(2,Z)
       |
edge(2,Y), path(Y,Z)
       |
   path(1,Z)
       |
 Infinite Cycle
```

Tabling in Prolog Systems

- **Tabling** is an implementation technique that overcomes some of the limitations of Prolog resolution.
  - Tabled subgoals are evaluated by storing their answers in an appropriate data space, called the **table space**.
  - Variant calls to tabled subgoals are resolved by **consuming** the answers already stored in the table instead of **being re-evaluated** against the program clauses.

Tabling in Prolog Systems

- **Tabling** is an implementation technique that overcomes some of the limitations of Prolog resolution.
  - Tabled subgoals are evaluated by storing their answers in an appropriate data space, called the **table space**.
  - Variant calls to tabled subgoals are resolved by **consuming** the answers already stored in the table instead of **being re-evaluated** against the program clauses.

- Implementations of Tabling are currently available in systems like:
  - XSB Prolog
  - **Yap Prolog**
  - B-Prolog
  - ALS-Prolog
  - Mercury
  - Ciao Prolog

Table Space - Overview

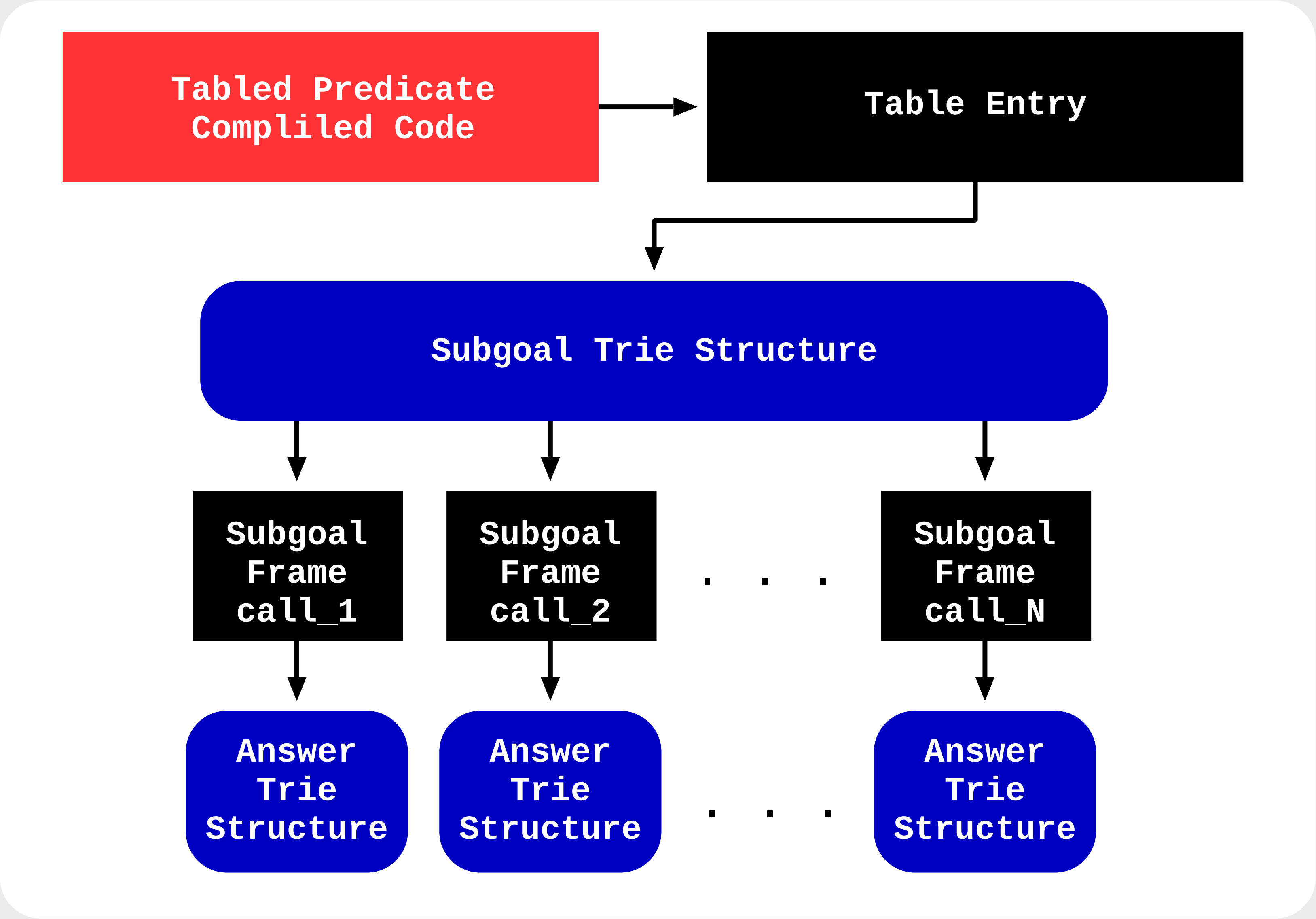


Table Space - Example

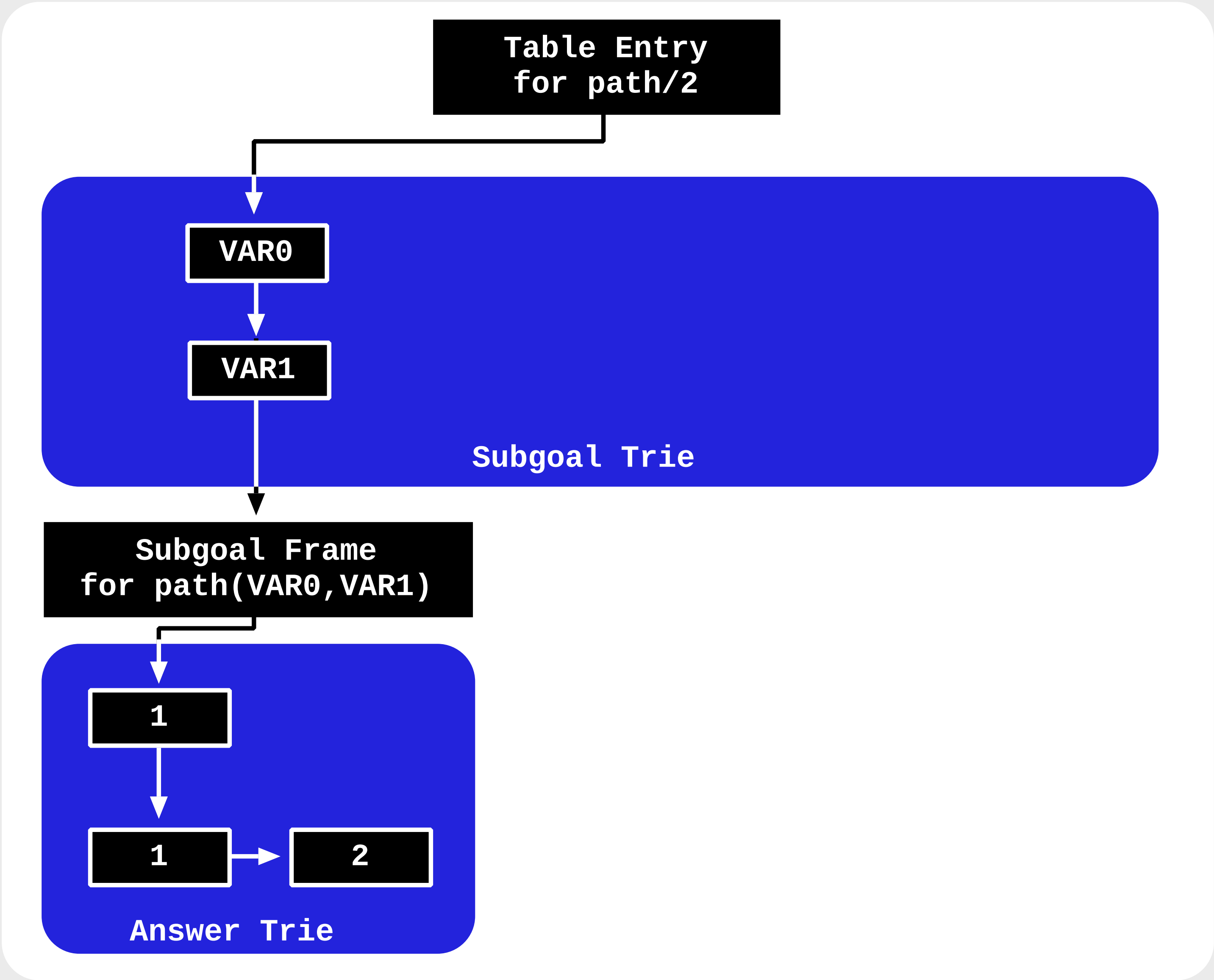
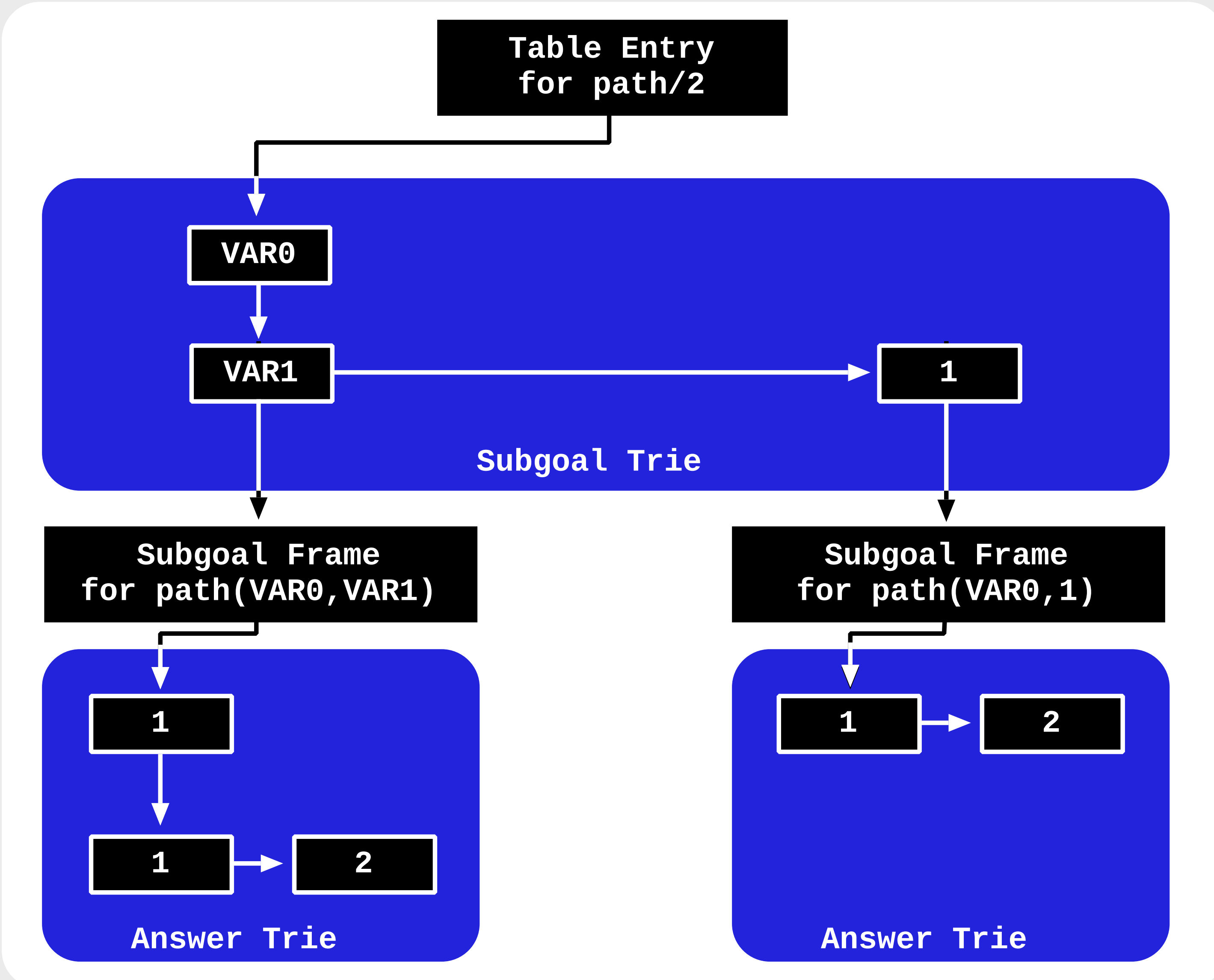


Table Space - Example



Multithreaded Tabling

- **Multithreading** in Prolog is the ability to **concurrently perform computations**, in which each computation runs independently but shares the program clauses.

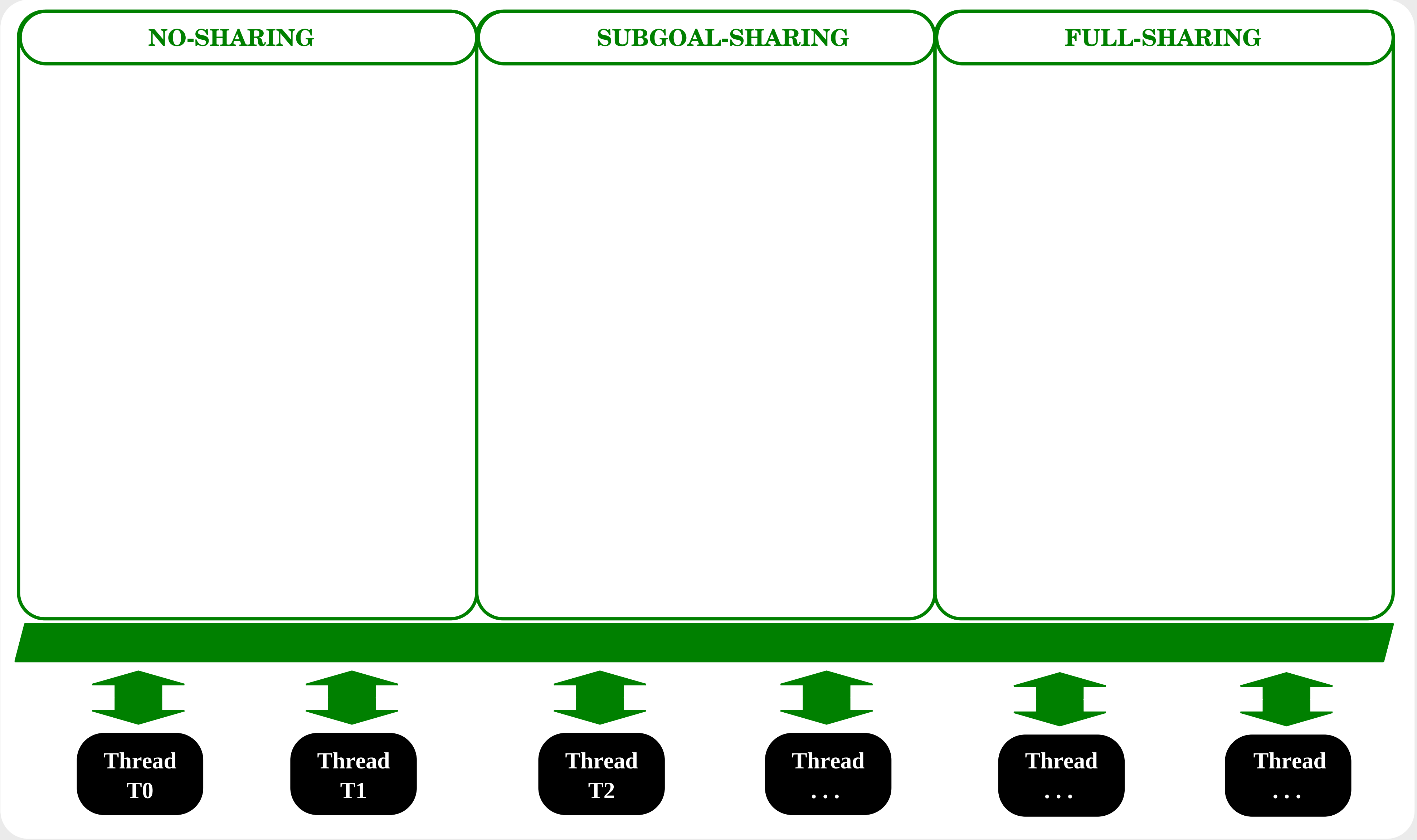
Multithreaded Tabling

- **Multithreading** in Prolog is the ability to **concurrently perform computations**, in which each computation runs independently but shares the program clauses.
