Linux-CR: Transparent Application Checkpoint-Restart in Linux

Oren Laadan
Columbia University
orenl@cs.columbia.edu

Serge E. Hallyn
IBM
serge@hallyn.com

Application C/R

- Application Checkpoint/Restart:

  a mechanism to save the state of a running application so that it can later resume its execution from that point

What is it good for ?

- Application roll back to the past
  - recover from faults
  - effective debugging
  - improved response time
  - retry a move in a game

What else is it good for ?

- Application suspend and resume
  - improved system utilization
  - suspend/resume a user's session

What more is it good for ?

- Application migration
  - load balancing and resource sharing
  - mobile desktop on a USB key
  - zero-downtime maintenance
  - improved availability

Application vs Virtual-Machine

	Application C/R	Virtual Machine
granularity	specific applications	operating system as a whole unit
saved state	application state only	entire operating system state
overhead	none	visible overhead

Some History

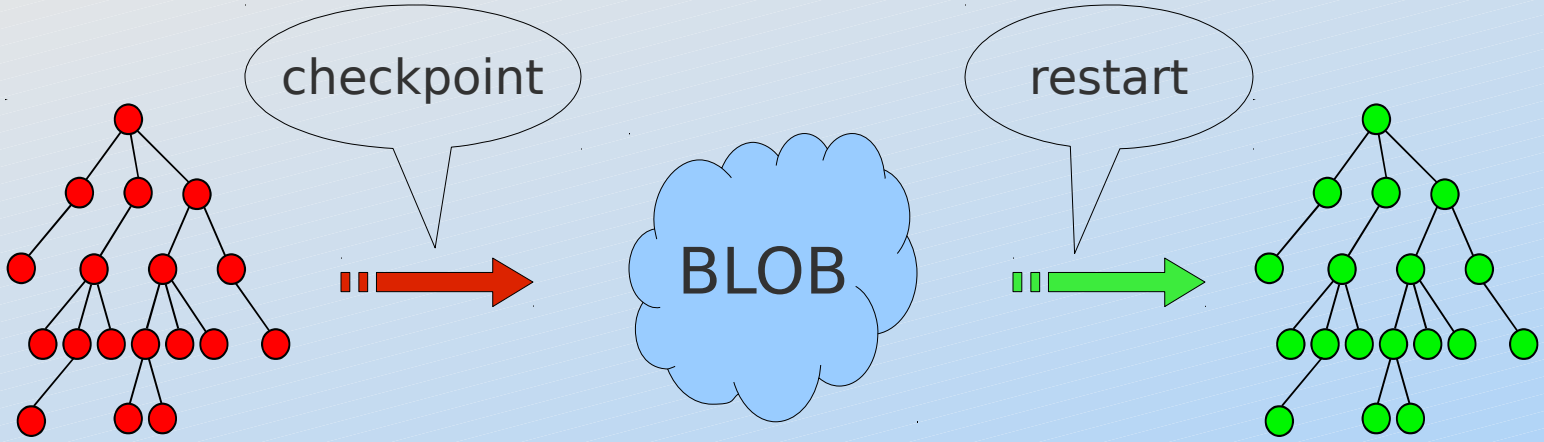
- Linux 2.4
  - EPCKPT (Rutgers)
  - CRAK (Columbia)
- Linux 2.6
  - BLCR (Berkeley)
  - OpenVZ (Parallels)
  - Zap (Columbia)

Requirements

- Reliable
  - if checkpoint succeeds – restart succeeds
- Transparent
  - applications are oblivious to operation
- Secure
  - must not introduce vulnerabilities
- Mainline
  - aim for inclusion in mainline kernel

Usage Model

- Checkpoint granularity
  - a process hierarchy
  - top-down traversal



Checkpoint Categories

- Container-checkpoint
- Subtree-checkpoint
- Self-checkpoint

Namespaces

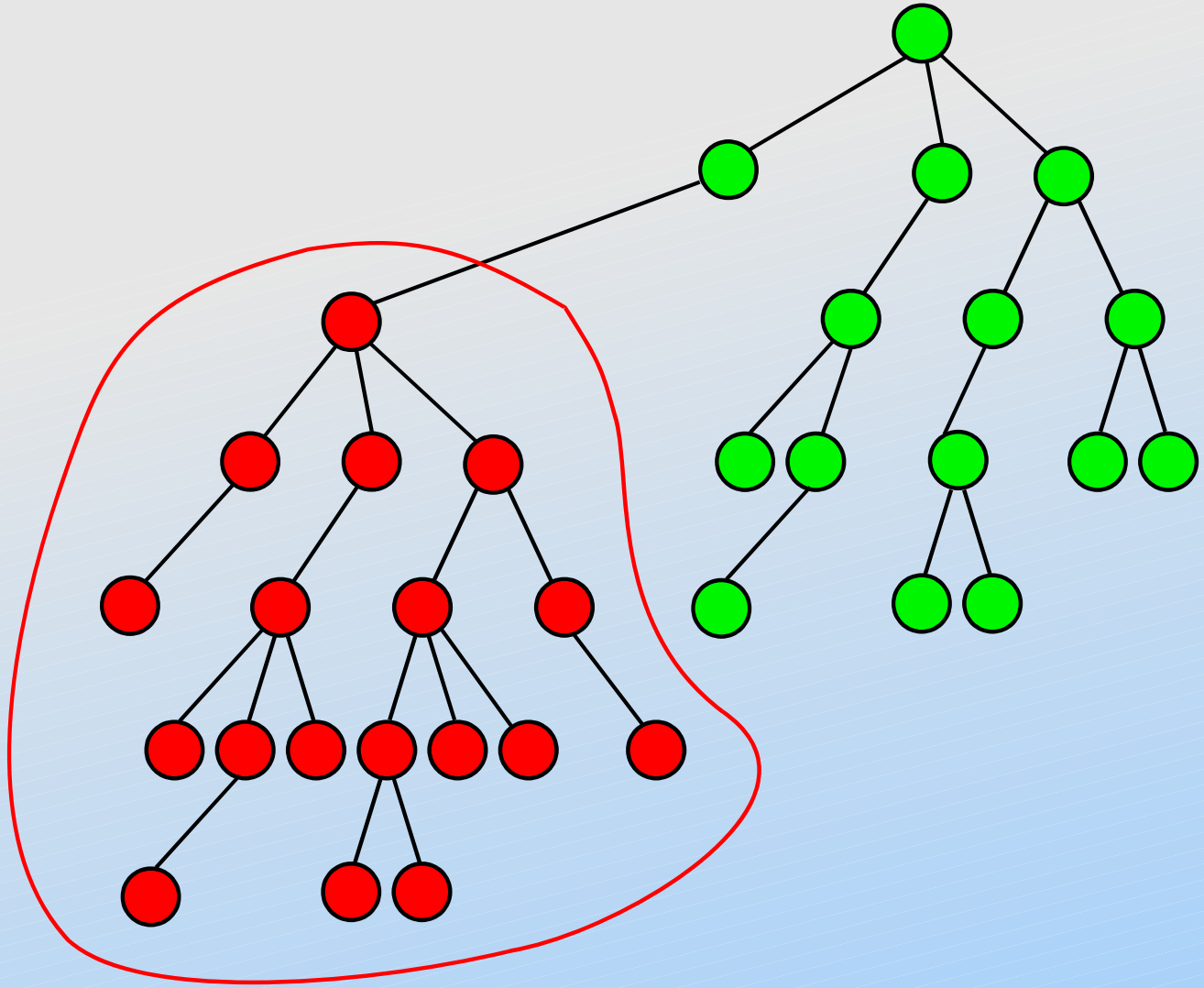
- Private and virtual view of resources
  - e.g. pid, mount, ipc, network...
- Private view
  - provide isolation from other processes
- Virtual view
  - decouple from underlying kernel instance

Container Checkpoint

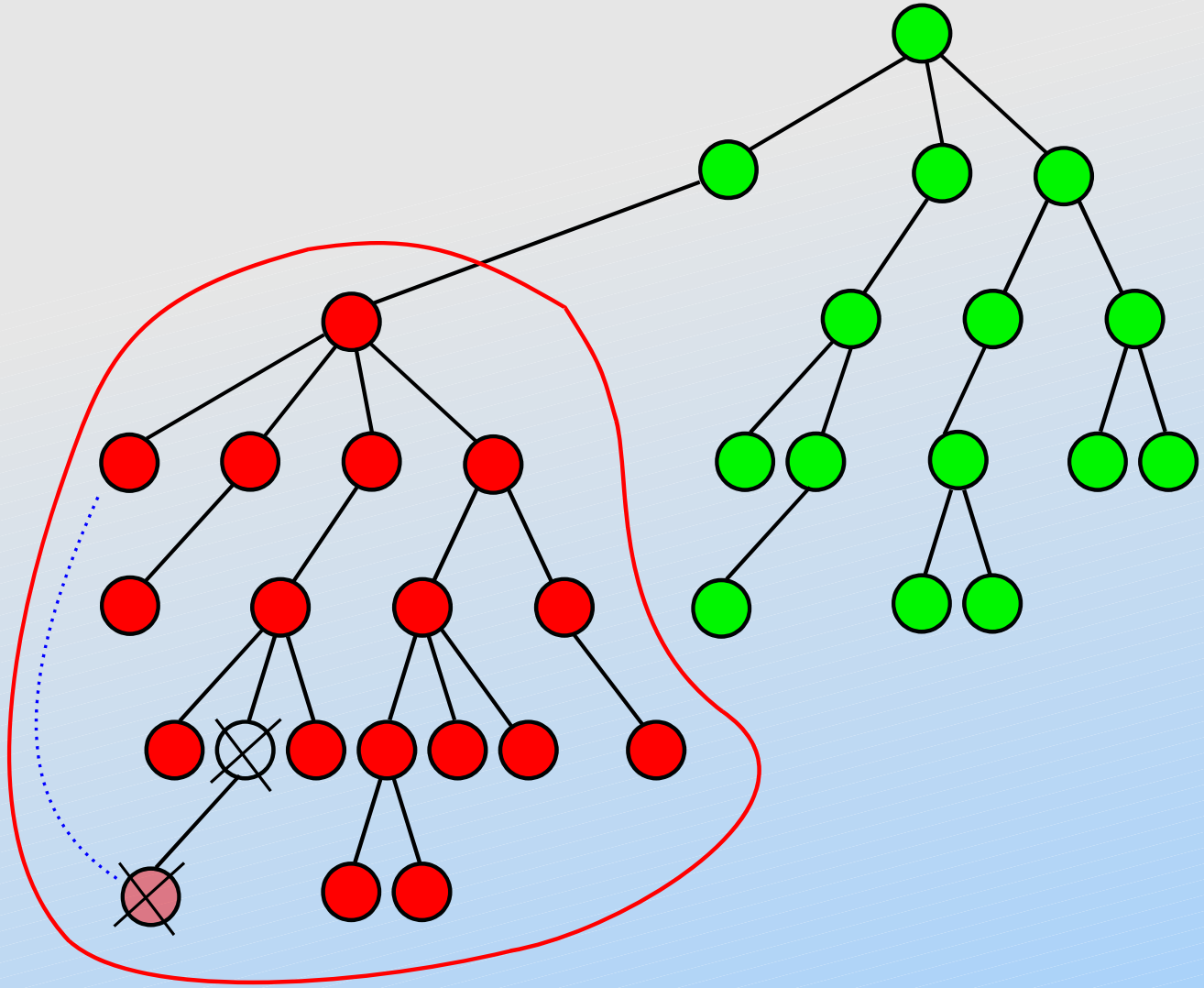
- Hierarchy is self-contained
  - includes all the processes that are referenced within the hierarchy
- Hierarchy is isolated
  - resources only referenced by processes that belong to the hierarchy

