Process Management: Current Status and Future Developments

Louis.Rilling@kerlabs.com

Focus

					Sched		
		IPC			EPM		
KerPipe	KerSocket	MM	Synchro		Proc	ProcFS	FS

Distributed Services Layer

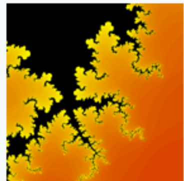
Dynamic Stream	Ghost
Container	

Basic Blocks

Service Manager	HotPlug
Communication Library	
KerNetDev	
Tools	

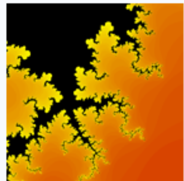
Comm Layer

Outline



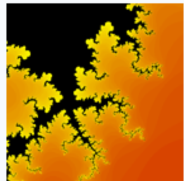
- Migration, Distant fork, Checkpoint (EPM)
- System containers
- Global scheduler
- Directions to investigate, but when?
- Porting issues
- Summarizing timeline

Outline



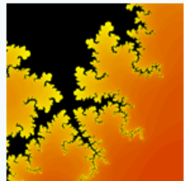
- Migration, Distant fork, Checkpoint (EPM)
- System containers
- Global scheduler
- Directions to investigate, but when?
- Porting issues
- Summarizing timeline

Outline



- Migration, Distant fork, Checkpoint (EPM)
  - What is working?
  - Making limitations safe
- System containers
- Global scheduler
- Directions to investigate, but when?
- Porting issues
- Summarizing timeline

Outline



- Migration, Distant fork, Checkpoint (EPM)
  - What is working?
  - Making limitations safe
- System containers
- Global scheduler
- Directions to investigate, but when?
- Porting issues
- Summarizing timeline

Clone Flags (until Linux 2.6.18)

CLONE_VM	Share VM (same **mm**) with caller
CLONE_FS	Share fs info (same **fs**) with caller
CLONE_FILES	Share open files (same **files**) with caller
CLONE_SIGHAND	Share signal handlers and ignored signals (same **sighand**) with caller
CLONE_PTRACE	Let tracing continue on the clone too
CLONE_VFORK	Caller wants the clone to wake it up on **mm_release** (**exit** or **exec**)
CLONE_PARENT	Clone has same parent (**real_parent**) as the caller
CLONE_THREAD	Same thread group as caller (**tgid**, **signal**, **sighand**, **real_parent**, **group_leader**, **process_keyring**)
CLONE_NEWNS	New namespace group
CLONE_SYSVSEM	Share system V SEM_UNDO semantics with caller
CLONE_SETTLS	Create a new TLS for the clone
CLONE_PARENT_SETTID	Set clone PID in the caller
CLONE_CHILD_CLEARTID	Clear the PID in the clone VM on **mm_release** (**exit** or **exec**)
CLONE_DETACHED	Unused, ignored
CLONE_UNTRACED	Tracing process won't force ptrace on this clone
CLONE_CHILD_SETTID	Set clone PID in the clone
CLONE_STOPPED	Start in stopped state

What is Working?

- Migration
  - Works for sequential processes
    - Nothing shared with another task, except file descriptors initially (POSIX semantics of `fork`)
  - Signals may be lost during migration
    - To be fixed shortly
  - Needs testing with Linux Test Project
- Distant fork
  - OK with clone flags `CLONE_CHILD_`{`SETTID`, `CLEARTID`} (needed by all `fork` in recent GNU libc)
  - Not used if any other clone flag specified
    - `CLONE_PARENT` and `CLONE_PARENT_SETTID` supported very soon
    - All threads of a thread group remain on a same node
  - Under heavy test with Linux Test Project

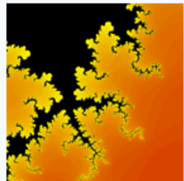
Checkpointing?

- Still not working
  - Who tried to make it work?
- Roadmap (sequential processes only, no communication)
  - Alpha version shortly (february, march)
    - No pid reservation: restart may fail if pid is reused!
  - Beta version during the summer
    - PID reservation as long as checkpoints remain valid
  - Robust in november 2007

Checkpointing: What is Needed?

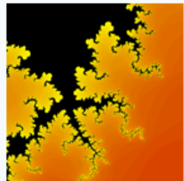
- Container support for checkpointing would be great :-)
- IO linker functions already do a similar job...
- Persistent storage for reserved PIDs
  - Security?
- File system support (not for november 2007)
  - File versioning
  - Stable storage
    - Checkpoints
    - Set of reserved PIDs
- Suggestions?