- When **Multithreading** is **combined** with **Tabling**, we have the best of both worlds, since we can exploit the combination of higher procedural control with higher declarative semantics.

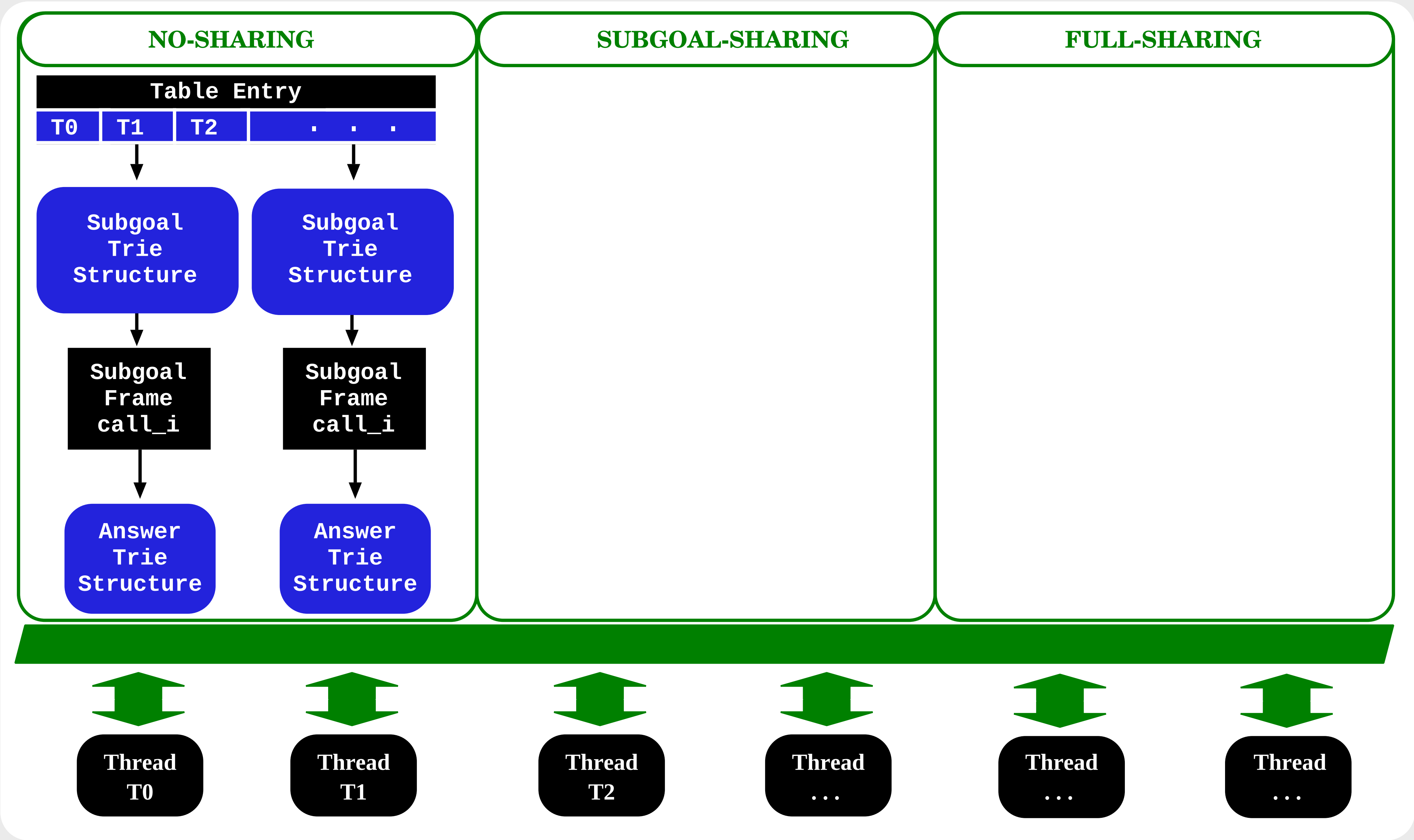
Multithreaded Tabling

- **Multithreading** in Prolog is the ability to **concurrently perform computations**, in which each computation runs independently but shares the program clauses.
- When **Multithreading** is **combined** with **Tabling**, we have the best of both worlds, since we can exploit the combination of higher procedural control with higher declarative semantics.
- **Multithreading** is currently supported by several well-known Prolog systems, but until now, XSB and **Yap Prolog** **[ICLP 2012]** were the only systems that were able to **combine** Multithreading with Tabling.
- In **Yap Prolog**, each thread views its tables as **private** but, at the **engine level**, Yap Prolog uses three different designs to support the **table space**.

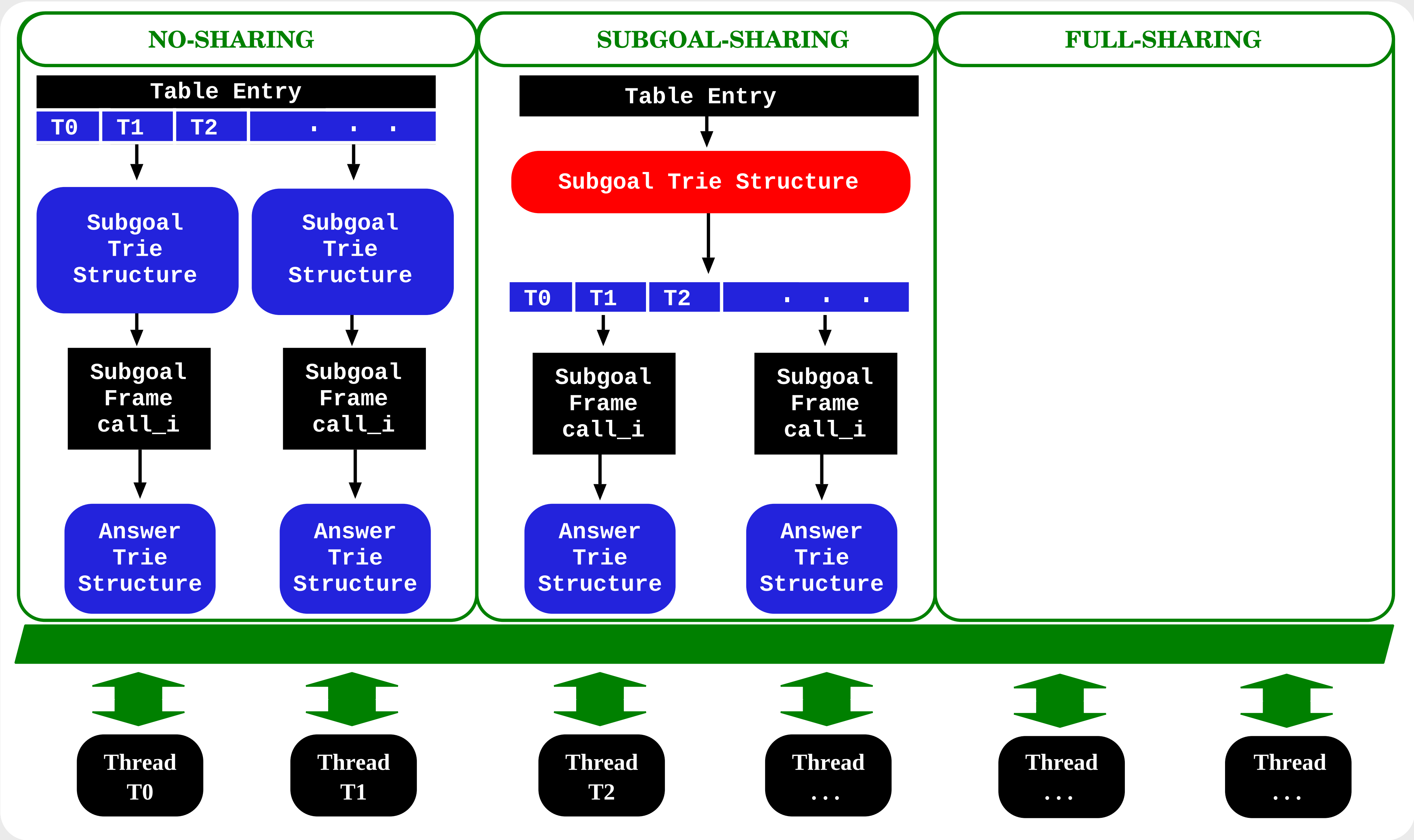
Multithreaded Tabling - Our Framework



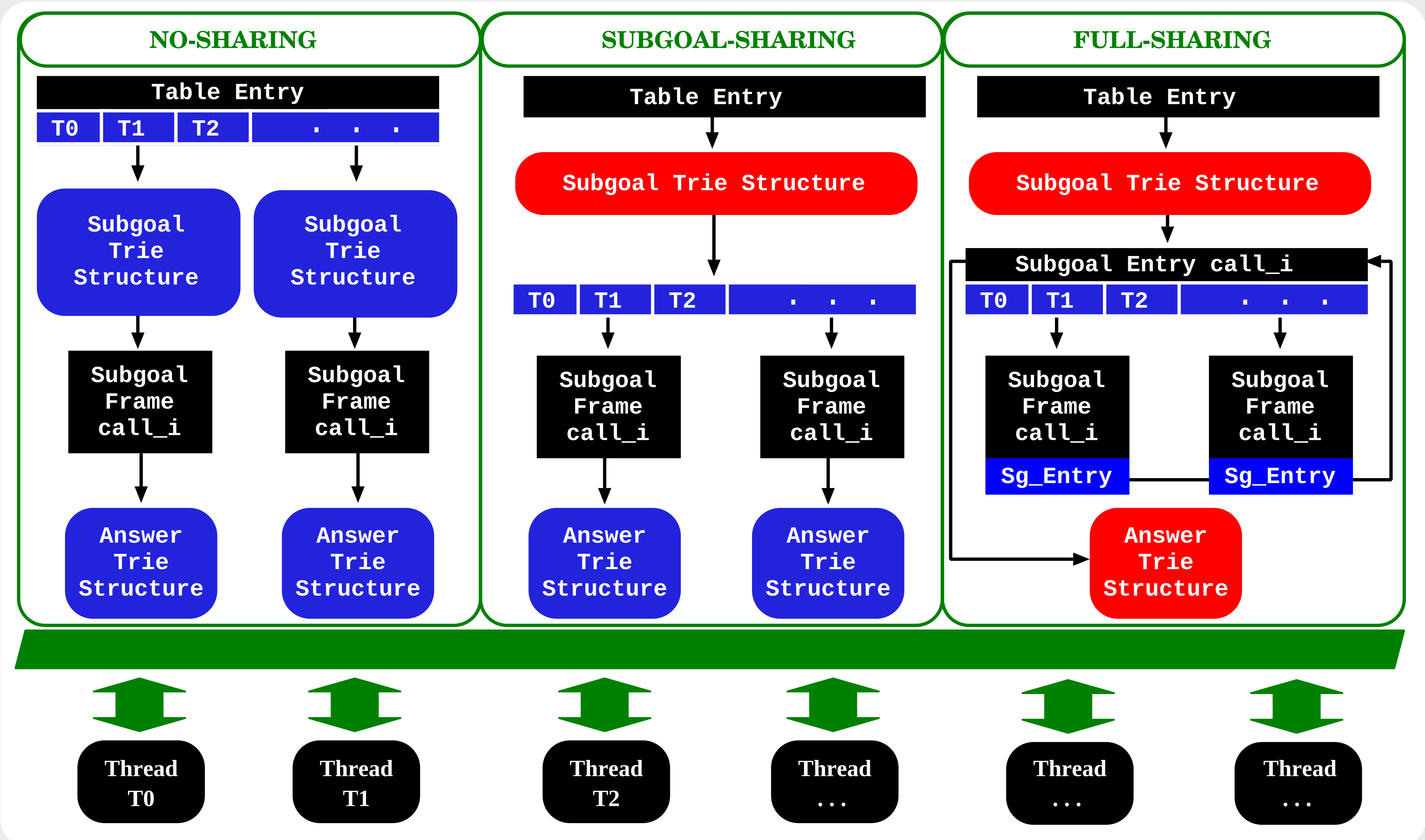
Multithreaded Tabling - Our Framework



Multithreaded Tabling - Our Framework



Multithreaded Tabling - Our Framework



User Level Memory Allocators - State-of-the-art

- A UMA is responsible for managing the **Heap**, which is an area of memory that is reserved for data.
- In a **Multithreaded** environment, all threads **share the same Heap**, thus the **allocation** and **deallocation** of objects on this area of memory must be performed **concurrently**.

User Level Memory Allocators - State-of-the-art

- A UMA is responsible for managing the **Heap**, which is an area of memory that is reserved for data.
- In a **Multithreaded** environment, all threads **share the same Heap**, thus the **allocation** and **deallocation** of objects on this area of memory must be performed **concurrently**.
- Several Concurrent UMAs are currently available:
  - Hoard
  - PtMalloc
  - TcMalloc
  - JeMalloc
  - . . .

User Level Memory Allocators - Characteristics

- **Hoard**:
  - Multiple **Global** and **Local Heaps**.
  - Per **Heap locking**.
  - **Emptiness groups** and **Blowup** avoidance algorithm.

User Level Memory Allocators - Characteristics

- **Hoard**:
  - Multiple **Global** and **Local Heaps**.
  - Per **Heap locking**.
  - **Emptiness groups** and **Blowup** avoidance algorithm.
- **PtMalloc**:
  - **Arenas** with different **Bins** for small and large objects.
  - Per **Arena locking**.
  - Allocation/Deallocation of objects inside the same arena.

User Level Memory Allocators - Characteristics

- **Hoard**:
  - Multiple **Global** and **Local Heaps**.
  - Per **Heap locking**.
  - **Emptiness groups** and **Blowup** avoidance algorithm.
