Software Persistent Memory

Jorge Guerra, *Leonardo Mármol*, Daniel Campello, Carlos Crespo, Raju Rangaswami, Jinpeng Wei
{jguerra, lmarm001, dcamp020, ccres008, raju, weijp}@cs.fiu.edu

School of Computing and Information Sciences
College of Engineering and Computing
Florida International University



Persistence is ubiquitous*

Metadata persistence in systems software

- Layout management (file systems, others)
- Journaling, Versioning, Snapshoting, CDP

Persistence is ubiquitous*

Metadata persistence in systems software

- Layout management (file systems, others)
- Journaling, Versioning, Snapshoting, CDP

Application data persistence

- End-user applications
- Databases
- Key-value stores

Persistence is ubiquitous*

Metadata persistence in systems software

- Layout management (file systems, others)
- Journaling, Versioning, Snapshoting, CDP

Application data persistence

- End-user applications
- Databases
- Key-value stores

Computational state persistence

- Scientific (HPC) applications
- “Big Data” applications

What is needed for persistence?

Required steps to persist data

- Tracking data dependencies

What is needed for persistence?

Required steps to persist data

- Tracking data dependencies
- Transferring data from memory to persistent media

What is needed for persistence?

Required steps to persist data

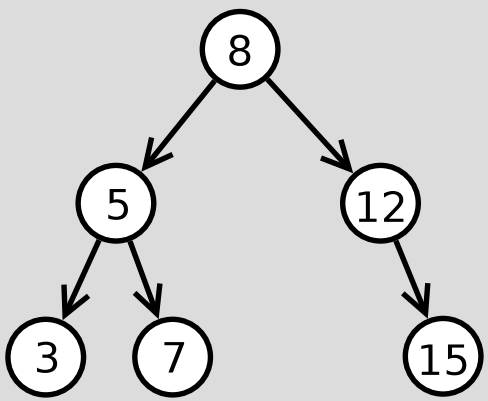
- Tracking data dependencies
- Transferring data from memory to persistent media
- Loading data back into memory

Challenges of persisting memory

Making a Balanced Binary Tree Persistent

Challenges of persisting memory

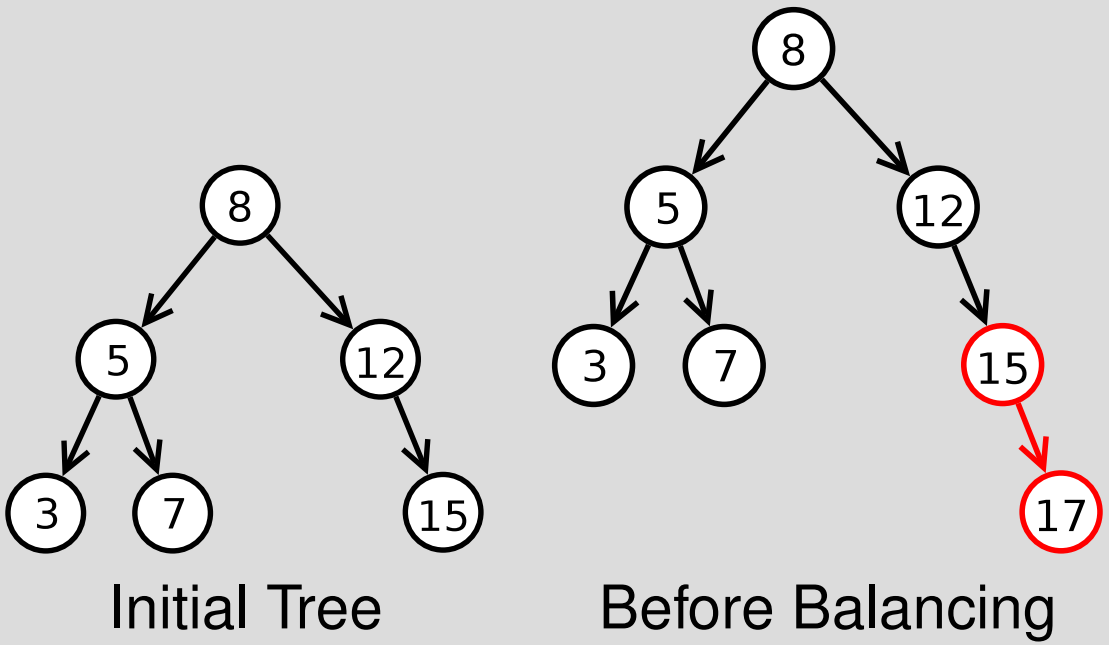
Making a Balanced Binary Tree Persistent



Initial Tree

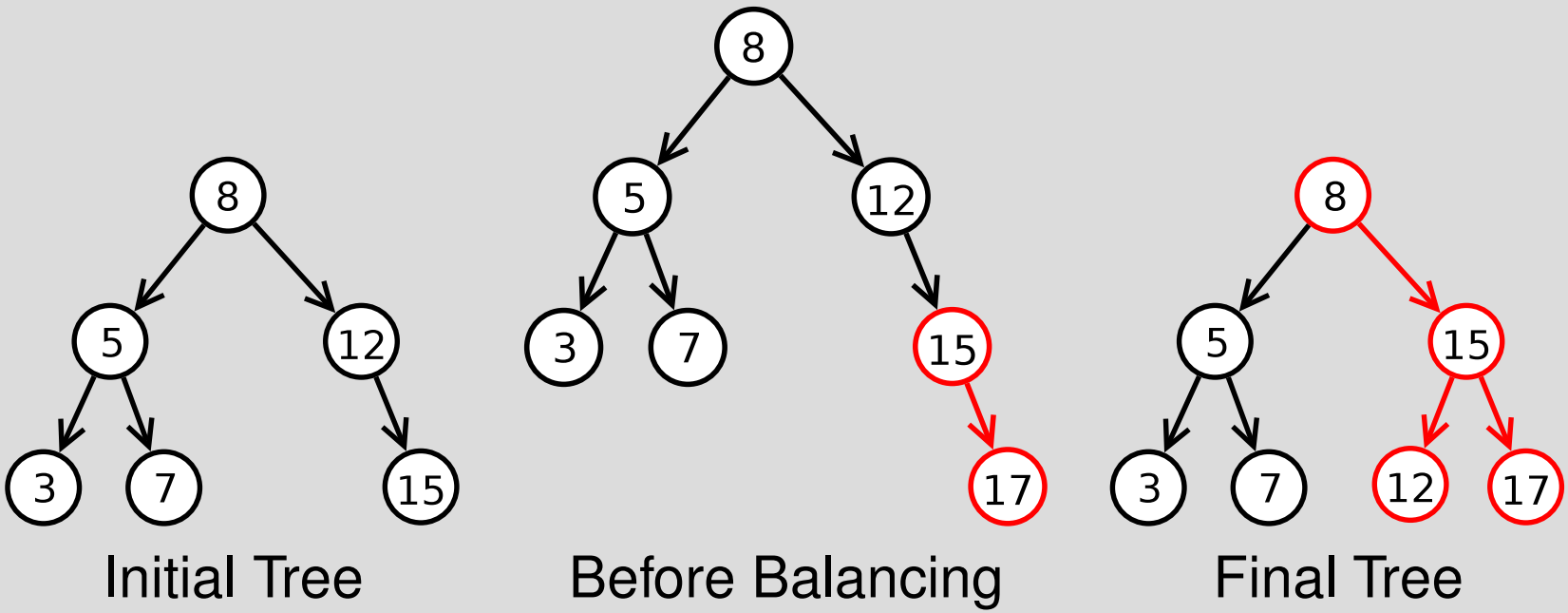
Challenges of persisting memory

Making a Balanced Binary Tree Persistent



Challenges of persisting memory

Making a Balanced Binary Tree Persistent



A possible solution

A Different Programming Model

- Automatically track data dependencies
  - ✓ Identify which memory regions must be made persistent

A possible solution

A Different Programming Model

- Automatically track data dependencies
  - ✓ Identify which memory regions must be made persistent
- Transfer data from memory to the persistent media
  - ✓ Transactional model

A possible solution

A Different Programming Model

- Automatically track data dependencies
  - ✓ Identify which memory regions must be made persistent
- Transfer data from memory to the persistent media
  - ✓ Transactional model
- Load data back into memory
  - ✓ Memory consistency

Introducing Software Persistent Memory (*SoftPM*)

SoftPM

- A type of application directed check-pointing
- Data to be check-pointed is automatically discovered
- Fast checkpoints and recovery
- Uses conventional volatile memory interfaces
- Works with existing technology

Outline

- Design
  - ✓ Container Abstractions
  - ✓ Automatic Data Discovery
  - ✓ Code Modification
  - ✓ Architecture Overview
- Evaluation
  - ✓ Development Complexity
  - ✓ Performance
- Summary
  - ✓ Related Work
  - ✓ Conclusions

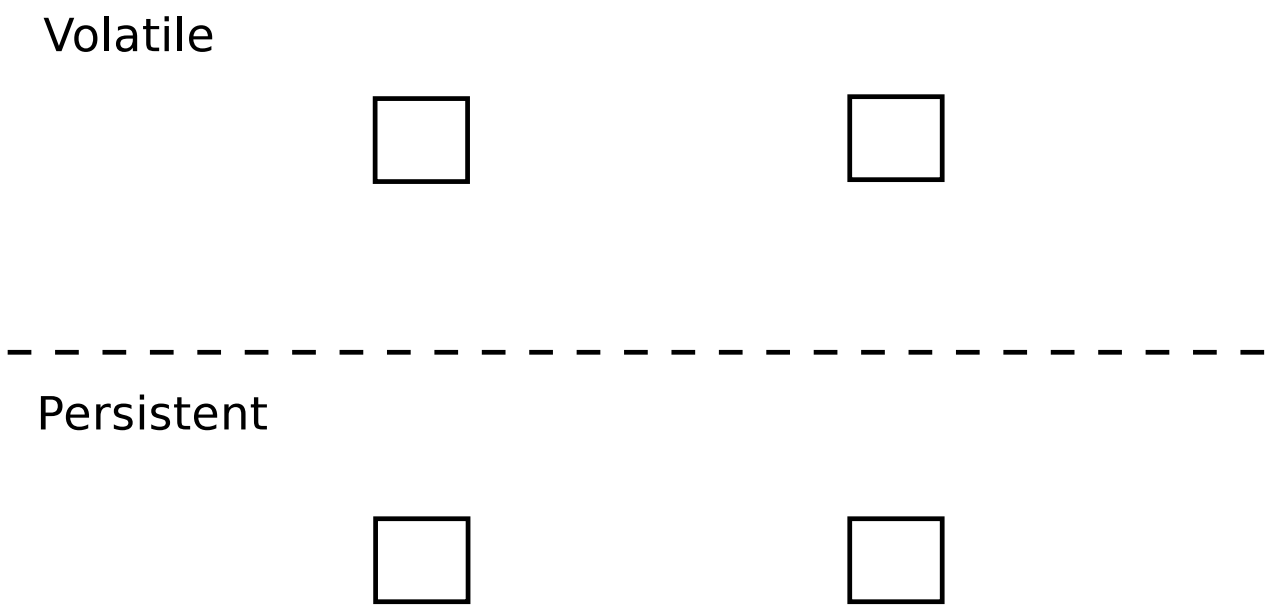
The Container abstraction

Simple container

- Self-describing unit of persistent memory
- One or more containers per application
- Applications specify a root structure
- All data reachable from the root automatically discovered
- Restoreable with a single operation

Automatically discovering data

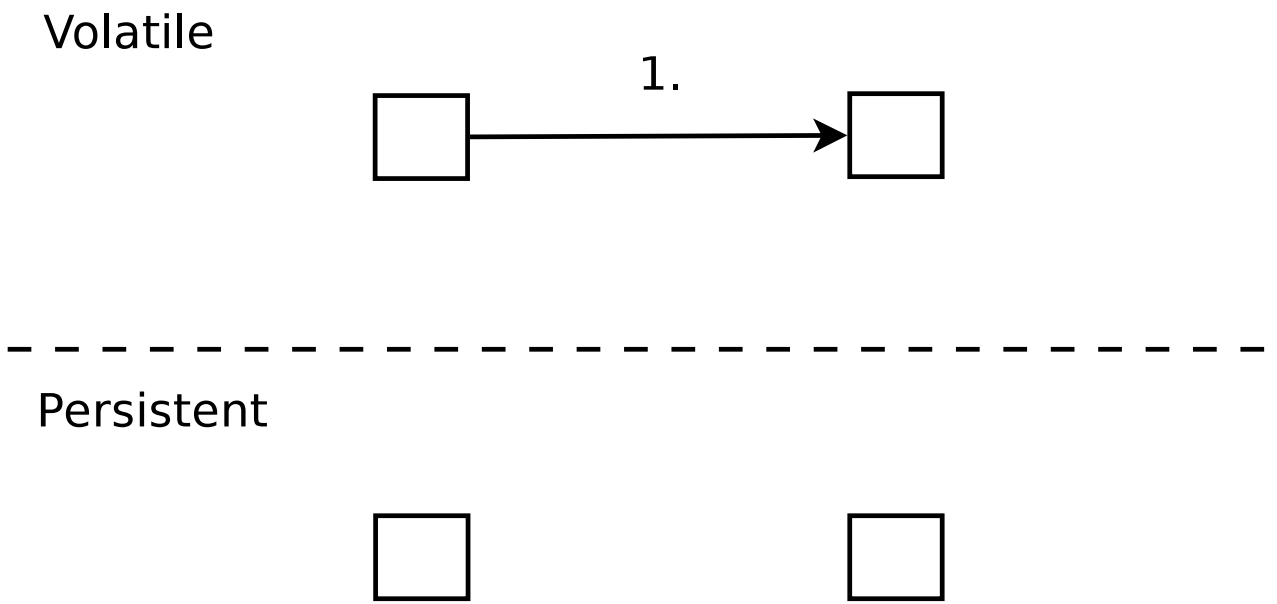
We divide memory into 2 regions: **volatile** and **persistent**.
Hence, there can be 4 types of pointers:



Automatically discovering data

We divide memory into 2 regions: **volatile** and **persistent**. Hence, there can be 4 types of pointers:

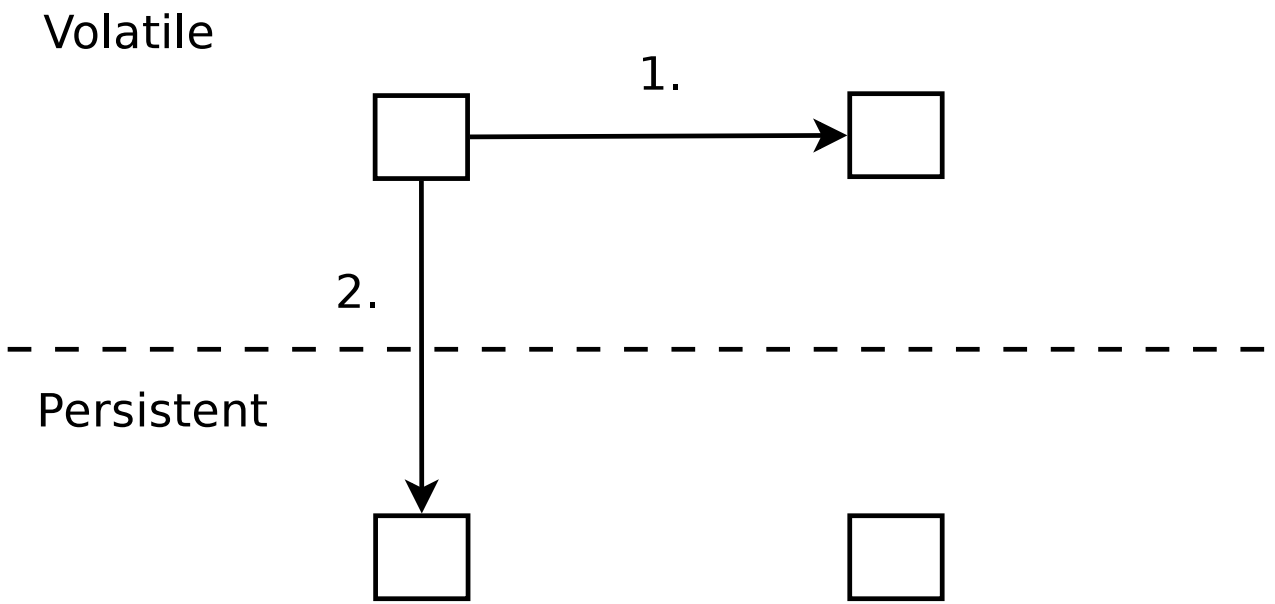
1. volatile $\rightarrow$ volatile



Automatically discovering data

We divide memory into 2 regions: **volatile** and **persistent**. Hence, there can be 4 types of pointers:

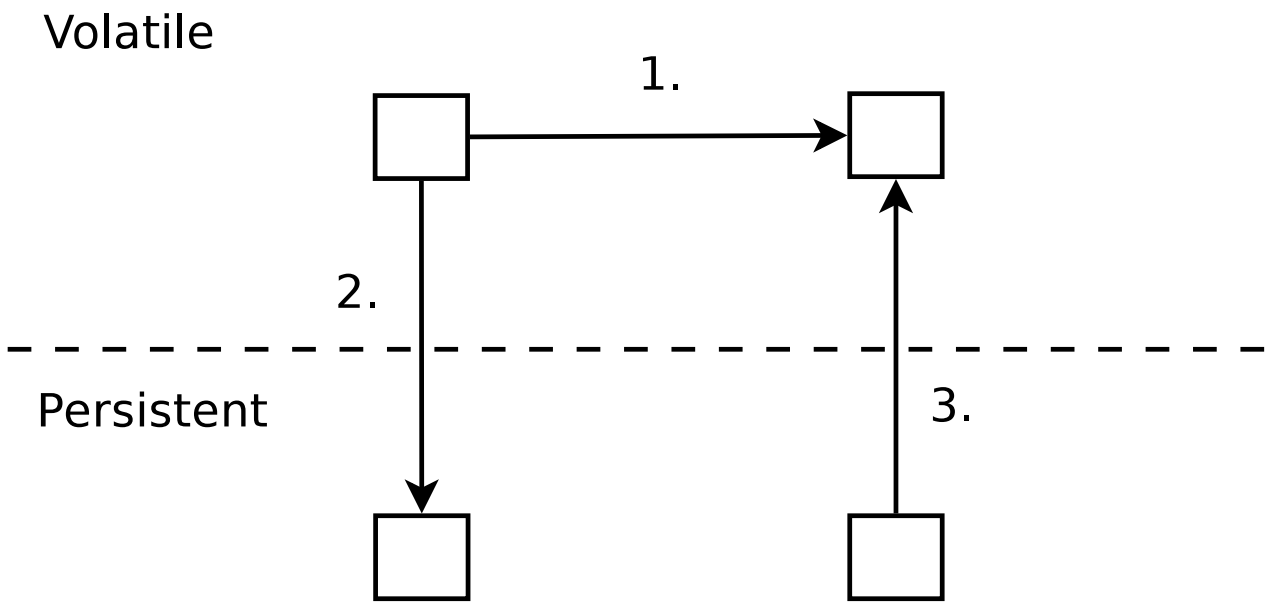
1. volatile $\rightarrow$ volatile
2. volatile $\rightarrow$ persistent