- Checkpoint is consistent and reliable

Container Checkpoint



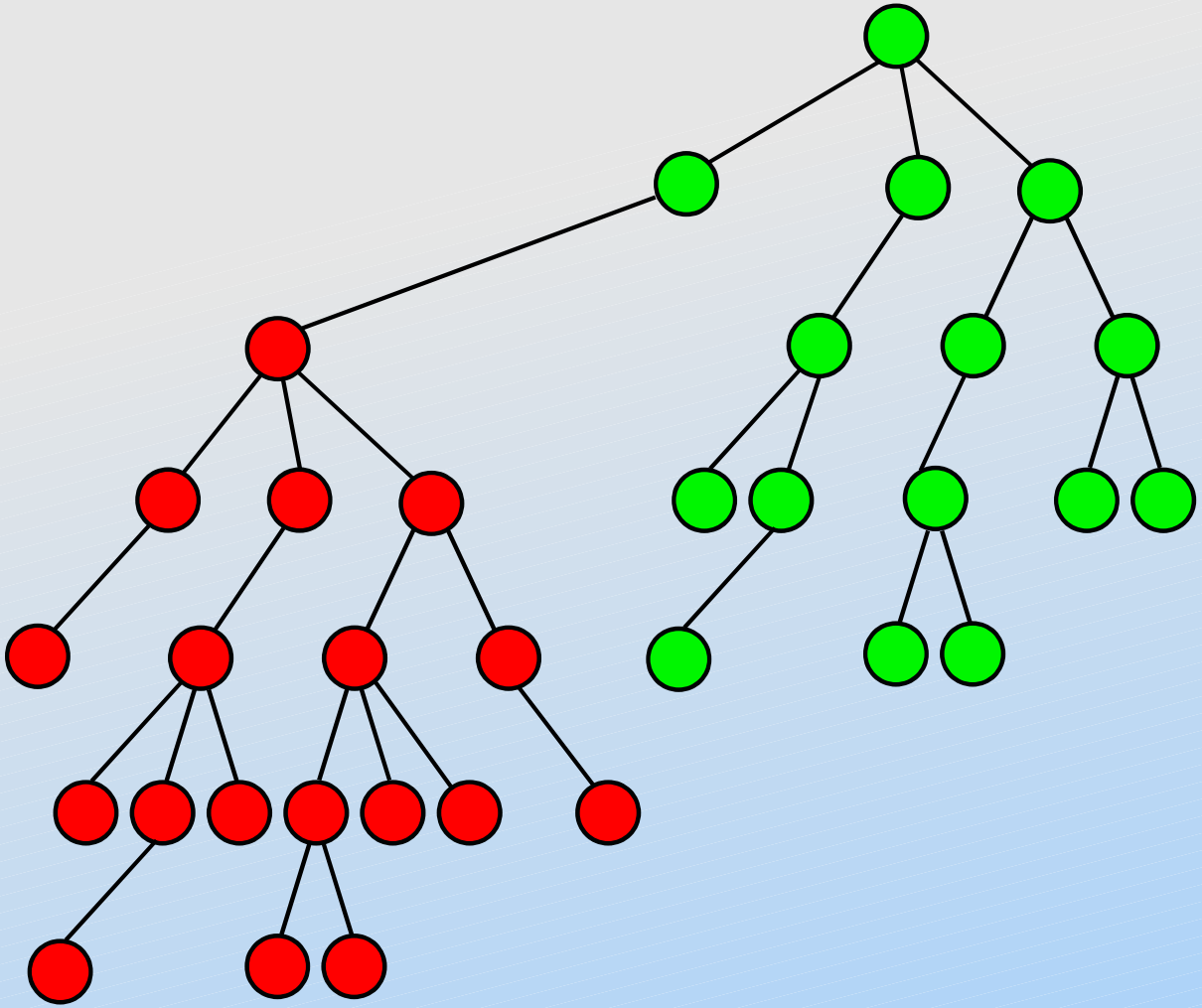
Container Checkpoint



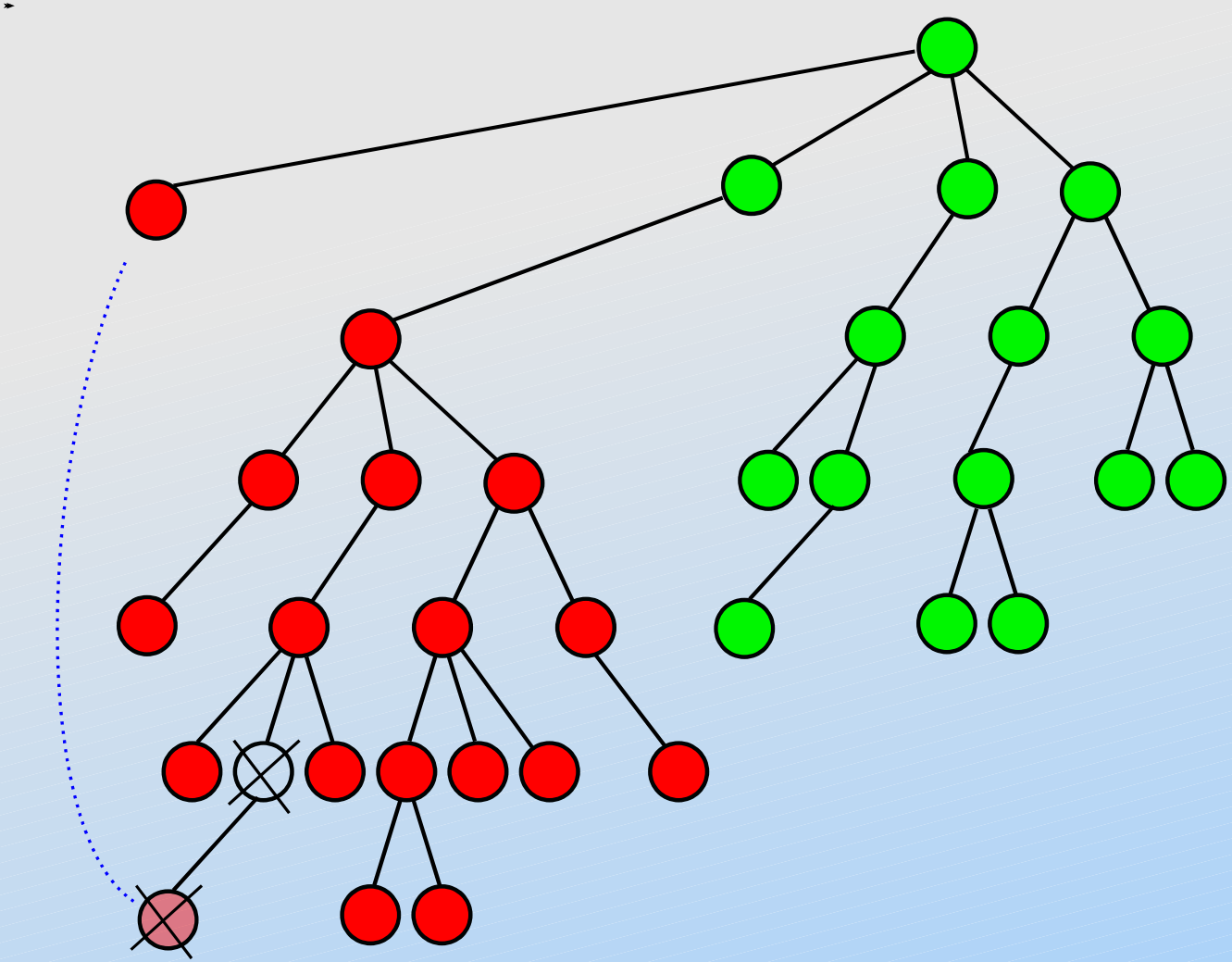
Subtree Checkpoint

- Arbitrary process hierarchy
  - no constraints on the target hierarchy
  - simplifies admin, but no guarantees
  - suitable for many use-cases

Subtree Checkpoint



Subtree Checkpoint



Self-Checkpoint

- For a process to save its state
  - record only current process
  - ignore sharing and dependencies
  - Analogous to *fork* syscall:

```
…
ret = checkpoint(0, fd, flags, -1);
if (ret < 0)
   return ret;
else if (ret)
   printf("checkpoint succeeded\n");
else
   printf("returned from restart\n");
…
```

System Calls

`long checkpoint(pid, fd, flags, logfd)`

- target hierarchy with root task `@pid`
- output to `@fd`, log to `@logfd`

`long restart(pid, fd, flags, logfd)`

- New hierarchy with coordinator `@pid`
- Input from `@fd`, log to `@logfd`

Example

```
cat > myscript.sh << EOF
#!/bin/sh
echo $$ > /cgroup/1/tasks
exec 0>&- ; exec 1>&- ; exec 2>&-
/usr/sbin/sshd -p 9999
screen -A -d -m -S mysession somejob.sh
EOF
```

Example

```
cat > myscript.sh << EOF
#!/bin/sh
echo $$ > /cgroup/1/tasks
exec 0>&- ; exec 1>&- ; exec 2>&-
/usr/sbin/sshd -p 9999
screen -A -d -m -S mysession somejob.sh
EOF

mkdir -p /cgroup
mount -t cgroup -o freezer cgroup /cgroup
mkdir /cgroup/1
```

Example

```
cat > myscript.sh << EOF
#!/bin/sh
echo $$ > /cgroup/1/tasks
exec 0>&- ; exec 1>&- ; exec 2>&-
/usr/sbin/sshd -p 9999
screen -A -d -m -S mysession somejob.sh
EOF

mkdir -p /cgroup
mount -t cgroup -o freezer cgroup /cgroup
mkdir /cgroup/1

nohup nsexec -tgcmpiUP pid.out myscript.sh &
```

Example

```
cat > myscript.sh << EOF
#!/bin/sh
echo $$ > /cgroup/1/tasks
exec 0>&- ; exec 1>&- ; exec 2>&-
/usr/sbin/sshd -p 9999
screen -A -d -m -S mysession somejob.sh
EOF

mkdir -p /cgroup
mount -t cgroup -o freezer cgroup /cgroup
mkdir /cgroup/1

nohup nsexec -tgcmpiUP pid.out myscript.sh &

PID=`cat pid.out`
echo FROZEN >  /cgroup/1/freezer.state
checkpoint $PID -l clog.out -o image.out
kill -9 $PID
echo THAWED > /cgroup/1/freezer.state
```

Example

```
cat > myscript.sh << EOF
#!/bin/sh
echo $$ > /cgroup/1/tasks
exec 0>&- ; exec 1>&- ; exec 2>&-
/usr/sbin/sshd -p 9999
screen -A -d -m -S mysession somejob.sh
EOF

mkdir -p /cgroup
mount -t cgroup -o freezer cgroup /cgroup
mkdir /cgroup/1

nohup nsexec -tgcmpiUP pid.out myscript.sh &

PID=`cat pid.out`
echo FROZEN >  /cgroup/1/freezer.state
checkpoint $PID -l clog.out -o image.out
kill -9 $PID
echo THAWED > /cgroup/1/freezer.state

restart -l rlog.out -i image.out
```

Architecture

- Reliability
- Transparency
- Kernel vs userspace
- Checkpoint image
- Shared resources
- Leak detection

Architecture: Reliability

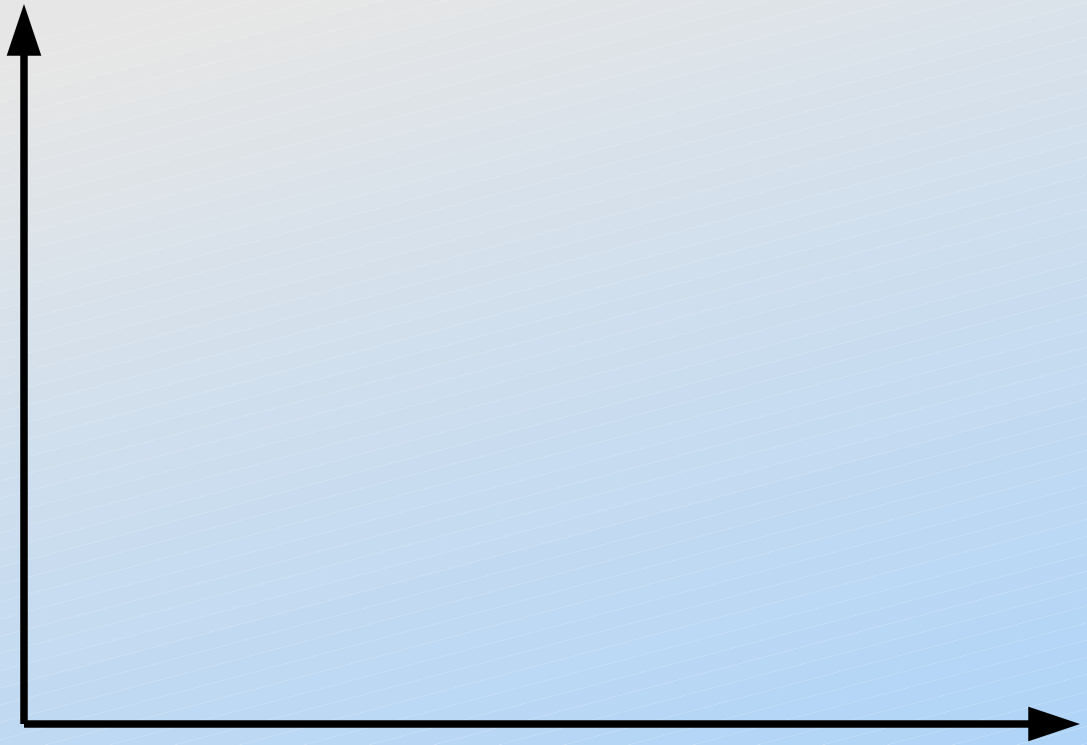
- How to maintain global consistency ?
- Requirements:
  - keep tasks frozen
  - keep resources unmodified
- Outcome: state is protected
  - from tasks in the hierarchy
  - from tasks outside the hierarchy

Architecture: Transparency

- How to maintain transparency ?
- Requirements:
  - include all resources in use by tasks
  - preserve resources identifiers on restart
- Outcome: state visible as before
  - all necessary state is restored
  - state accessible via same identifiers

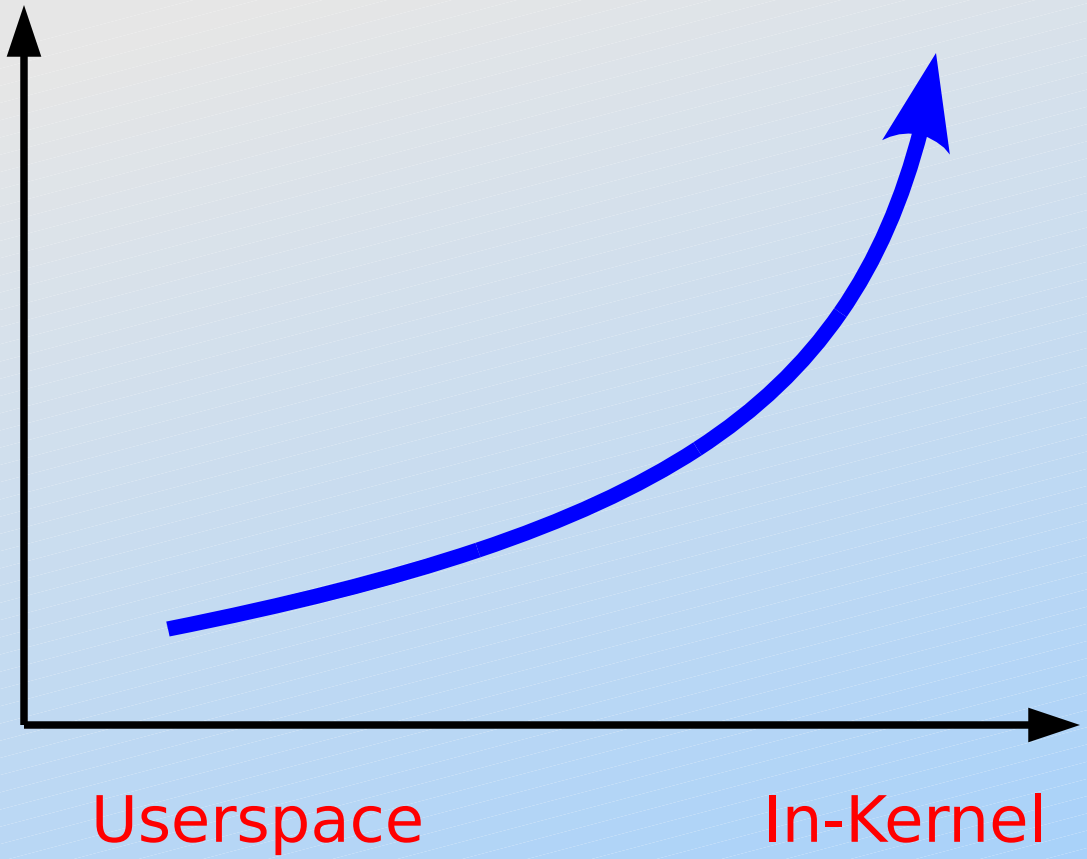
Kernel vs. Userspace

Completeness
Transparency
Extensibility



Kernel vs. Userspace

Completeness
Transparency
Extensibility



Kernel vs. Userspace

- The rule: in-kernel implementation
  - transparency, completeness
  - leverage extensive kernel API
- The exception: userspace possible
  - if straightforward with existing APIs
  - if provides significant added value
  - If occurs before entering the kernel

Checkpoint

(1) Freeze process hierarchy

(2) Save global data

(3) Save process hierarchy

(4) Save state of all tasks

In-Kernel

(?) Filesystem snapshot

(5) Thaw/kill process hierarchy

Restart

(1) Create container

(?) Restore (stage) filesystem

(3) Create process hierarchy

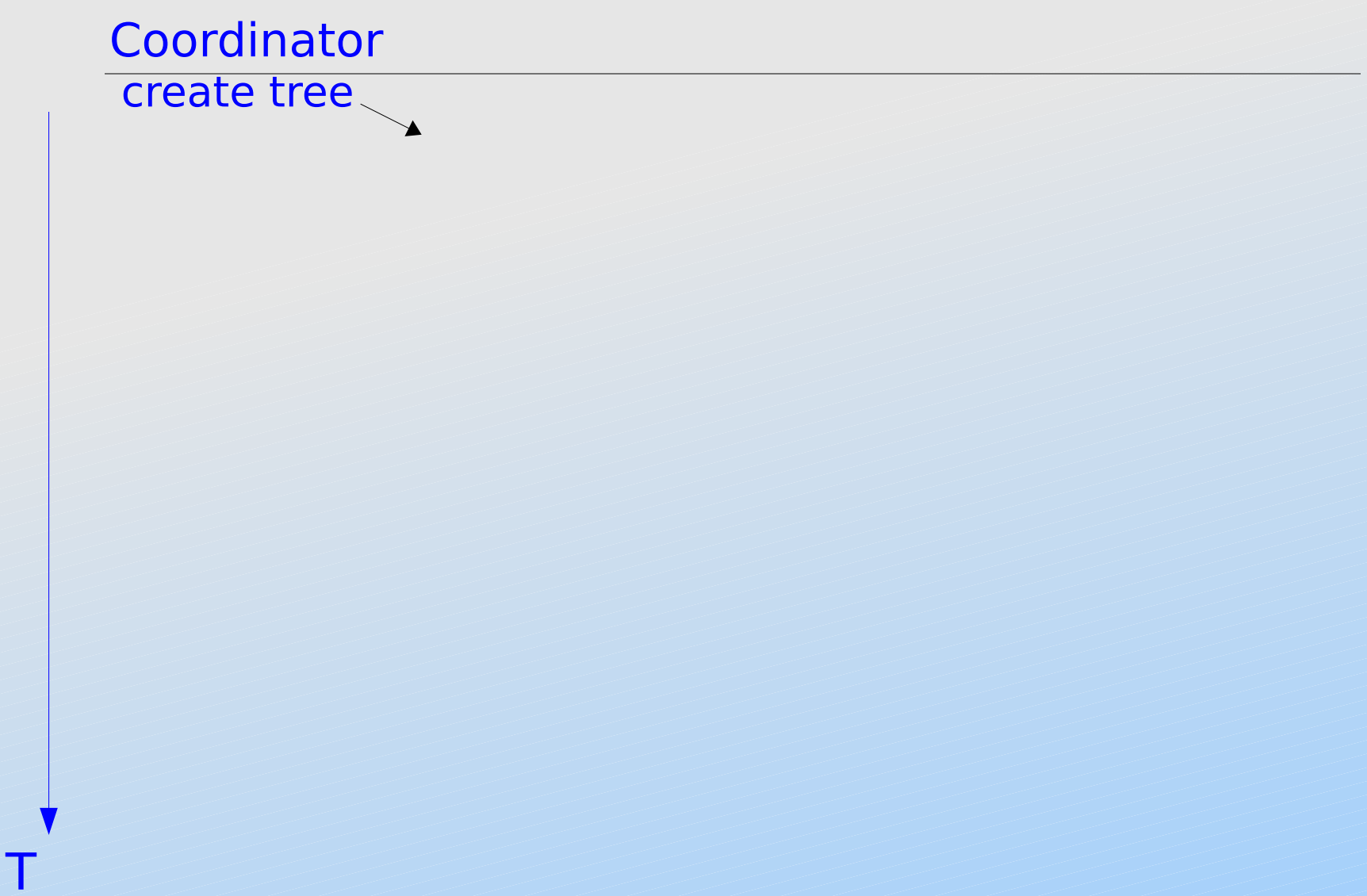
(4) Restore state of all tasks } In-Kernel

