Understanding Linux Lists

Nic Volanschi and Julia Lawall (Inria)

August 25, 2020

Paper to appear at ASE 2020

Lists

A fundamental data structure to make a collection of objects.

Concepts:

- List elements: the data values contained in the list
- List element connector: how to get from one element to the next
- List head: how to find the start of the list

Challenges for typing:

- Different lists contain different types of elements.
  - Work queues contain work, run queues contain tasks, etc
- Want one list type and operations for the thousands of list element types.

Linux lists

Lists in code:

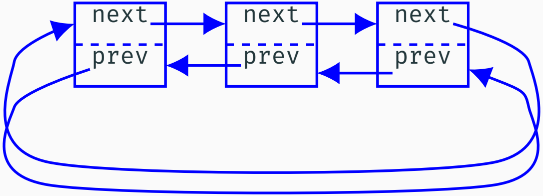
```
struct list_head {
  struct list_head *next, *prev;
};
```

Linux lists

Lists in code:

```
struct list_head {
  struct list_head *next, *prev;
};
```

Lists in pictures:

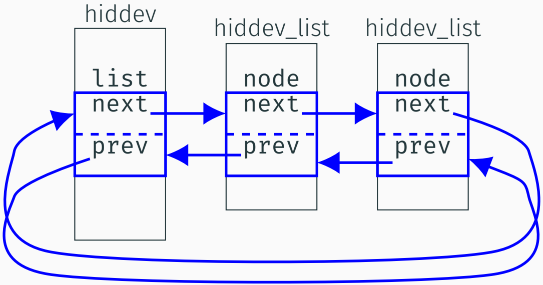


Lists in code

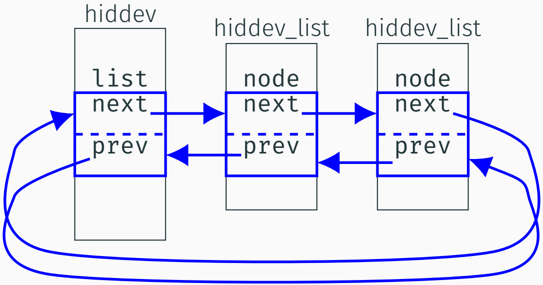
```c
struct hiddev {
  int minor;
  ...
  struct list_head list;
  spinlock_t list_lock;
  ...
};
```

```c
struct hiddev_list {
  struct hiddev_usage_ref buffer[HIDDEV_BUFFER_SIZE];
  ...
  struct list_head node;
  ...
}
```

Lists in pictures



Lists in pictures



List elements retrieved using `list_entry()`, *i.e.*, `container_of()`.

Assessment

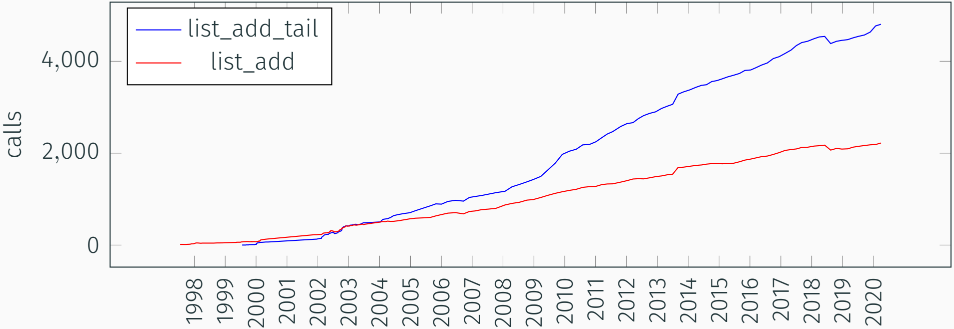
+ One API for all kinds of lists.

```
void list_add(struct list_head *new, struct list_head *head);
void list_add_tail(struct list_head *new, struct list_head *head);

list_entry(ptr, type, member)

list_for_each(pos, head) ...
list_for_each_entry(pos, head, member) ...
```

+ Embedded list connectors improve locality.