Automatically discovering data

We divide memory into 2 regions: **volatile** and **persistent**. Hence, there can be 4 types of pointers:

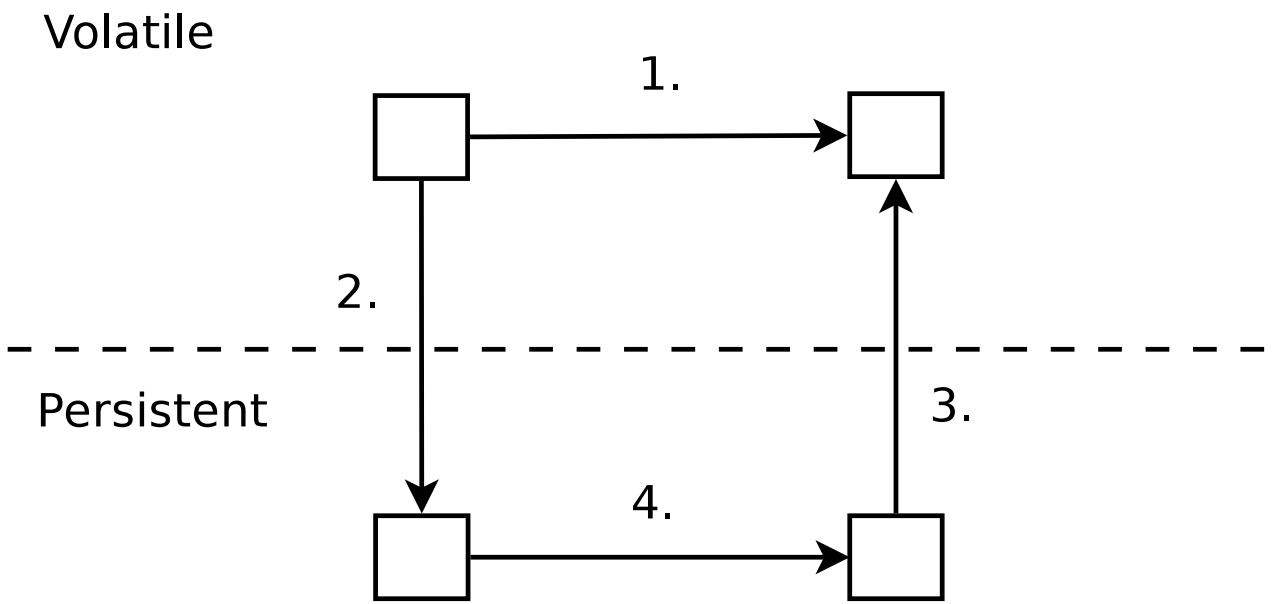
1. volatile $\rightarrow$ volatile
2. volatile $\rightarrow$ persistent
3. persistent $\rightarrow$ volatile



Automatically discovering data

We divide memory into 2 regions: **volatile** and **persistent**.
Hence, there can be 4 types of pointers:

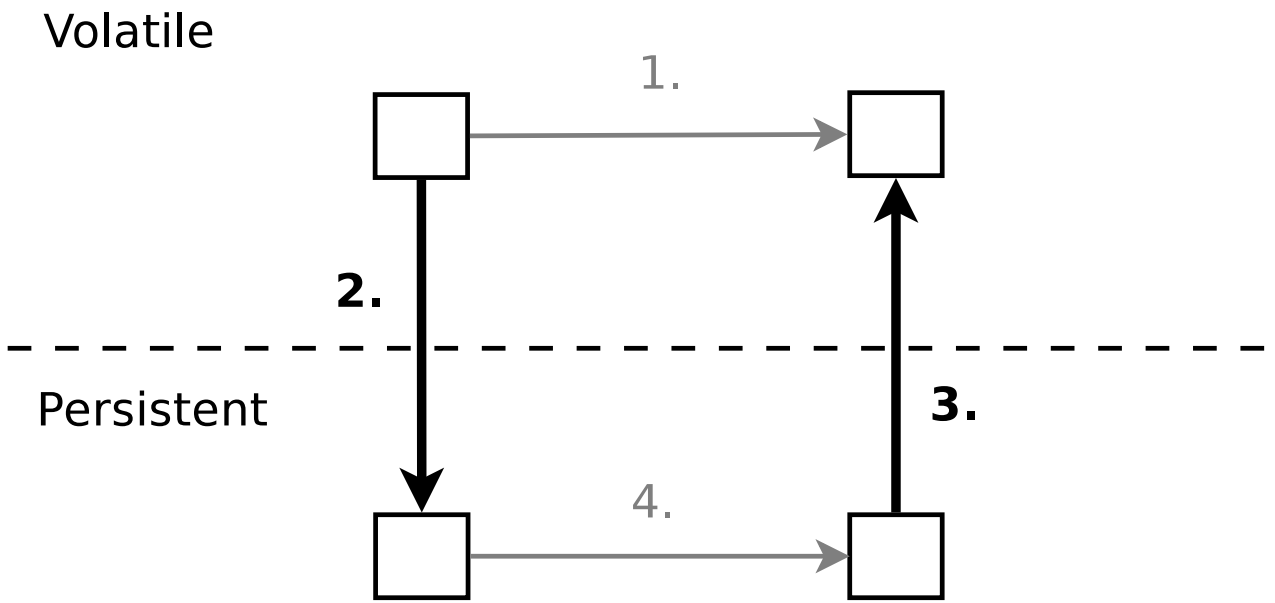
1. volatile → volatile
2. volatile → persistent
3. persistent → volatile
4. persistent → persistent



Automatically discovering data

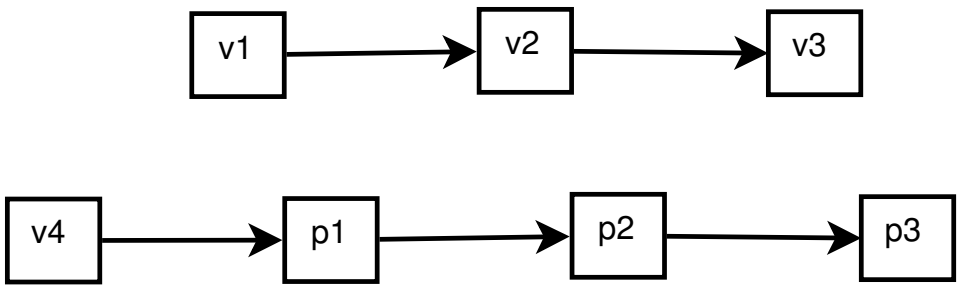
We divide memory into 2 regions: **volatile** and **persistent**.
Hence, there can be 4 types of pointers:

1. volatile → volatile
2. volatile → persistent ✓
3. persistent → volatile ✓
4. persistent → persistent



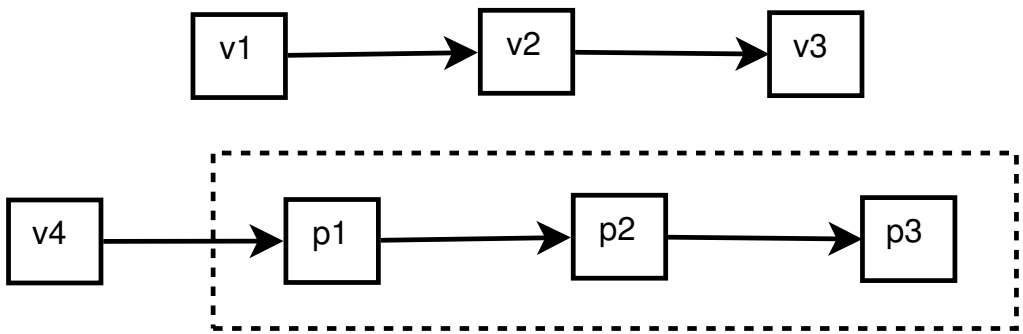
Automatically discovering data

Initially all memory is volatile except a *root* structure.



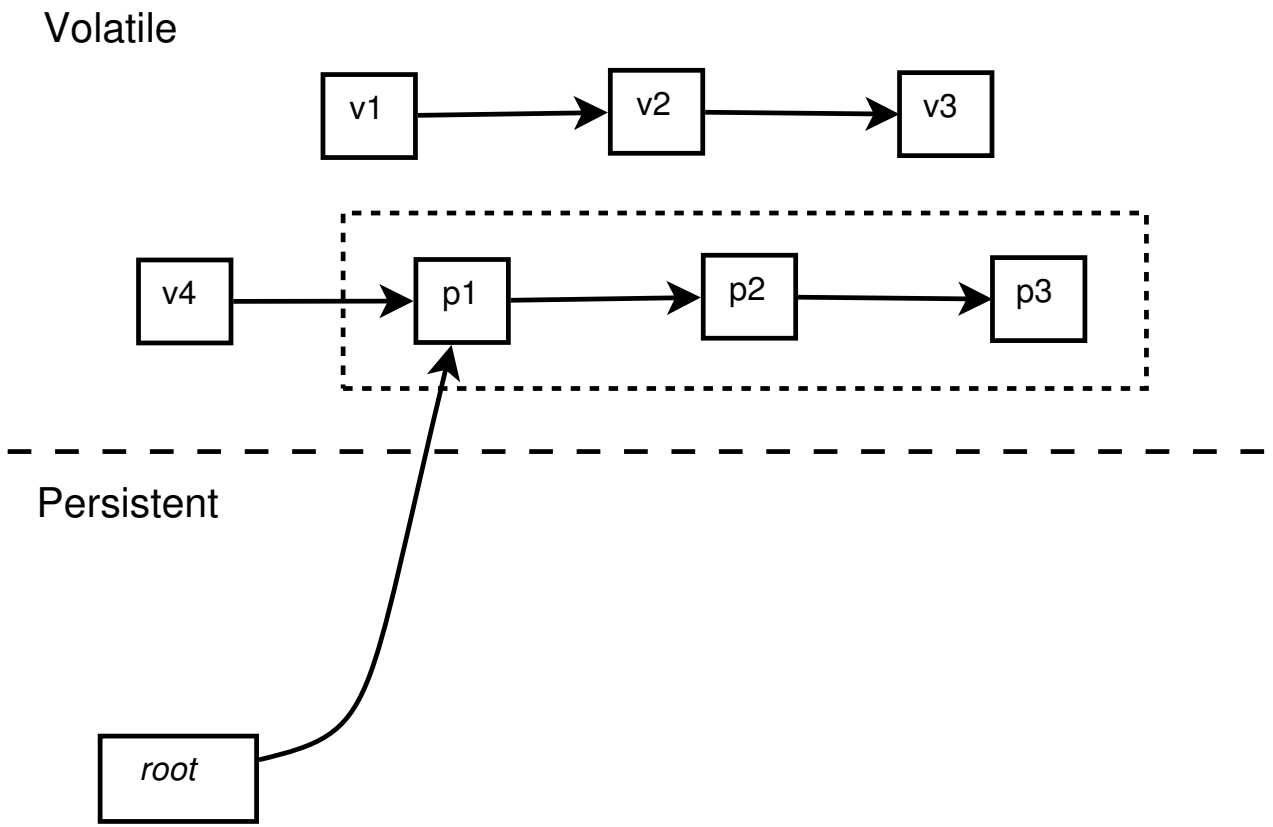
Automatically discovering data

Initially all memory is volatile except a *root* structure.



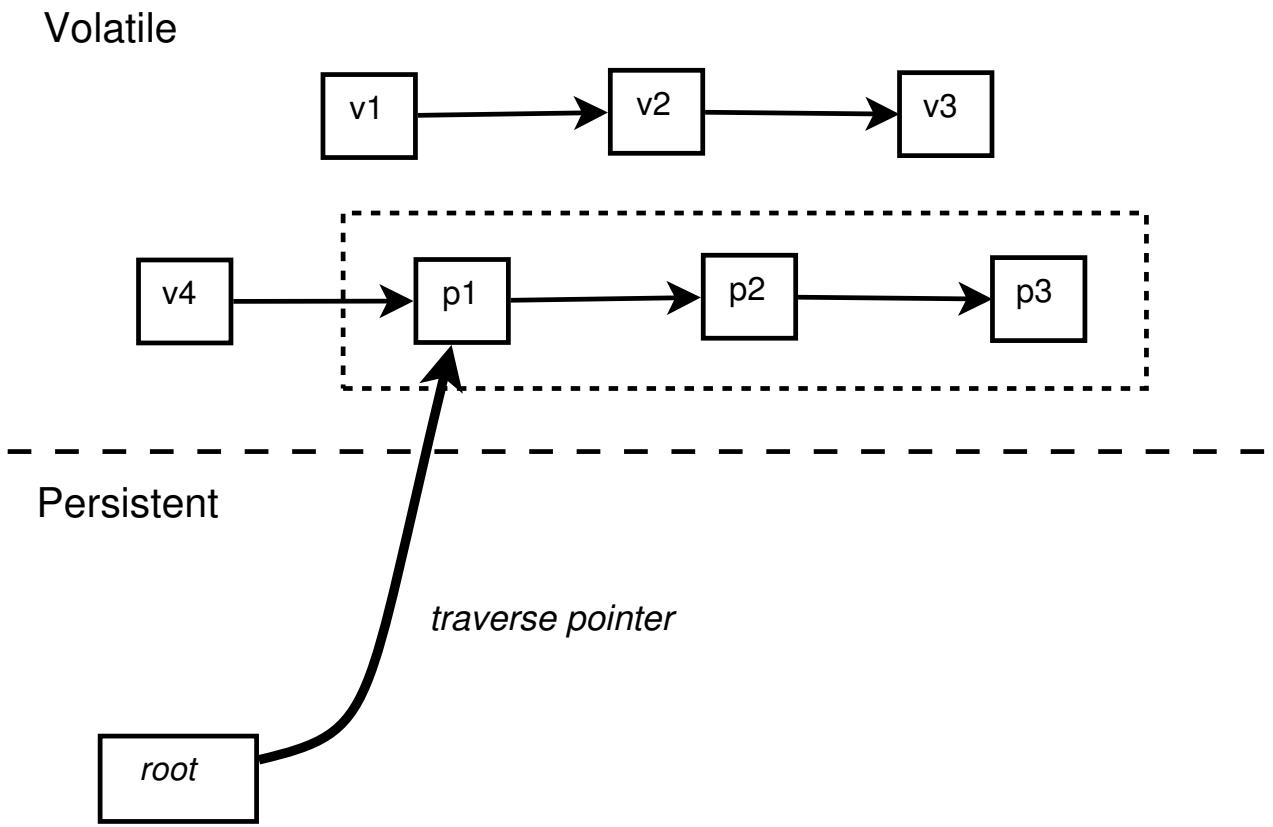
Automatically discovering data

Initially all memory is volatile except a *root* structure.



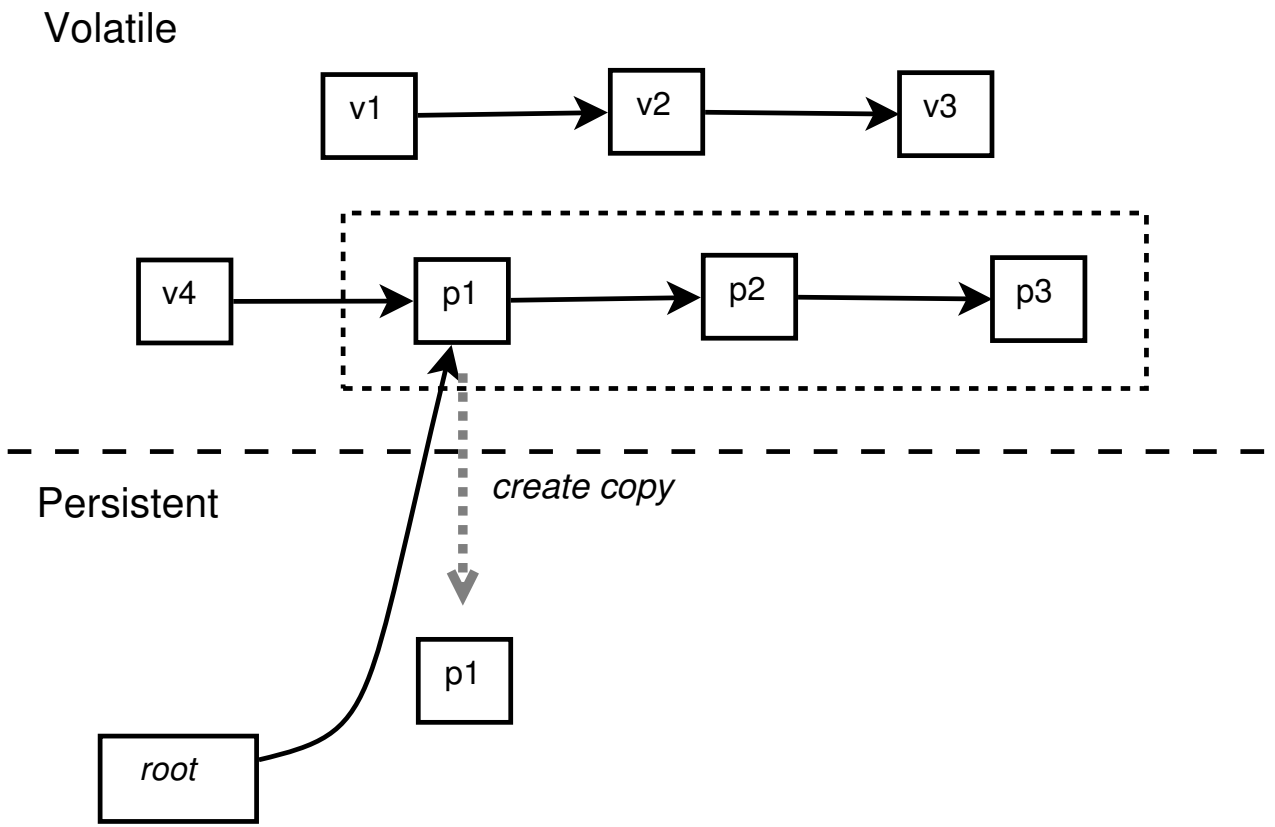
Automatically discovering data

Initially all memory is volatile except a *root* structure.