- **PtMalloc**:
  - **Arenas** with different **Bins** for small and large objects.
  - Per **Arena locking**.
  - Allocation/Deallocation of objects inside the same arena.
- **TcMalloc**:
  - **Thread Cache** used for small objects ($\leq$ 32KB).
  - **Central Heap** for large objects.
  - **Locking** required when accessing the **Central Heap**.
  - A **garbage collection** operation is done by a thread when a deallocation makes a **Thread Cache** bigger than a adjustable threshold.

User Level Memory Allocators - Characteristics

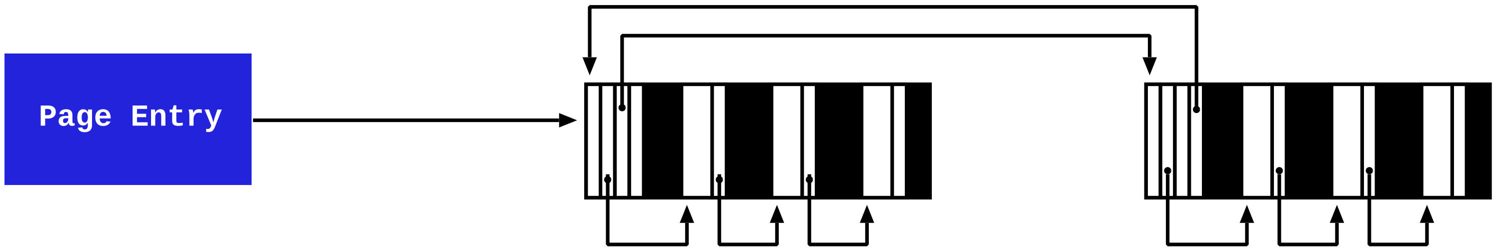
- **JeMalloc**:
  - **Thread Cache** used for small objects ($\leq$ 32KB).
  - **Arenas** with different **Bins** for small and large objects.
  - Per **Bin locking** for small objects and/or per **Arena locking**.
  - Allocation/Deallocation of objects inside the same Arena.
  - **Red-black trees** improve Allocation/Deallocation of objects.

User Level Memory Allocators - Characteristics

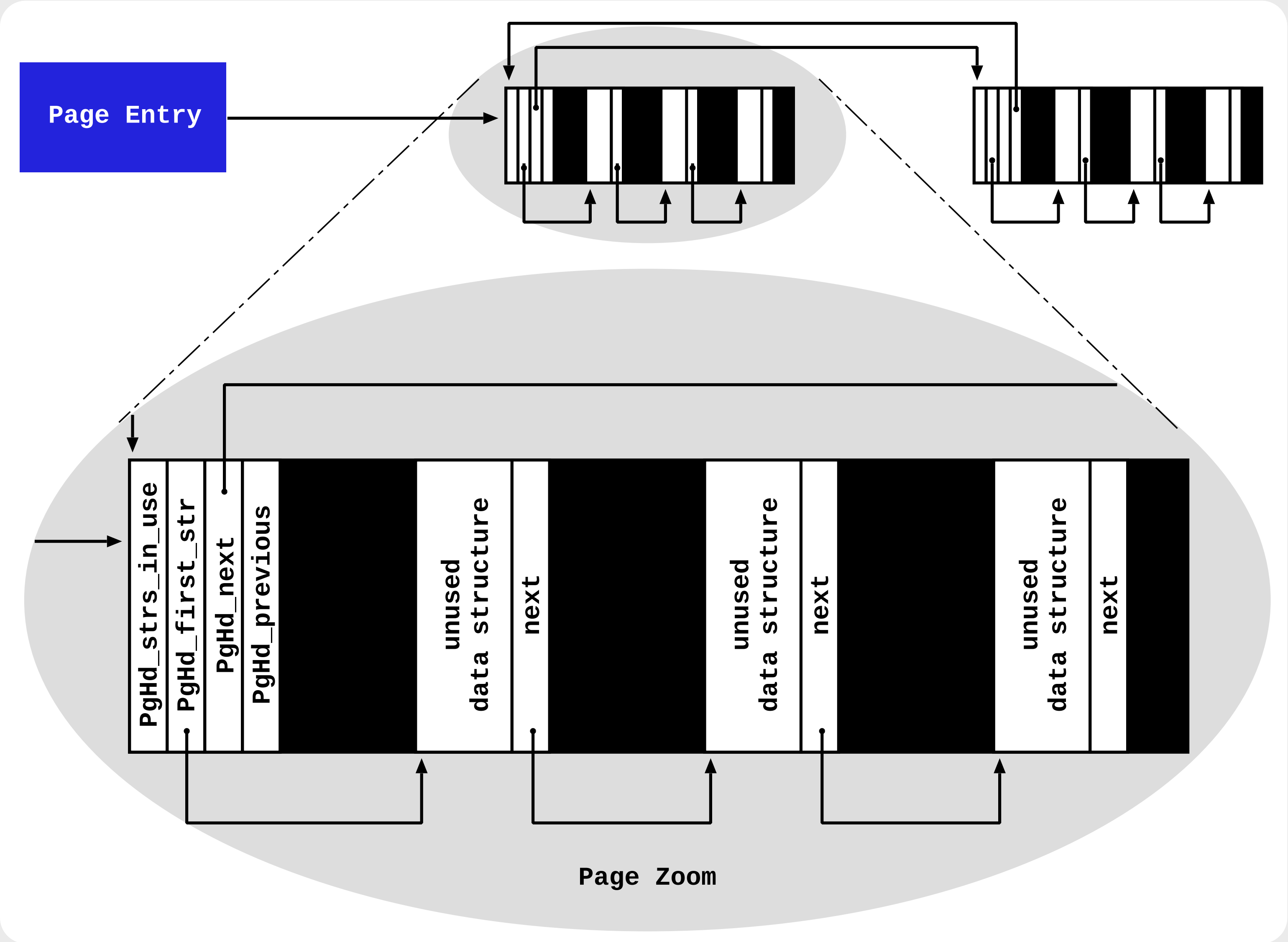
- **JeMalloc**:
  - **Thread Cache** used for small objects ($\leq$ 32KB).
  - **Arenas** with different **Bins** for small and large objects.
  - Per **Bin locking** for small objects and/or per **Arena locking**.
  - Allocation/Deallocation of objects inside the same Arena.
  - **Red-black trees** improve Allocation/Deallocation of objects.

- **TabMalloc**:
  - **Local** and **Global Page Heaps** per object type.
  - **Global** and **Local Void Heaps** for the allocation of objects when **Local Page Heaps** run empty.
  - Per **Global Heap locking**.
  - **Global Page Heaps** used for the deallocation of shared objects.
  - Allocation/Deallocation of objects is always done via Local Page Heaps, except for the main thread that performs garbage collection on the Global Page Heaps.

User Level Memory Allocators - TabMalloc

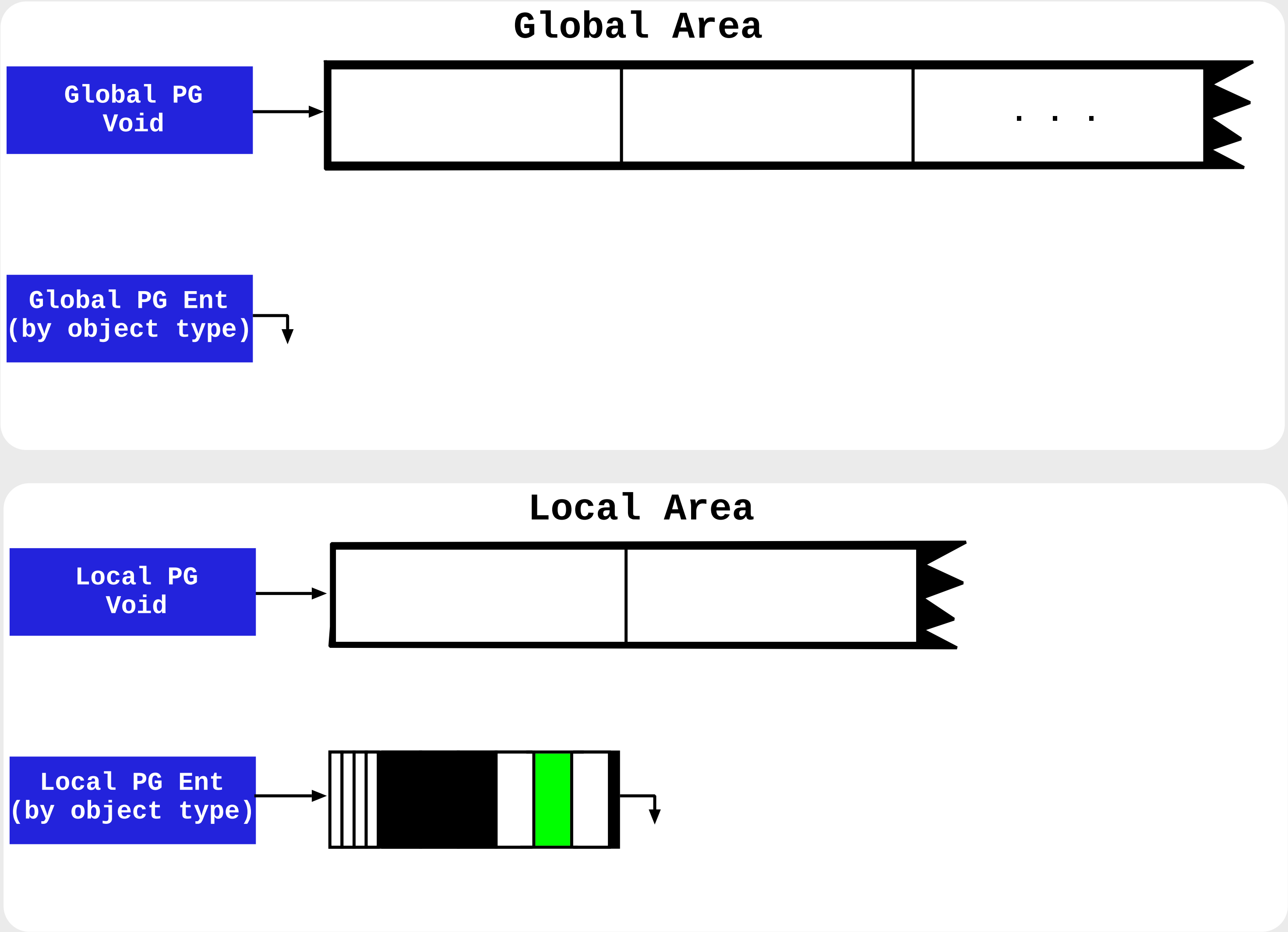


User Level Memory Allocators - TabMalloc



User Level Memory Allocators - TabMalloc

- **Case 1**: **Allocation of objects**. Local Page Heap.

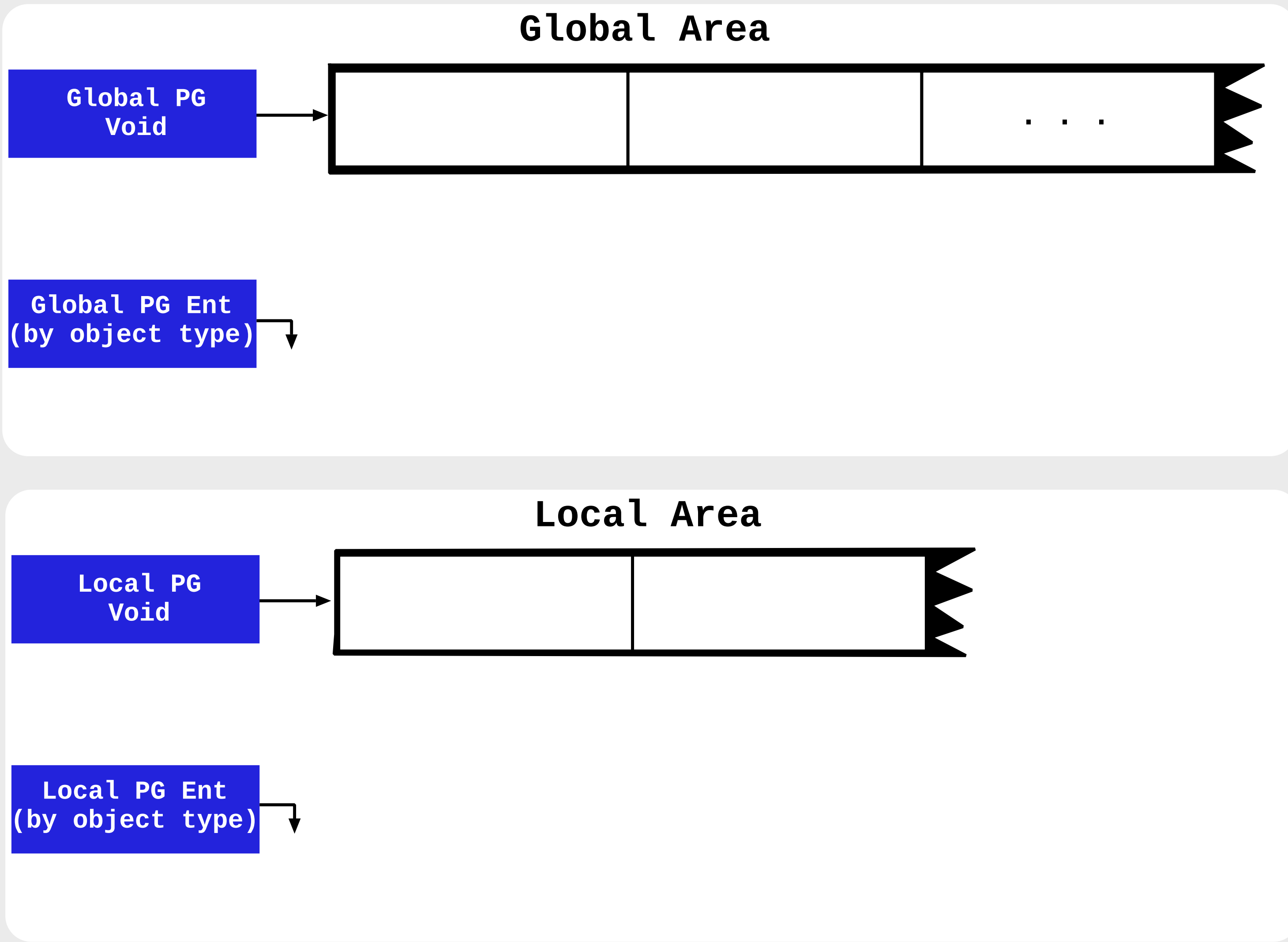


User Level Memory Allocators - TabMalloc

- **Case 2**: **Allocation of objects**. Local Void Heap.