+ List operations provide some concurrency guarantees.

List usage over time



Problems

`list_head`s everywhere!

- What is their role?
  - List head?
  - List element connector?
- What are the involved types?
  - For a list head, what is the type of the elements?
  - For a list element, from what types of heads is it reachable?

Example

```c
struct hiddev {
  int minor;
  ...
  struct list_head list;
  spinlock_t list_lock;
  ...
};
```

```c
struct hiddev_list {
  struct hiddev_usage_ref buffer[HIDDEV_BUFFER_SIZE];
  ...
  struct list_head node;
  ...
};
```

Example

```
struct hiddev {
  int minor;
  ...
  struct list_head list;
  spinlock_t list_lock;
  ...
};
```

```
struct hiddev_list {
  struct hiddev_usage_ref buffer[HIDDEV_BUFFER_SIZE];
  ...
  struct list_head node;
  ...
};
```

No comments, and the structures are defined in different files.

Example

```c
struct hiddev {
  int minor;
  ...
  struct list_head list;
  spinlock_t list_lock;
  ...
};
```

```c
struct hiddev_list {
  struct hiddev_usage_ref buffer[HIDDEV_BUFFER_SIZE];
  ...
  struct list_head node;
  ...
}
```

No comments, and the structures are defined in different files.

Only 35-40% of `list_head` fields have comments, depending on the version.

- Some useful: “`head of waiting srb list`”
- Some obscure or irrelevant: “`submitted to pdma fifo`”

Observation

List operators give type information:

```
struct hiddev *hiddev = hid->hiddev;
struct hiddev_list *list;
...
list_for_each_entry(list, &hiddev->list, node) {
        ...
}
```

Observation

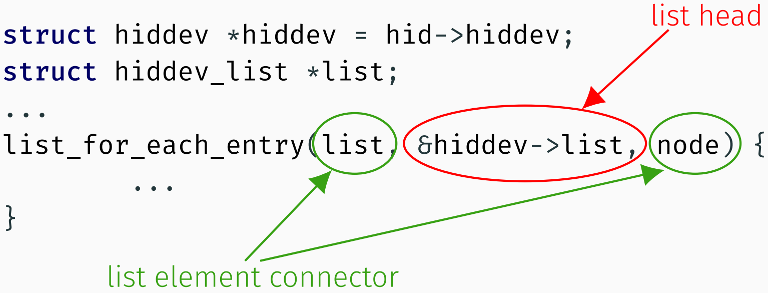
List operators give type information:

```
struct hiddev *hiddev = hid->hiddev;
struct hiddev_list *list;
...
list_for_each_entry(list, &hiddev->list, node) {
        ...
}
```

list head

Observation

List operators give type information:



```
struct hiddev *hiddev = hid->hiddev;
struct hiddev_list *list;
...
list_for_each_entry(list, &hiddev->list, node) {
        ...
}
```

list head

list element connector

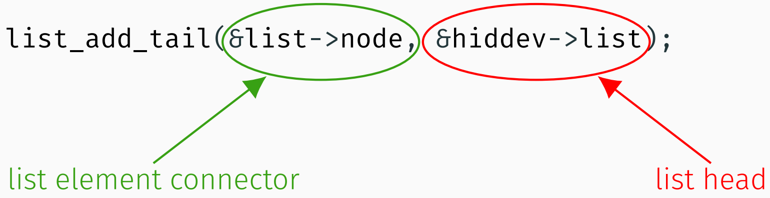
Observation, continued

List operators give type information:

```
list_add_tail(&list->node, &hiddev->list);
```

Observation, continued

List operators give type information:



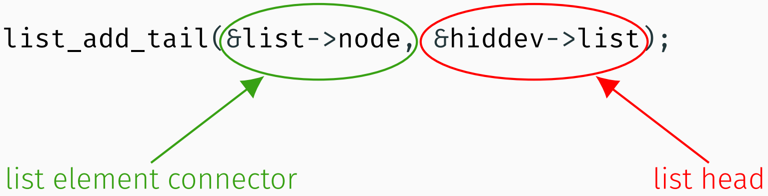
```
list_add_tail(&list->node, &hiddev->list);
```

list element connector

list head

Observation, continued

List operators give type information:



```
list_add_tail(&list->node, &hiddev->list);
```

list element connector

list head

Assessment:

- The role and type information is available in the source code.
- But scattered in different files and functions, and requires C type information.