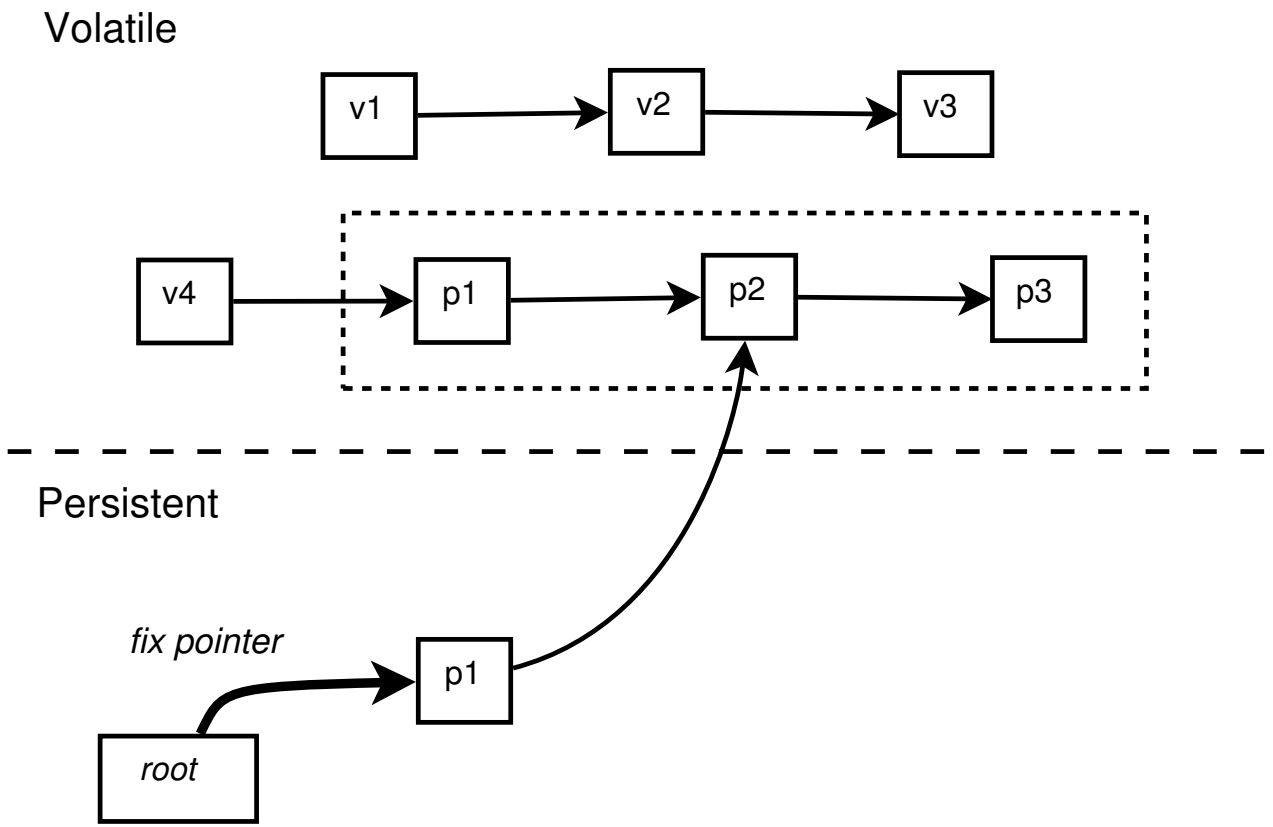
Automatically discovering data

Initially all memory is volatile except a *root* structure.



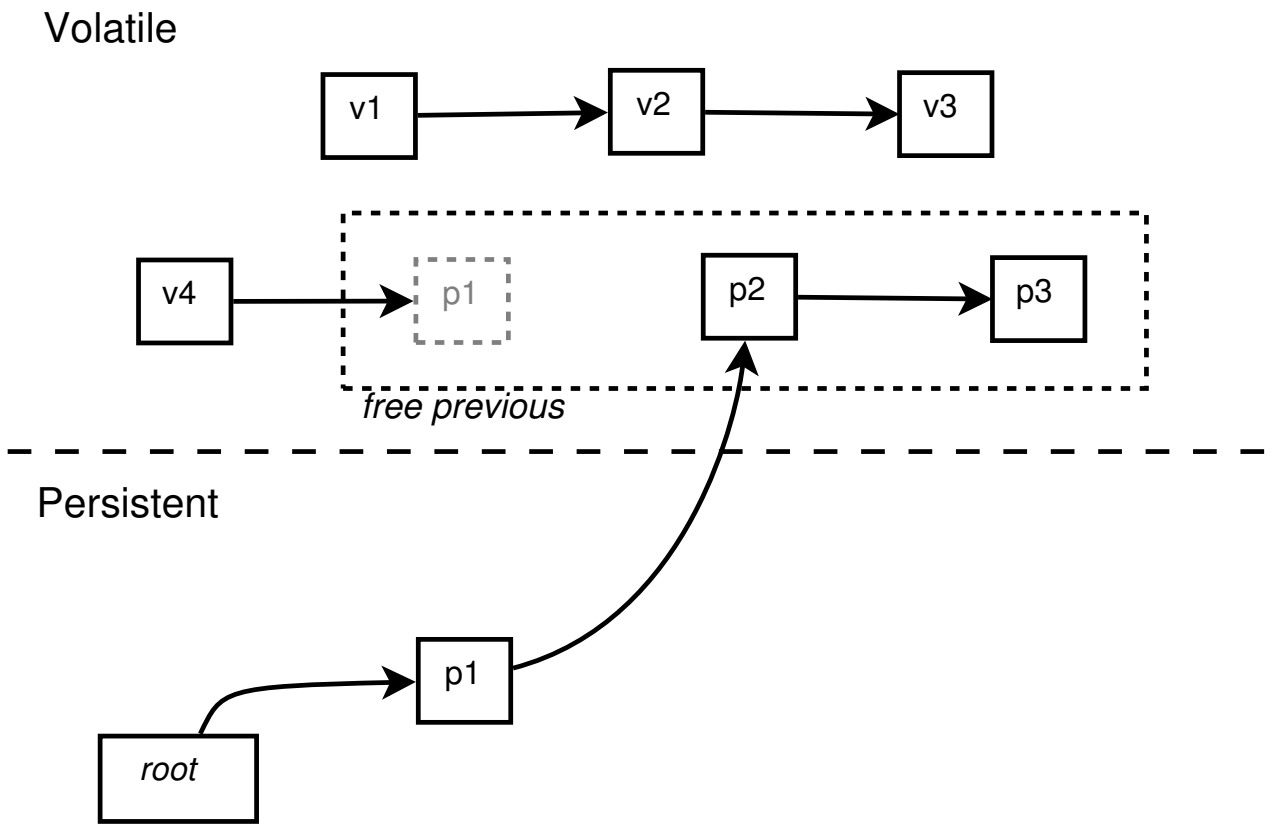
Automatically discovering data

Initially all memory is volatile except a *root* structure.



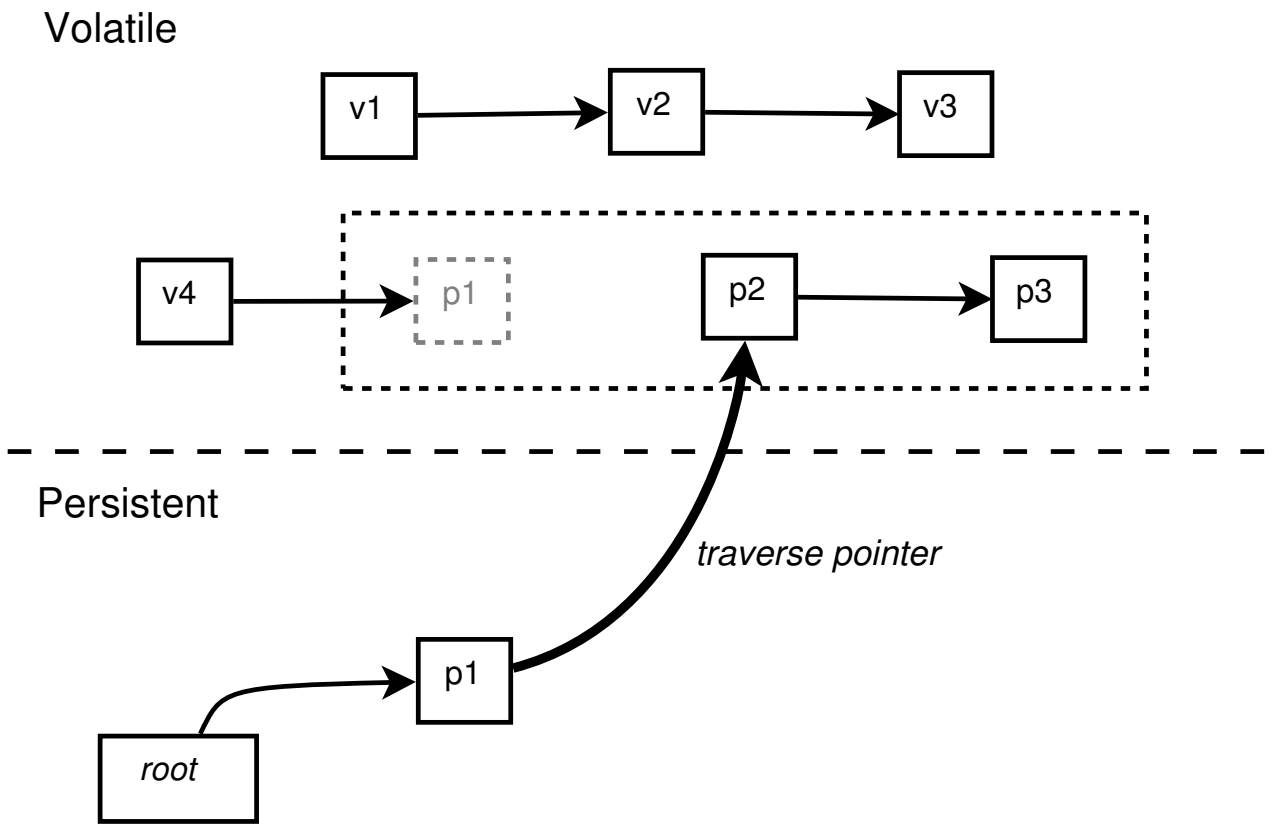
Automatically discovering data

Initially all memory is volatile except a *root* structure.



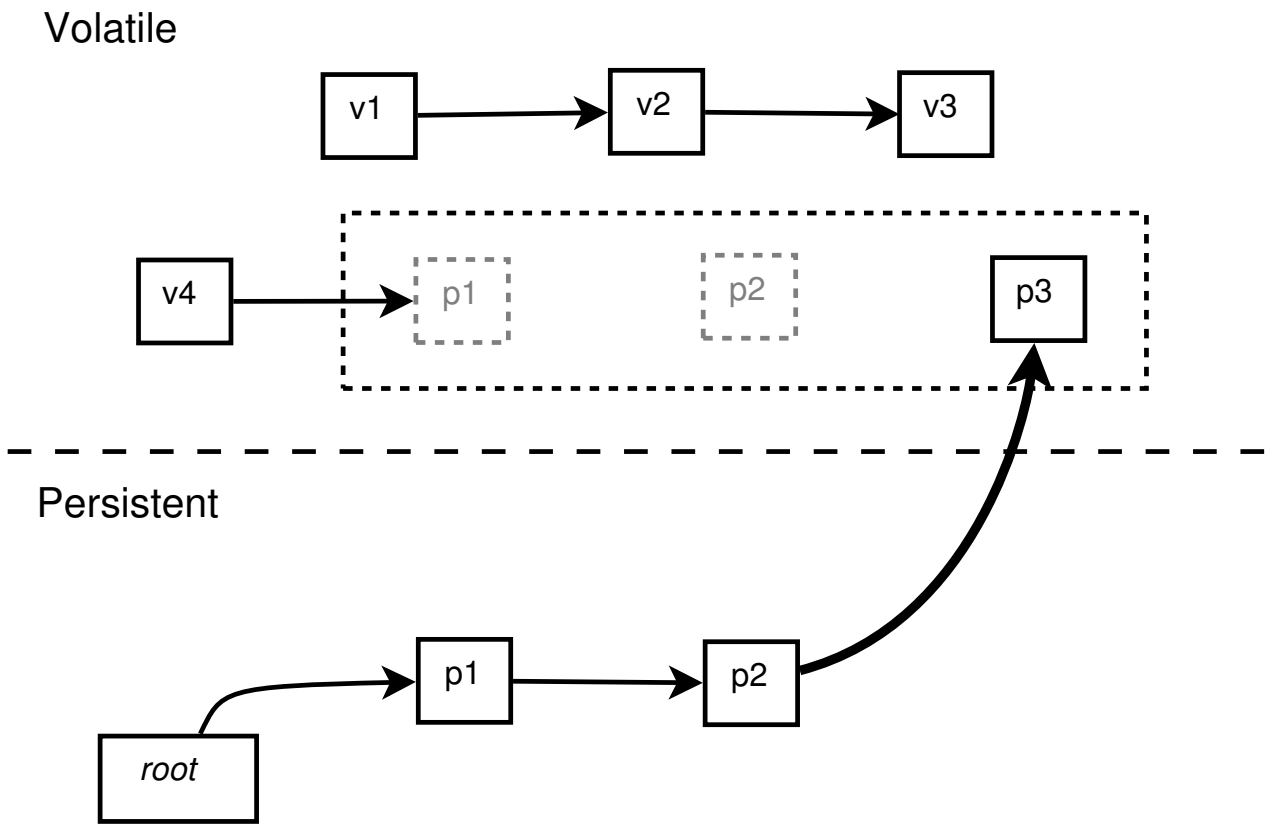
Automatically discovering data

Initially all memory is volatile except a *root* structure.



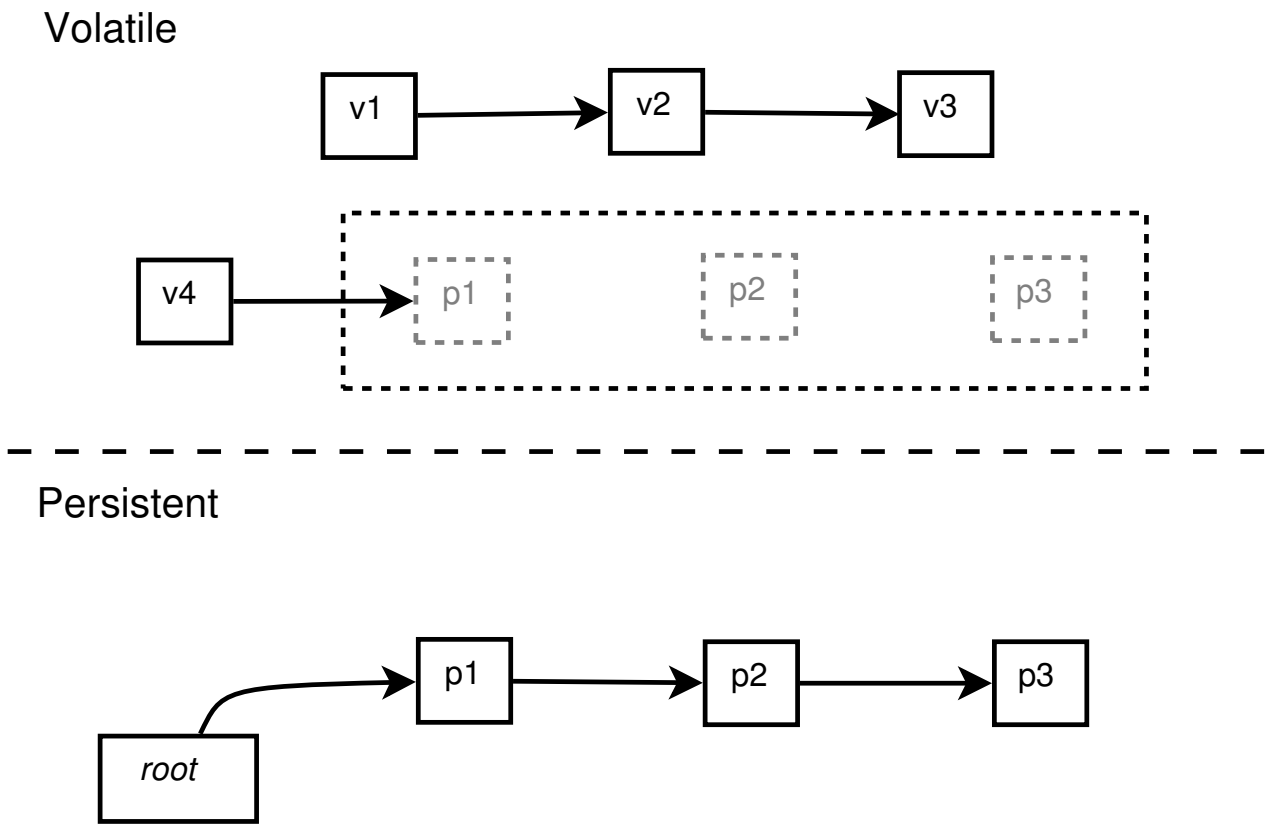
Automatically discovering data

Initially all memory is volatile except a *root* structure.