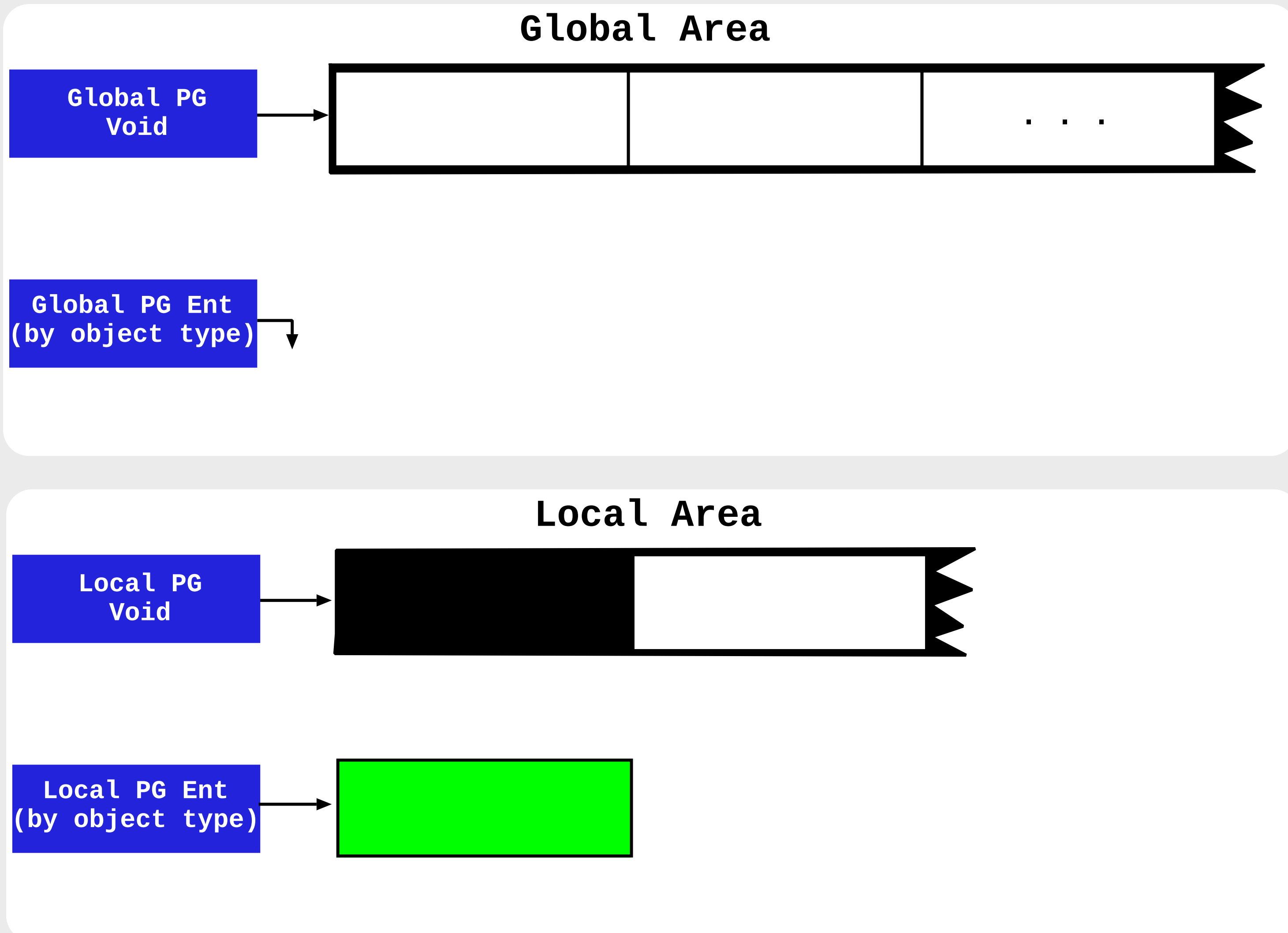
User Level Memory Allocators - TabMalloc

- **Case 2**: **Allocation of objects**. Local Void Heap.



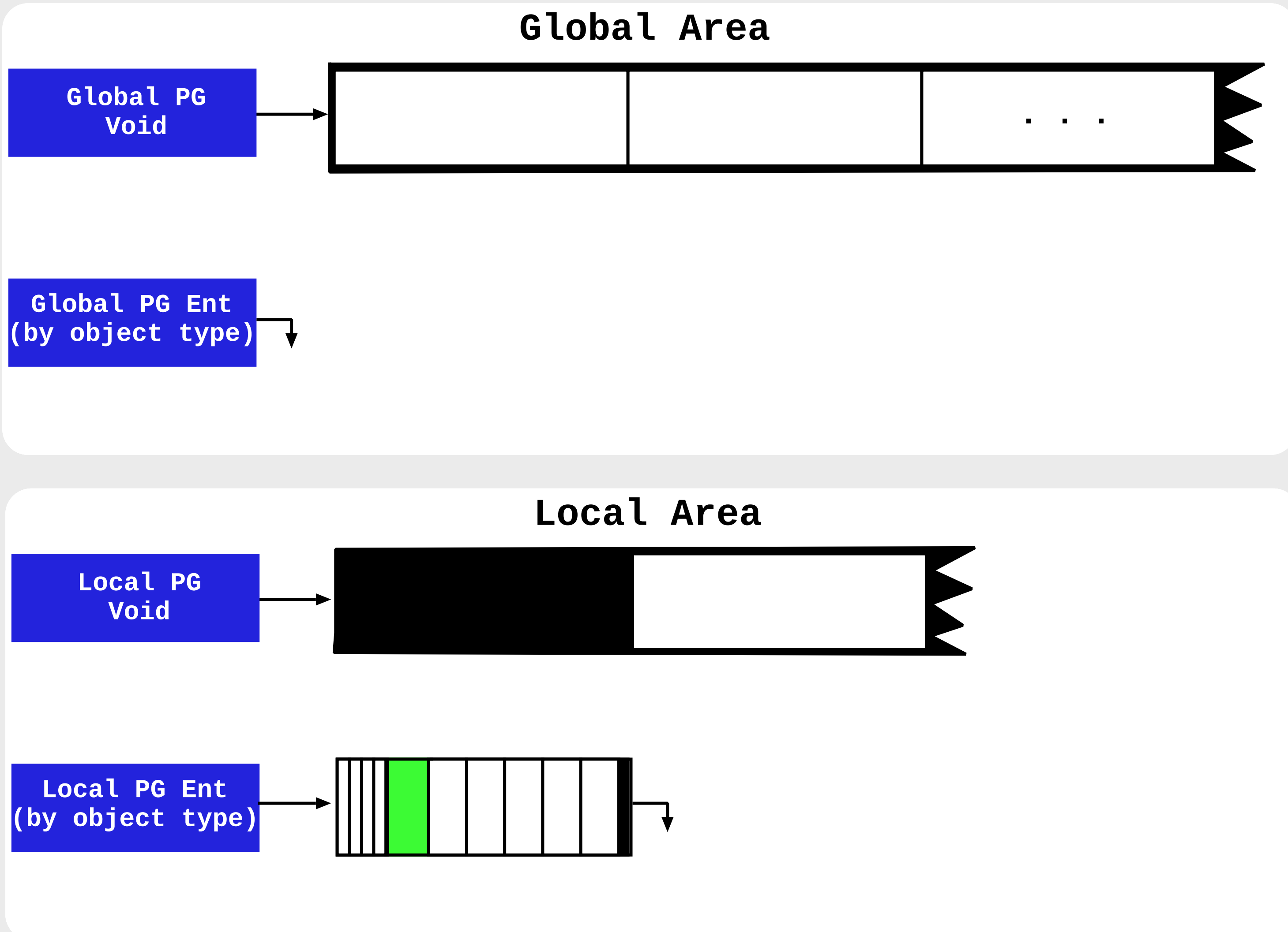
User Level Memory Allocators - TabMalloc

➤ **Case 2**: **Allocation of objects**. Local Void Heap.



User Level Memory Allocators - TabMalloc

➤ **Case 2**: **Allocation of objects**. Local Void Heap.

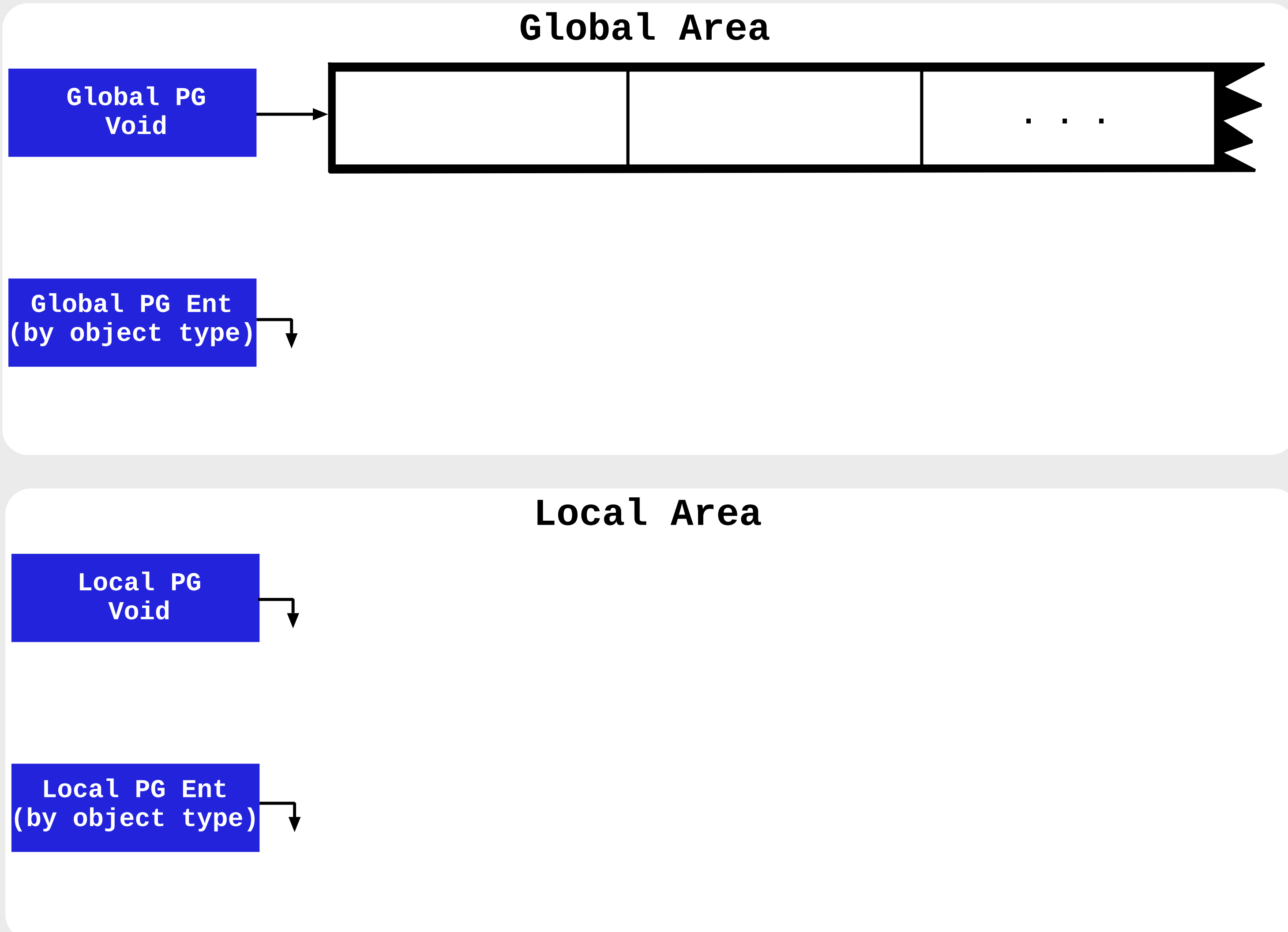


User Level Memory Allocators - TabMalloc

- **Case 3**: **Allocation of objects**. Global Void Heap.

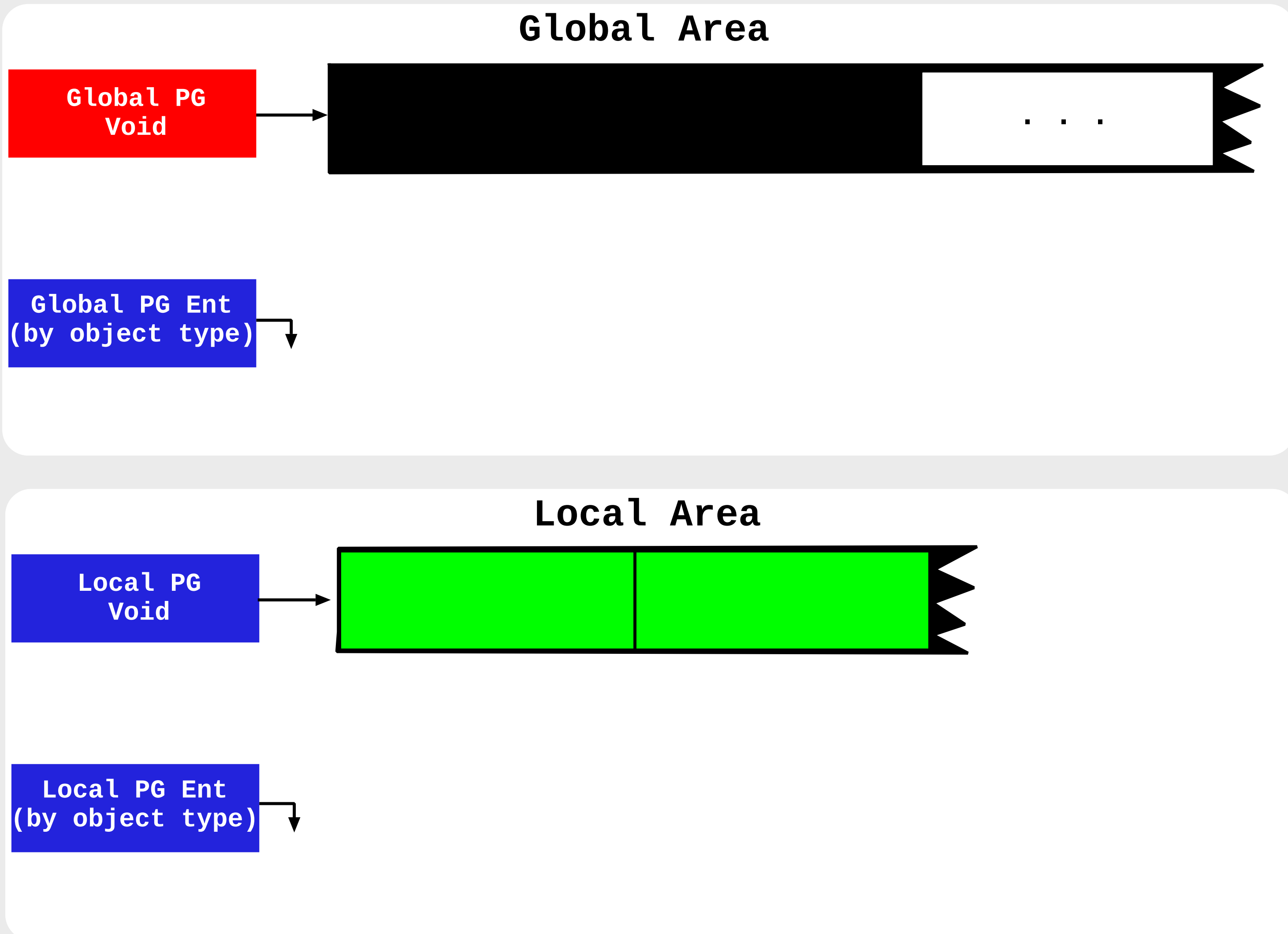
User Level Memory Allocators - TabMalloc

- **Case 3**: **Allocation of objects**. Global Void Heap.