Outline



- Migration, Distant fork, Checkpoint (EPM)
  - What is working?
  - Making limitations safe
- System containers
- Global scheduler
- Directions to investigate, but when?
- Porting issues
- Summarizing timeline

Making Limitations Safe



- Clone flags easy to check at task creation, but after?
  - Ex: disable migration of a process whose parent does not have a children ctnr object
  - Ex: disable migration of a process being ptraced
- 2 generic mechanisms
  - **`krg_cap_unavailable*`** capability arrays
  - **`krg_action_*`** functions family

krg_cap_unavailable Capability Array

- One array per task
- Inherited at fork
- One counter for each capability
  - Ex: # of inheritable objects used that prevent from using cap

```
sys_open()
{
      ...
      if (... /* special file */)
            /* Disable migration since it would break access to the file */
            atomic_inc(&current->krg_cap_unavailable[CAP_CAN_MIGRATE]);
      ...
}
sys_close()
{
      ...
      if (... /* special file */)
            /* Closing the special file does not prevent migration anymore */
            atomic_dec(&current->krg_cap_unavailable[CAP_CAN_MIGRATE]);
}
```

krg_cap_unavailable_private Capability Array

- Similar to **krg_cap_unvailable**, but not inherited at fork
  - Ex: giving system ctnr objects to processes is a per process decision

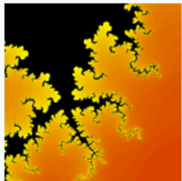
```
copy_process()
{
      struct task_struct *p;       /* New task */
      ...
      if (!p->task_ctnr)
            /* Disable parent migration since p could not notify it at exit */
            atomic_inc(&p->parent->
                              krg_cap_unavailable_private[CAP_CAN_MIGRATE]);
      ...
      if (!p->parent->children_ctnr)
            /* Disable migration since p could not notify parent at exit */
            atomic_inc(&p->krg_cap_unavailable_private[CAP_CAN_MIGRATE]);
      ...
      if (!p->children_ctnr)
            /* Disable distant fork since child could not notify p at exit */
            atomic_inc(&p->krg_cap_unavailable_private[CAP_DISTANT_FORK]);
      ...
}
```

krg_action_* Family (built on top of krg_cap_unvailable)

```
#include <epm/action.h>

typedef enum {
   EPM_NO_ACTION,
   EPM_MIGRATE,
   EPM_REMOTE_CLONE,
   EPM_CHECKPOINT,
   EPM_ACTION_MAX    /* Always in last position */
} krg_epm_action_t;

/* Disable action, if not one already in progress */
int krg_action_disable(struct task_struct *task, krg_epm_action_t action);
/* Re-enable action */
int krg_action_enable(struct task_struct *task, krg_epm_action_t action);

/* Start action if not disabled */
int krg_action_start(struct task_struct *task, krg_epm_action_t action);
/* Notify action end */
int krg_action_stop(struct task_struct *task, krg_epm_action_t action);
```

- Non-blocking, can be nested
- Result != 0 means “Abort or do something else!”

Outline



- Migration, Distant fork, Checkpoint (EPM)
- System containers
- Global scheduler
- Directions to investigate, but when?
- Porting issues
- Summarizing timeline

System Containers: Tasks

- Objects are facultative for any task, but EPM actions on a task need the task being attached to all containers
  - Not created for local PIDs and kernel-created kernel threads
- PID container
  - Object <-> PID, lazy creation
  - PID allocation and recycling
  - May provide PID location in the future
  - Implementation may change
- Task container
  - Object <-> PID, facultative
  - Share fields of a `task_struct`
    - Remote child reaping
  - PID Location (to be transferred elsewhere)

System Containers: Signals

- signal_struct container
  - Object <-> TGID, facultative, depends on Task ctnr objects
  - Share a signal_struct
    - Provide parent with process resource usage at child's exit
    - Will allow distributed threads to share signals
- sighand_struct container
  - Object <-> custom unique ID, facultative, depends on Task ctnr objects
  - Share signal handlers
    - Will allow distributed tasks to share signal handlers

System Containers: Children Container

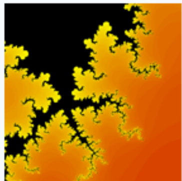
- Object <-> TGID, facultative, depends on Task ctnr objects
- Reparent children to remaining threads of a thread group
- Know who is parent without making exit unscalable
  - Children list of a thread group rather than one for each thread
  - **parent**, **real_parent**, and **real_parent_tgid** fields of task ctnr objects need not being always up to date

```
/* Lock children ctnr object of parent, and get up to date real parent TGID */
/* Result == NULL => no need to unlock */
/* Must be used on a live task (not reaped yet) */
struct children_ctnr_object *kh_parent_children_writelock(
      struct task_struct *child, pid_t *real_parent_tgid);

/* Can be used on a dead task (real_parent_tgid will point to 0) */
struct children_ctnr_object *kh_parent_children_readlock(
      struct task_struct *child, pid_t *real_parent_tgid);

void kh_children_unlock(pid_t tgid);
```