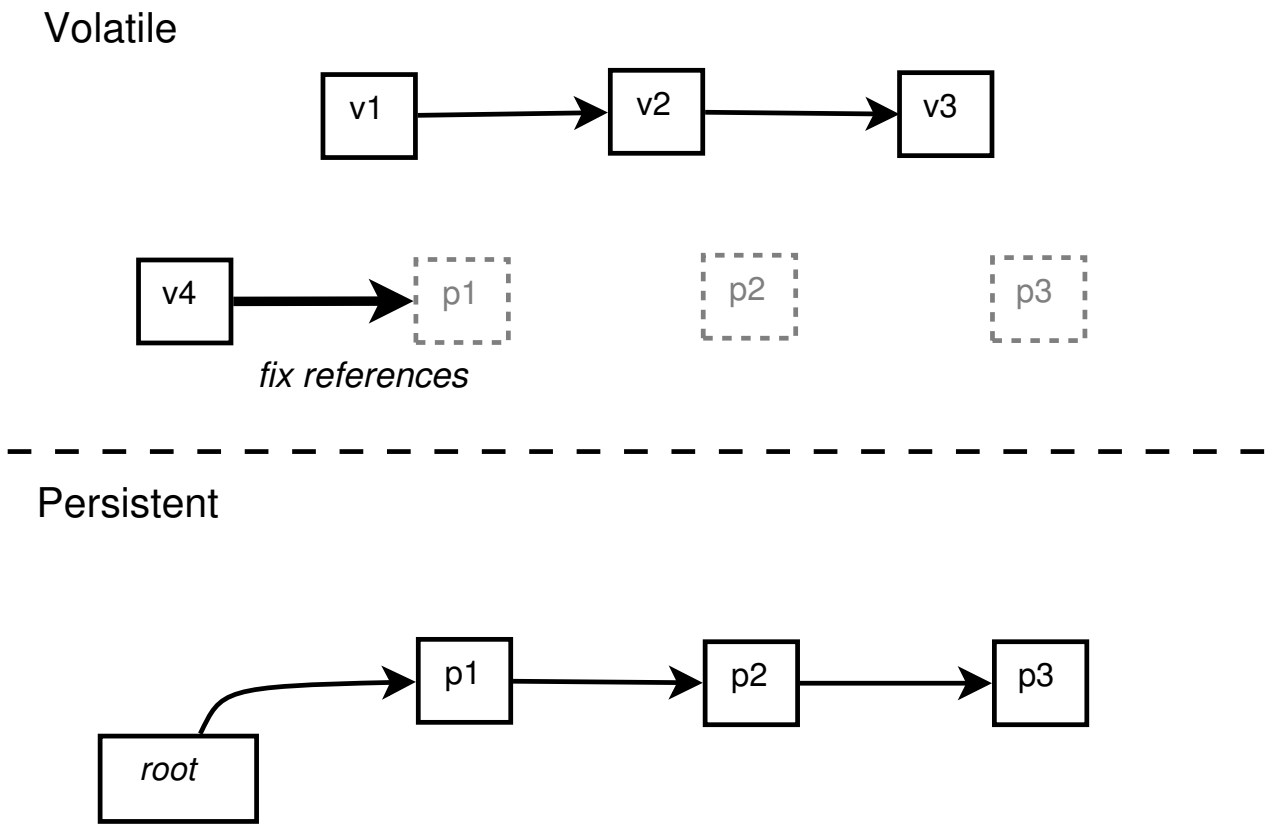
Automatically discovering data

Initially all memory is volatile except a *root* structure.



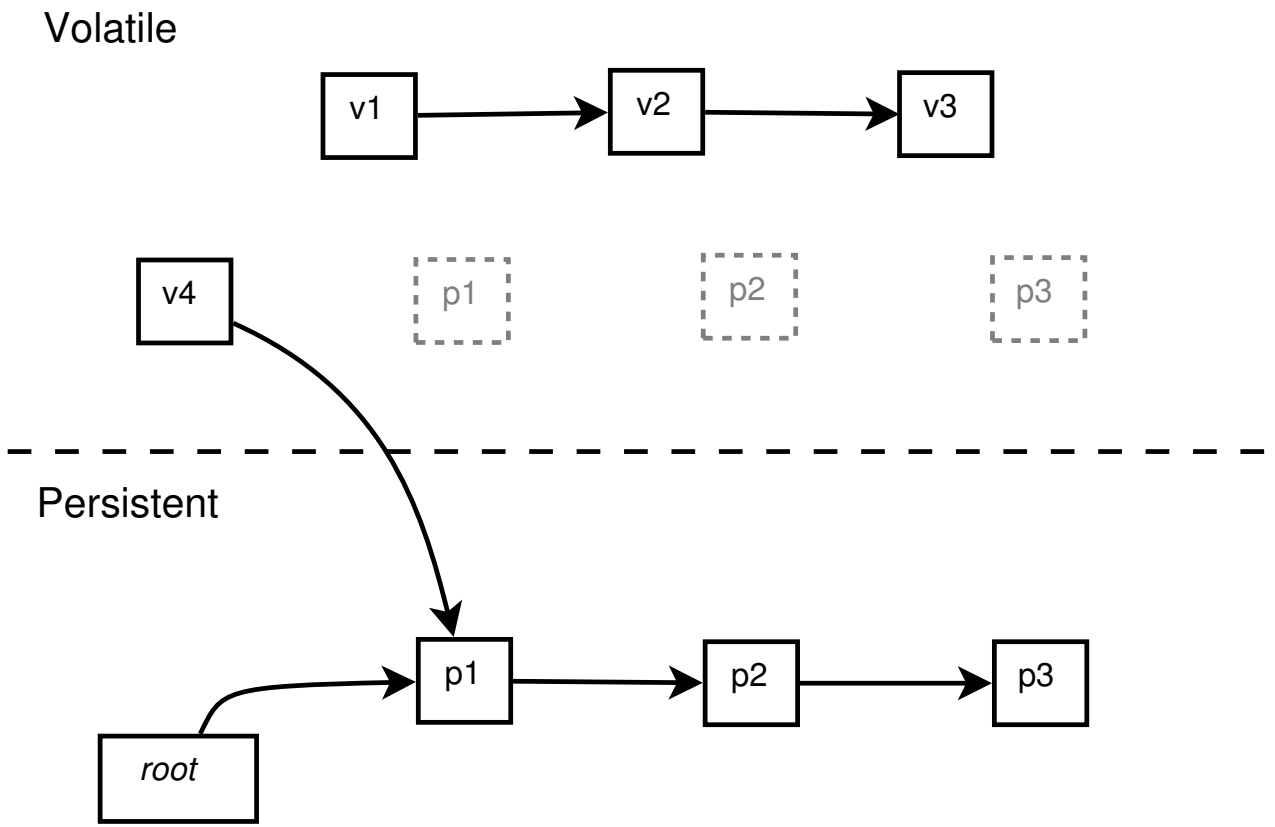
Automatically discovering data

Initially all memory is volatile except a *root* structure.



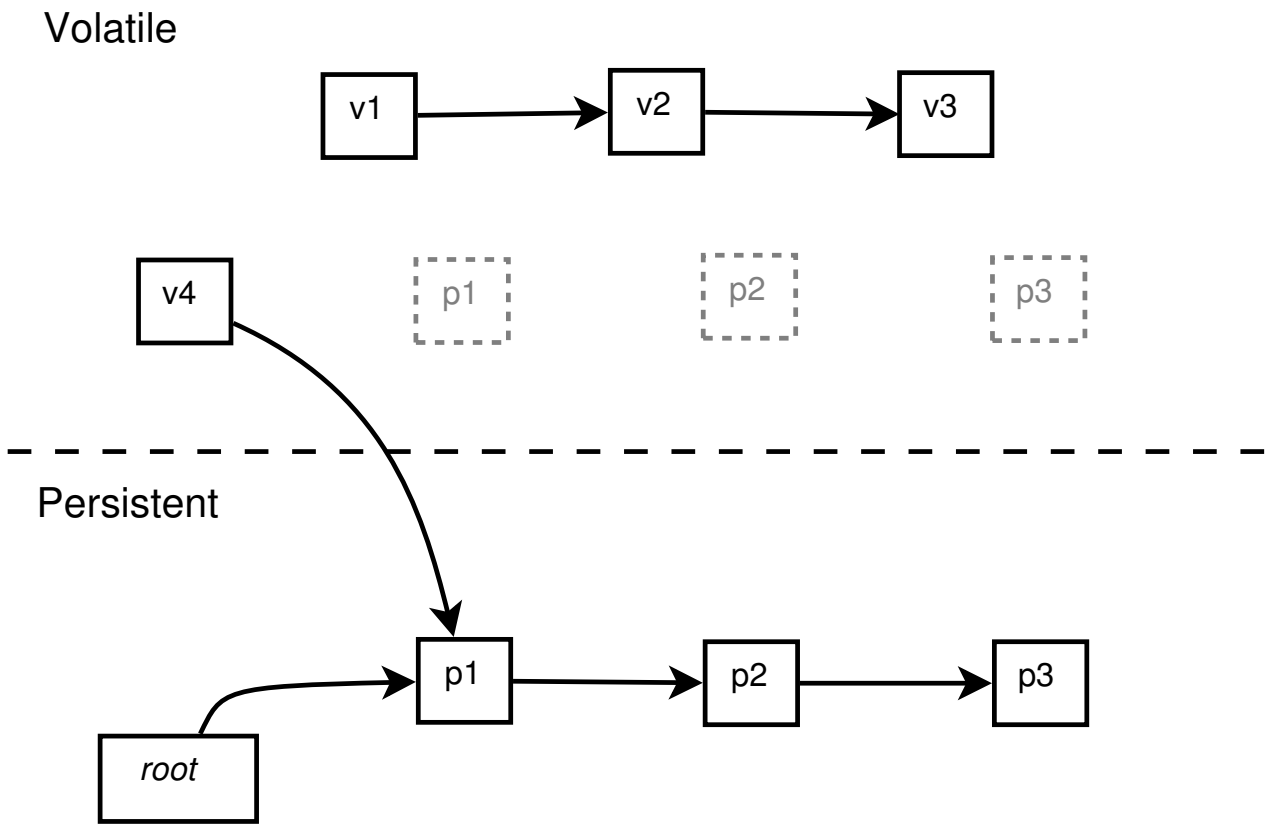
Automatically discovering data

Initially all memory is volatile except a *root* structure.



Automatically discovering data

Initially all memory is volatile except a *root* structure.



This root structure and it's closure form the **container**.

Creating a persistence point

```
typedef struct l {
  int v;
  struct l * next;
} list;



foo() {

  list *n1 = malloc(...);

  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;

  :
  :
}
```

Creating a persistence point

```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {

  list *n1 = malloc(...);

  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;


  :
}
```

Creating a persistence point

```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);

  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;


  :
}
```

Creating a persistence point

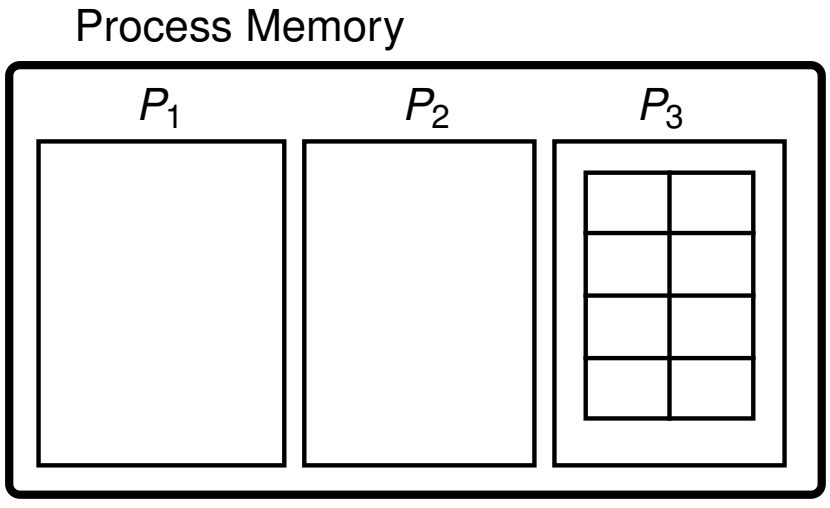
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;


  :
}
```

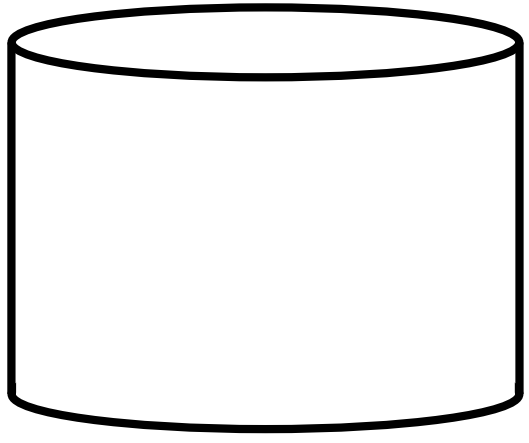
Creating a persistence point

```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  :
}
```

Creating a persistence point

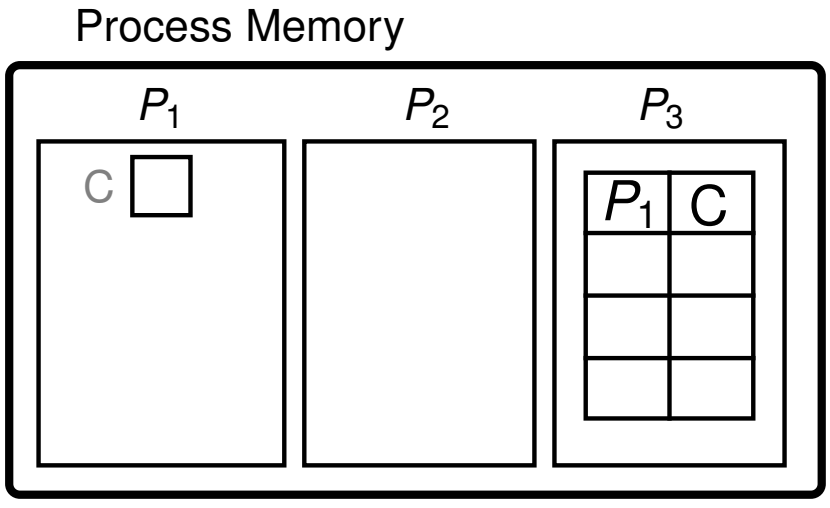
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  :
}
```

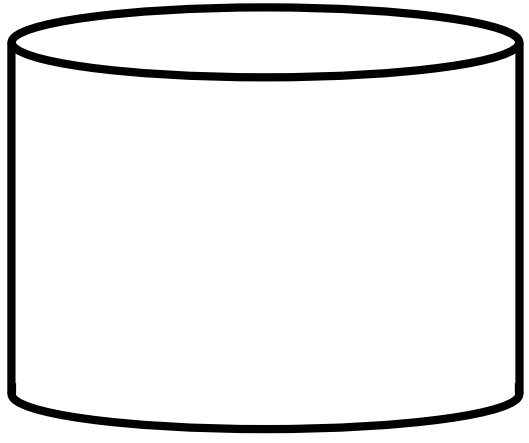




Creating a persistence point

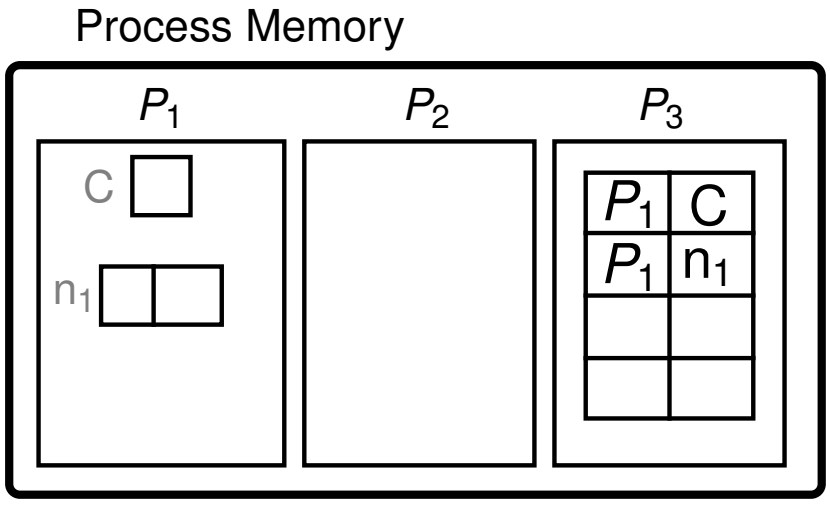
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  ⋮
}
```

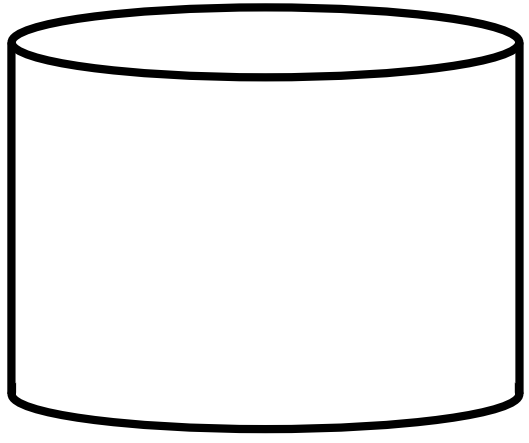




Creating a persistence point

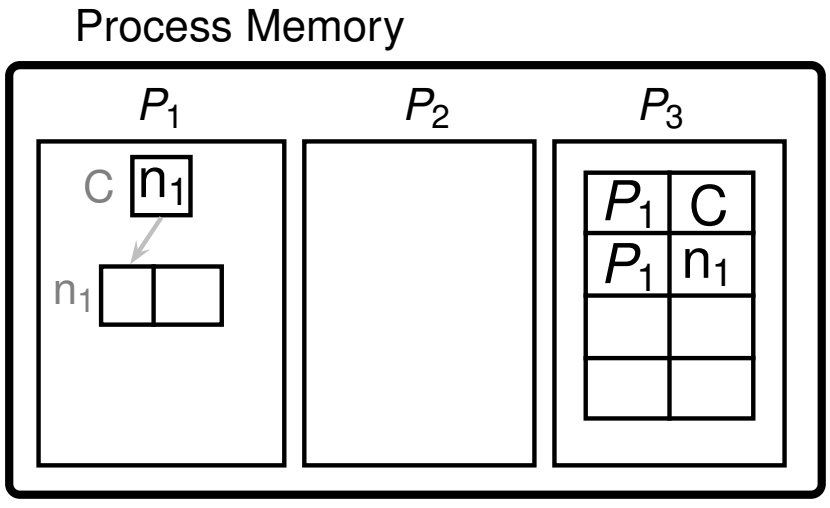
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
▶   list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  ⋮
}
```

