Process Migration via Remote Fork: a Viable Programming Model?

Branden J. Moore

cse 598z: Distributed Systems
December 02, 2004
What is a Remote Fork?

- Creates an “exact” copy of the process on a remote system
- Attempts to duplicate semantics of POSIX `fork()`
Why Remote Fork?

- Computational Codes are **BIG**.
- Parallel Algorithms have been proved very effective.
- “Clustering” computers provides for a high degree of potential parallelism and resources.
- Parallelism across disjoint systems is difficult to harness.
- `fork()` is a well-known programming model.
- [rs]h - Remote “exec”, not “fork”.
  - No “single-initialization” advantage.

- Condor, PBS, etc. - Job Schedulers
  - No guarantee of Parallelism.

- MPI - Not dynamic.
  - Hosts determined and initialized at start.

- POSIX fork - Single-host only.
Schedule of Events

- Why Study Remote Fork?
- Exploring the Design Space
- Sample Implementation
- Conclusion
Components of a Remote Fork

- Expected Semantics
- Checkpoint & Recovery
- File Descriptors
- Signals
- Fault Tolerance
fork() Semantics

- fork() creates an exact copy of the calling process except for the following:
  - The child process has a unique process ID and different parent process ID.
  - The child process has its own copy of the parent’s descriptors.
- Memory mappings remain valid.
- Parent receives process ID of child process.
- Parent should wait(2) for child’s exit.
Create a copy of a process on disk which can be restored to become an exact copy of the original process.

rfork() Checkpoints, and restores on remote system.
- Recording process state is tricky.
  - Heisenburg is at work.
  - Internal (Library) - Can read most all process state, but the act of doing so changes state.
  - External (Debugger) - Can read most state via \texttt{ptrace(2)}), but cannot access kernel state.

- Dynamic linking and loading

- Open File Descriptors
  - Shared memory (\texttt{mmap(2)})
  - Sockets & unnamed pipes, missing files.
File Descriptors & Signals

- What do file descriptors connect to?
  - Local copies of the files
    - Even on Global FS, breaks semantics
    - unnamed pipes, sockets cannot connect
  - Tunnel back to host computer
    - Use a Shadow Process
  - Ostrich

- Signals & Process IDs
  - kill(2), waitpid(2)
  - getppid(2), exit(2)
Fault Tolerance

- Distributed systems increase chance of faults
  - Remote system failure, network failure
  - Restart failure

- Checkpoints give ability for recovery
  - Store & periodic checkpoint
  - Single checkpoint, retry until success
  - What about communication?
libcr, librfork & RFD
Checkpointing with librcr

- Must link to library
- Handles Dynamic Libraries*

Trivial API:
- `pid_t cr_checkpoint(const char*)`;
  - Create child, dump image (internal & external)
- `pid_t cr_restart(const char*)`;
  - forks, and restores image
Remote Fork Architecture

- Link against librfork and liber
  - `pid_t rfork(const char* rhostname);`
- `rfork()` creates a shadow process “cpp”
- Remote computer runs Remote Fork Daemon (RFD)
- `cpp` converses with RFD and parent to tunnel communication
- Open files are tunneled, and signal information is passed
Remote Fork Architecture
librfork & cpp

- librfork
  - checkpoint, fork
    - child: exec cpp
    - parent: return pid when signaled by cpp

- cpp
  - Connect to RFD, send checkpoint
  - Loop
    - select() on open FDs
    - pass signal information
    - exit with same status as remote child
Remote Fork Daemon

- Accept Connections
- Spawn new Children & Restore from Checkpoint
- Forward reads/writes & signal information to/from Child & CPP
- Monitor existence of child, and send waitpid()'s status back to CPP
Benchmark: Parallel Matrix Multiply

- Trivial Parallel Matrix Multiply algorithm
  - Divide into segments, each process computes over that segment, and returns result to parent

- Allocates and \( \text{random}(3) \)-ly fill matrices A & B

- \text{rfork()} child processes to compute sections
In Summary

- Remote Fork offers a method to harness available parallelism from clustered computers.

- There are many challenges in designing a production-quality Remote Fork system
  - Semantics of distributed systems don’t quite match local systems.

- The act of `rfork()` is EXPENSIVE
  - Best suited to jobs with large computation, minimal communication, relatively small memory footprints.
Questions?