(5) Resume execution

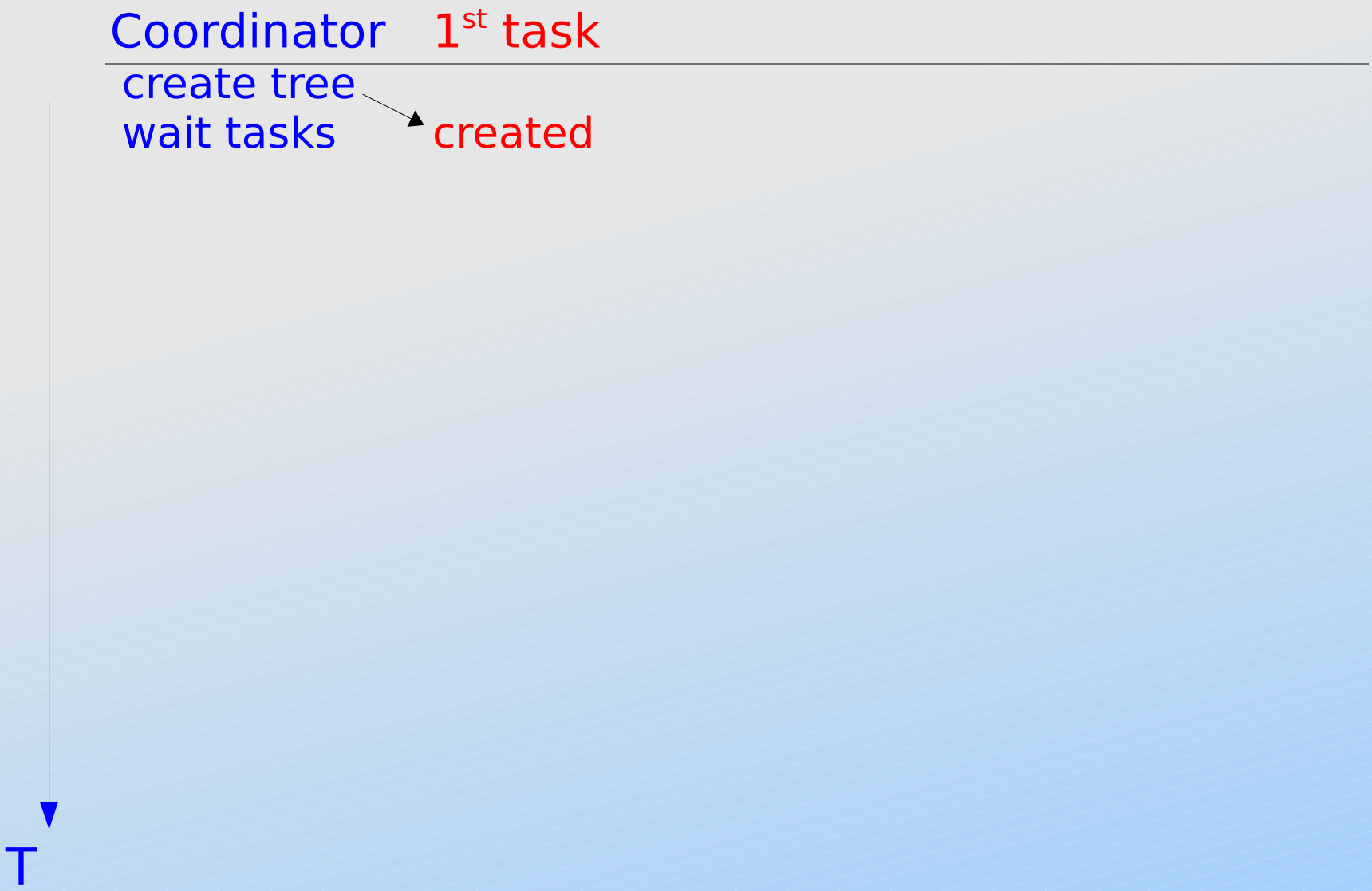
Restart: Create Hierarchy

- *DumpForest*
  - convert hierarchy data to instructions
- *CreateForest*
  - execute instructions to re-create tasks
  - proceeds from root task recursively
- Curious how it works ?
  - see USENIX 2007 paper (Zap)
  - read comments in code