Approach

- Scan the code base to collect information about list operator arguments.
- Make inferences from this information.

Approach

- Scan the code base to collect information about list operator arguments.
- Make inferences from this information.

Type language:

$l_1 : l_2$, i.e., head : element

where $l ::= s.f \mid v$

for structure name s, field name f, and variable v

Approach

- Scan the code base to collect information about list operator arguments.
- Make inferences from this information.

Type language:

$l_1 : l_2$, i.e., head : element

where $l ::= s.f \,|\, v$

for structure name s, field name f, and variable v

Example: `hiddev.list : hiddev_list.node`

Results

- Over 10,000 `list_head`s detected in Linux v5.6.
- Some are not used with standard operators, so no type is inferred (7.2%).
- A few hundred `list_head`s per version appear to be unused (2.9%).

Results

- Over 10,000 `list_head`s detected in Linux v5.6.
- Some are not used with standard operators, so no type is inferred (7.2%).
- A few hundred `list_head`s per version appear to be unused (2.9%).

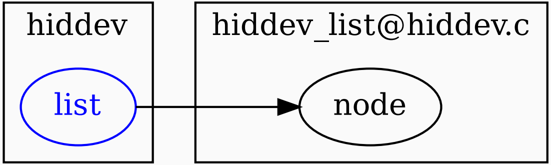
Experiment	Typed (total)	Head only	Element only	Head & element
v4.19	8601	4797 (55.8%)	3600 (41.9%)	204 (2.4%)
v5.6	9125	5078 (55.6%)	3823 (41.9%)	224 (2.5%)

Visualization tool

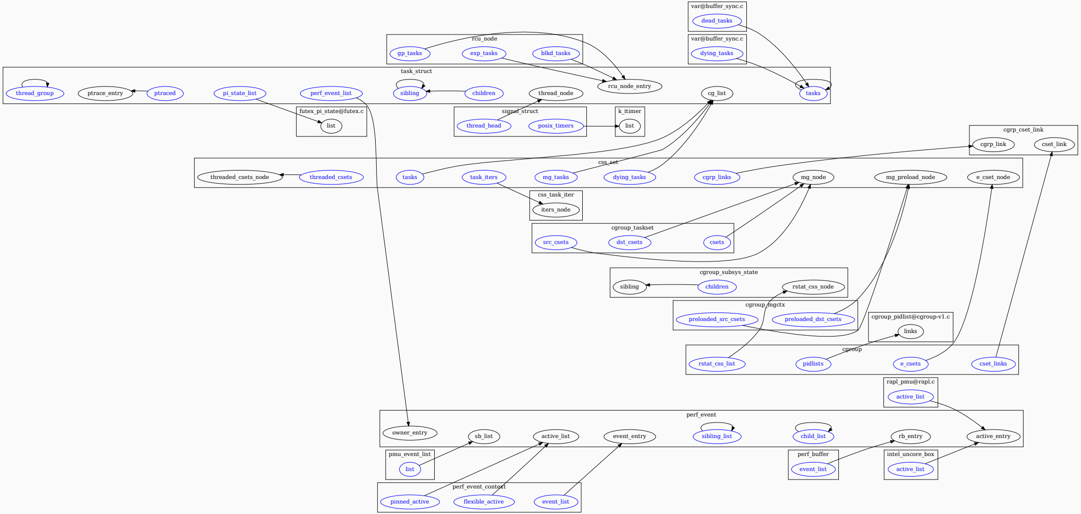
- Graphical representation of the inferred types based on GraphViz.
- Boxes for structures, circles for fields.
- Blue circles for list heads, black circles for list element connectors.