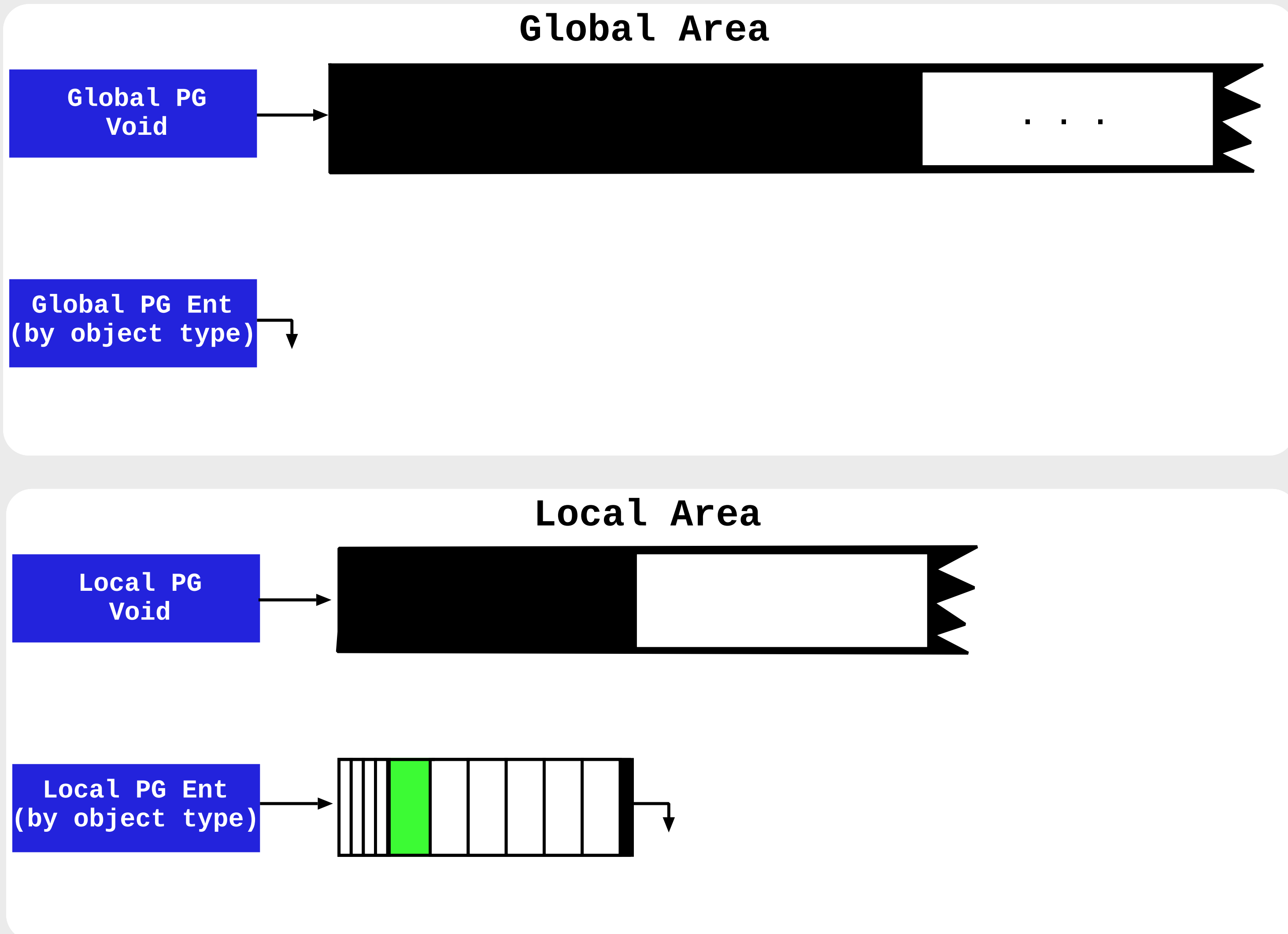
User Level Memory Allocators - TabMalloc

- **Case 3**: **Allocation of objects**. Global Void Heap.



User Level Memory Allocators - TabMalloc

- **Case 3**: **Allocation of objects**. Global Void Heap.

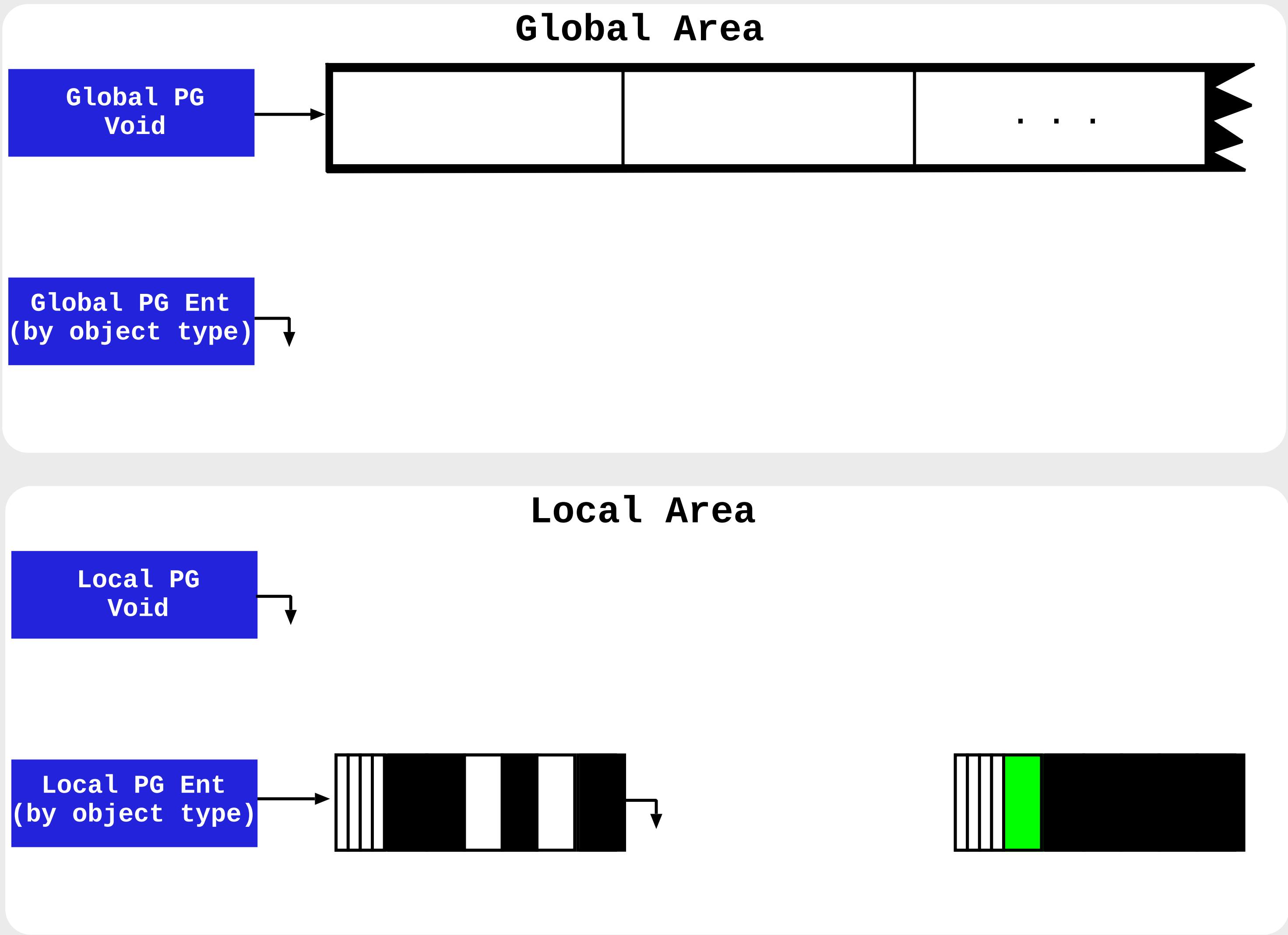


User Level Memory Allocators - TabMalloc

- **Case 4**: **Deallocation of objects**.

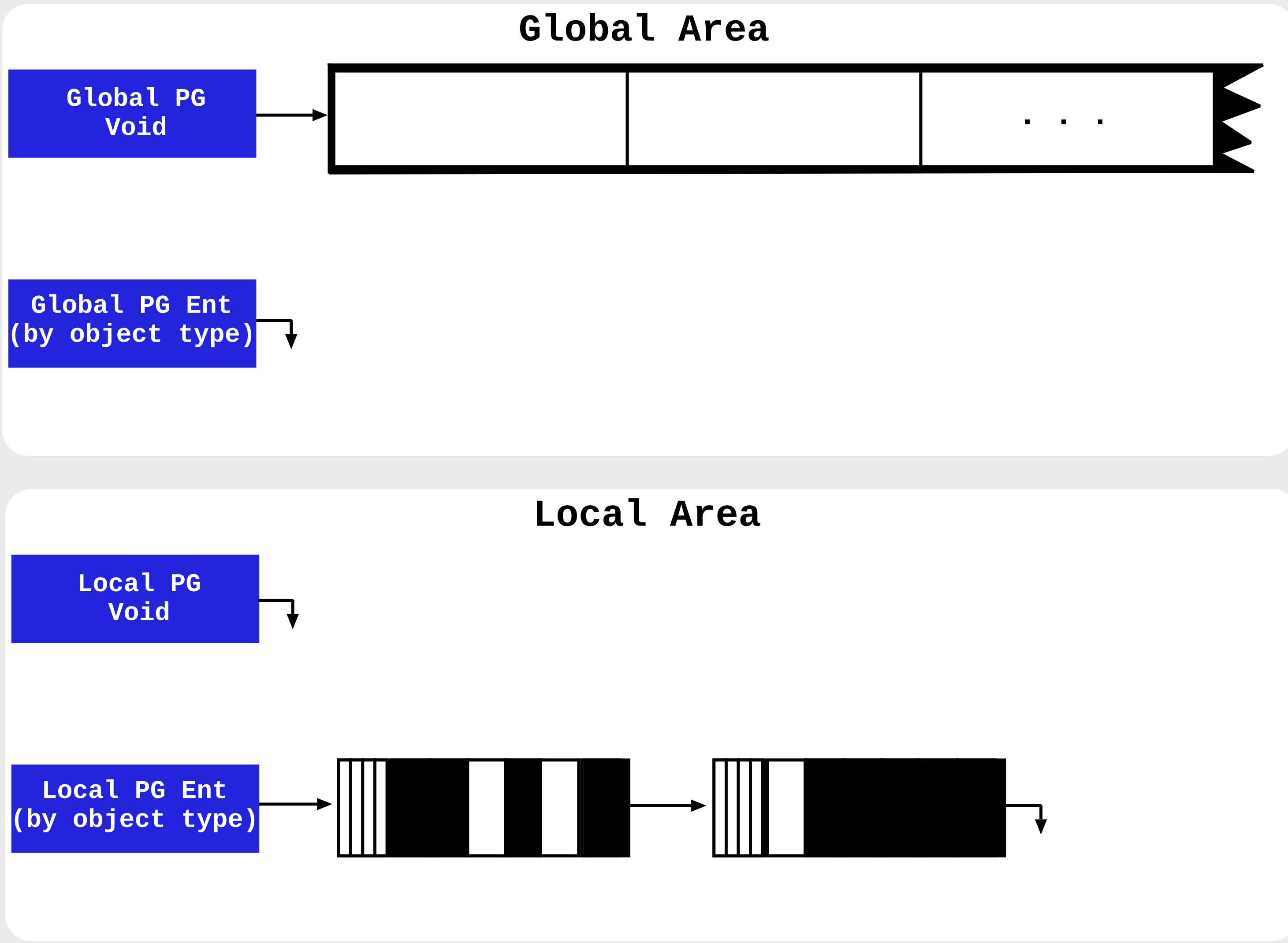
User Level Memory Allocators - TabMalloc

➤ **Case 4**: **Deallocation of objects**.



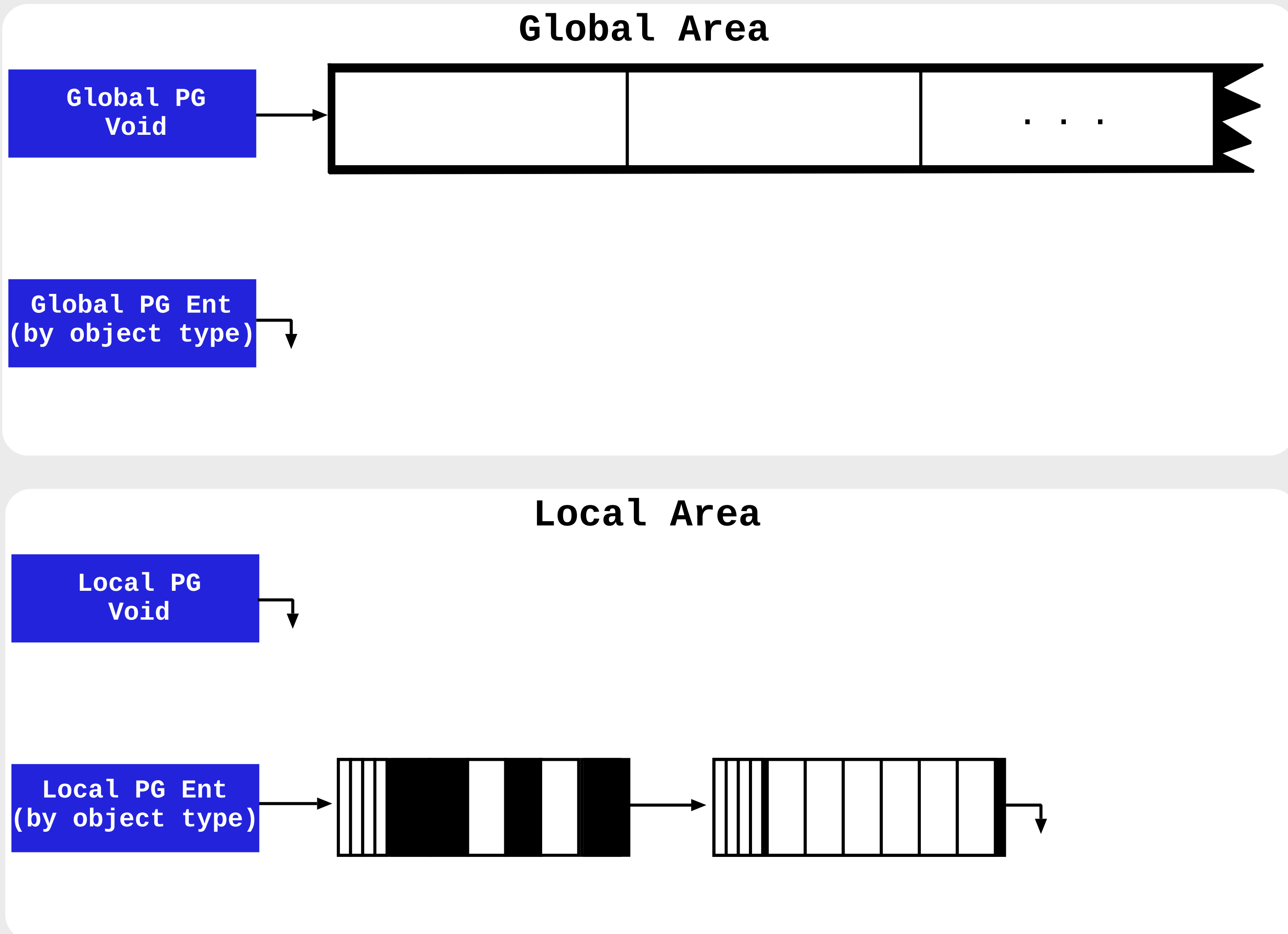
User Level Memory Allocators - TabMalloc

- **Case 4**: Deallocation of objects.