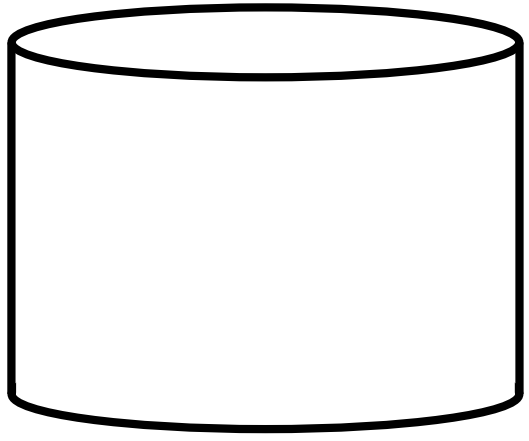




Creating a persistence point

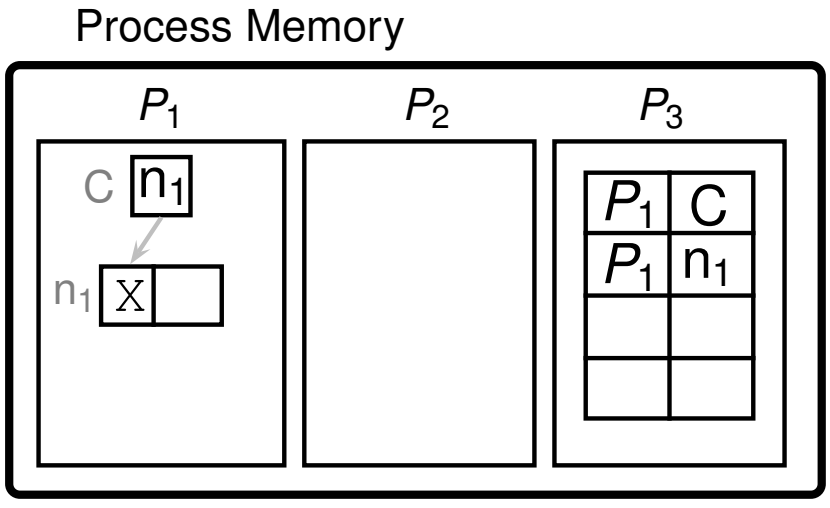
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
▶ C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  ⋮
}
```

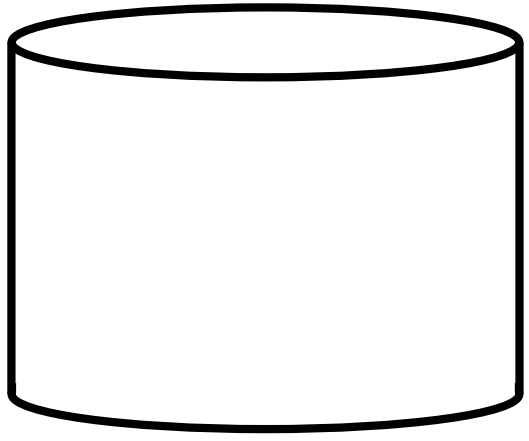




Creating a persistence point

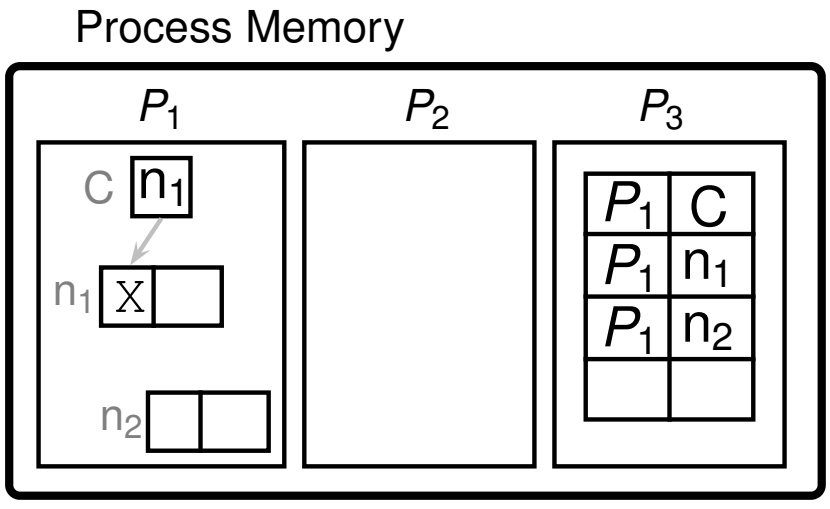
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
▶ n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  ⋮
}
```

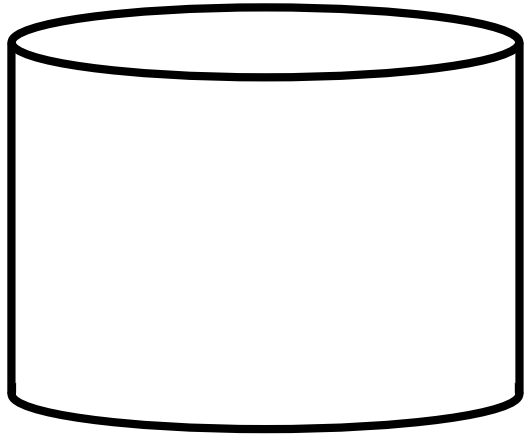




Creating a persistence point

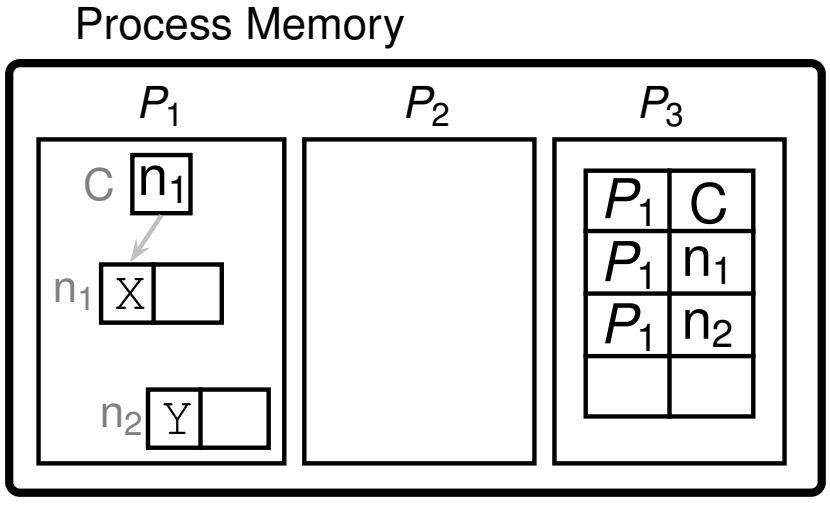
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
▶ list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  ⋮
}
```

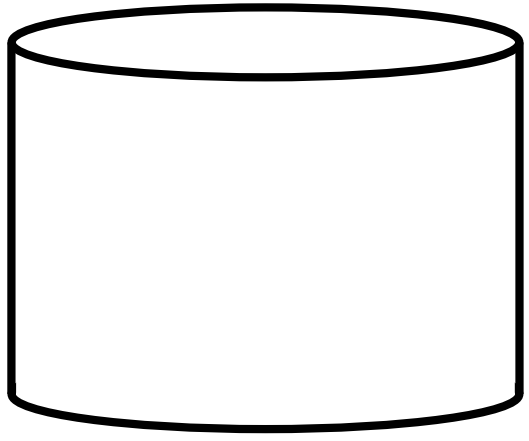




Creating a persistence point

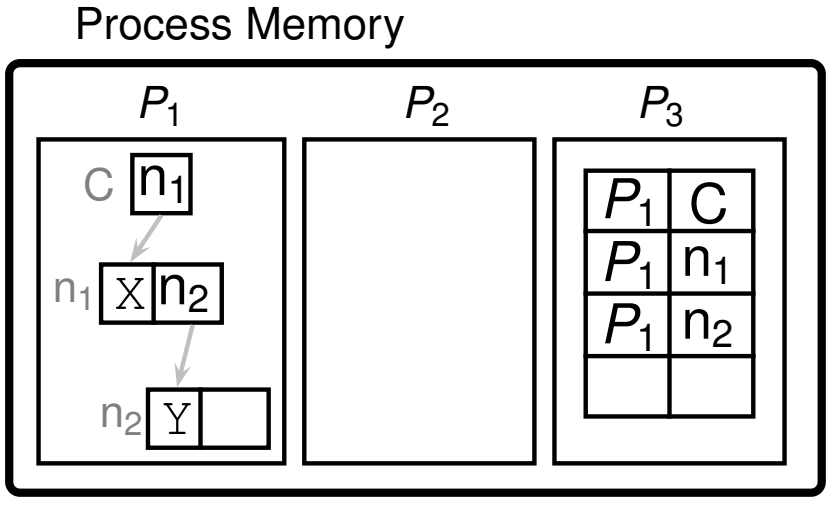
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
▶ n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  ⋮
}
```

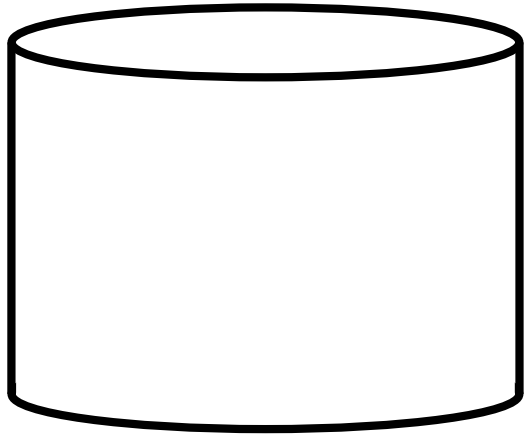




Creating a persistence point

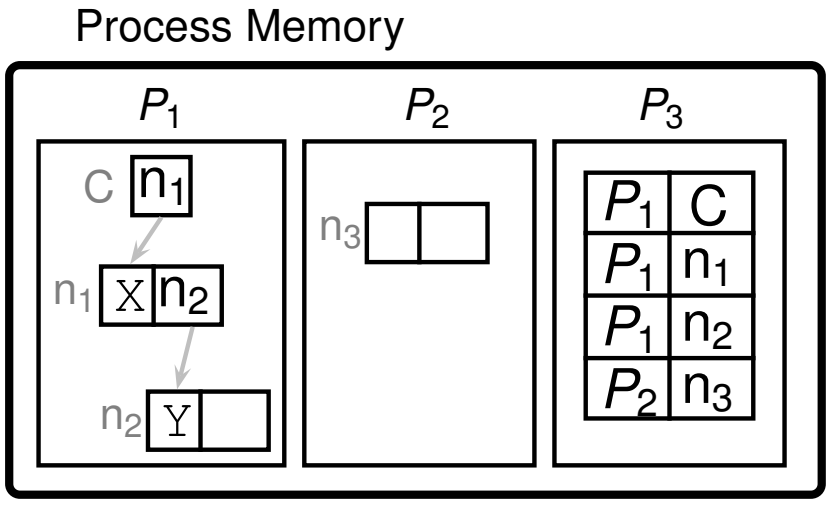
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
▶ l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  ⋮
}
```

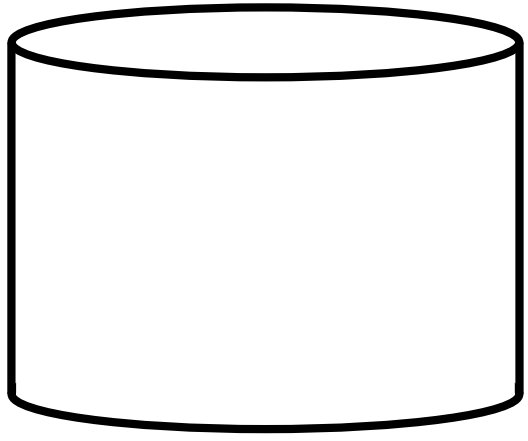




Creating a persistence point

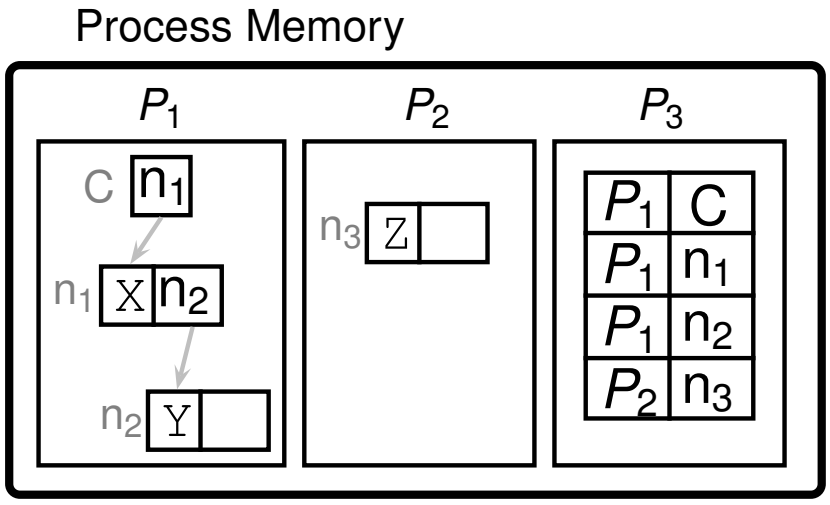
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
▶ list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  ⋮
}
```

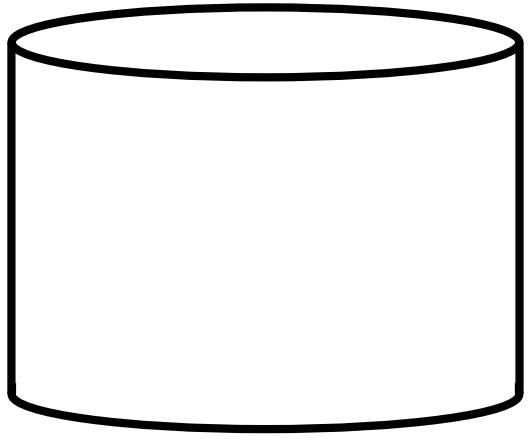




Creating a persistence point

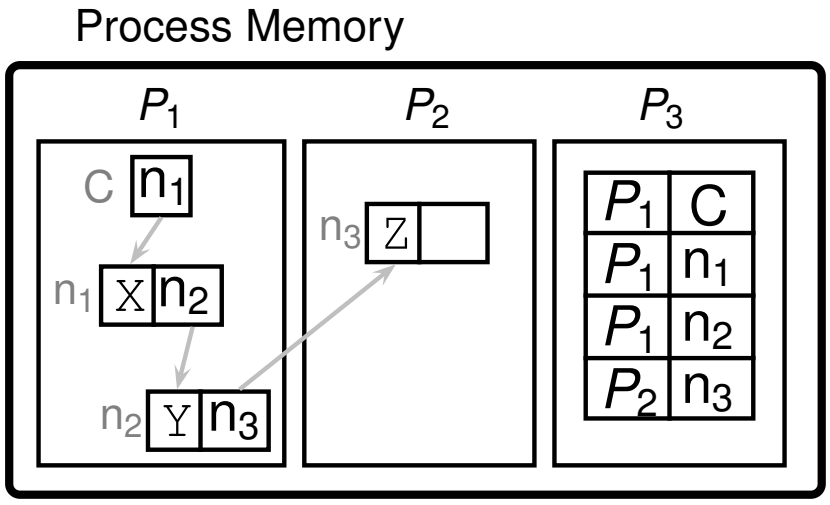
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
▶ n2->v = Z;
  n1->next = n2;
  pPoint(cid);
  ⋮
}
```

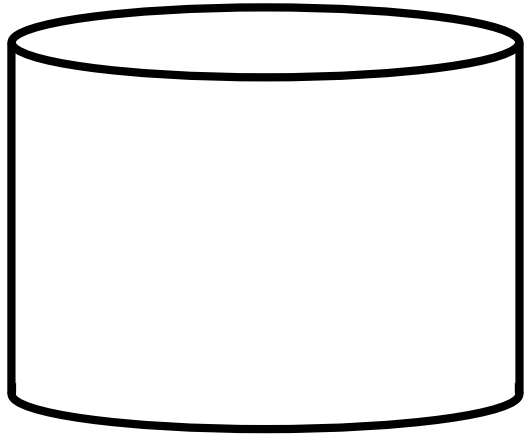




Creating a persistence point

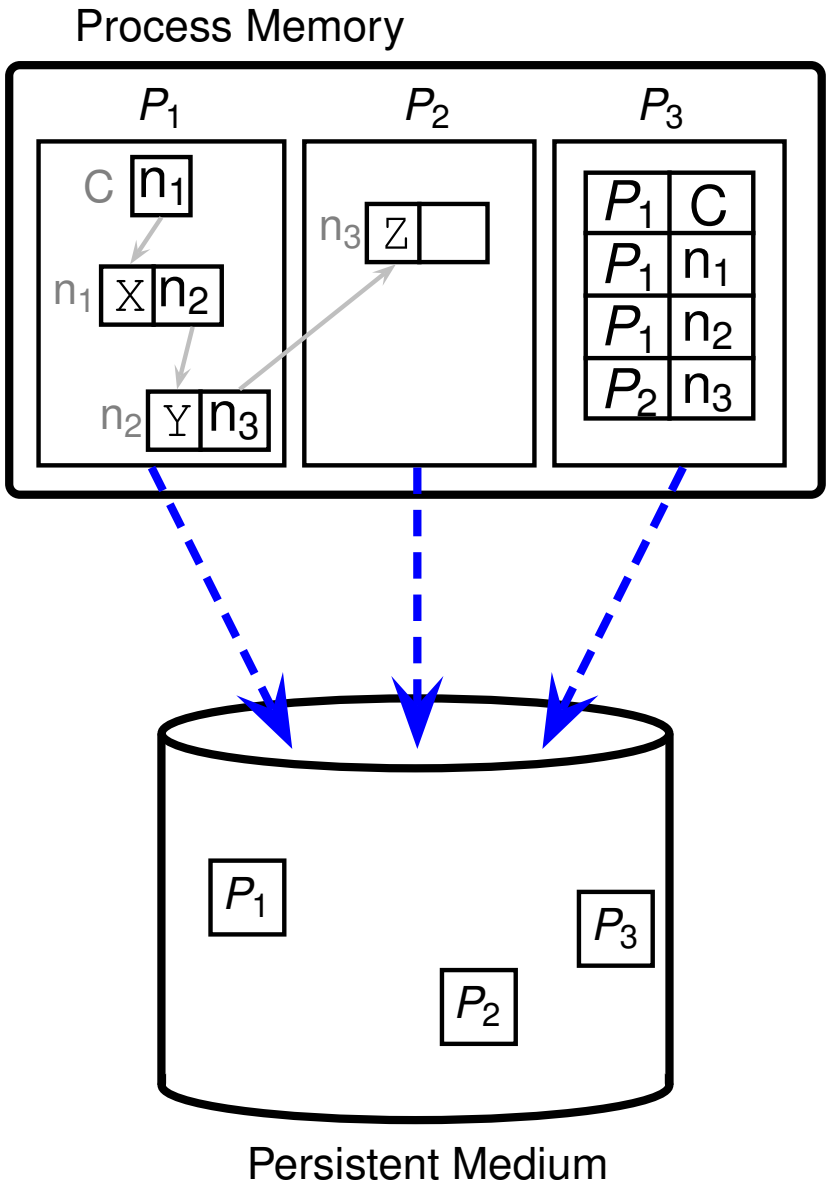
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
▶ n1->next = n2;
  pPoint(cid);
  ⋮
}
```