Visualization tool

- Graphical representation of the inferred types based on GraphViz.
- Boxes for structures, circles for fields.
- Blue circles for list heads, black circles for list element connectors.



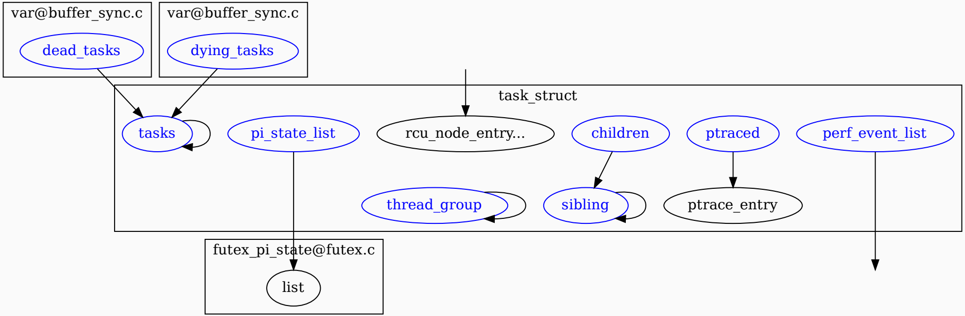
Visualization tool: task_struct



`list_heads` that are both heads and element connectors

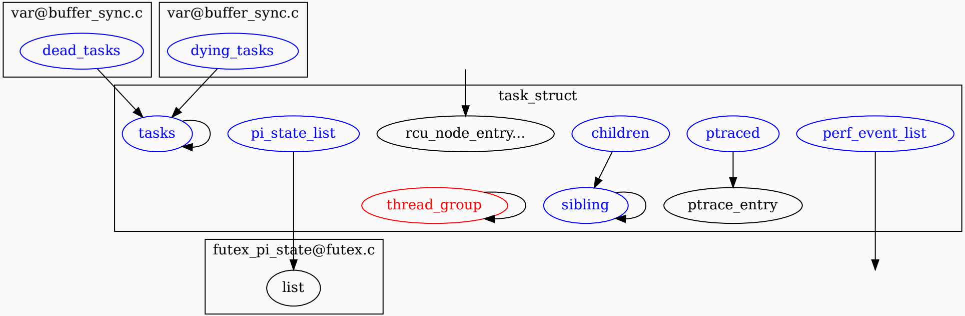
Experiment	Hd & elm (total)	Self-lists	Mutual pairs	Other cases
v4.19	204	164 (80.4%)	11 (10.8%)	18 (8.8%)
v5.6	224	179 (79.9%)	13 (11.6%)	19 (8.5%)

Some interesting examples



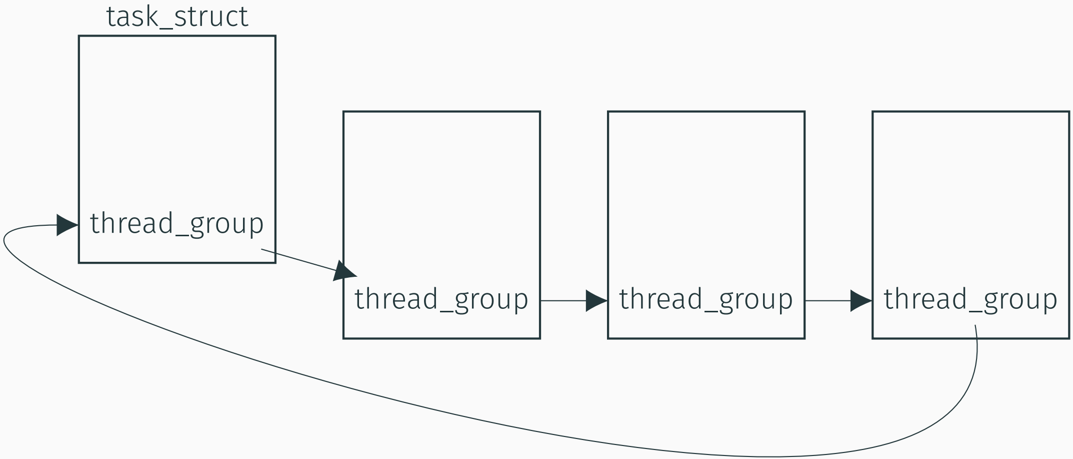
- List elements that are also list heads.
- Self loops.
- etc.