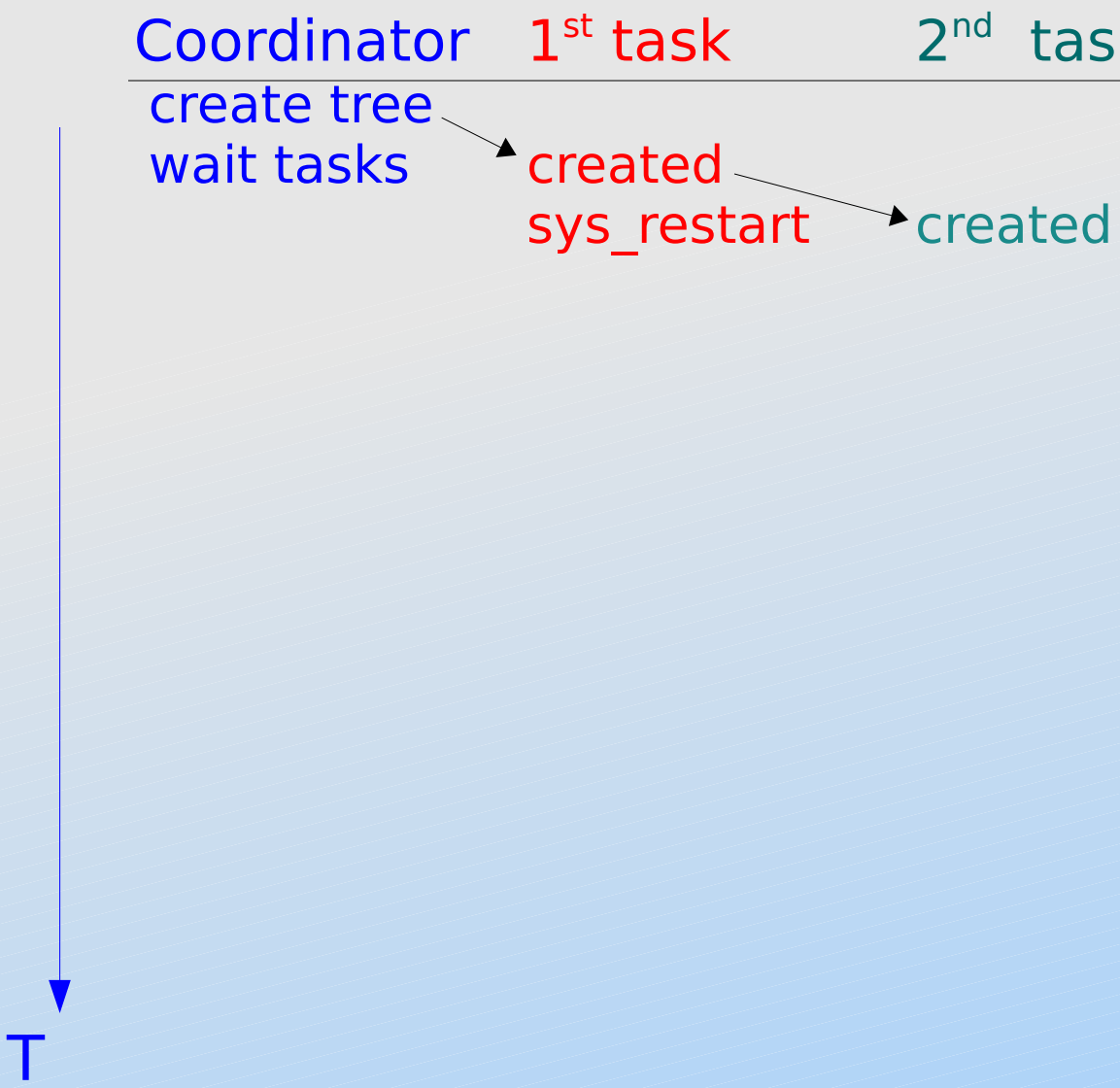
Restart Coordination



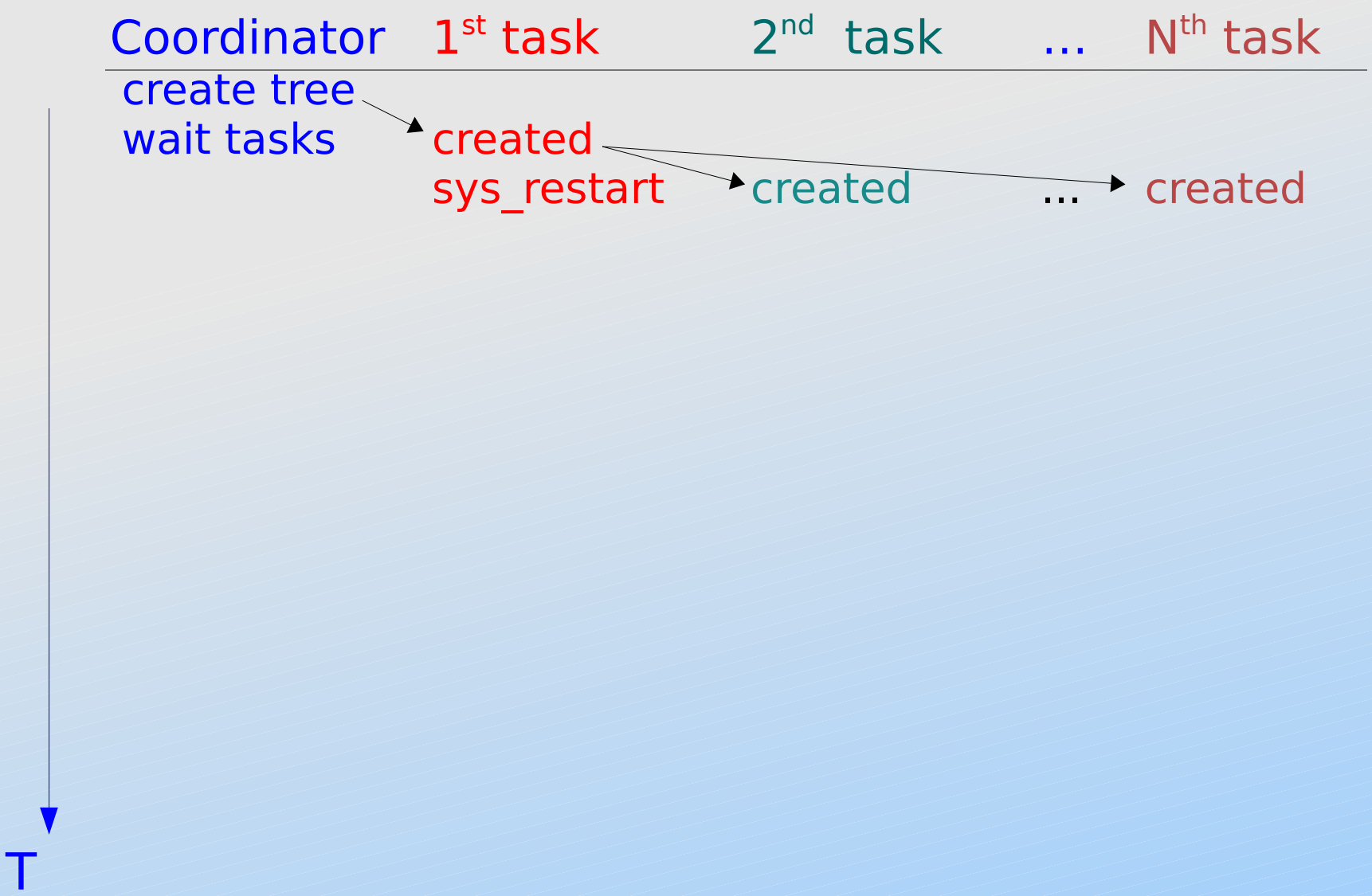
Restart Coordination



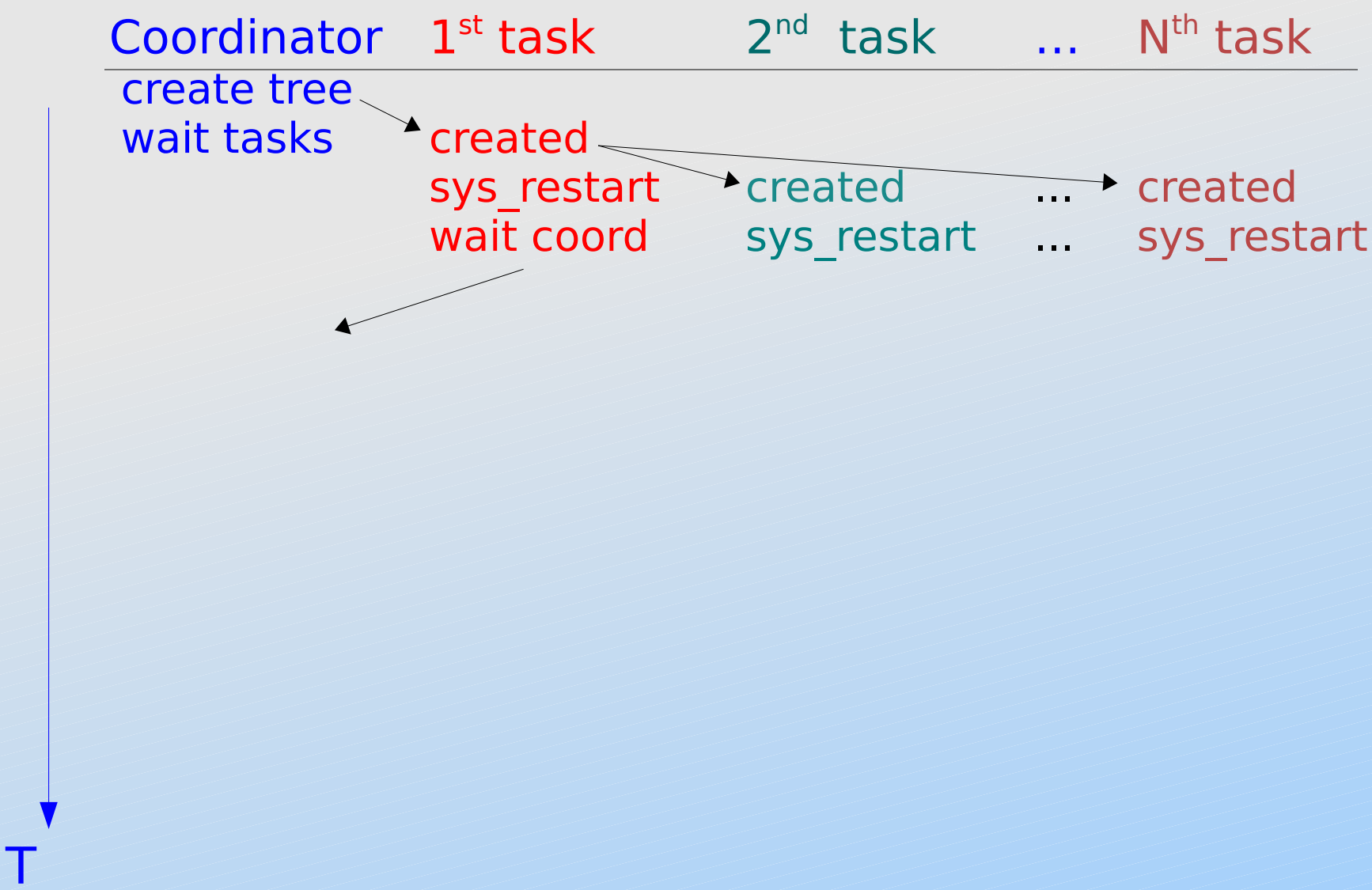
Restart Coordination



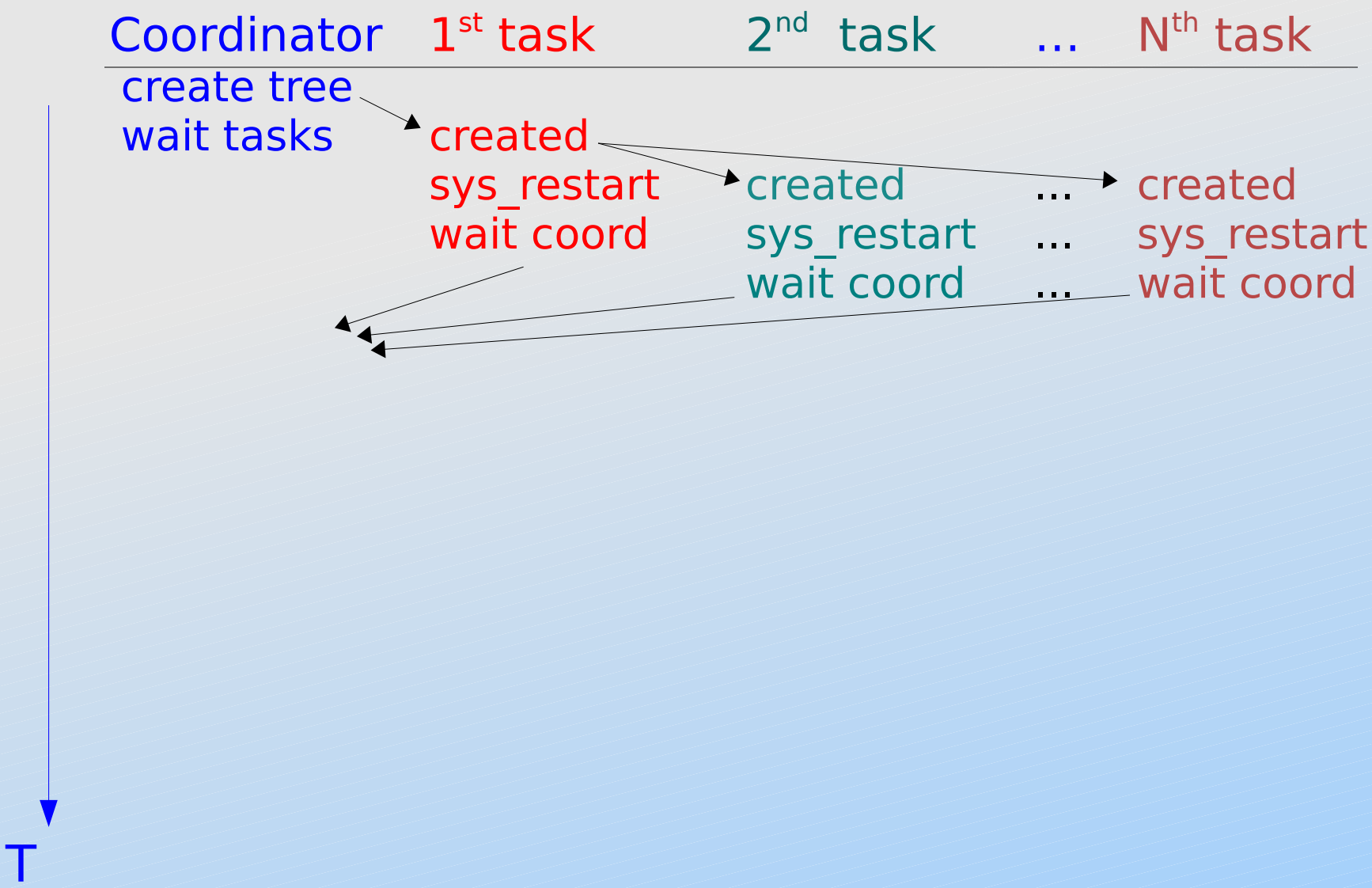
Restart Coordination



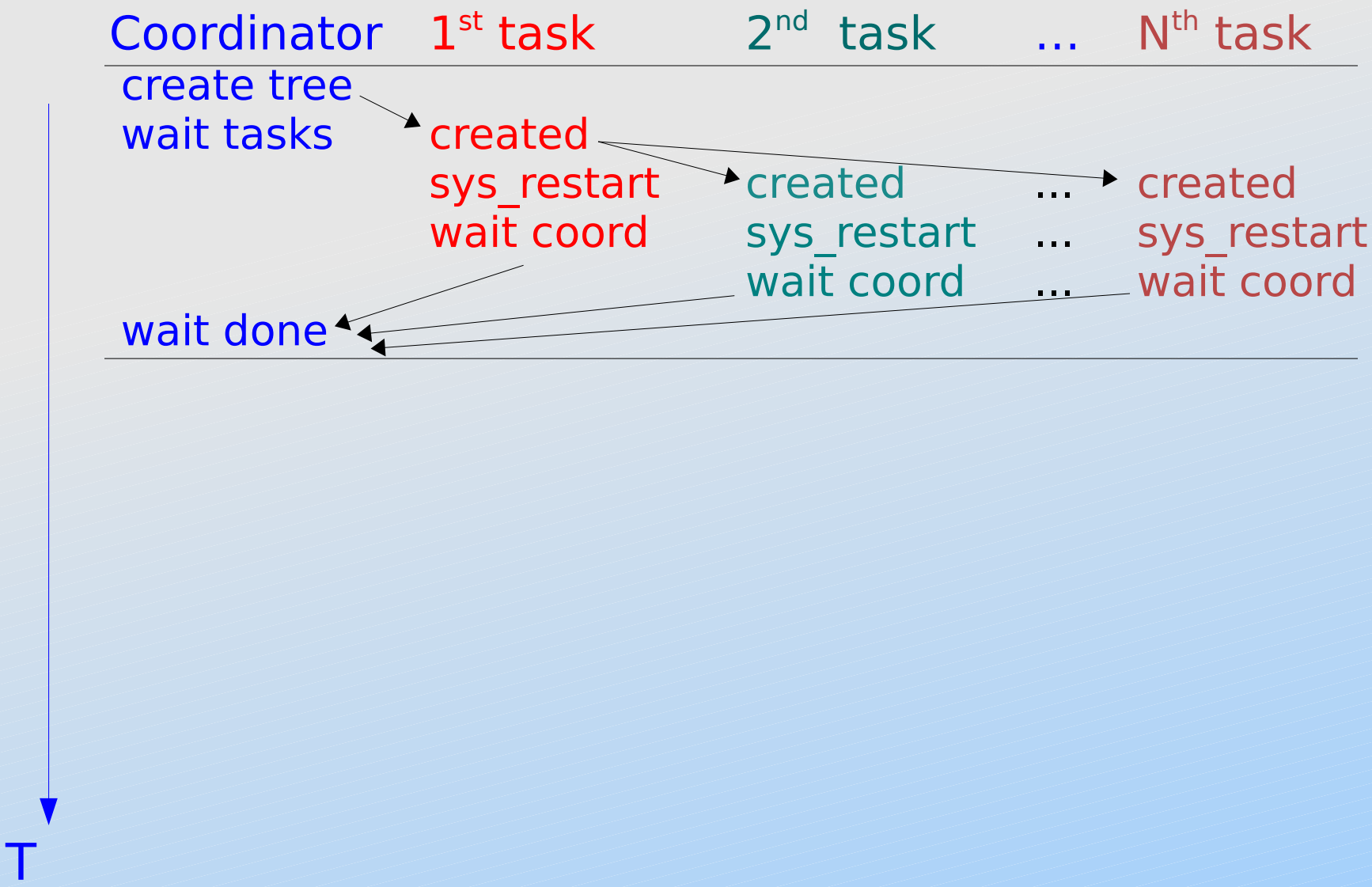
Restart Coordination



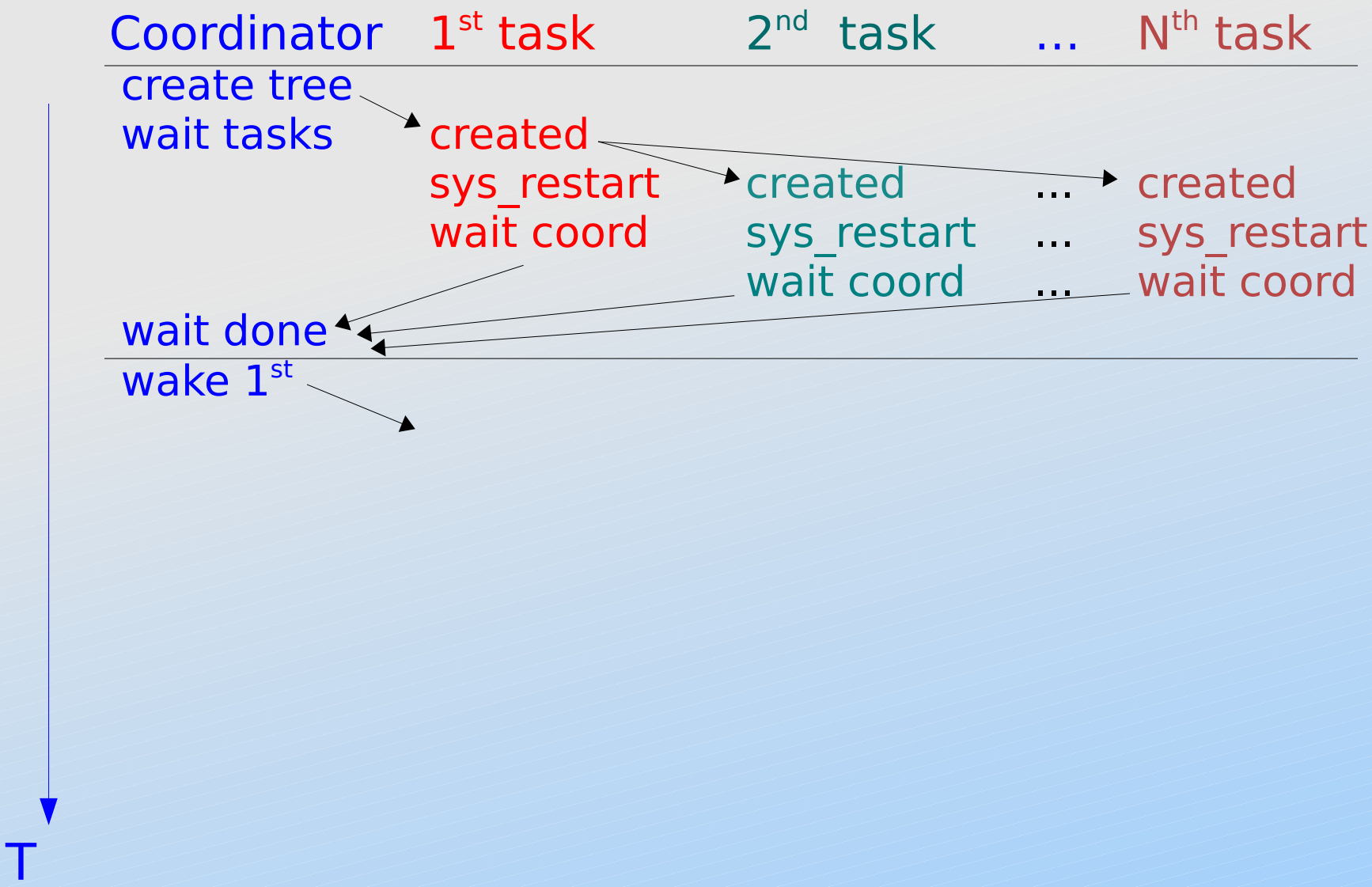
Restart Coordination



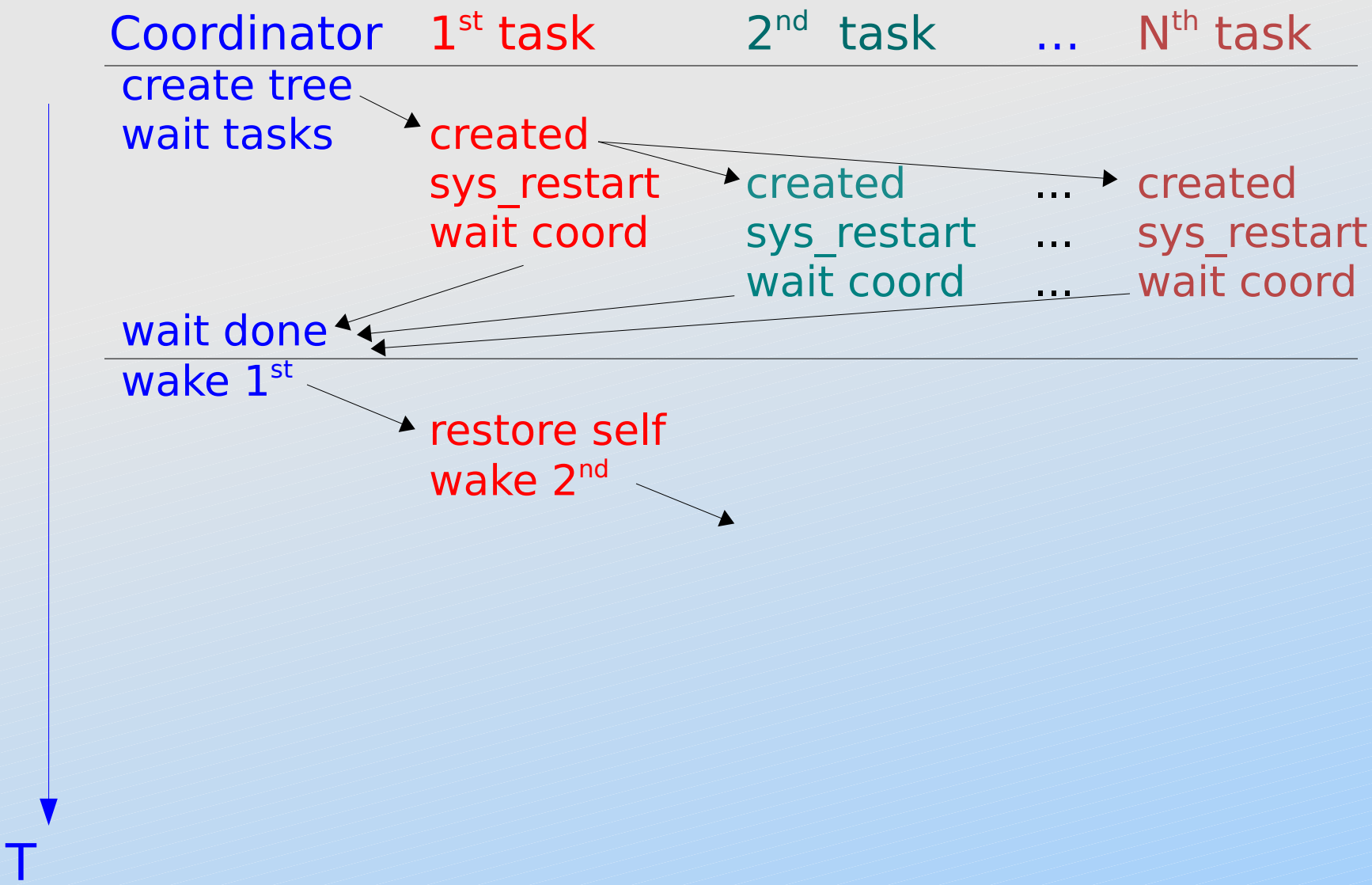
Restart Coordination



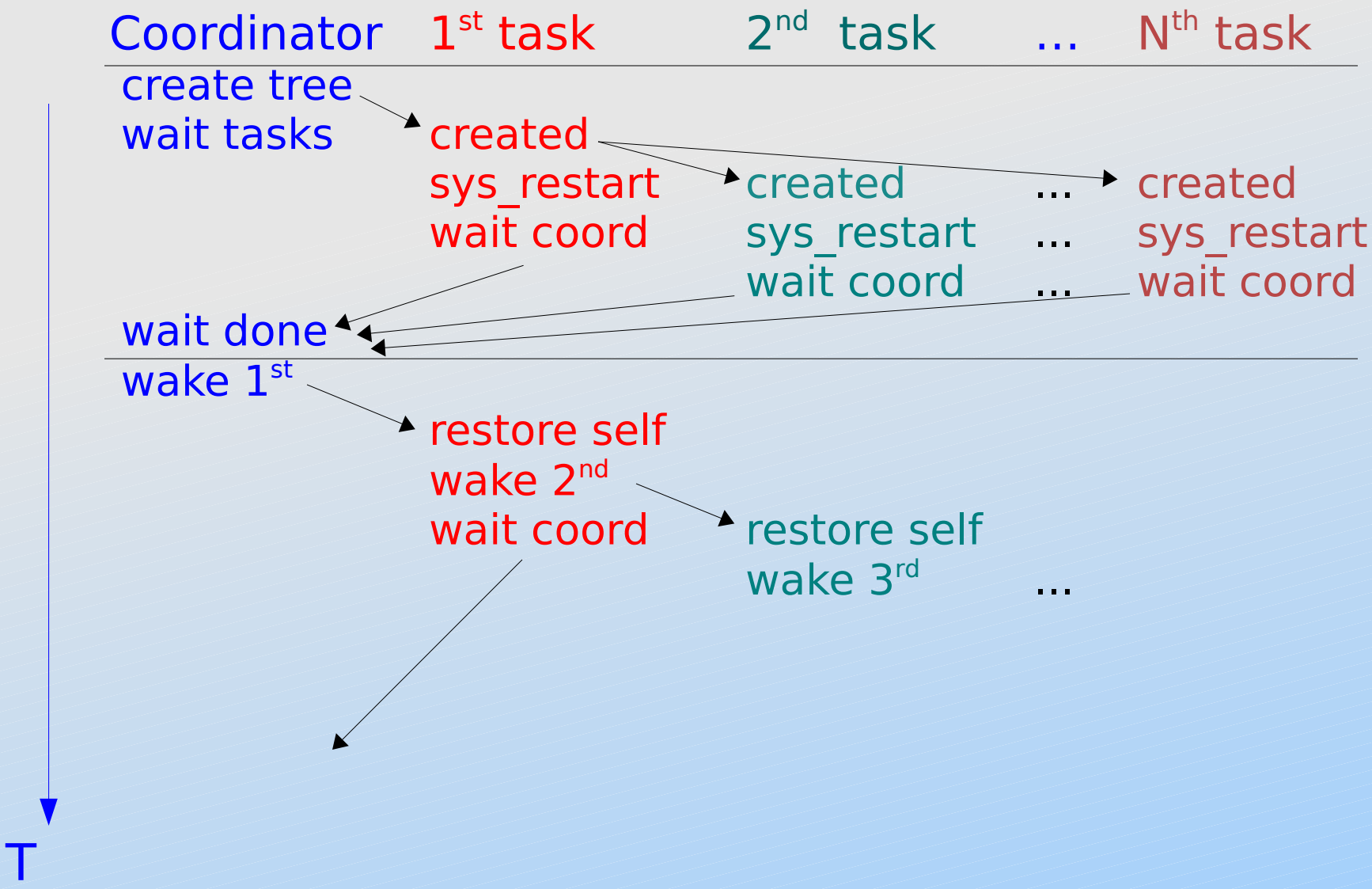
Restart Coordination