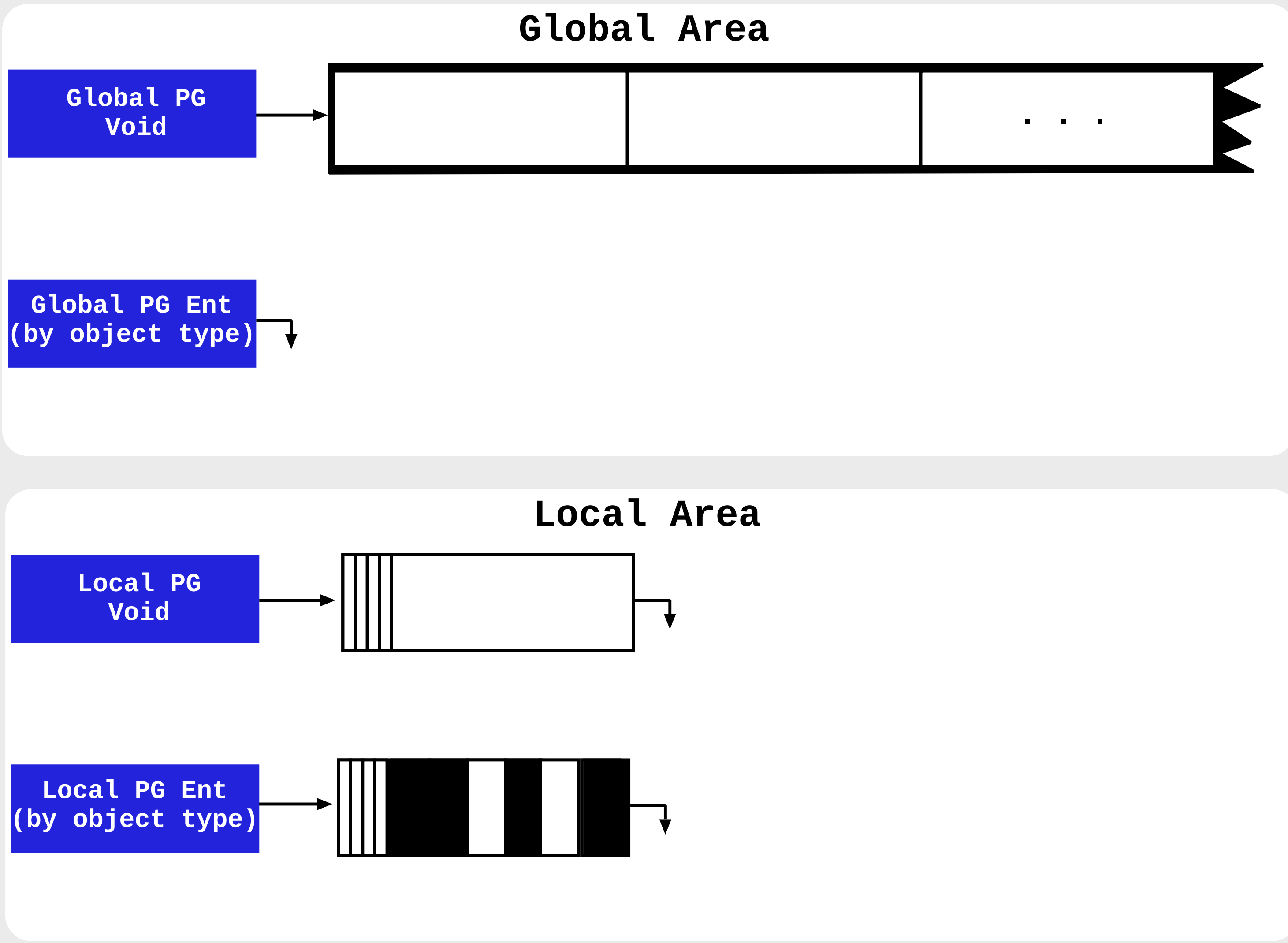
User Level Memory Allocators - TabMalloc

➤ **Case 4**: **Deallocation of objects**.



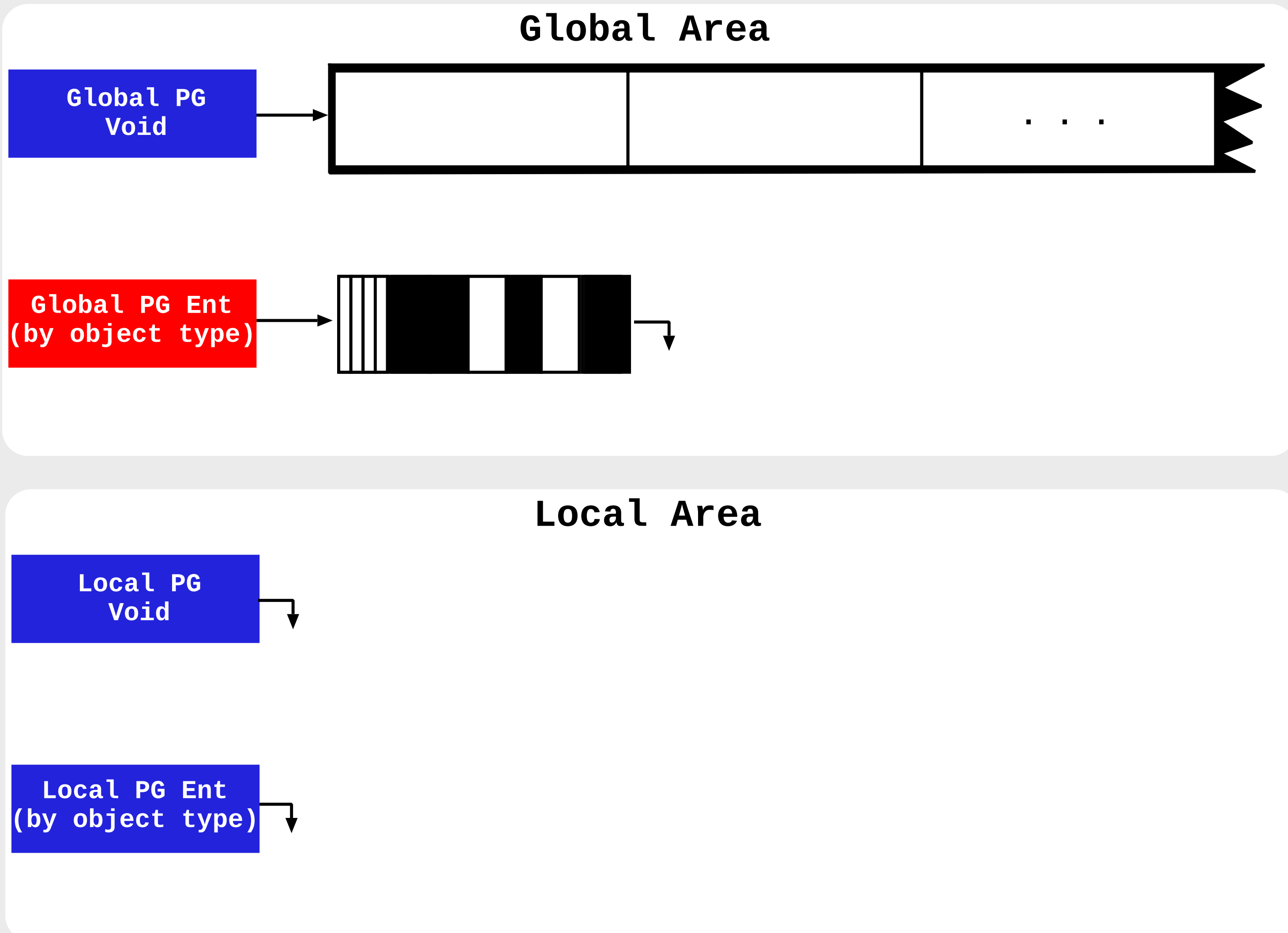
User Level Memory Allocators - TabMalloc

- **Case 4**: Deallocation of objects.



User Level Memory Allocators - TabMalloc

➤ Case 4: Deallocation of objects.



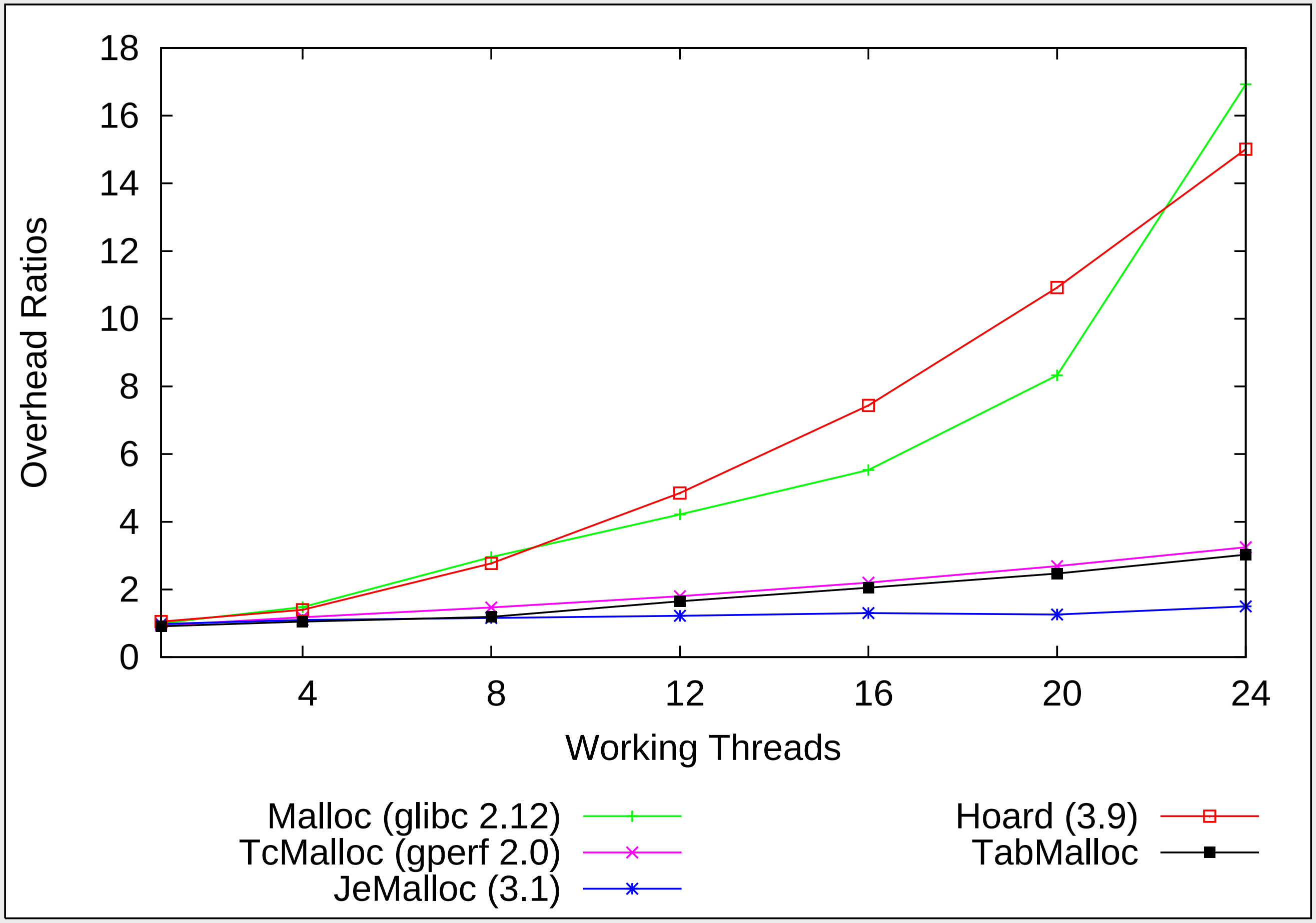
Experimental Results - Benchmark Statistics

Characteristics Of The Benchmarks: 1 Working Thread

Bench	Tabled Subgoals			Tabled Answers				Time (s)
	Calls	Trie Nodes	Trie Depth	Unique	Repeated	Trie Nodes	Trie Depth	NS
Model Checking								
IProto	1	6	5/5/5	134,361	385,423	1,554,896	4/51/67	2.4
Leader	1	5	4/4/4	1,728	574,786	41,788	15/80/97	3.7
Sieve	1	7	6/6/6	380	1,386,181	8,624	21/53/58	24.6
Large Joins								
Join2	1	6	5/5/5	2,476,099	0	2,613,660	5/5/5	3.7
Mondial	35	42	3/4/4	2,664	2,452,890	14,334	6/7/7	0.7
Path Left								
BTree	1	3	2/2/2	1,966,082	0	2,031,618	2/2/2	1.5
Pyramid	1	3	2/2/2	3,374,250	1,124,250	3,377,250	2/2/2	3.3
Cycle	1	3	2/2/2	4,000,000	2,000	4,002,001	2/2/2	4.0
Grid	1	3	2/2/2	1,500,625	4,335,135	1,501,851	2/2/2	1.9
Path Right								
BTree	131,071	262,143	2/2/2	3,801,094	0	3,997,700	1/2/2	2.3
Pyramid	3,000	6,001	2/2/2	6,745,501	2,247,001	6,751,500	1/2/2	2.7
Cycle	2,001	4,003	2/2/2	8,000,000	4,000	8,004,001	1/2/2	3.0
Grid	1,226	2,453	2/2/2	3,001,250	8,670,270	3,003,701	1/2/2	2.3

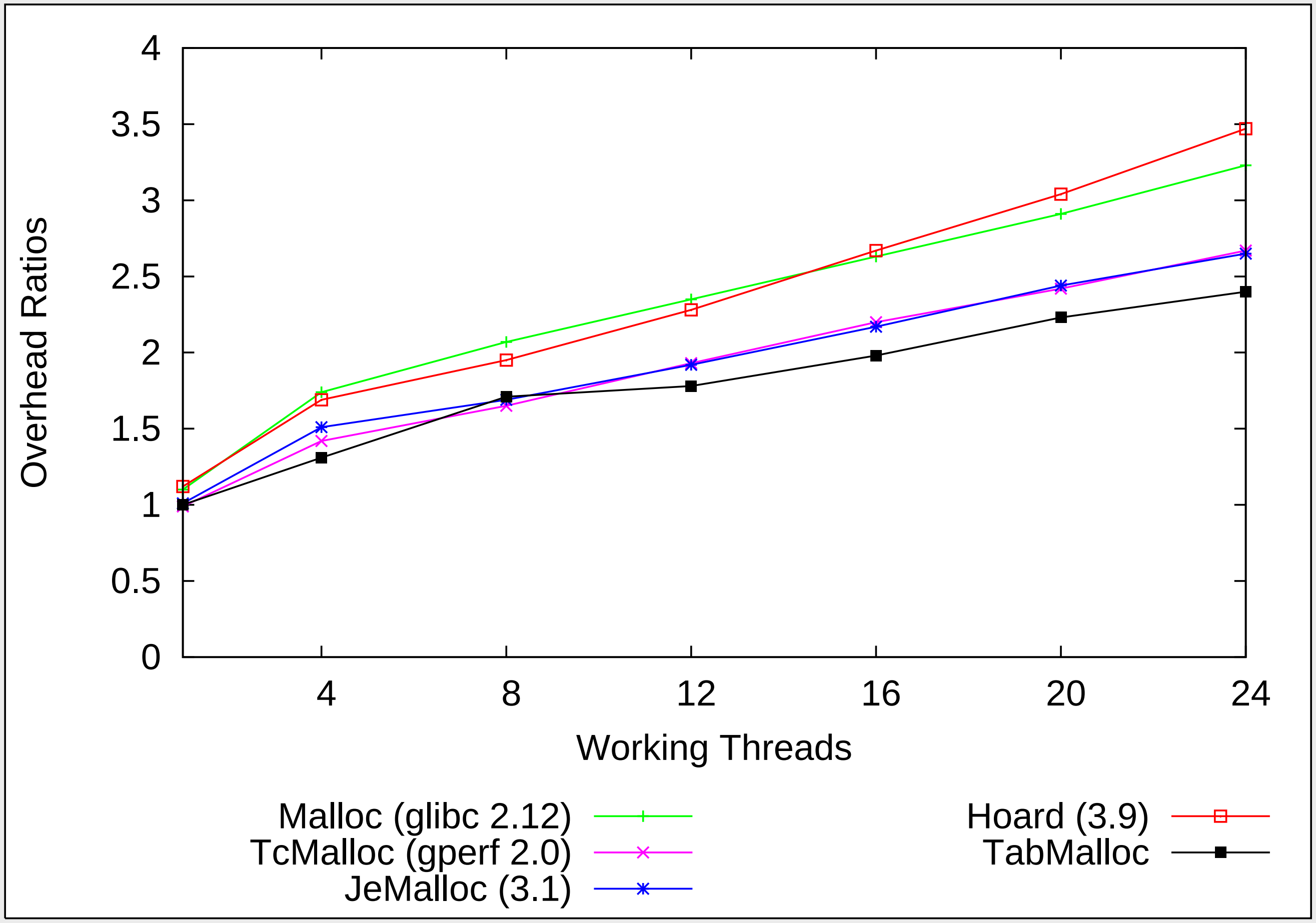
Experimental Results - First Runs (NS Design)

- Comparison between UMAs (using LD_PRELOAD).



