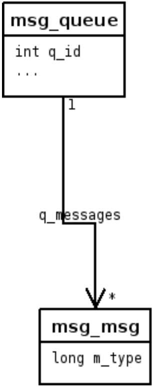
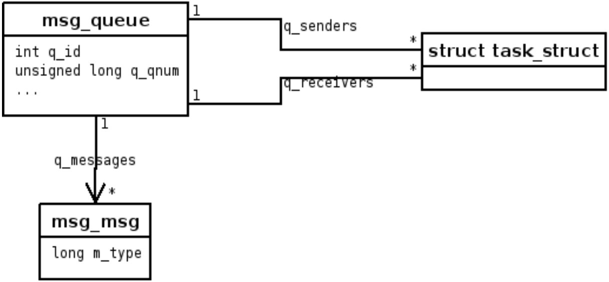
User interface

IPC mechanisms allowing to send/receive *typed* messages to a messages queue.



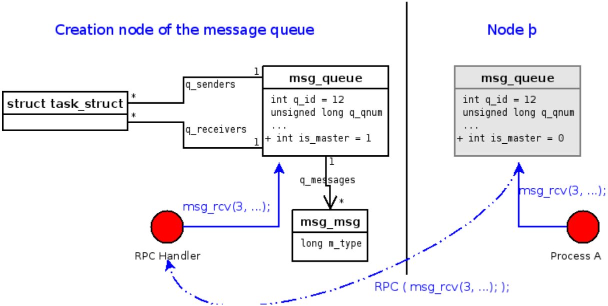
```
int msgsnd(int msqid,
           const void *msgp,
           size_t msgsz,
           int msgflg);

ssize_t msgrcv(int msqid,
               void *msgp,
               size_t msgsz,
               long msgtyp,
               int msgflg);
```

Linux Implementation



- Sending processes may block because messages queue is full. (`q_senders`)
- Receiving processes may block because msg queue does not contain any message of corresponding type(s). (`q_receivers`)

Kerrighed Implementation



Send or receive operations are forwarded to the creation node of the msq queue.

“Checkpointing” IPC messages queue

- `krgipcmsg [-save|-s] msqid`
- `krgipcmsg [-load|-l] msqid`

Saves the message queue information and the messages already in the queue in `/var/chkpt/msgq/<id>/<vers>/msgq<id>_v<vers>.bin`

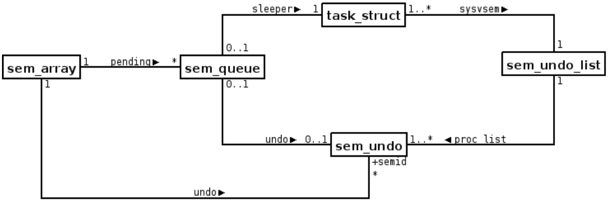
No information about waiting processes are saved.

User interface

- Quite complicated interface
- Allow to operate on many semaphores at the same time.

```
int semget(key_t key, int nsems, int semflg);

int semctl(int semid, int semnum, int cmd, ...);

int semop(int semid, struct sembuf *sops,
    unsigned nsops);

int semtimedop(int semid, struct sembuf *sops,
    unsigned nsops, struct timespec *timeout);
```

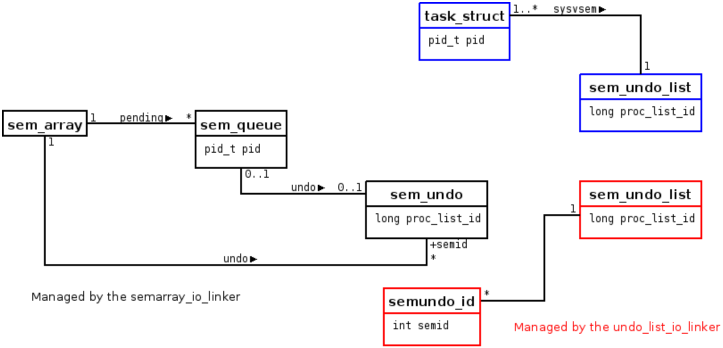
Linux Implementation



Major problem to make a distributed version: undo list objects are linked

- per semaphore
- per process (not exactly in fact!)