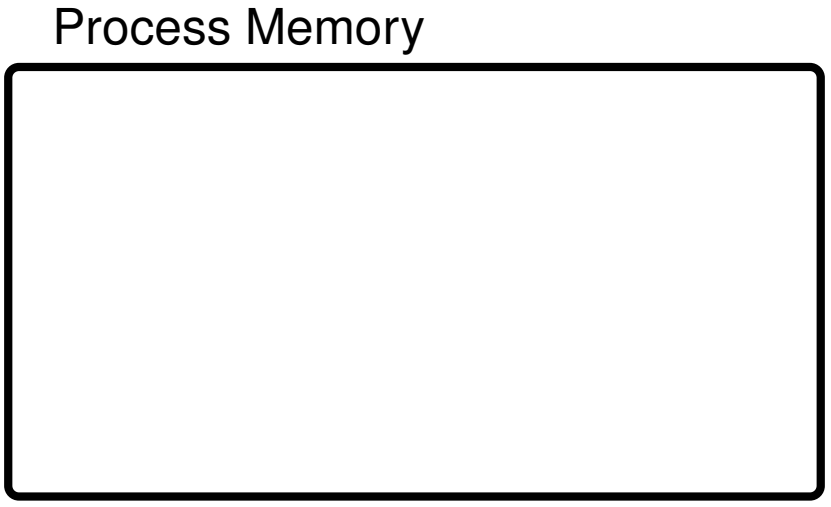
Creating a persistence point

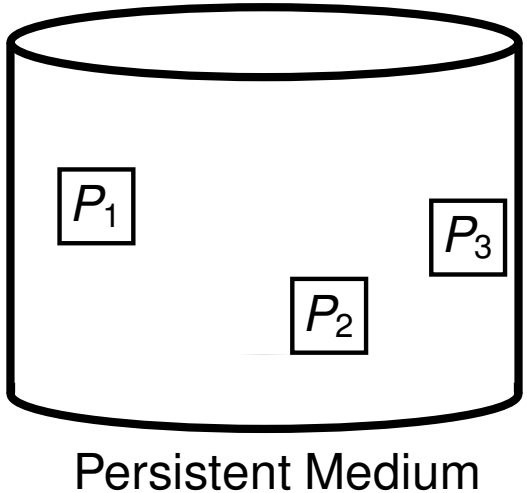
```
typedef struct l {
  int v;
  struct l * next;
} list;
struct pcont {
  list *head;
} *C;
foo() {
  cid = pCAlloc(&C, sizeof(*C));
  list *n1 = malloc(...);
  C->head = n1;
  n1->v = X;
  list *n2 = malloc(...);
  n1->v = Y;
  l->next = n1;
  list *n3 = malloc(...);
  n2->v = Z;
  n1->next = n2;
▶ pPoint(cid);
  ⋮
}
```



Restoring a persistence point

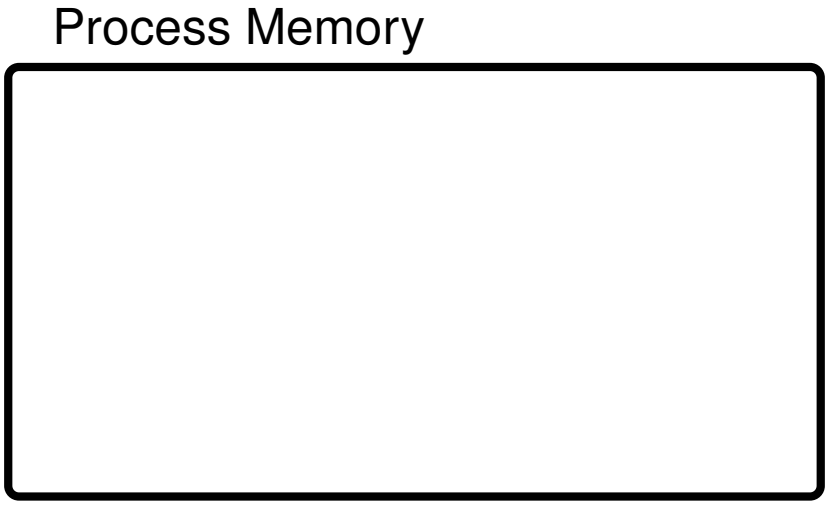
```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    ⋮
  }
  ⋮
}
```



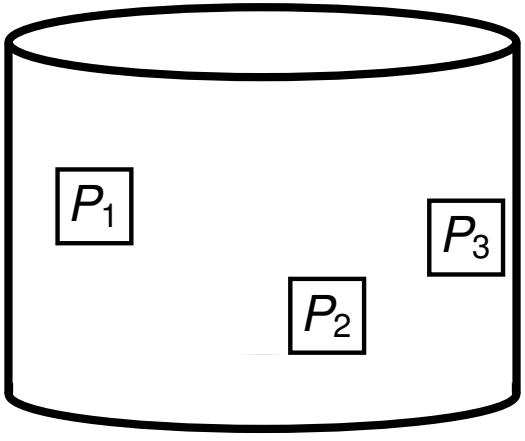


Restoring a persistence point

```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    :
  }
  :
}
```

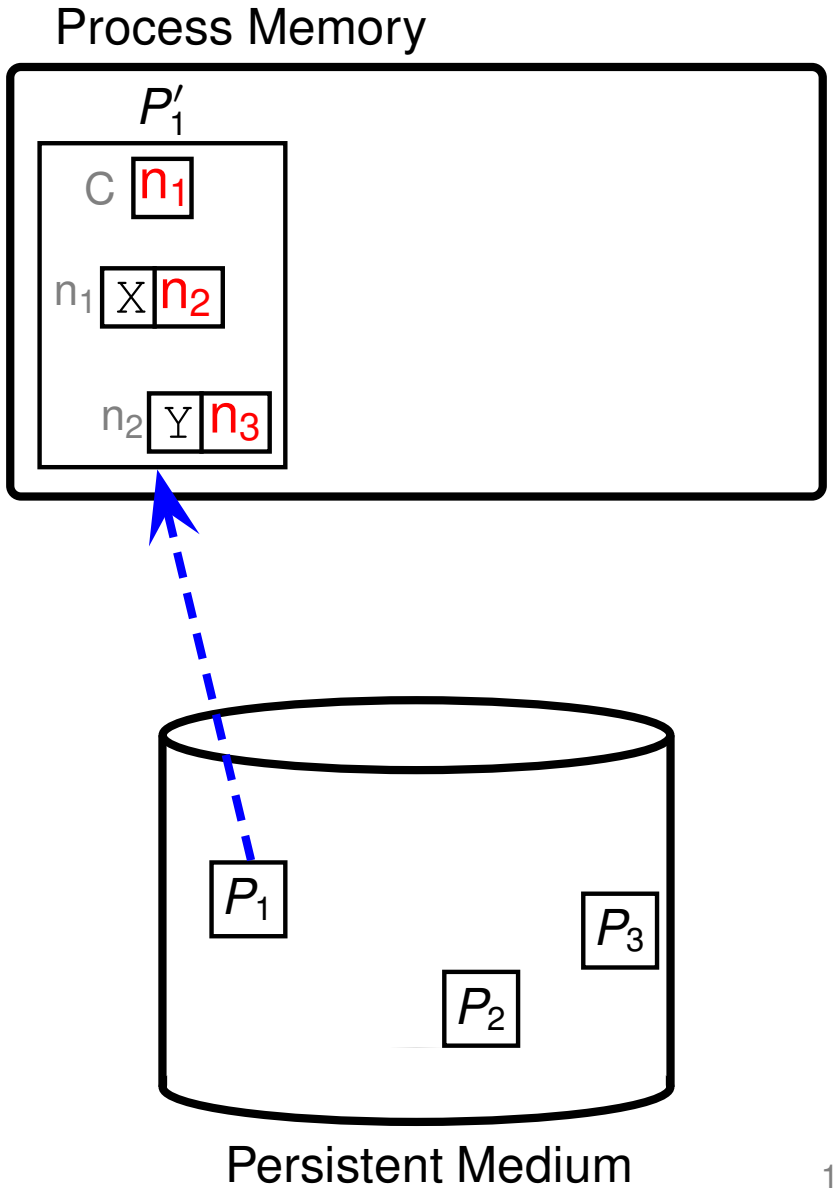


Prev	Curr



Restoring a persistence point

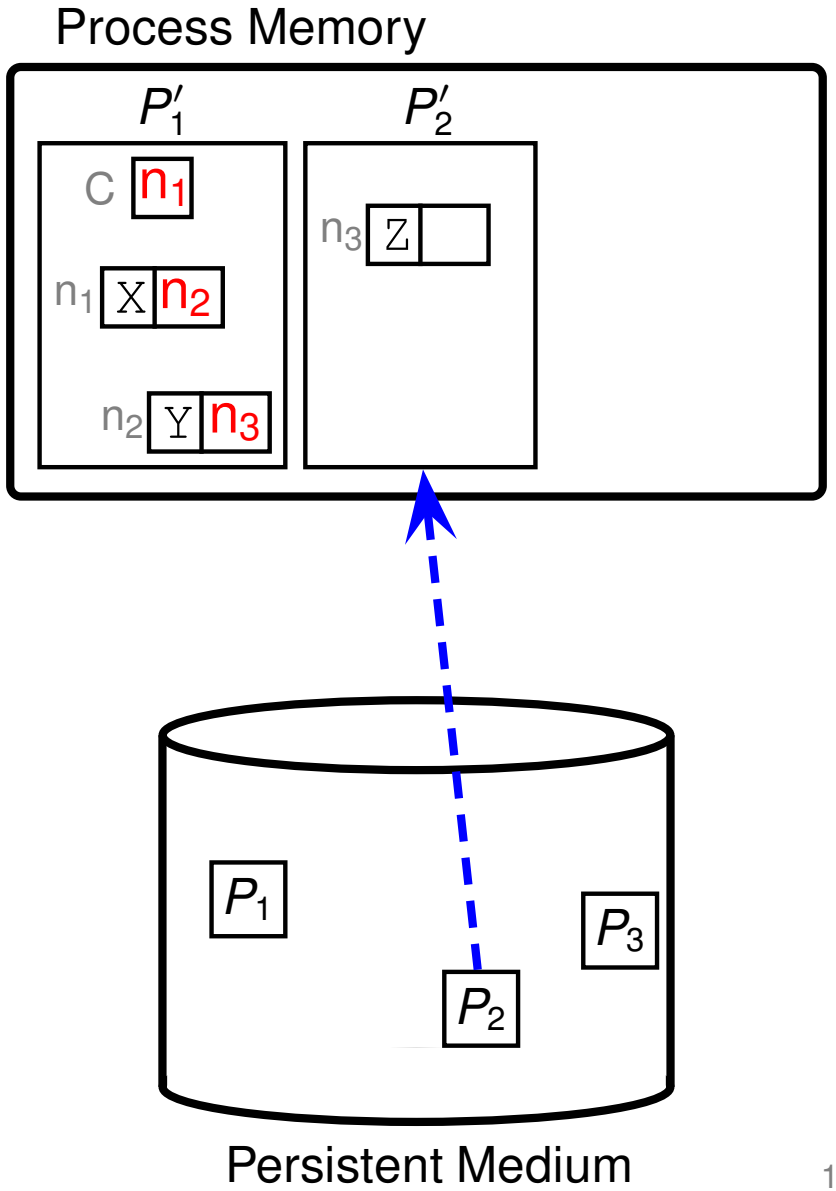
```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    :
  }
  :
}
```



Prev	Curr
P_1	P'_1

Restoring a persistence point

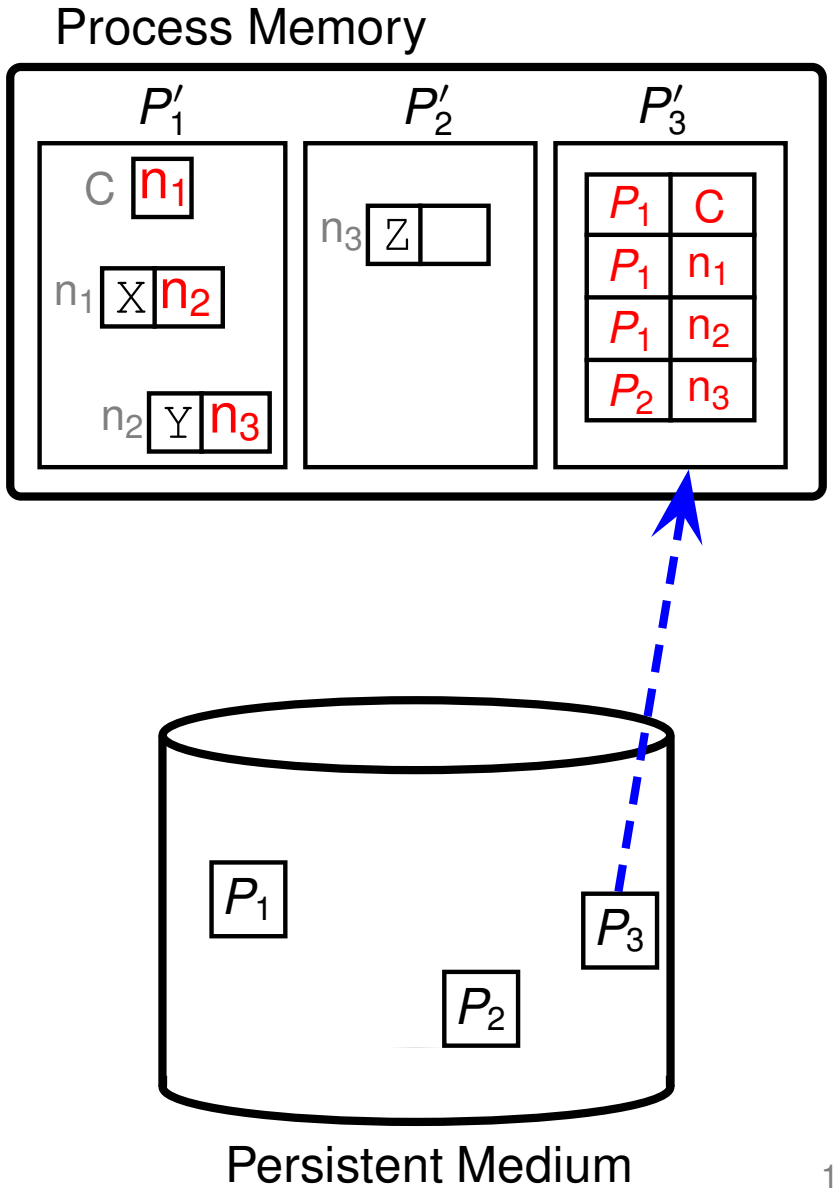
```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    :
  }
  :
}
```



Prev	Curr
P_1	P'_1
P_2	P'_2

Restoring a persistence point

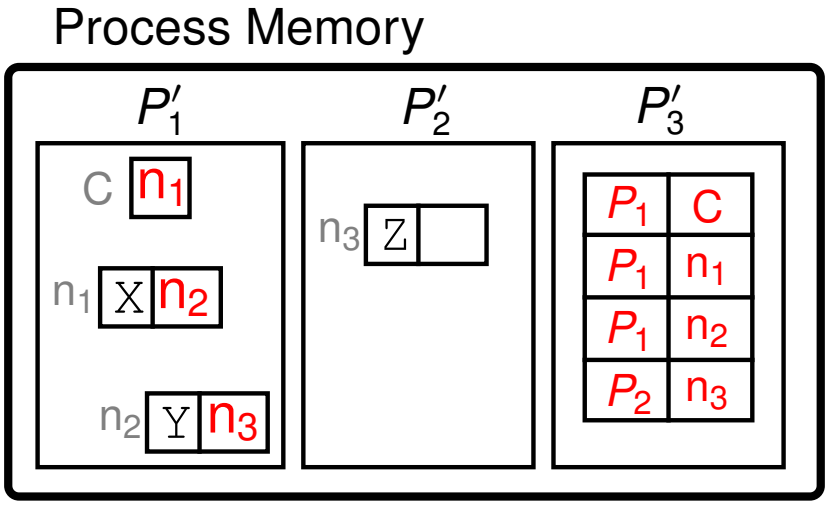
```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    :
  }
  :
}
```



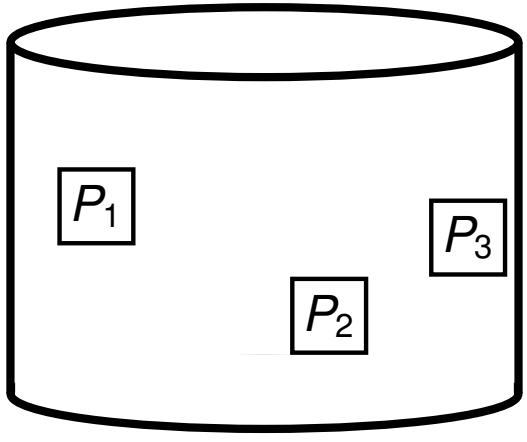
Prev	Curr
P_1	P'_1
P_2	P'_2
P_3	P'_3

Restoring a persistence point

```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    :
  }
  :
}
```