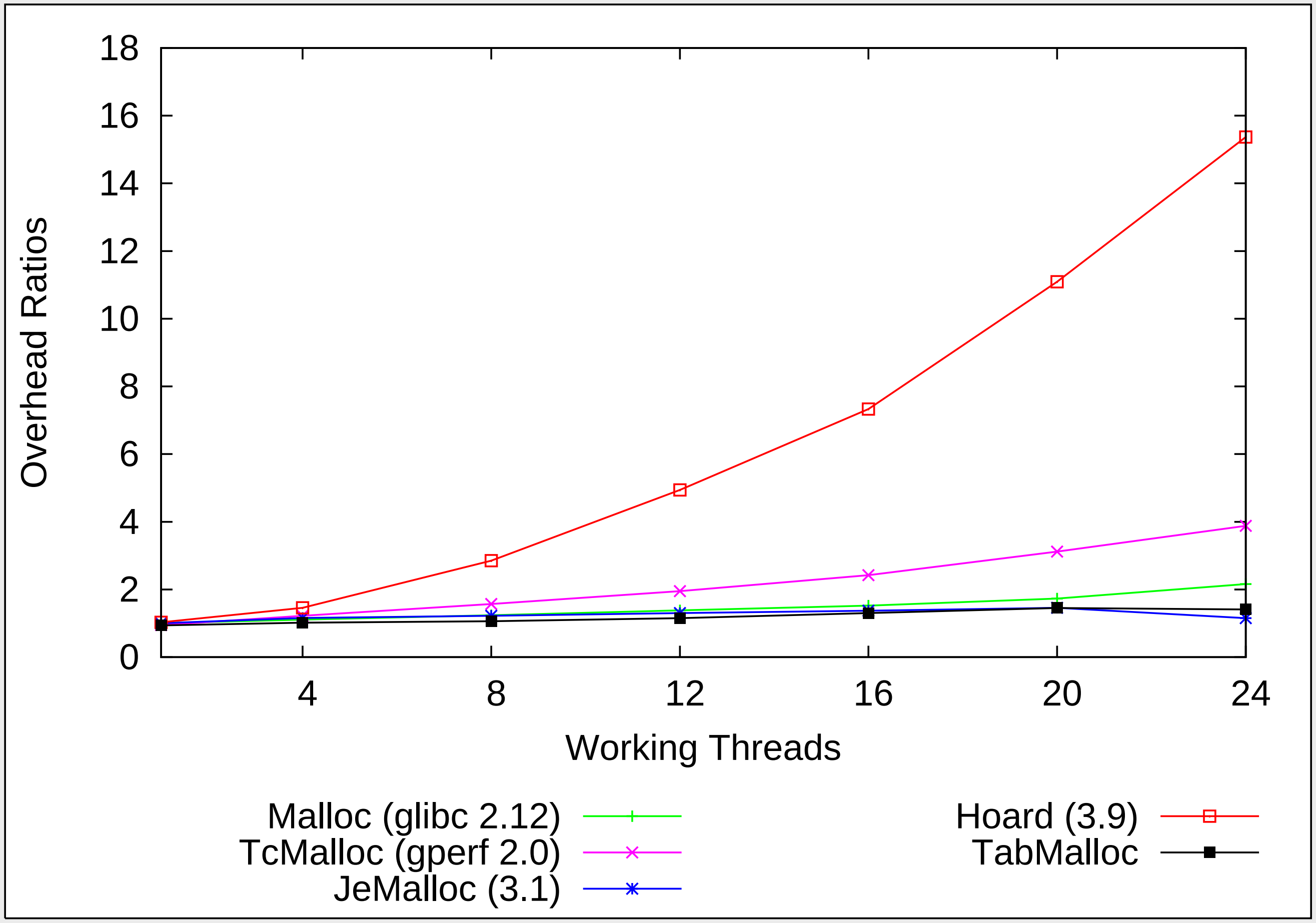
Experimental Results - First Runs (FS Design)

- Comparison between UMAs (using LD_PRELOAD).



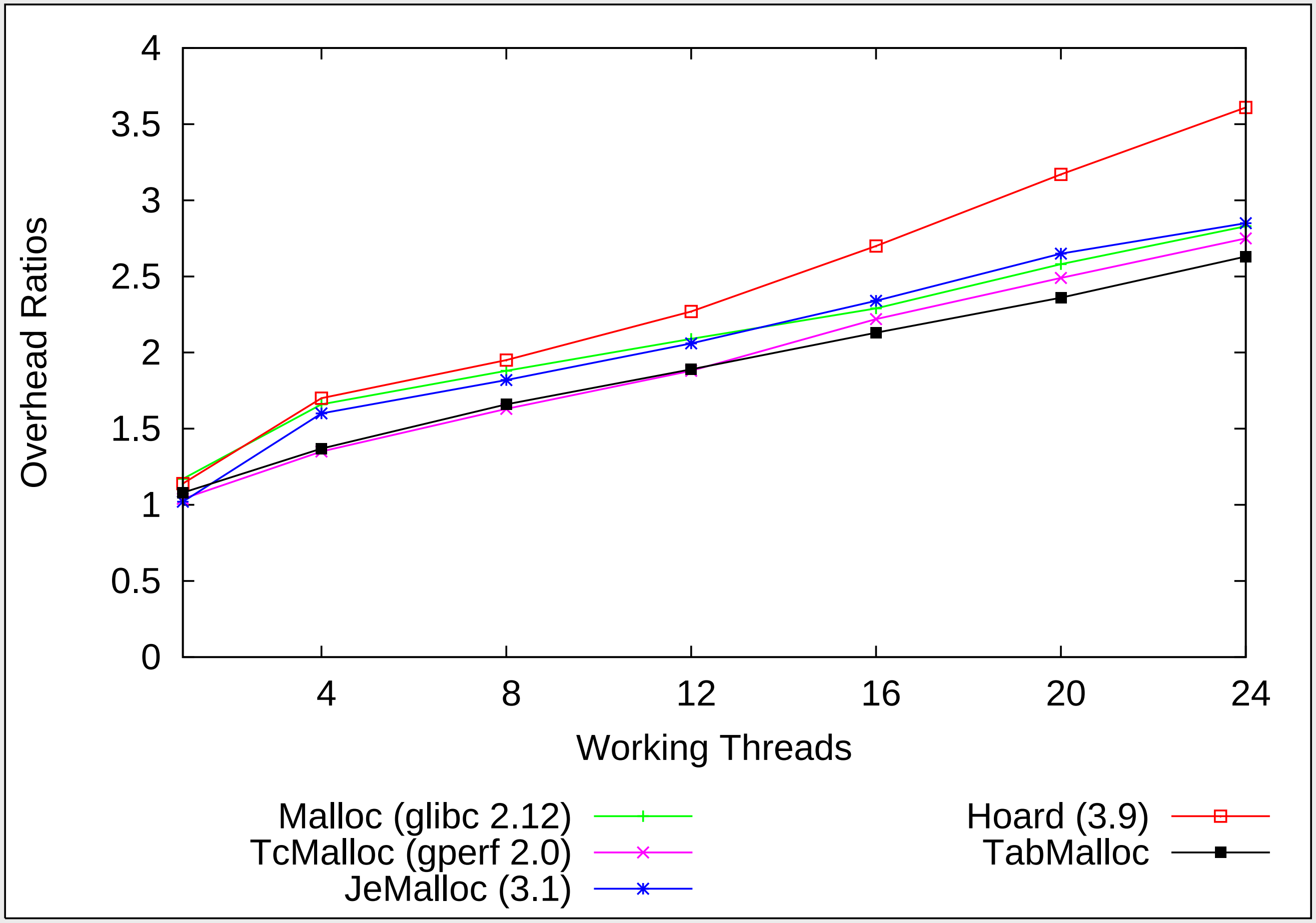
Experimental Results - Second Runs (NS Design)

- Comparison between UMAs (using LD_PRELOAD).



Experimental Results - Second Runs (FS Design)

- Comparison between UMAs (using LD_PRELOAD).



Conclusions

- We have presented a **novel**, **efficient** and **scalable** memory allocator for Multithreaded tabled evaluations of logic programs:
  - Has a **page based mechanism**, where data structures of the same type are allocated within a page.
  - **Splits memory** among specific data structures and different threads, by using a different Global and Local Heaps.

Conclusions

- We have presented a **novel**, **efficient** and **scalable** memory allocator for Multi-threaded tabled evaluations of logic programs:
  - Has a **page based mechanism**, where data structures of the same type are allocated within a page.
  - **Splits memory** among specific data structures and different threads, by using a different Global and Local Heaps.

- Experimental results show a **good performance** in the running time, when compared with other state-of-the-art UMAs.

- Further work will include following features:
  - Studying even further the alternative memory allocators.
  - Support for **wait-free synchronization**.

Thank You !!!