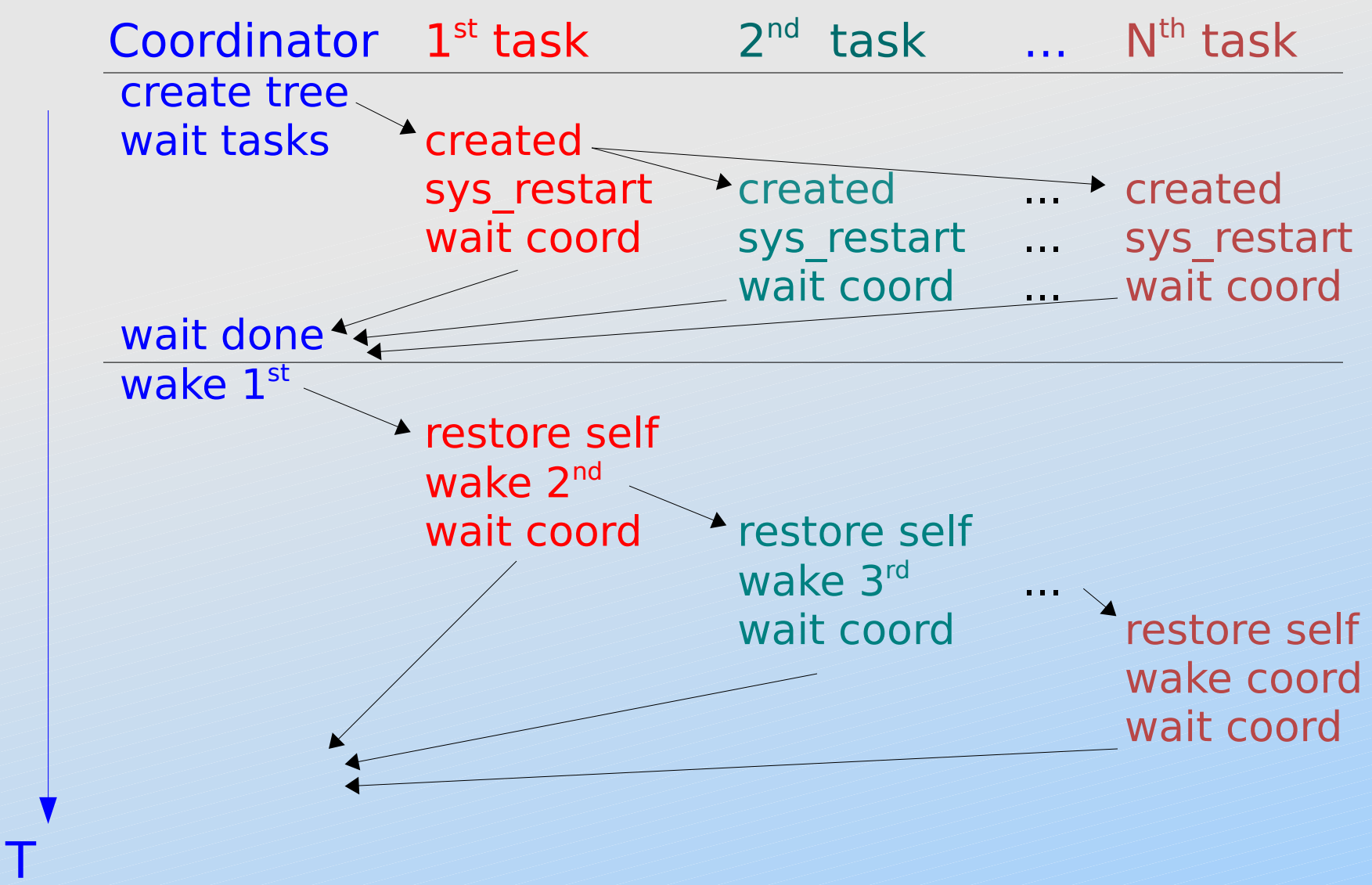
Restart Coordination



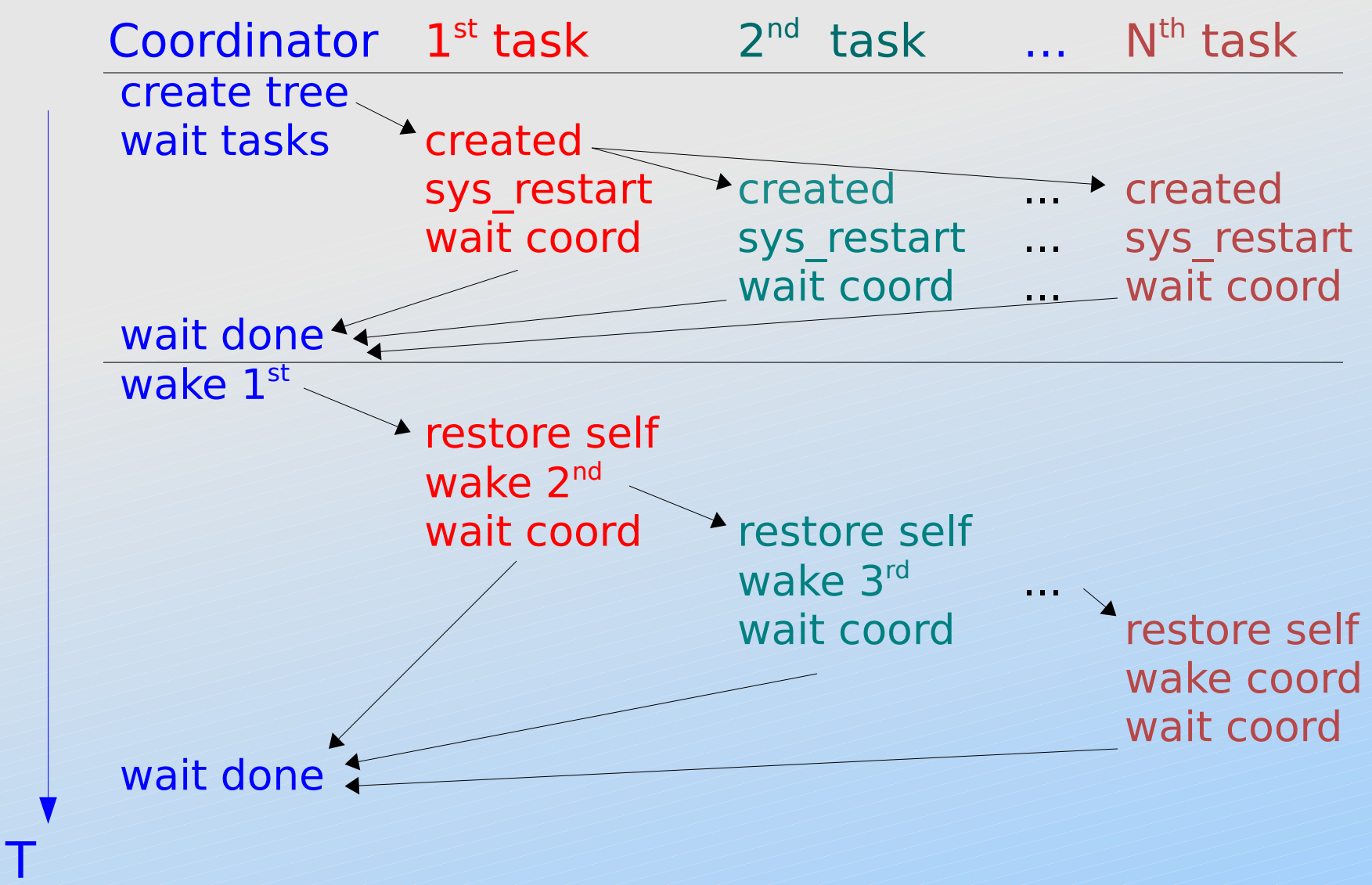
Restart Coordination



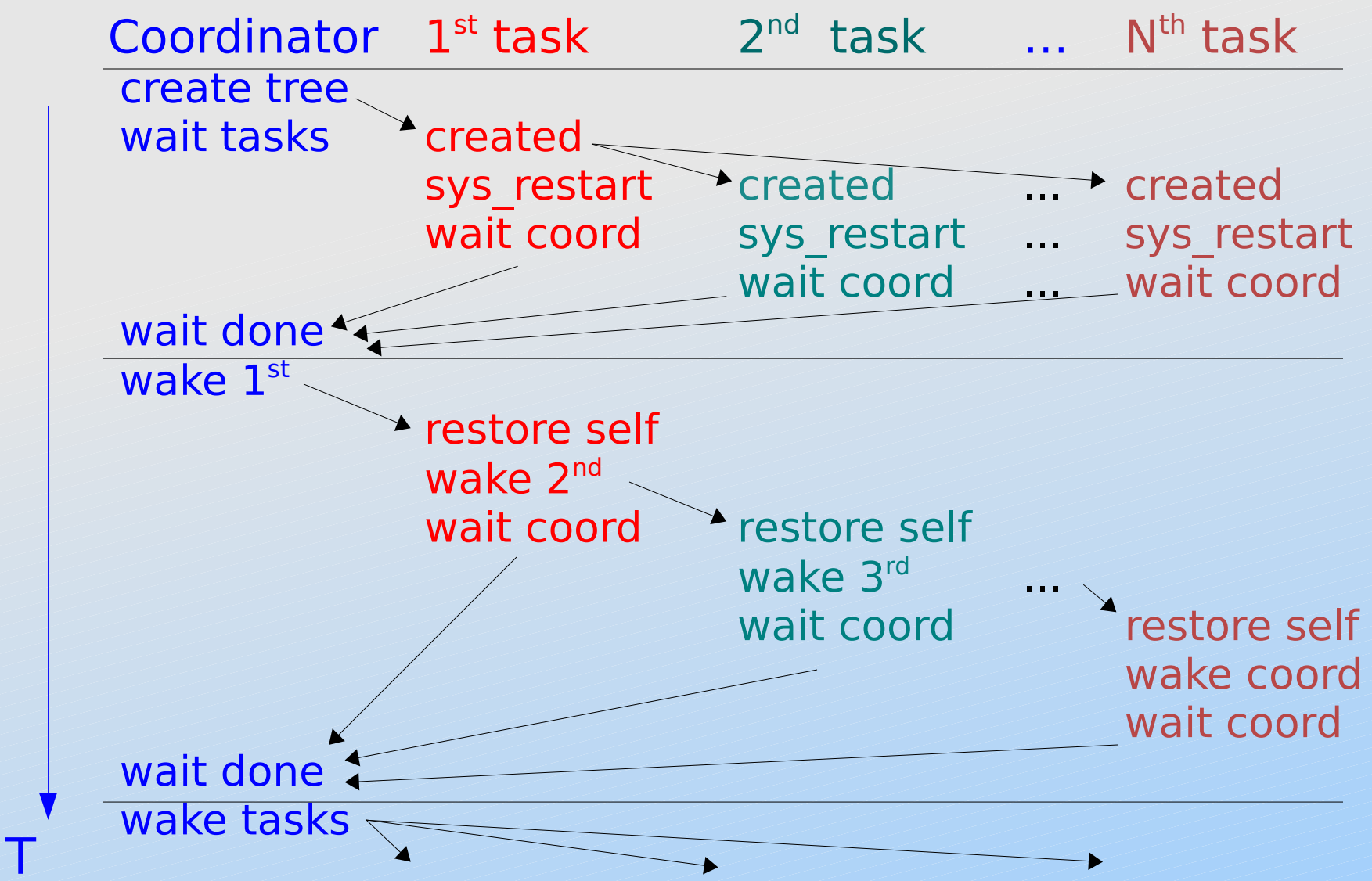
Restart Coordination



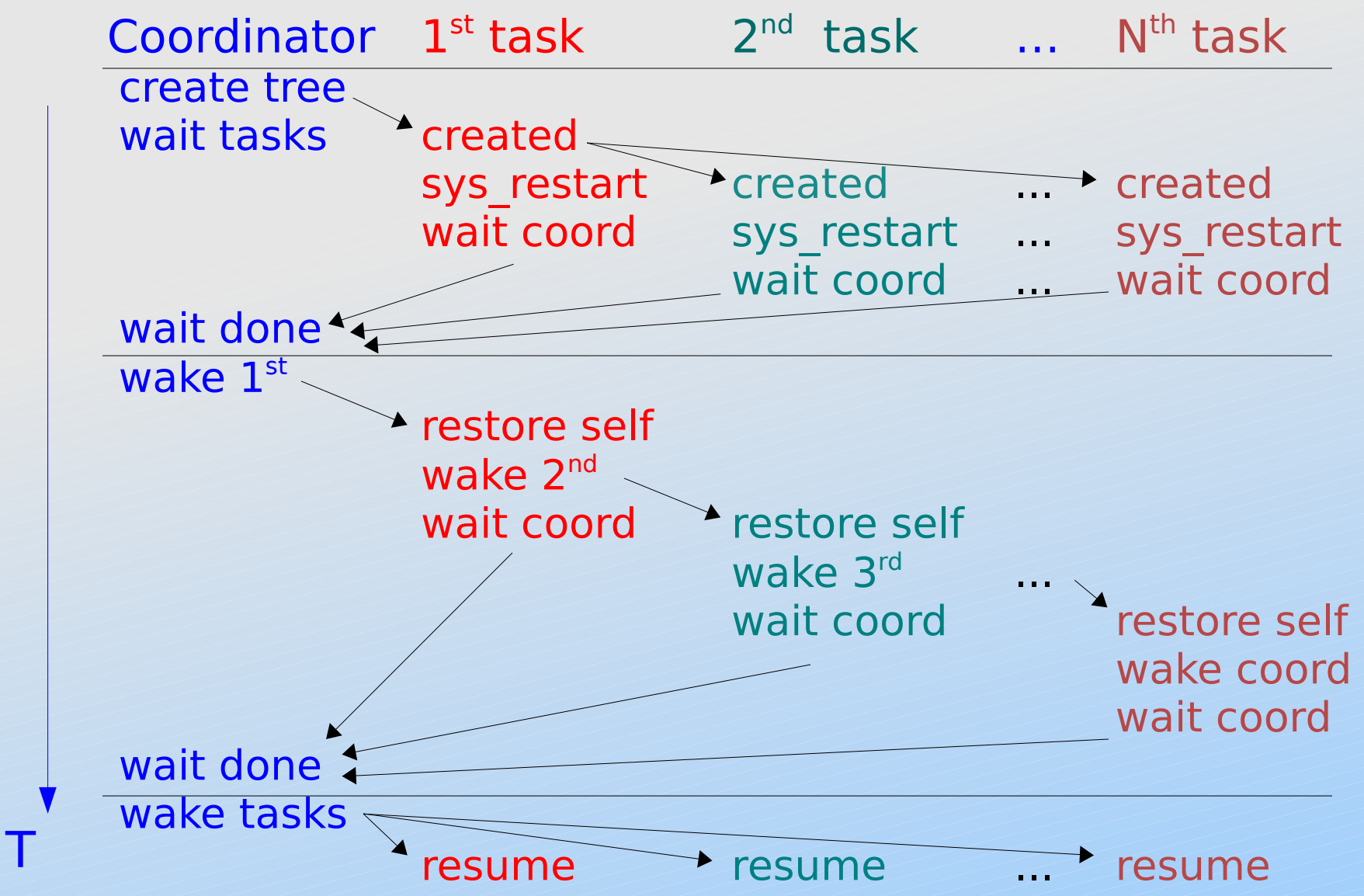
Restart Coordination



Restart Coordination



Restart Coordination



Checkpoint/Restart Compared

Checkpoint	Restart
auxiliary process	restore in context
tasks are passive	tasks participate
detect non-restartable	detect non-secure
non-intrusive (errors)	cleanup (errors)