Some interesting examples

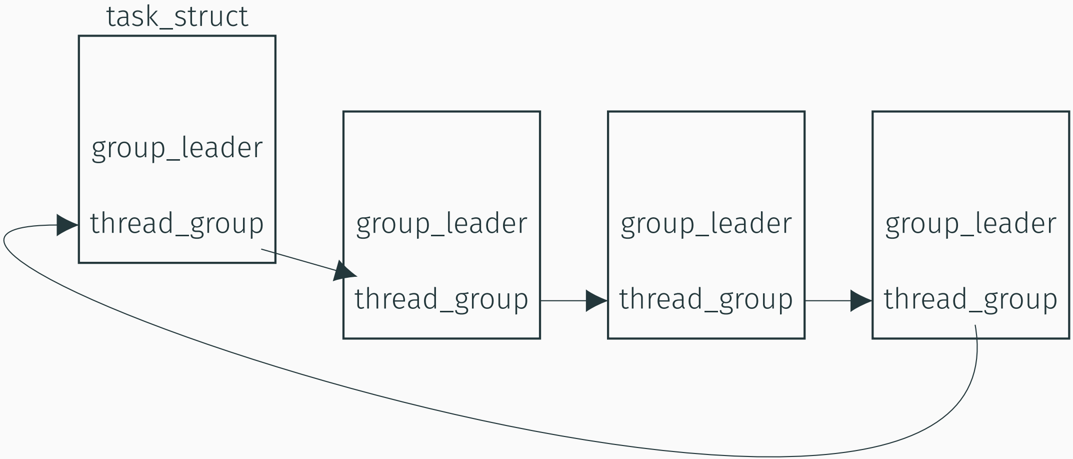


- List elements that are also list heads.
- Self loops.
- etc.