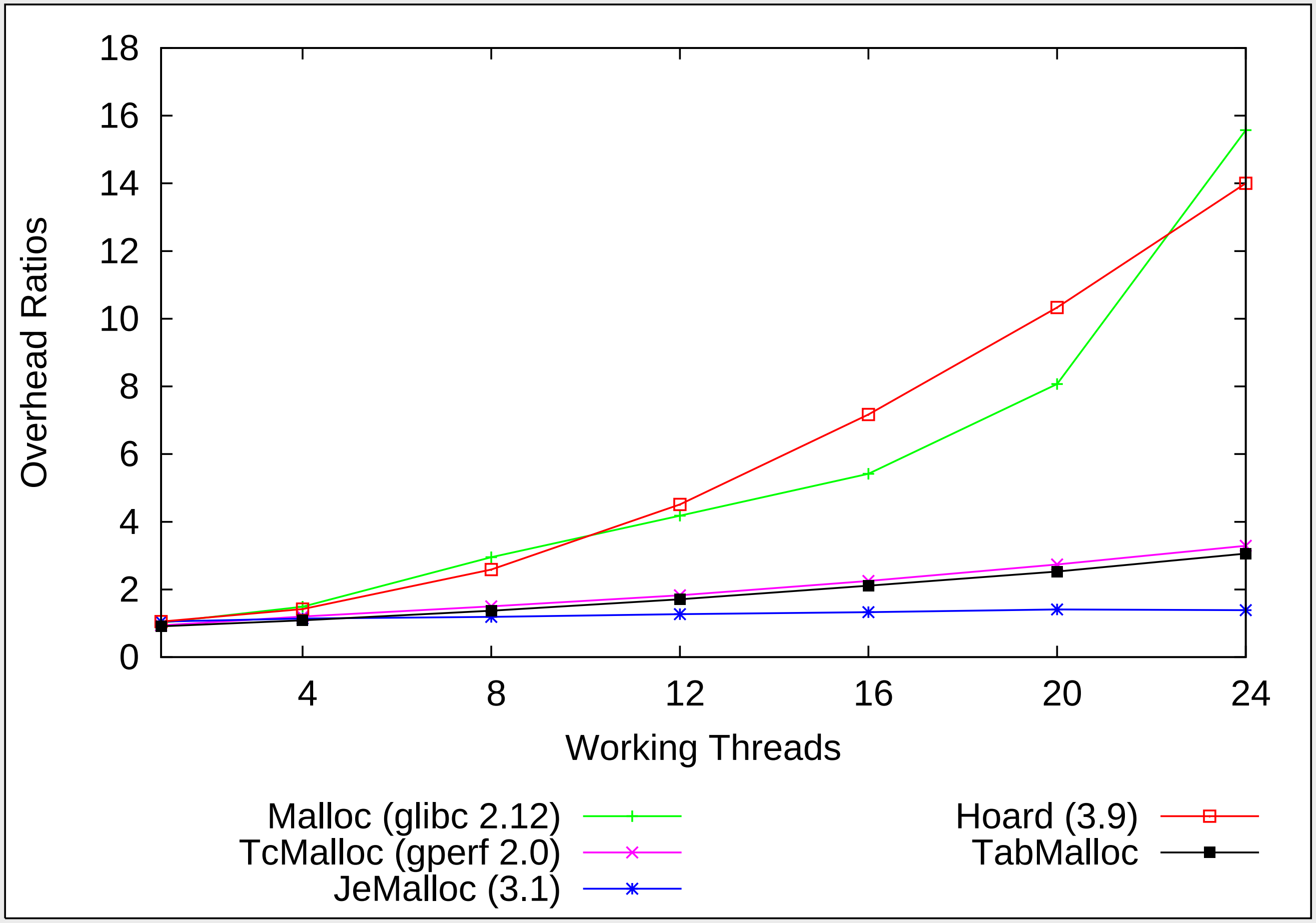
Miguel Areias and Ricardo Rocha
CRACS & INESC-TEC LA
University of Porto, Portugal
miguel-areias@dcc.fc.up.pt *ricroc@dcc.fc.up.pt*

Yap Prolog : *http://www.dcc.fc.up.pt/~vsc/Yap*

Projects LEAP and HORUS: *http://cracs.fc.up.pt/*

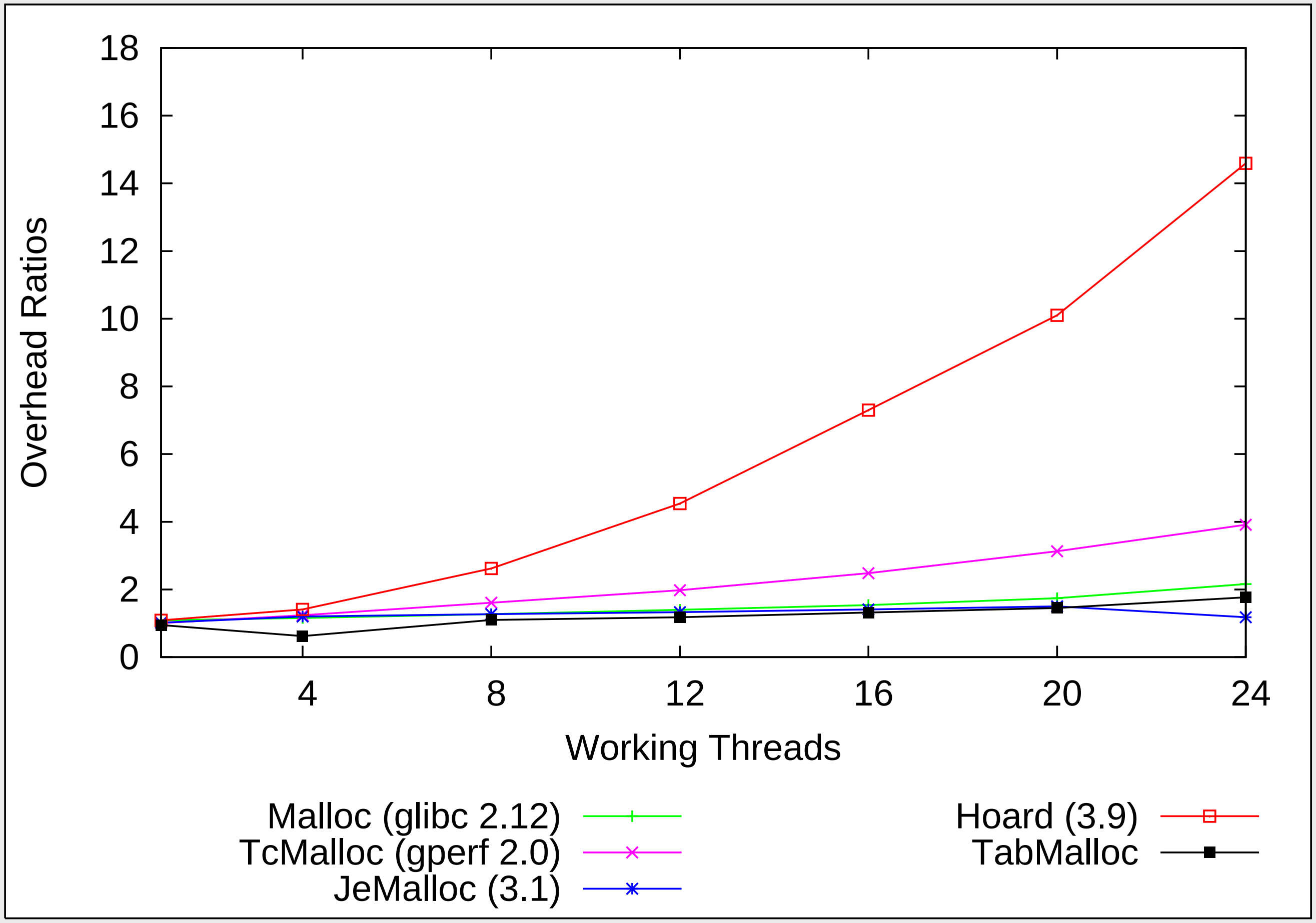
Experimental Results - First Runs (SS Design)

- Comparison between UMAs (using LD_PRELOAD).

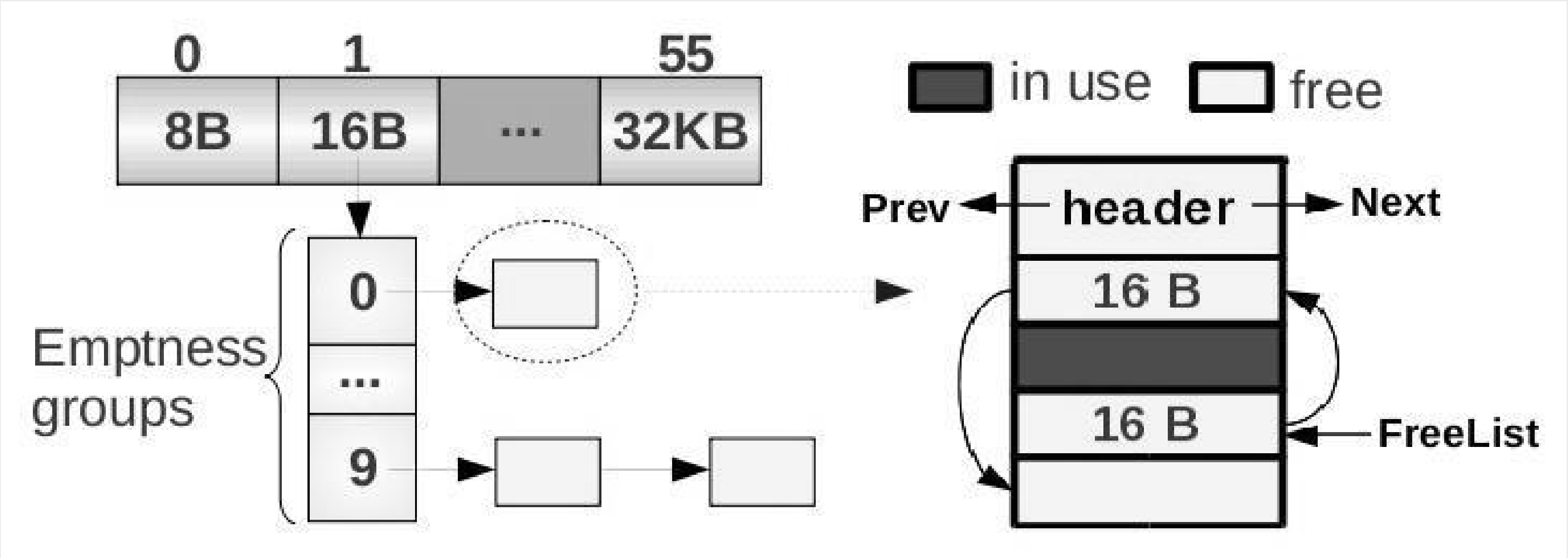


Experimental Results - Second Runs (SS Design)

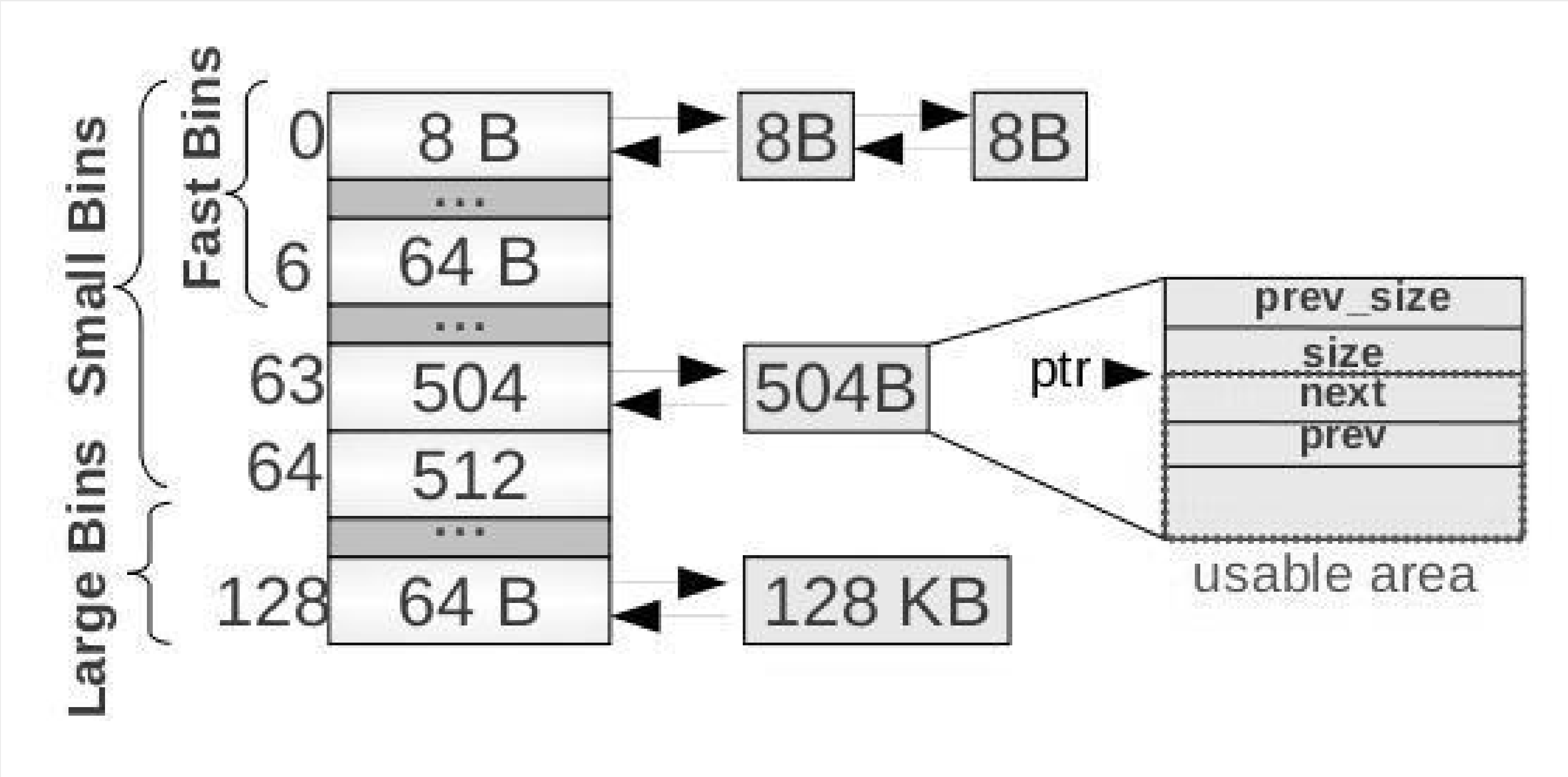
- Comparison between UMAs (using LD_PRELOAD).



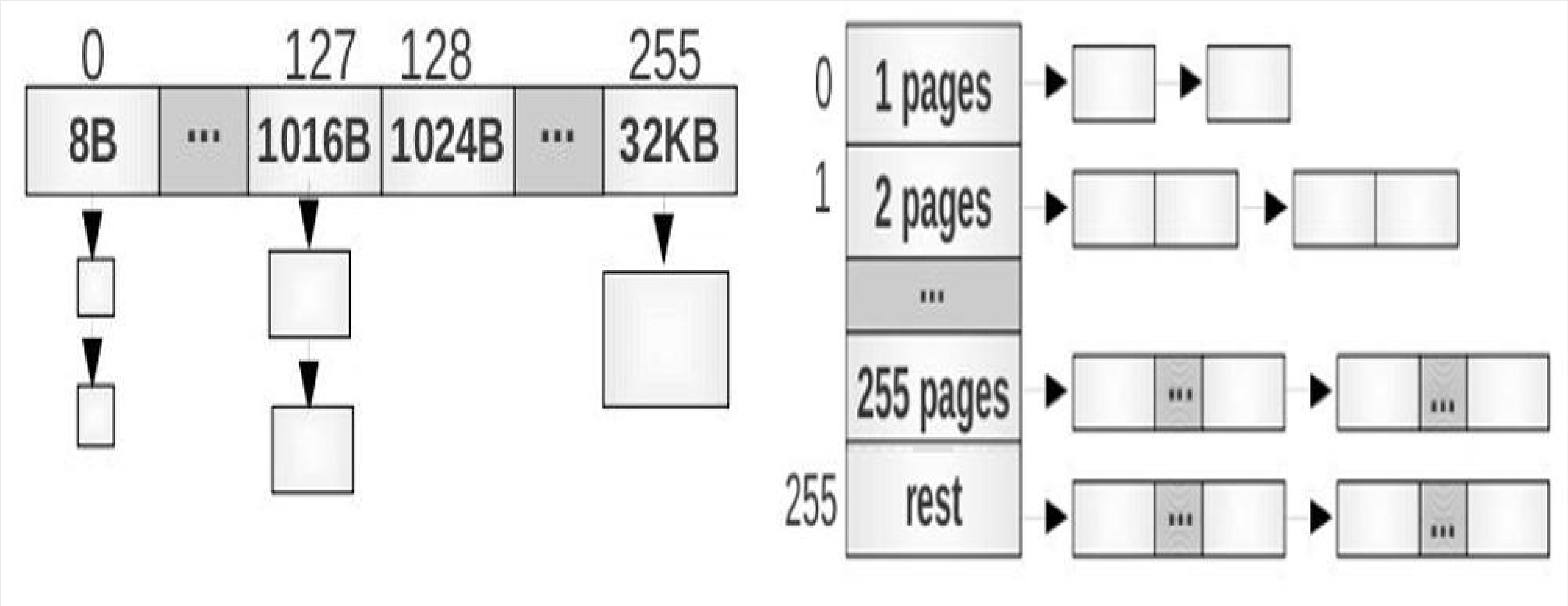
UMA - Hoard


0
1
55
8B
16B
...
32KB
in use
free
Prev
header
Next
0
...
9
Emptness
groups
16 B
16 B
FreeList

UMA - PtMalloc



UMA - TcMalloc



UMA - JeMalloc