Checkpoint Image

- The image is a BLOB
  - internals may change over time
  - conversion to be done in userspace
- Designed for streaming
  - for migration, or for image filters: sign, compress, encrypt, convert, etc.

Checkpoint Image

- Representation of kernel data
  - already need to inspect on restart
  - compatibility across kernel versions
  - does not save unnecessary fields
  - unified format for 32/64 bit architectures

Checkpoint Image

- a sequence of object records
- records have header and payload

```
struct ckpt_hdr {
   __u32 type;
   __u32 len;
};

struct ckpt_hdr_task {
   struct ckpt_hdr h;
   __u32 state;
   ...
};
```

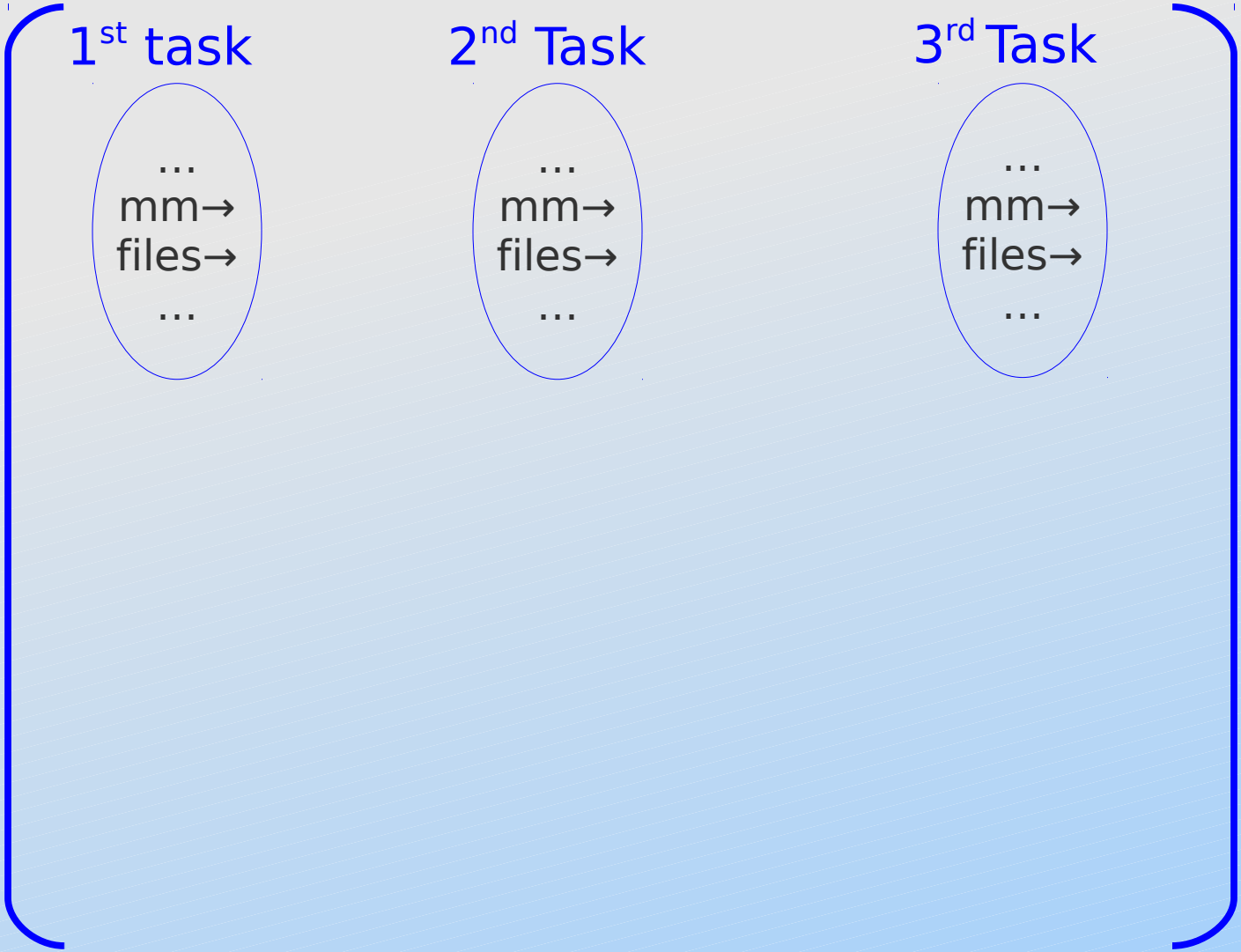
Shared Resources

- Resources in use by multiple tasks
  - open files, namespaces, signals, handlers, memory descriptor
  - only checkpoint/restore once each
  - use a hash-table to track instances

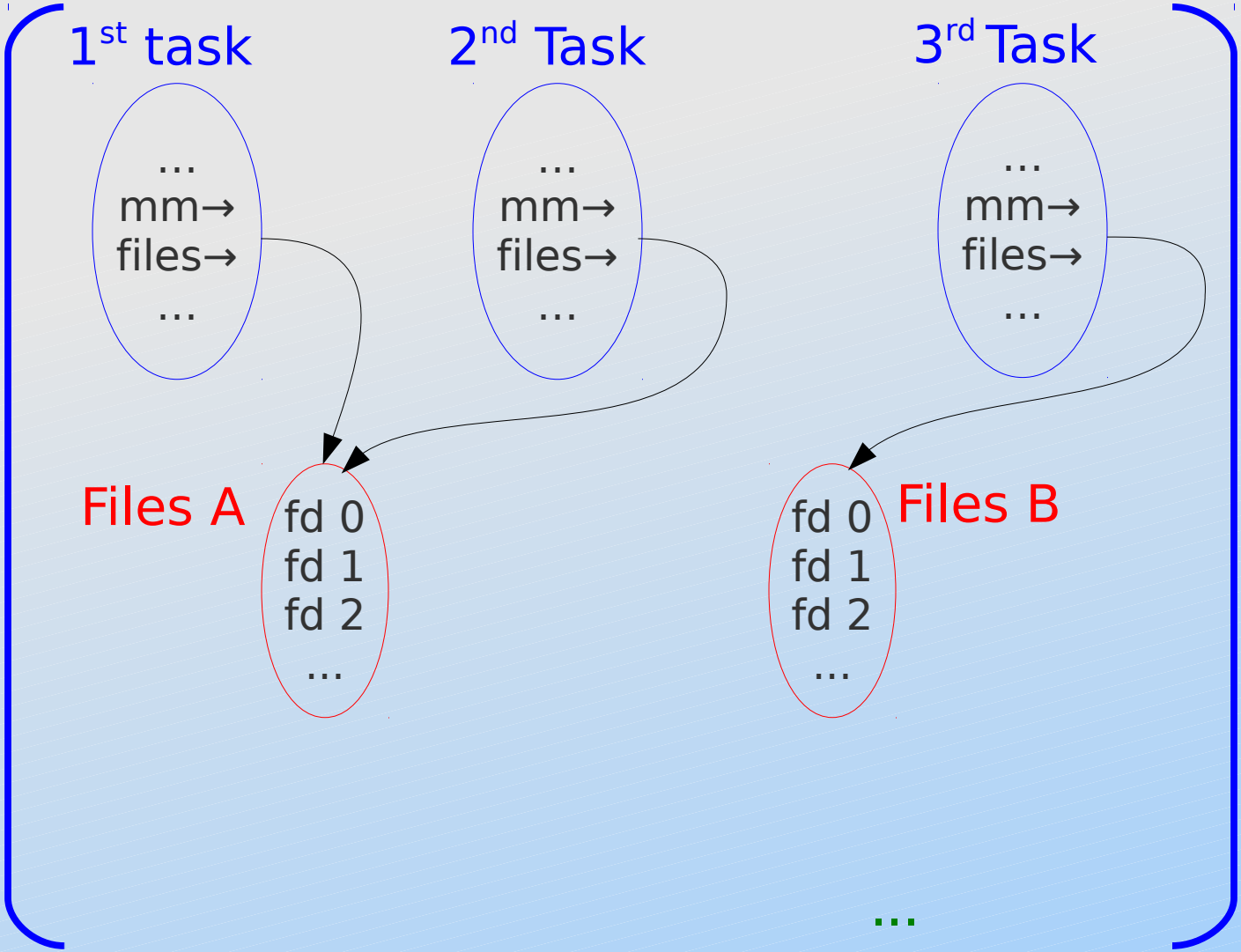
Shared Resources

- Checkpoint:
  - physical pointer → unique tag
  - save before the “parent” object
  - “parent” objects saves only tag
- Restart
  - unique tag → (new) physical pointer
  - restore before the “parent” object
  - use tag in “parent” to locate instance