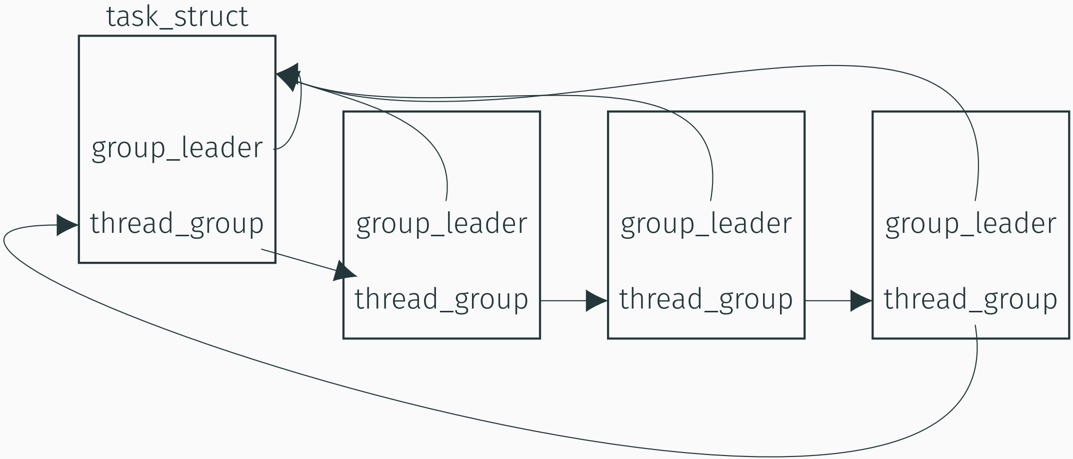
Umbrellas


task_struct
thread_group
thread_group
thread_group
thread_group

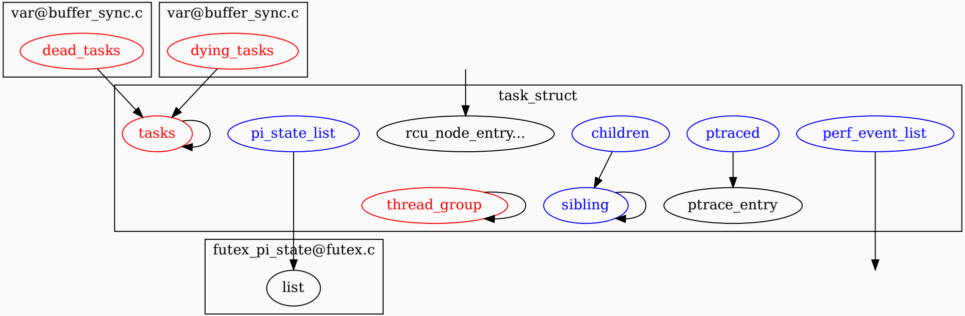
Umbrellas


task_struct
group_leader
thread_group
group_leader
thread_group
group_leader
thread_group
group_leader
thread_group

Umbrellas

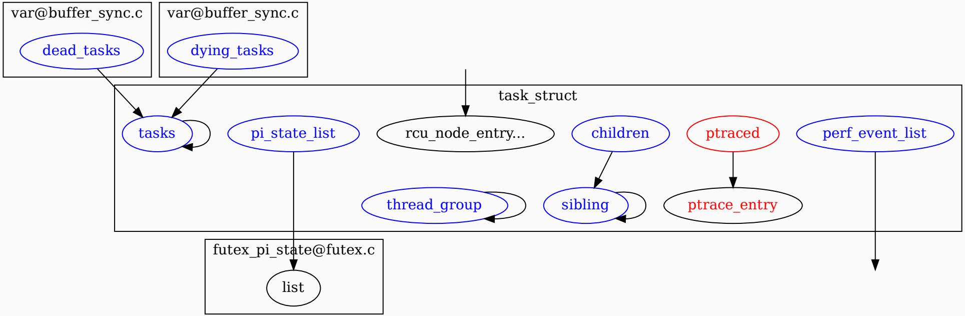

task_struct
group_leader
thread_group
group_leader
thread_group
group_leader
thread_group
group_leader
thread_group

Some interesting examples



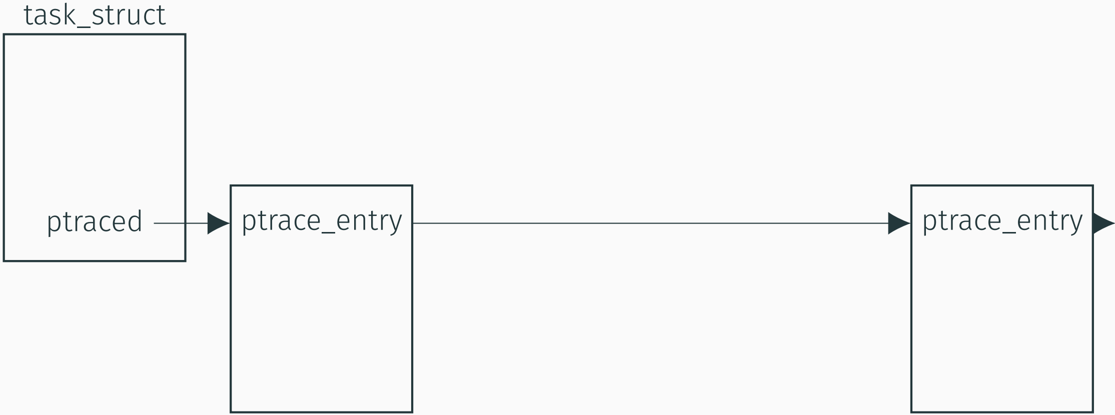
- List elements that are also list heads.
- Self loops.
- etc.