Future System Containers

- Pgrp container
  - Object <-> PGID (subset of TGIDs having existed so far)
  - Know which nodes a process group spans
  - POSIX compliant job control
    - Detect orphaned process groups and send them `SIGHUP+SIGCONT` if this results from a process death
  - Support syscall **`setpgid`**
- Thread group container
  - Object <-> TGID
  - Know the PIDs of all threads in a distributed thread group
  - Ease reparenting of children when a thread exits
  - Support wait syscalls family with distributed thread groups
- `user_struct` container, `group_info` container

Outline



- Migration, Distant fork, Checkpoint (EPM)
- System containers
- Global scheduler
- Directions to investigate, but when?
- Porting issues
- Summarizing timeline

Global Scheduler Architecture

Process group a
Process group b
Process group c

Scheduling policy X
Scheduling policy Y
Scheduling policy Y

Filter Filter Filter Filter Filter Filter Filter

...

Filter Filter Filter Filter Filter Filter Filter

Probe A1
Probe B
Probe A2
Probe B
Probe A2

Node 1
Node 2

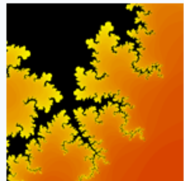
Dynamic Configuration

- Configfs
  - Quoting Linux documentation:

    “configfs is a ram-based filesystem that provides the converse of sysfs's functionality. Where sysfs is a filesystem-based view of kernel objects, configfs is a filesystem-based manager of kernel objects, or `config_items`. “

  - `mkdir` -> create a `config_item`
  - `read/write` -> see/set `config_item` attributes
  - `symlink` -> aggregate `config_items` from different subtrees
- Map scheduler component connections to configfs operations
- Background work...

Outline



- Migration, Distant fork, Checkpoint (EPM)
- System containers
- Global scheduler
- Directions to investigate, but when?
- Porting issues
- Summarizing timeline

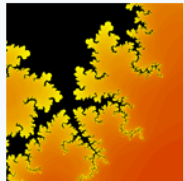
Distributed Linux

- Consistent time management
  - Wallclocks, jiffies
- Distributed threads
  - Distant fork must support `CLONE_THREAD`, `CLONE_VM`, `CLONE_SIGHAND`, `CLONE_SETTLS`, `CLONE_FILES`, `CLONE_FS`, `CLONE_SYSVSEM`
- Remote `ptrace`
  - Manage parent != real parent, one or both remote
  - Access to the VM of a remote task

Enhancements to Kerrighed

- Application fault tolerance
  - Parallel checkpointing/restart
  - High availability for applications
  - User-level API to customize / optimize fault tolerance
- Multi-localized tasks
  - Improve SSI performance by keeping operations local
  - Init, already in some way
    - Orphaned children reaping remains local as much as possible
  - Servers
    - apache, inetd, nscd

Outline



- Migration, Distant fork, Checkpoint (EPM)
- System containers
- Global scheduler
- Directions to investigate, but when?
- Porting issues
- Summarizing timeline

Latest Linux 2.6

- Probably Linux 2.6.20
- Namespaces (vservers)
  - PID allocation, IPC, ...
- New clone flags
  - `CLONE_NEWUTS`, `CLONE_NEWIPC`
- `unshare` system call
- Track changes in `task_struct` for EPM
- New task flags
  - Find place for the 5 Kerrighed flags
    - `PF_MIGRATING`, `PF_CHECKPOINTING`, `PF_DISTANT_FORKING`, `PF_AWAY`, `PF_EXIT_NOTIFYING`
    - New `krg_flags` field?

SMP / Multi-core

- PROC “prepared”
- EPM “half-prepared”
  - Real big issue is robust error handling
- PROCFS to be rewritten(!)
- SCHEDULER easy to port in its current status
- Nothing tested yet!
- My own philosophy
  - Better have deadlocks than hidden race conditions

64 bits (x86-64)

- In process management, only a matter for EPM
- Kerrighed signal
  - Arch-independent Kerrighed code that hooks in vanilla Linux arch-dependent code
- Export/import of Arch-specific task state
  - Already written, but did someone test?
- 32 bits compatibility for Kerrighed syscalls (ioctls)

Summarizing Timeline

- Personal view
- Who knows what can happen?

Kerrighed summit		spring 07		summer 07		nov 07		?
alpha proc. mngt				beta proc. mngt		robust proc. mngt		distributed threads remote ptrace
	alpha sequential checkpoint			beta sequential checkpoint with PID reservation		robust sequential checkpoint		parallel checkpoint high-availability
		alpha scheduler arch			beta scheduler arch		robust scheduler arch	multi-node tasks consistent time mngt
	Linux 2.6.20			SMP	x86-64			