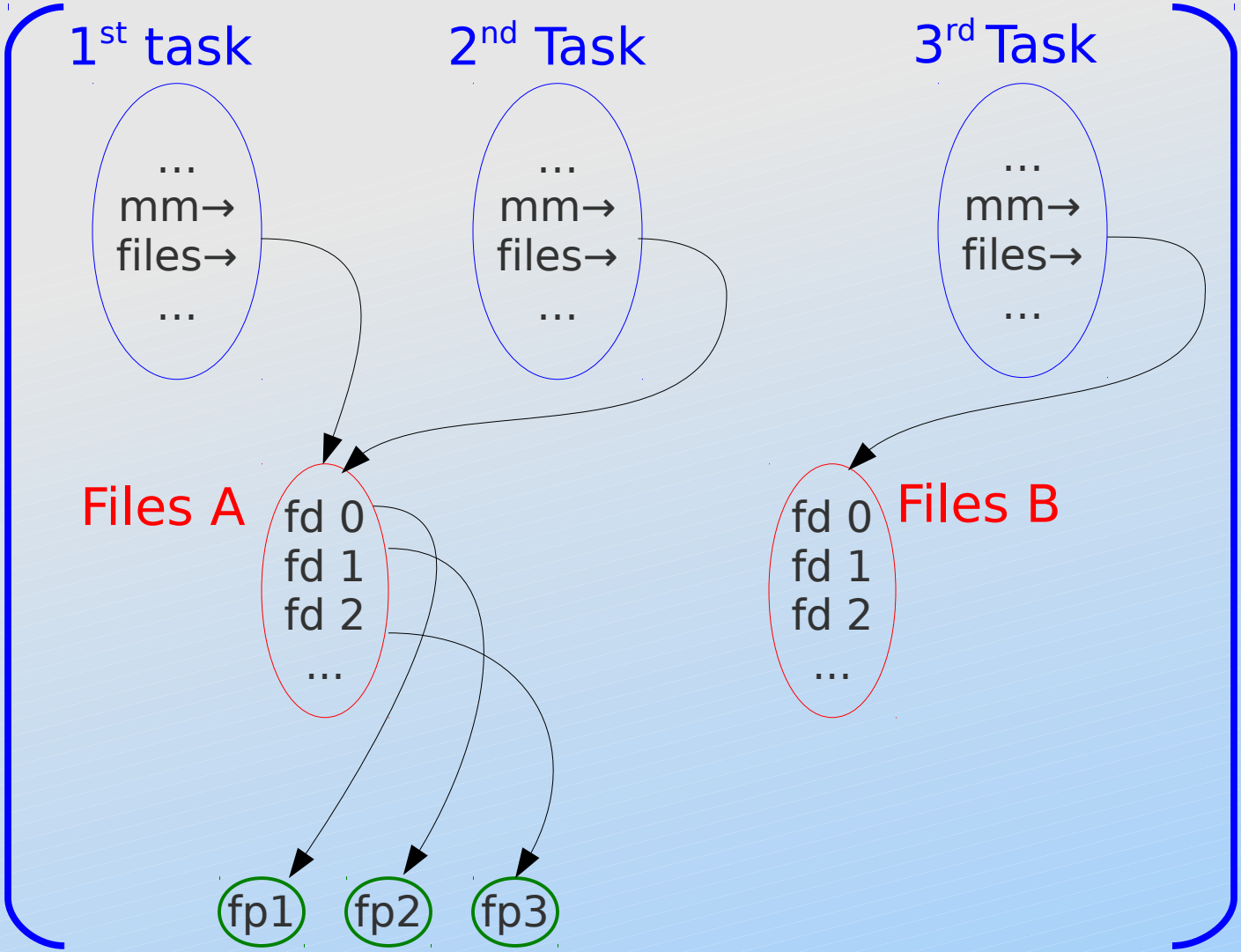
Shared Resources



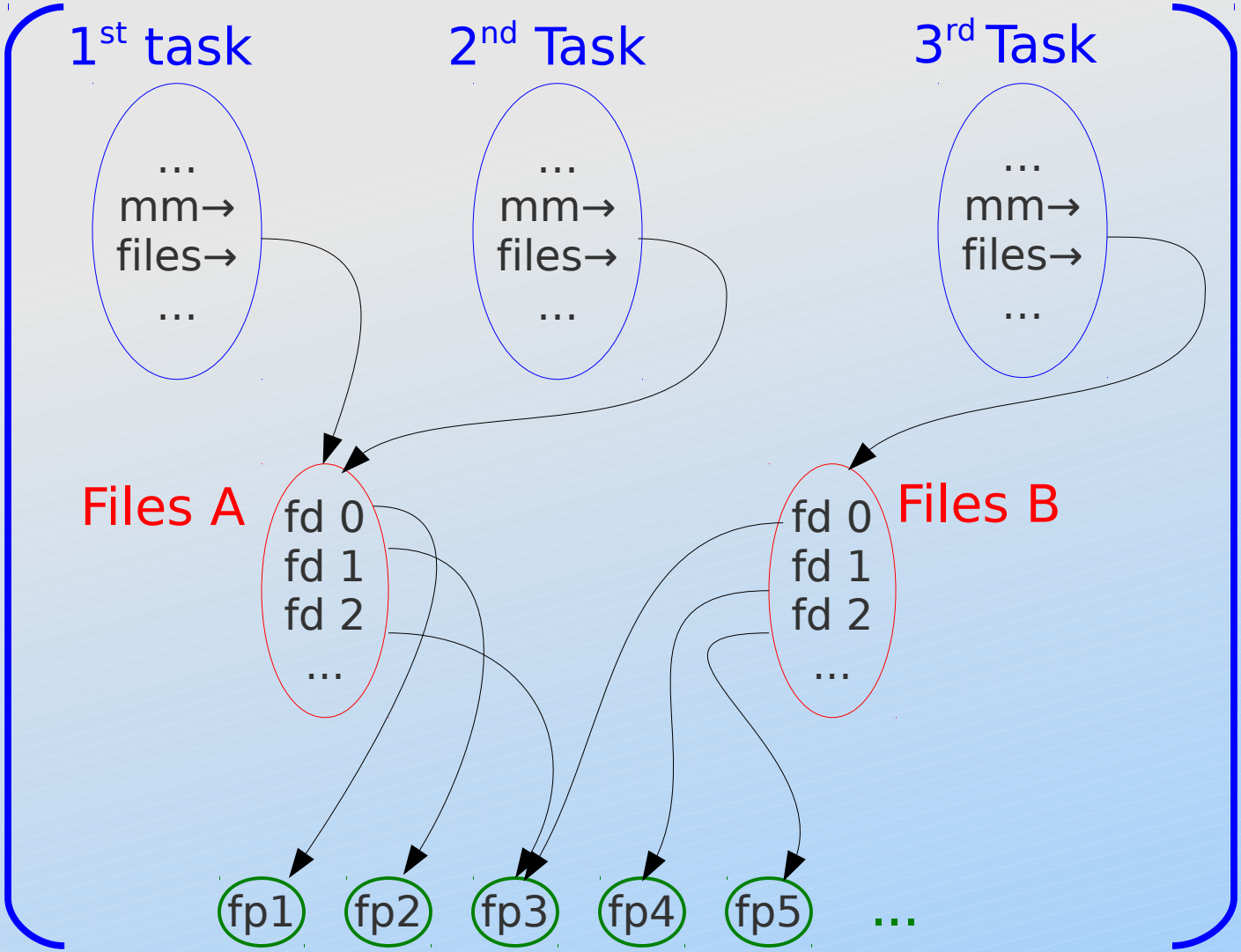
Shared Resources



Shared Resources

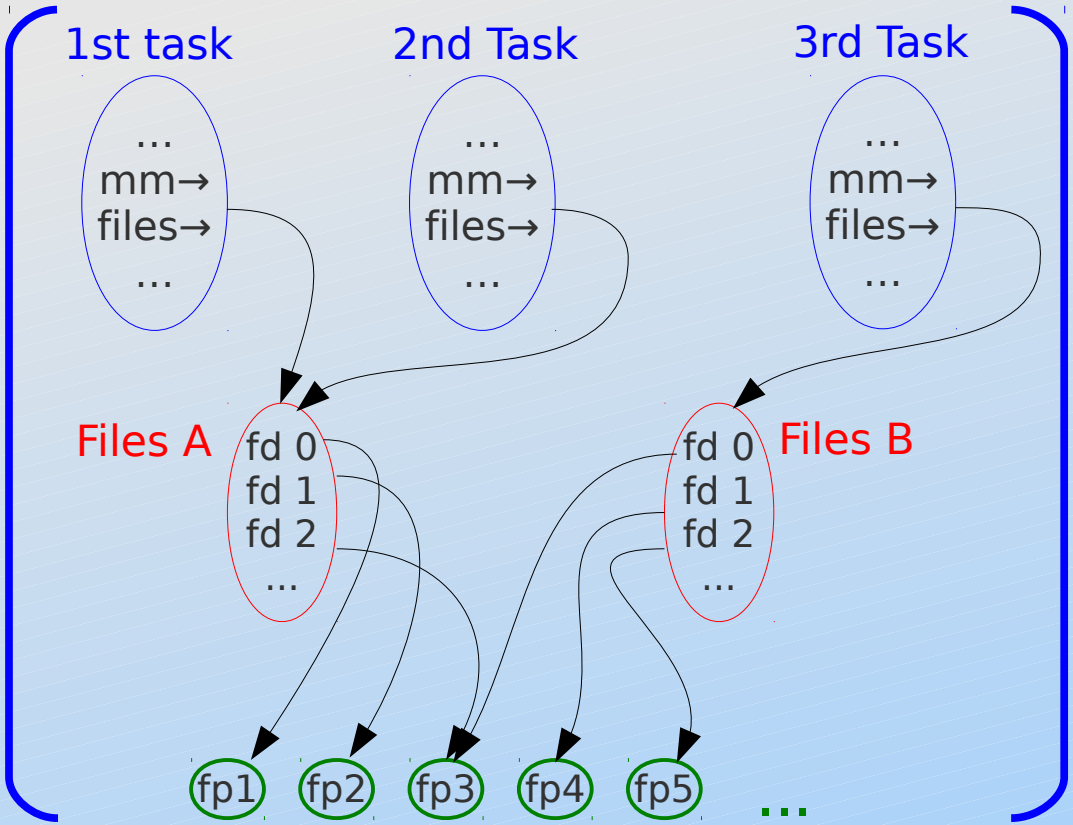


Shared Resources



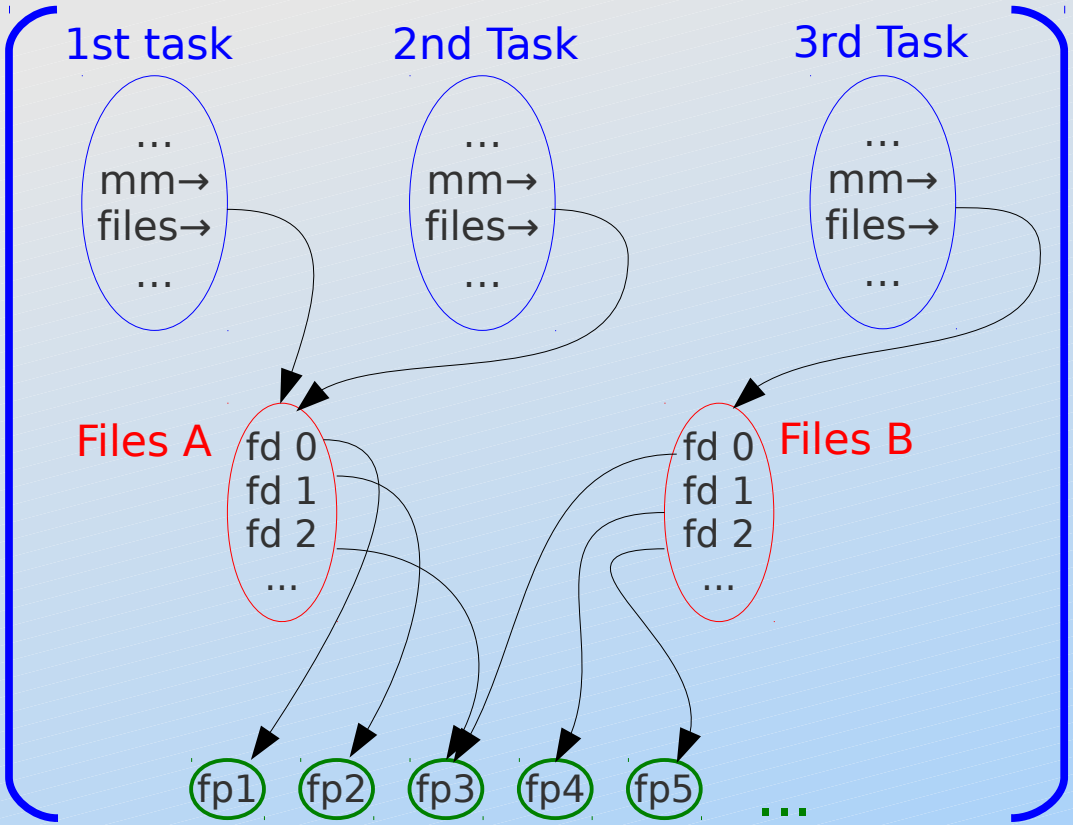
Checkpoint Image Example

...
hdr_mm [1st]
hdr_fd [fp1]
hdr_fd [fp2]
hdr_fd [fp3]
hdr_files [files A]
hdr_task [1st]
...



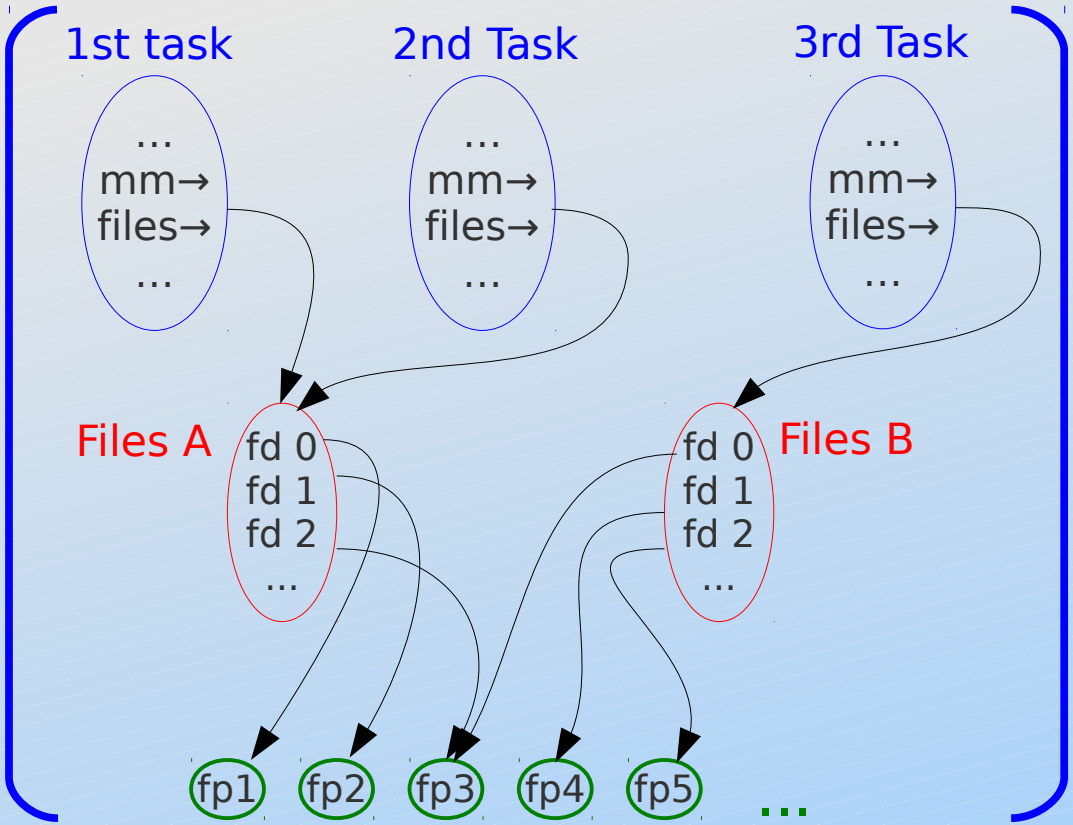
Checkpoint Image Example

...
hdr_mm [1st]
hdr_fd [fp1]
hdr_fd [fp2]
hdr_fd [fp3]
hdr_files [files A]
hdr_task [1st]
...
hdr_mm [2nd]
hdr_task [2st]
...



Checkpoint Image Example

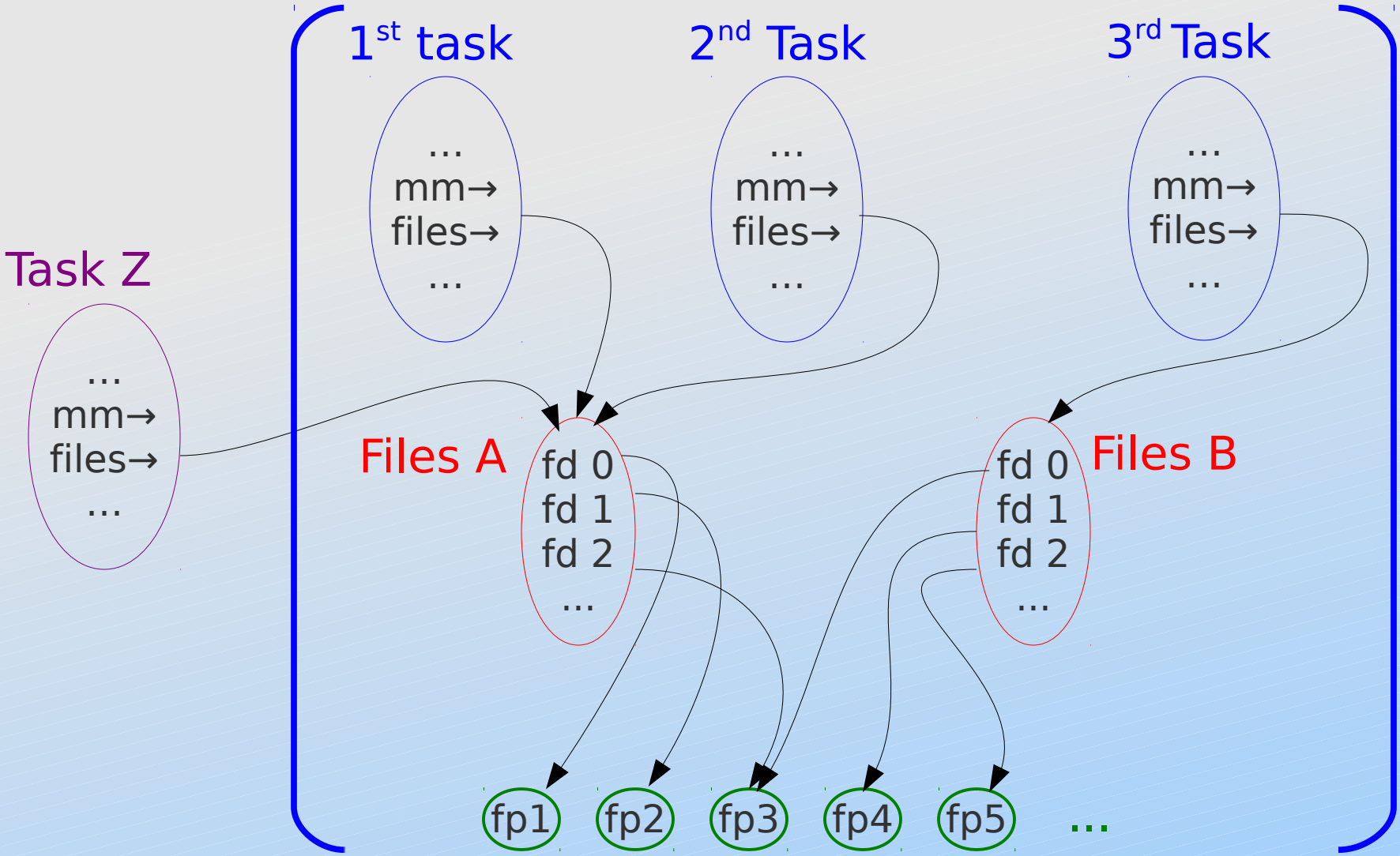
...
hdr_mm [1st]
hdr_fd [fp1]
hdr_fd [fp2]
hdr_fd [fp3]
hdr_files [files A]
hdr_task [1st]
...
hdr_mm [2nd]
hdr_task [2st]
...
hdr_mm [3st]
hdr_fd [fp4]
hdr_fd [fp5]
hdr_files [files B]
hdr_task [3rd]
...