Some interesting examples

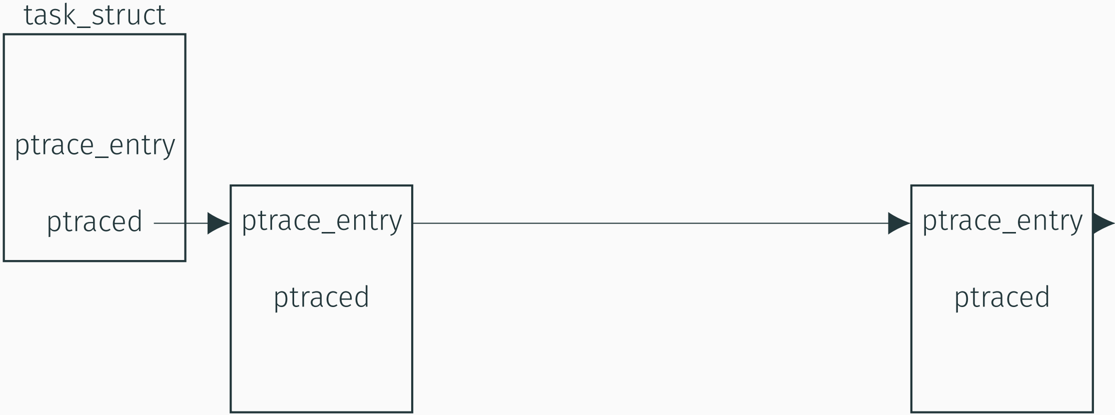


- List elements that are also list heads.
- Self loops.
- etc.