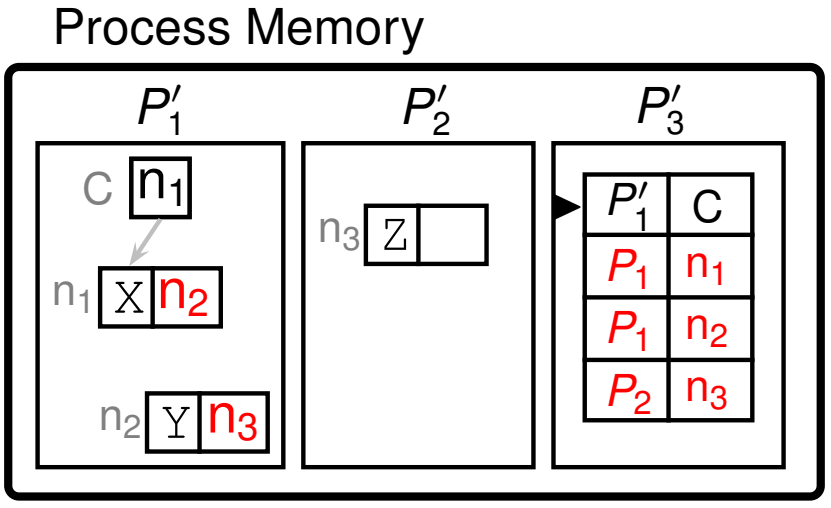


Prev	Curr
P_1	P'_1
P_2	P'_2
P_3	P'_3

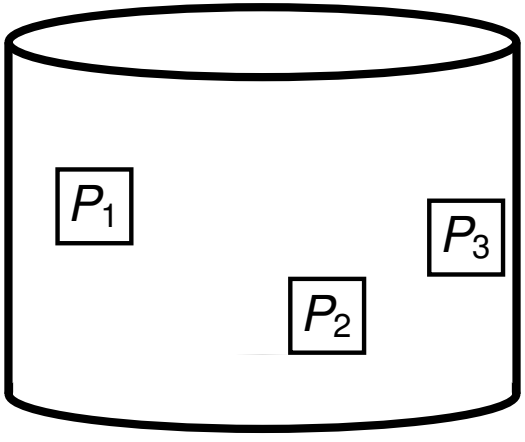


Restoring a persistence point

```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    :
  }
  :
}
```

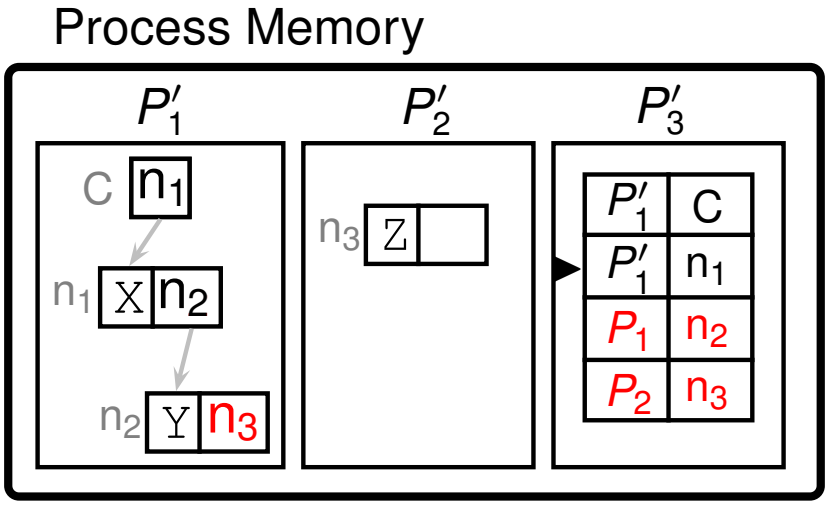


Prev	Curr
P_1	P'_1
P_2	P'_2
P_3	P'_3

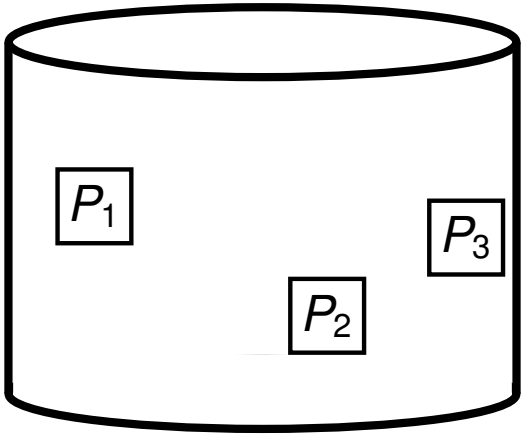


Restoring a persistence point

```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    :
  }
  :
}
```

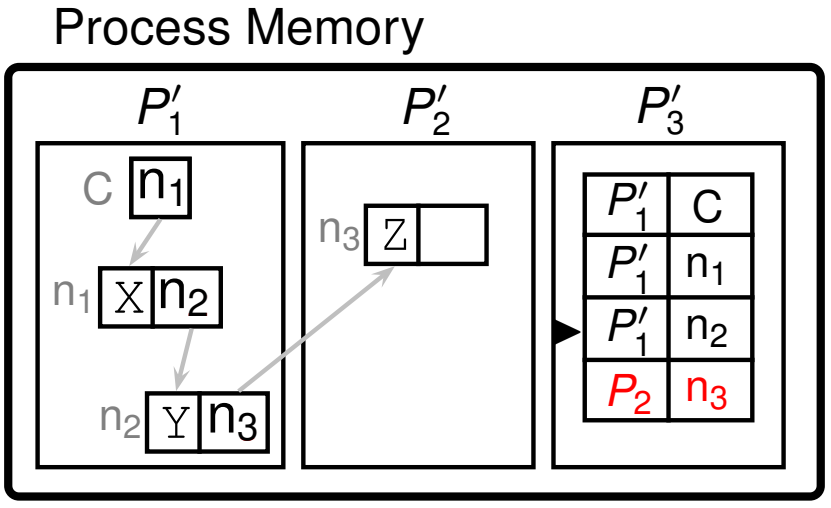


Prev	Curr
P_1	P'_1
P_2	P'_2
P_3	P'_3

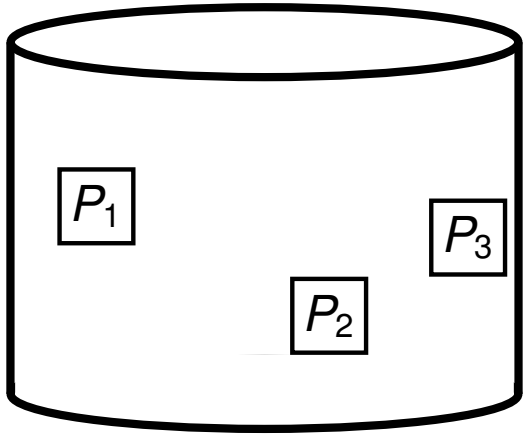


Restoring a persistence point

```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    :
  }
  :
}
```

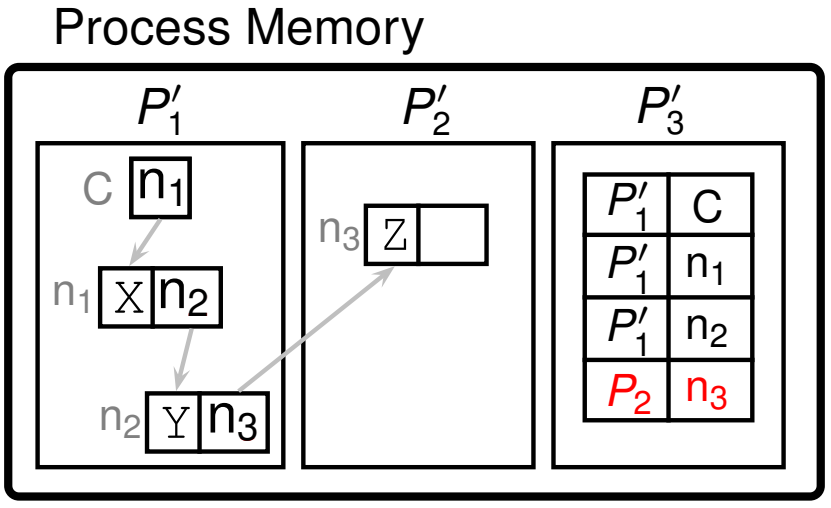


Prev	Curr
P_1	P'_1
P_2	P'_2
P_3	P'_3

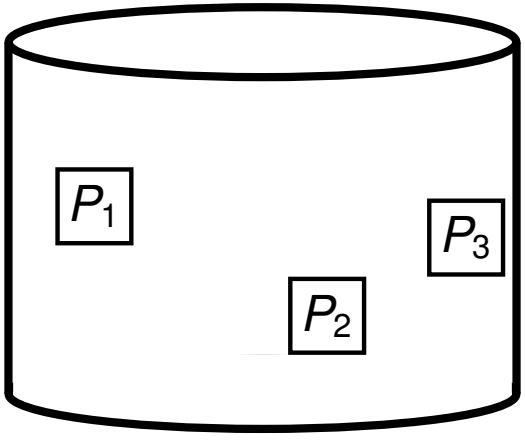


Restoring a persistence point

```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    :
  }
  :
}
```

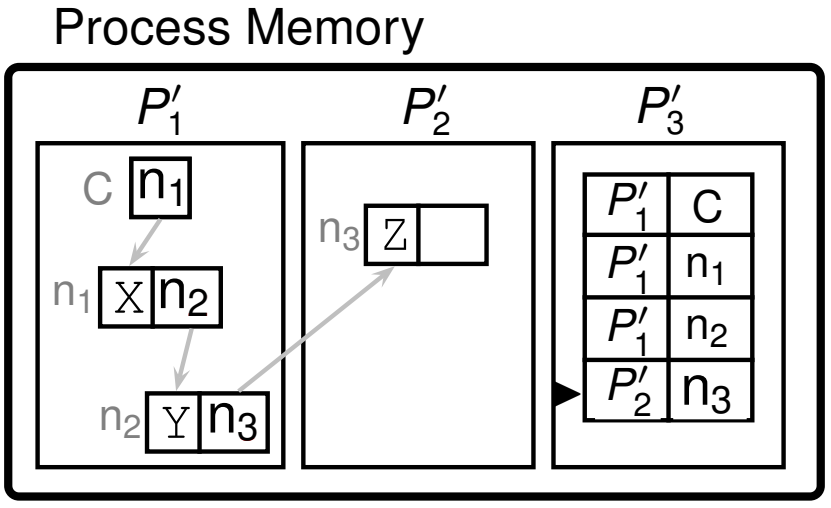


Prev	Curr
P_1	P'_1
P_2	P'_2
P_3	P'_3

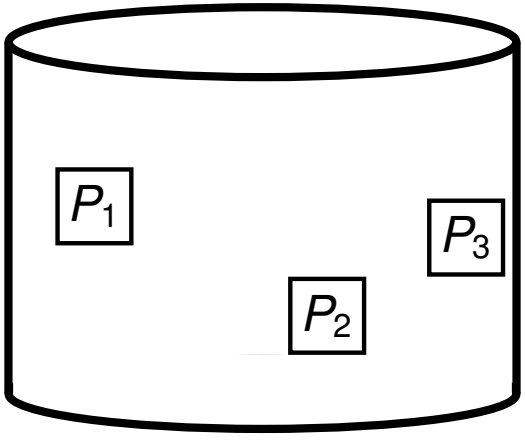


Restoring a persistence point

```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    ⋮
  }
  ⋮
}
```

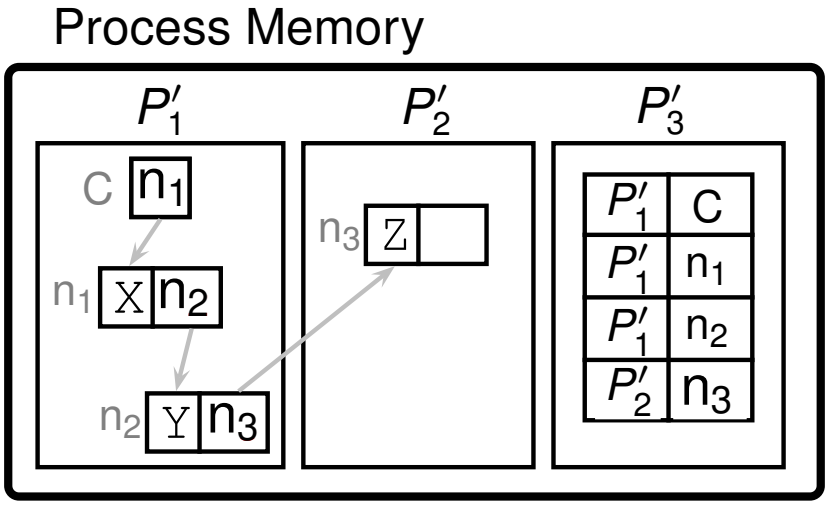


Prev	Curr
P_1	P'_1
P_2	P'_2
P_3	P'_3

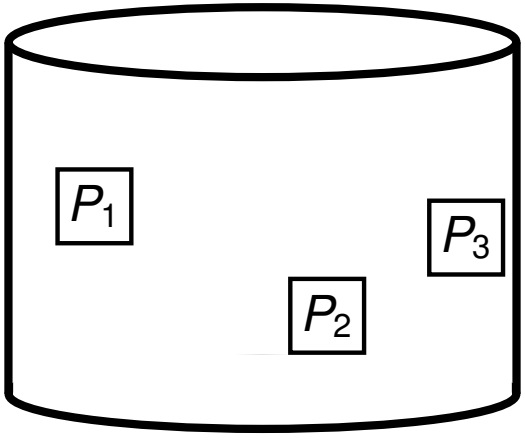


Restoring a persistence point

```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    :
  }
  :
}
```

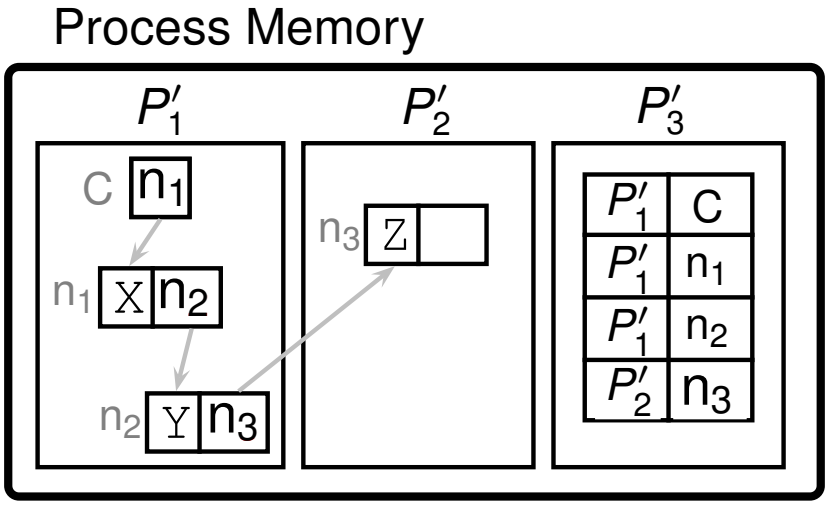


Prev	Curr
P_1	P'_1
P_2	P'_2
P_3	P'_3

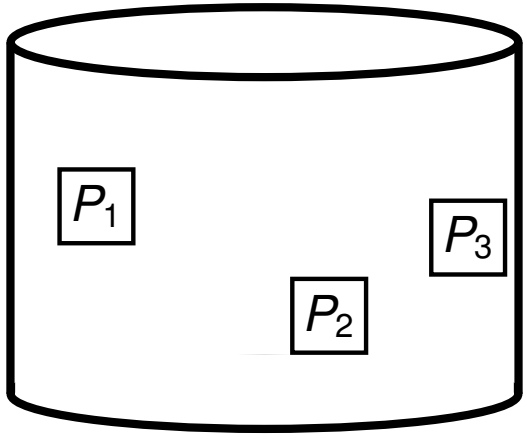


Restoring a persistence point

```
struct pcont {
  list *head;
} *C;
foo() {
  if (!(cid = pCRestore(&C)))
    ⋮
  }
  ⋮
}
```



Prev	Curr
P_1	P'_1
P_2	P'_2
P_3	P'_3



SoftPM Static code analysis

Type Definition	Usage

`struct list {`
`  int val;`
 	
`  struct list *next;`
 	
`};`
	`struct list *l = malloc(n);`

SoftPM Static code analysis

Type Definition	Usage

`struct list {`
`  int val;`
 	
`  struct list *next;`
 	
`};`
	`struct list *l = malloc(n);`

Source-to-source translation

SoftPM Static code analysis

Type Definition	Usage

`struct list {`
`  int val;`
 	
`  struct list *next;`
 	
`};`
	`struct list *l = malloc(n);`
	
	`mallocat(l, n);`

Source-to-source translation

- Records the location and size of all memory allocations

SoftPM Static code analysis

Type Definition	Usage

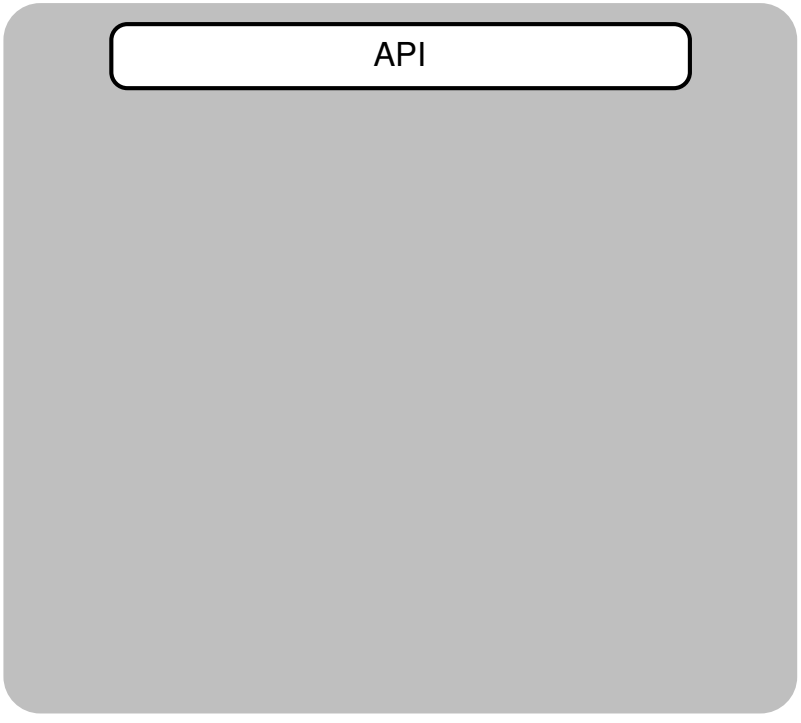
`struct list {`
`  int val;`
`  struct list *next;`
`};`
	`struct list *l = malloc(n);`
	`mallocat(l, n);`
	`pointerat(&l);`