Leak Detection

- Resources in use must not be modified from outside the container
  - collect: count reference in hierarchy
  - compare with kernel reference count
  - leverage shared instances repository

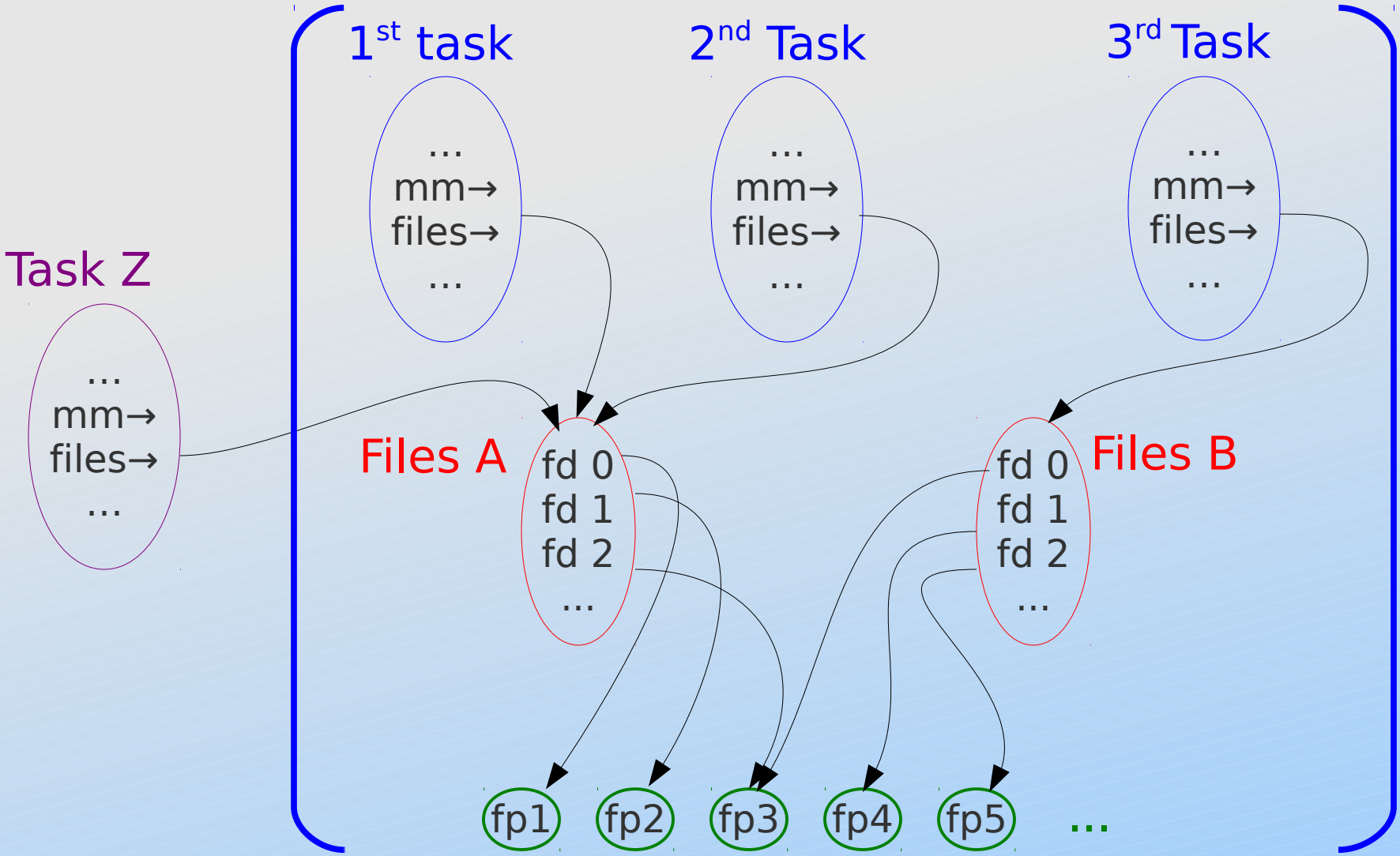
Leak Detection



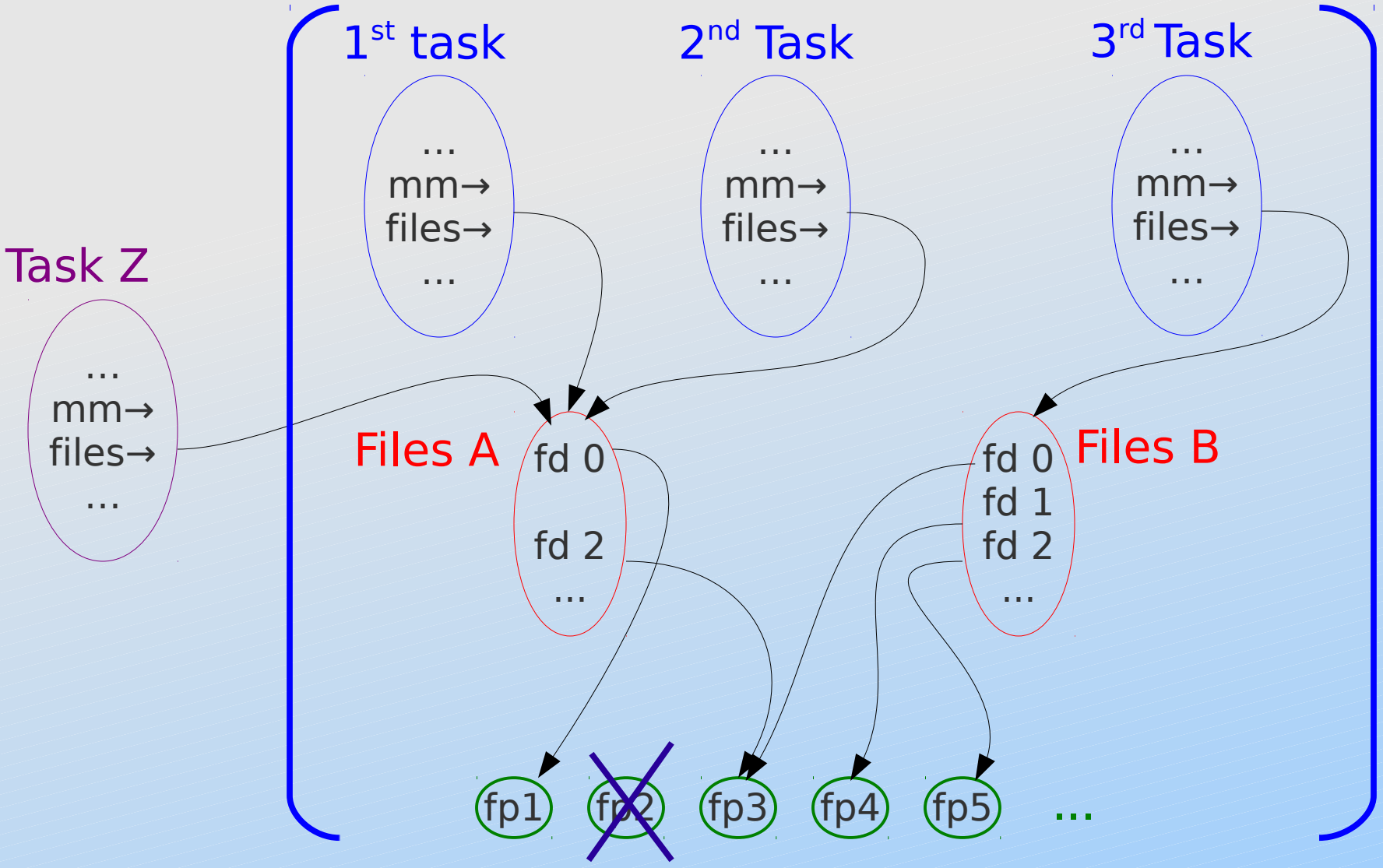
Leak Detection

- Resources in use must no be modified from outside the container
  - collect: count references in hierarchy
  - compare with kernel reference count
  - leverage shared instances repository
- What about races during collection ?

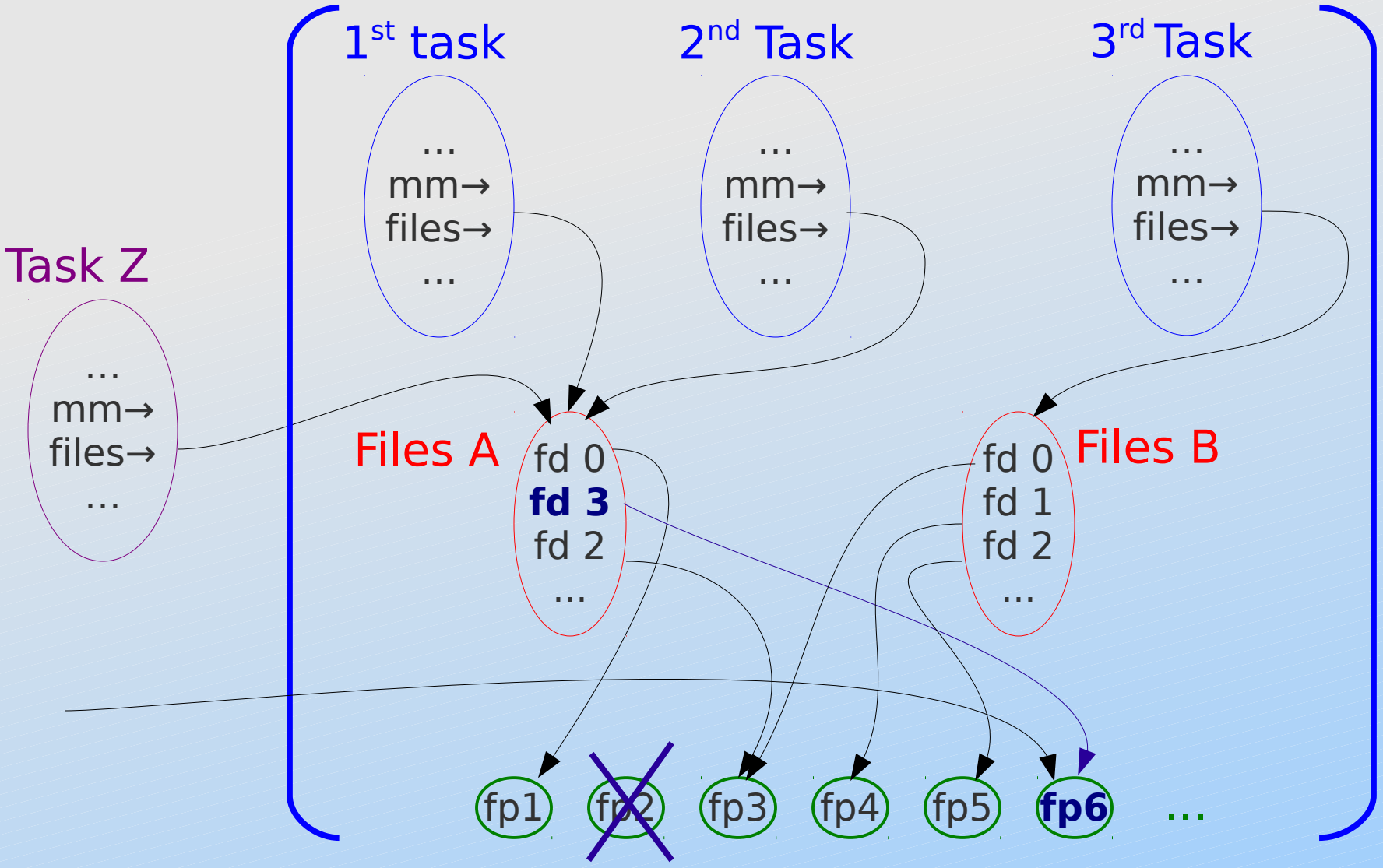
Leak Detection



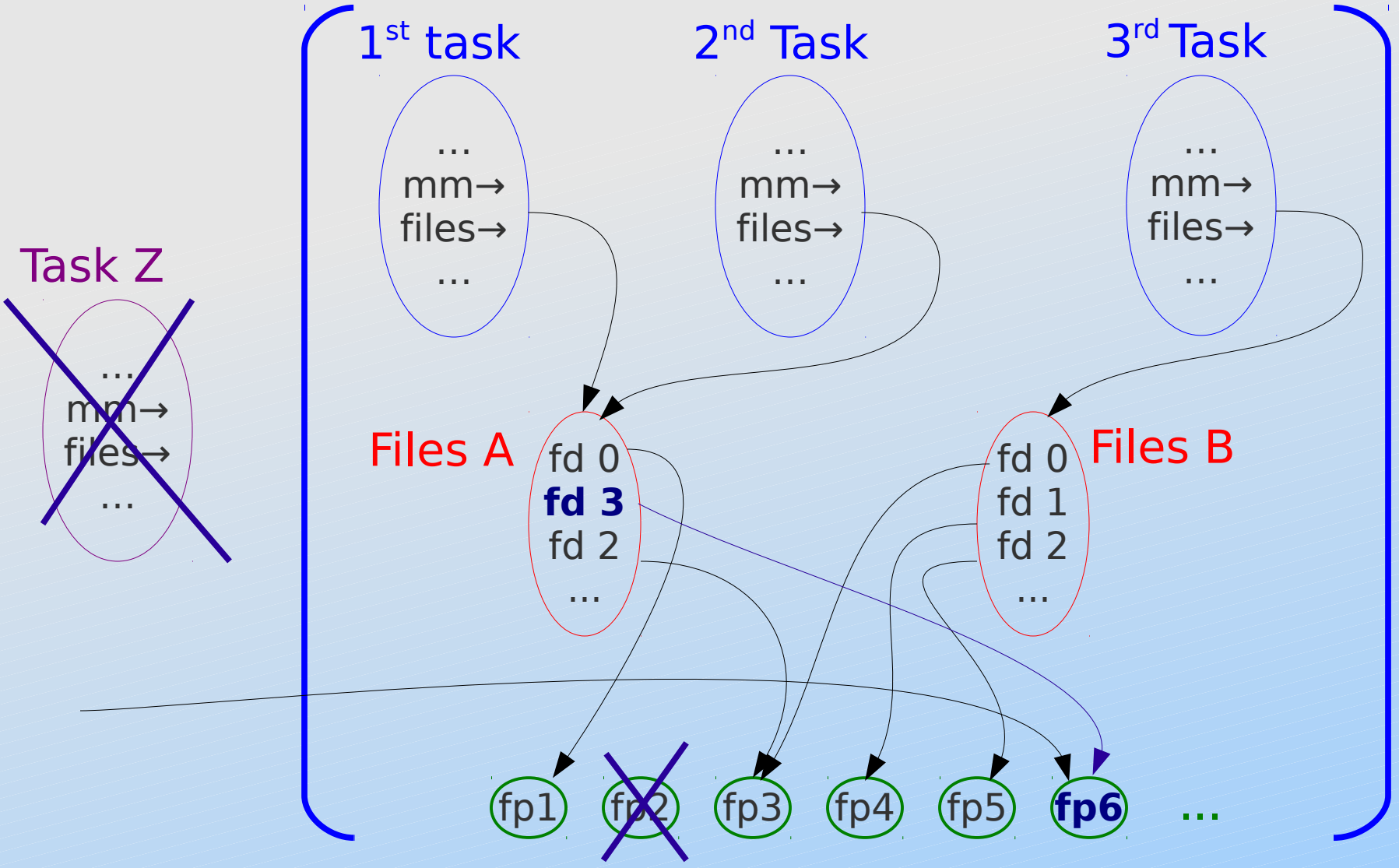
Leak Detection



Leak Detection



Leak Detection



Error Handling

- When error occurs:
  - syscall reports single error value
  - detailed log written to `@logfd`
  - users can examine the log

Kernel API - Overview

`ckpt_hdr_...()`: record handling (eg alloc/dealloc)

`ckpt_write_...()`: write records/data to image

`ckpt_read_...()`: read records/data from image

`ckpt_msg_...()`: output to log file (and debug)

`ckpt_err_...()`: report an error condition

`ckpt_obj_...()`: manage objects and hash-table

Kernel API – Shared Objects

```
struct ckpt_obj_ops {
    char *obj_name;
    Int obj_type;
    void (*ref_drop)(...);
    int (*ref_grab)(...);
    int (*ref_users)(...);
    int (*checkpoint)(...);
    void (*restart)(...);
};
```

register/unregister object handlers

```
register_checkpoint_obj(ops):
unregister_checkpoint_obj(ops):
```

Current State

- Supported architectures:
  - x86-32, x86-64, s390x, PowerPC, ARM
- Features:
  - see up to date information at
    https://ckpt.wiki.kernel.org/index.php/Checklist
  - experimental integration with LXC

Contributions

- Sukadev Bhattiprolu, Serge Hallyn, Dave Hansen, Matt Helsley, Nathan Lynch, Dan Smith, and myself...
- Suggestion, ideas and reviews from many other people ... Thank You !

Join the Effort !

- Implement more features [kernel]
- Checkpoint optimizations [kernel]
- Convert between kernel versions [user]
- Inspection of checkpoint image [user]
- Plug-in architecture for restart [user]
- ... and more ...

Questions ?

- More information
  - Web page: http://www.linux-cr.org/
  - Git tree(s): git://www.linux-cr.org/git/
  - Email: orenl@cs.columbia.edu

Thanks You !