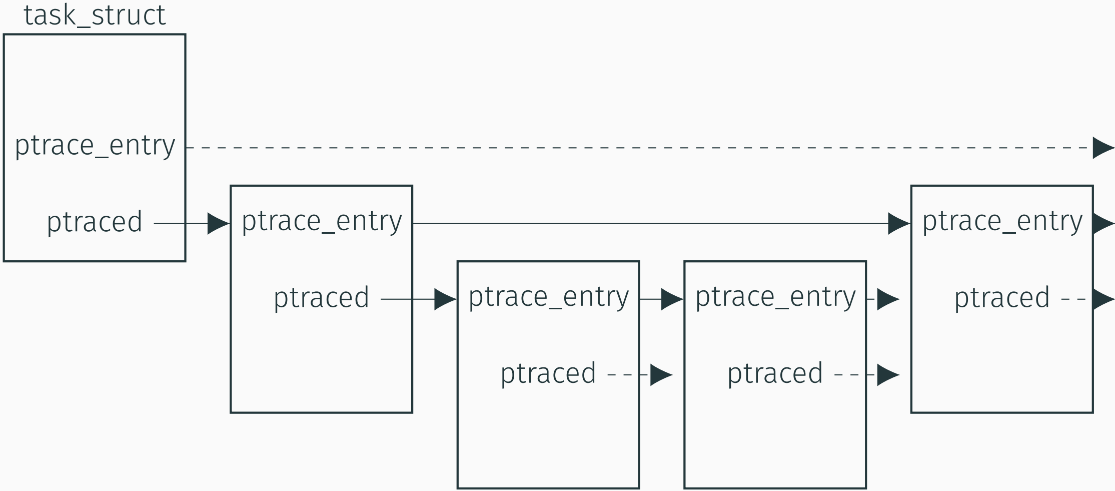
Trees



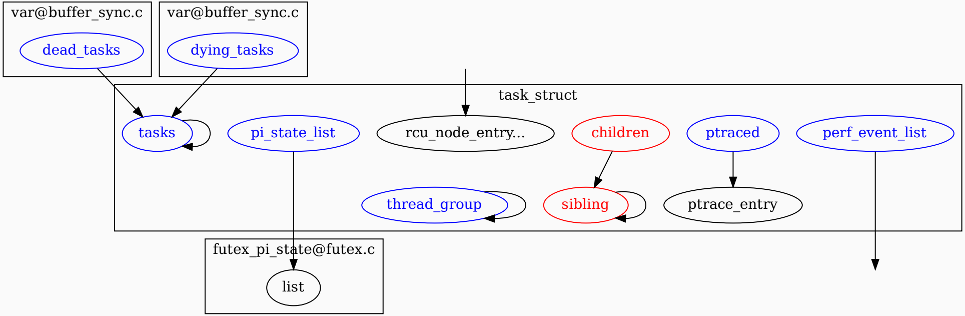
Trees


task_struct
ptrace_entry
ptraced
ptrace_entry
ptraced
ptrace_entry
ptraced

Trees


task_struct
ptrace_entry
ptraced
ptrace_entry
ptraced
ptrace_entry
ptraced
ptrace_entry
ptraced
ptrace_entry
ptraced

An umbrella tree!



- Simplings are sometimes accessed from the parent via `children`, and sometimes from the head of the list of siblings via `sibling`.
- `sibling` uses `group_leader` to find the head of the list of siblings.

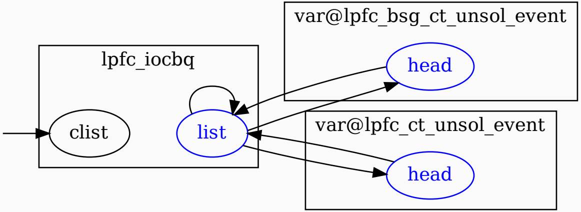
Rings

- Some self-loops really are loops, with no distinguished leader.
- Iteration becomes complex, because list iteration operators assume a head.
- Solution: add a head temporarily.

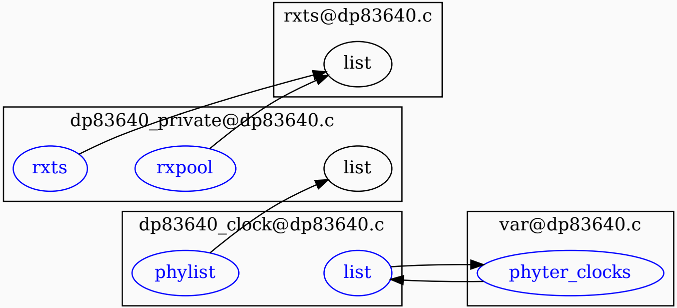
```
list_add_tail(&head, &piocbq->list);
list_for_each_entry(iocbq, &head, list) {
        icmd = &iocbq->iocb;
        if (icmd->ulpBdeCount == 0)
                lpfc_ct_unsol_buffer(phba, iocbq, NULL, 0);
        ...
}
list_del(&head);
```

Rings

- Some self-loops really are loops, with no distinguished leader.
- Iteration becomes complex, because list iteration operators assume a head.
- Solution: add a head temporarily.



Bugs!



- 6 new bugs found.
- Could have detected at least 8 out of 11 previous `list_add`/`list_add_tail` argument swap bugs.

Phyter bug in more detail

```
list_for_each(this, &phyter_clocks) {
        tmp = list_entry(this, struct dp83640_clock, list);
        if (tmp->bus == bus) {
                clock = tmp;
                break;
        }
}

list_for_each_safe(this, next, &phyter_clocks) {
        ...
}

list_add_tail(&phyter_clocks, &clock->list);
```

Phyter bug in more detail

```
list_for_each(this, &phyter_clocks) {
        tmp = list_entry(this, struct dp83640_clock, list);
        if (tmp->bus == bus) {
                clock = tmp;
                break;
        }
}

list_for_each_safe(this, next, &phyter_clocks) {
        ...
}

list_add_tail(&phyter_clocks, &clock->list);
```

Conclusion

- Simple type system for lists, distinguishing heads and elements.
- Tool for visualizing list types.
- Tool for collecting list uses.

Conclusion

- Simple type system for lists, distinguishing heads and elements.
- Tool for visualizing list types.
- Tool for collecting list uses.

- Are there other patterns besides umbrellas, trees, and rings?
- Are there other C types that need higher-level descriptions?
- Could these types be enforced, e.g. to avoid list_add argument swap bugs?
- If not enforced, should they be systematically documented?

https://gitlab.inria.fr/lawall/liliput