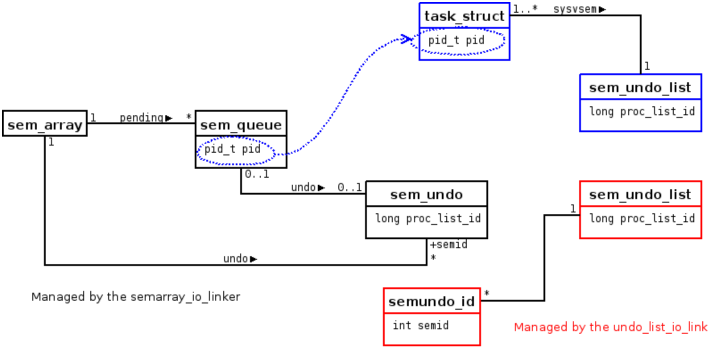
Kerrighed Implementation



Managed by the semarray_io_linker

Managed by the undo_list_io_linker

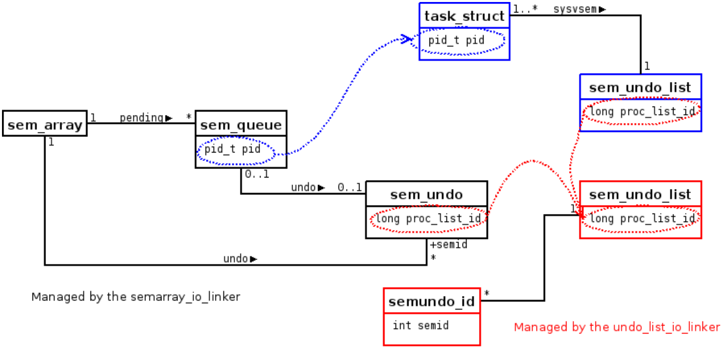
Kerrighed Implementation



Managed by the semarray_io_linker

Managed by the undo_list_io_linker

Kerrighed Implementation



"Checkpointing" IPC semaphores

- `krgipcsem [-save|-s] semid`
- `krgipcsem [-load|-l] semid`

Saves only the states of the semarray in
`/var/chkpt/sem/<id>/<vers>/sem<id>_v<vers>.bin`

No information (sempending, semundo) about waiting processes are saved.

Typical scenario of IPC checkpointing

Checkpoint an application with its needed IPC application

1. Defines what are the used IPC objects (using `ipcs`)
2. Finds one pid of your application processes
3. `checkpoint -kill=19 <PID>` (where 19 equals to SIGSTOP)
4. `krgipcsem -s <SEMID>`
5. `krgipcmsg -s <MSGID>`
6. `killall -k SIGCONT myapp`

Typical scenario of IPC checkpointing

Restart an application with its needed IPC application

1. `krgipcsem -l <SEMID> [<version>]`
2. `krgipcmsg -s <MSGID> [<version>]`
3. `restart <APPID> [<version>]`

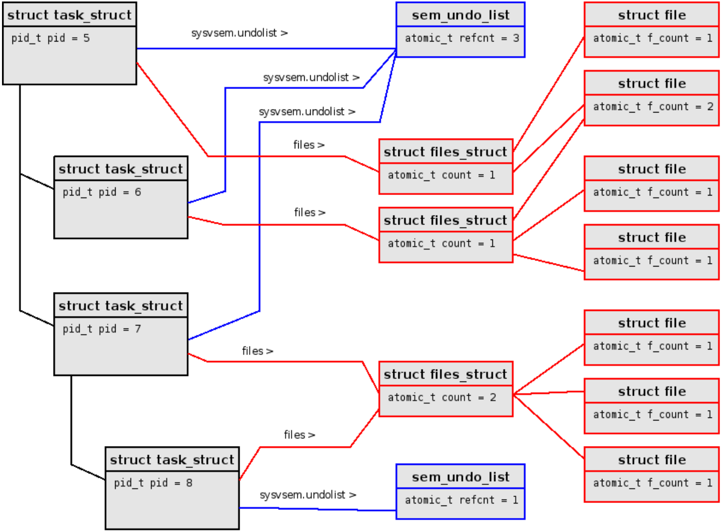
Application checkpoint

Checkpointing a tree of processes != Checkpointing each process of the tree

Some data may be shared between processes:

- fs_struct
- files_struct
- files pointer
- sysvsem undolist
- mm_struct
- ...

Application checkpoint



struct task_struct
pid_t pid = 5

struct task_struct
pid_t pid = 6

struct task_struct
pid_t pid = 7

struct task_struct
pid_t pid = 8

sysvsem.undolist >

sysvsem.undolist >

sysvsem.undolist >

sysvsem.undolist >

files >

files >

files >

files >

sem_undo_list
atomic_t refcnt = 3

sem_undo_list
atomic_t refcnt = 1

struct files_struct
atomic_t count = 1

struct files_struct
atomic_t count = 1

struct files_struct
atomic_t count = 2

struct file
atomic_t f_count = 1

struct file
atomic_t f_count = 2

struct file
atomic_t f_count = 1

struct file
atomic_t f_count = 1

struct file
atomic_t f_count = 1

struct file
atomic_t f_count = 1

struct file
atomic_t f_count = 1

Goal

- Avoid to checkpoint the same data twice or more
- Ensure a consistent restart with correct data sharing between the processes
- Minimize modification on current `import_*/export_*` functions
- Add a generic way to handle checkpoint/restart with data sharing

Current status

Work in progress!

Handles only data shared on one node

Only `export_/import_sysvsem` have been updated to use the framework

Open issue:
How to handle data shared between several nodes? (with minimum of communication)

Current implementation

1. Checkpoint each process one by one but do not save shared structs.
2. Instead, add relevant information in a local rbtree to save it later.
3. After each process has been checkpointed, the local coordinator creates a new ghost file and for each shared structs, calls the corresponding export_ functions.

Current implementation

```
int export_sysv_sem(struct epm_action *action,
    ghost_t *ghost, struct task_struct *task)
{
  int r = 0;

  switch (action->type) {
  case EPM_CHECKPOINT:
+     if (action->checkpoint.shared ==
    CR_SAVE_DELAYED) {
+         r = export_shared_sysv_sem(action, ghost,
    task);
+         goto end;
+     }
      /* really do the checkpoint */
      ...
      break;
```

Current implementation

```
int export_shared_sysv_sem(struct epm_action *
    action, ghost_t *ghost, struct task_struct *
    task)
{
  int r;
  long key = task->sysvsem.undo_list_id;
  r = ghost_write(ghost, &key, sizeof(long));
  if (r)
    goto error;

  r = add_to_shared_structs_list(task,
      SEM_UNDO_STRUCT, key);
error:
  return r;
}
```