Source-to-source translation

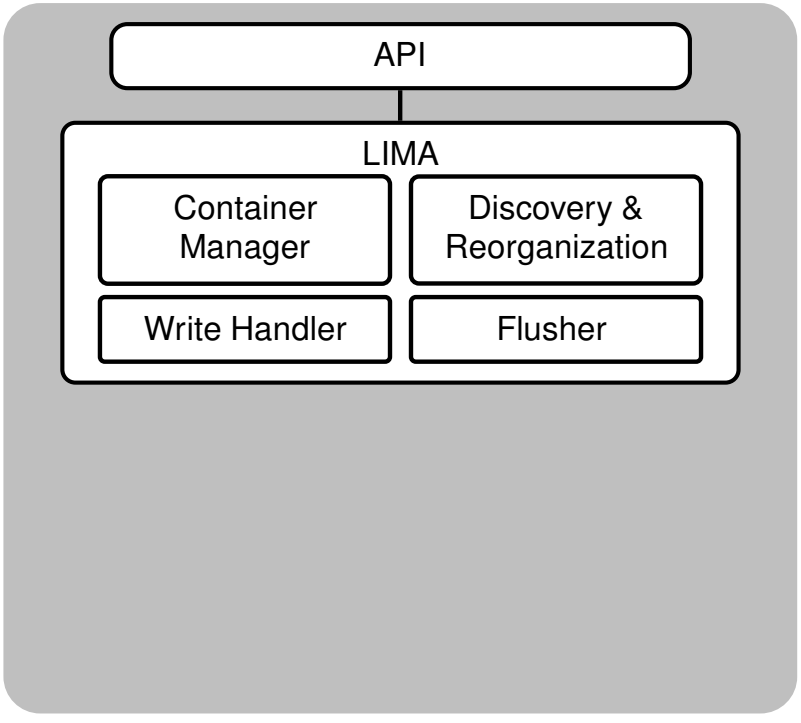
- Records the location and size of all memory allocations
- Registers the location of all pointers **at initialization**
  - ✓ Explicit initialization: *l-value* of assignment operations
  - ✓ Implicit initialization: *memory copying/move* operations

SoftPM Architecture

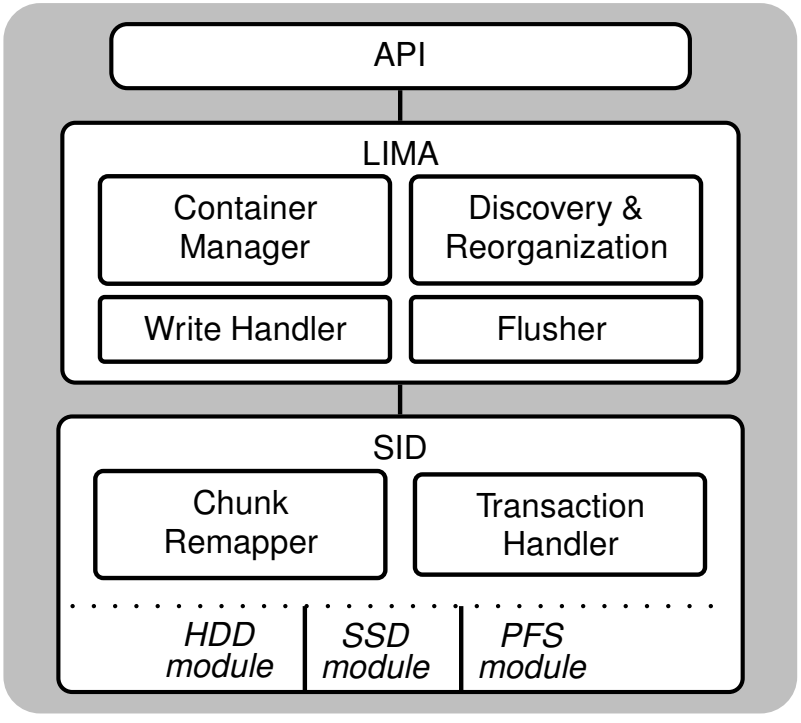
SoftPM Architecture



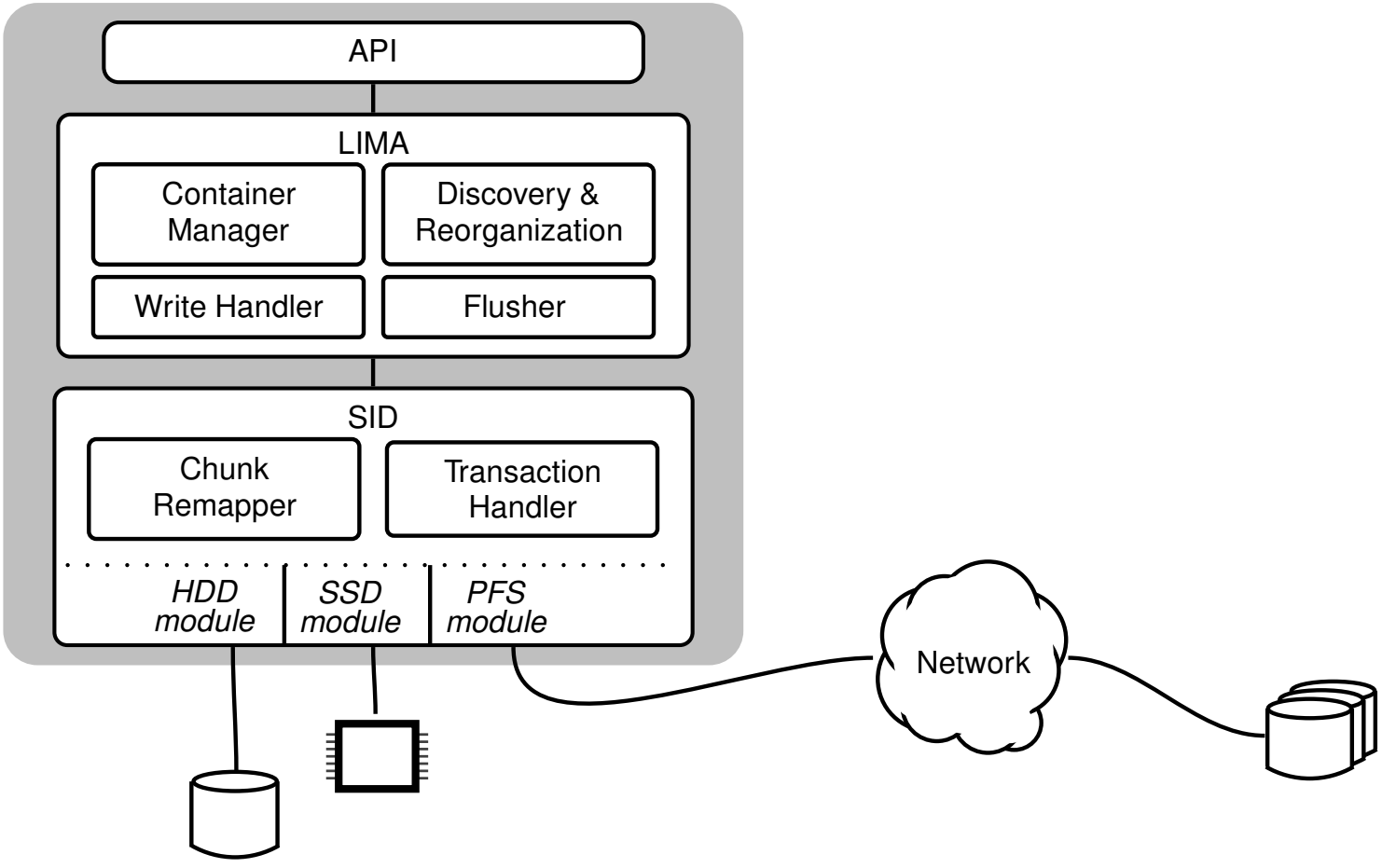
SoftPM Architecture



SoftPM Architecture



SoftPM Architecture



Evaluation

Configurations

- **Native**: Original unmodified code (with/without persistence)
- **TPL**: Using a data structure serialization library
- ***SoftPM***: Using *SoftPM* to identify and persist data

Evaluation

Configurations

- **Native**: Original unmodified code (with/without persistence)
- **TPL**: Using a data structure serialization library
- ***SoftPM***: Using *SoftPM* to identify and persist data

What we want to evaluate

Evaluation

Configurations

- **Native**: Original unmodified code (with/without persistence)
- **TPL**: Using a data structure serialization library
- ***SoftPM***: Using *SoftPM* to identify and persist data

What we want to evaluate

- Development complexity

Evaluation

Configurations

- **Native**: Original unmodified code (with/without persistence)
- **TPL**: Using a data structure serialization library
- ***SoftPM***: Using *SoftPM* to identify and persist data

What we want to evaluate

- Development complexity
- Application performance when using *SoftPM*

Evaluating development complexity

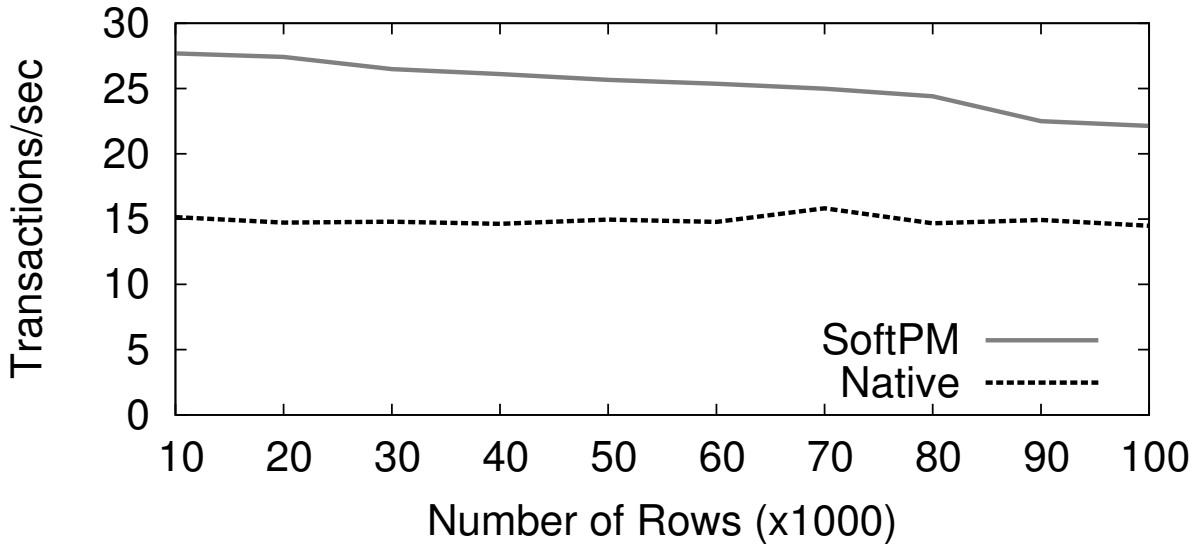
Data Structure or Application	Original LOC	LOC for Persistence	LOC to use *SoftPM*
Arrays	102	17	3
Linked List	188	24	3
RB Tree	285	22	3
Hash Table	396	21	3
SQLite	73042	6696	9
memcachedb	1894	205	40

NOTE: SQLite and memcachedb implement their own persistence.

SQLite transaction performance

Workload

Benchmark creates a DB and performs random insert, select, update, and delete transactions.

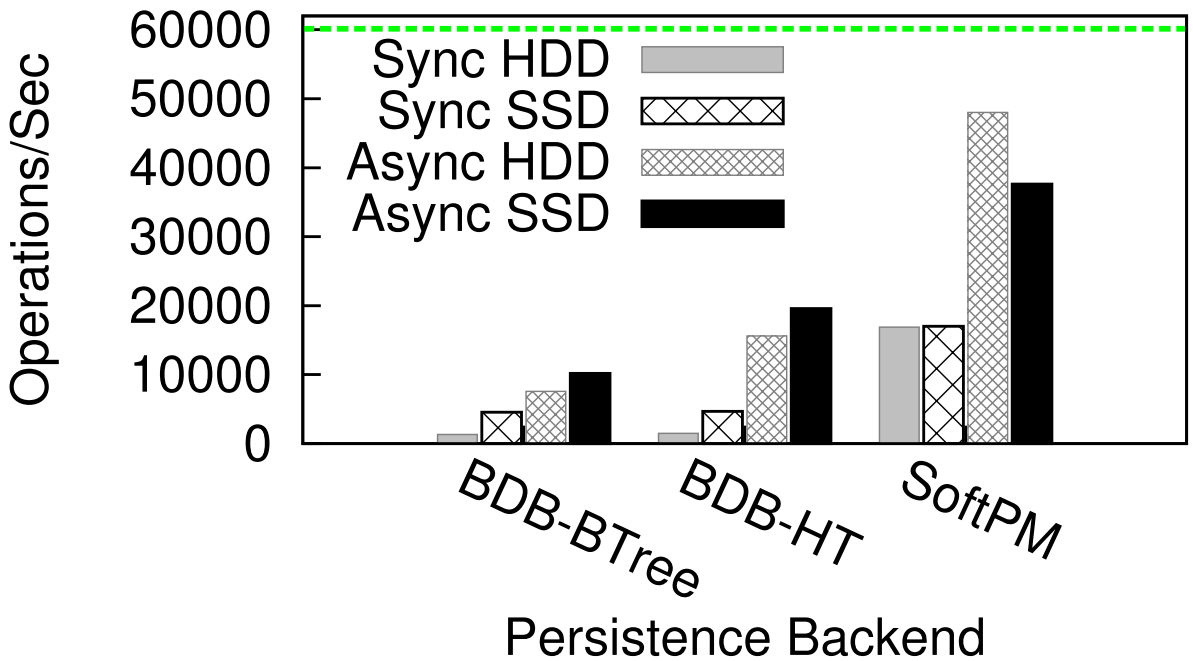


s

Memcachedb using different back-ends

Workload

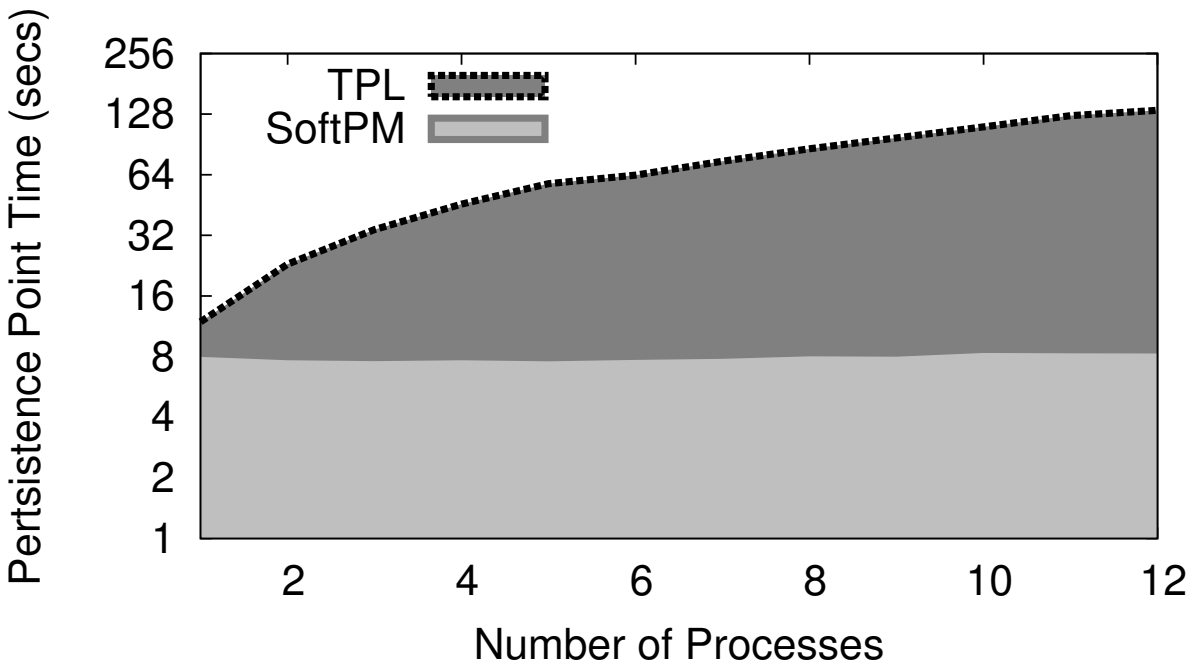
The workload randomly adds, queries, and deletes 512 byte elements with 16 byte keys. The dashed line represents a memory only solution.



Scalability of *SoftPM*

Workload

- Multiple processes; multiple containers
- Each container stores a linked list of size 1GB/processes



Related Work

Process Checkpointing

- libckpt[USENIX'95]
- Zap[OSDI'02]

Related Work

Process Checkpointing

- libckpt[USENIX'95]
- Zap[OSDI'02]

Object Oriented DBs

- ObjectStore[CACM'91]
- Thor[SIGMOD'96]

Related Work

Process Checkpointing

- libckpt[USENIX'95]
- Zap[OSDI'02]

Object Oriented DBs

- ObjectStore[CACM'91]
- Thor[SIGMOD'96]

Persistent Memory

- RVM[CACM'91]
- SSDAlloc[USENIX'10]
- NVHeap and Mnemosyne[ASPLOS'11]

Conclusions

SoftPM Offers a Different Programming Model

- A memory abstraction for persistent data
- Fully automated
- Transactional
- High-performance
- Traditional volatile interfaces
- Works with existing technologies

Thank you!

(I'll be happy to take questions)





