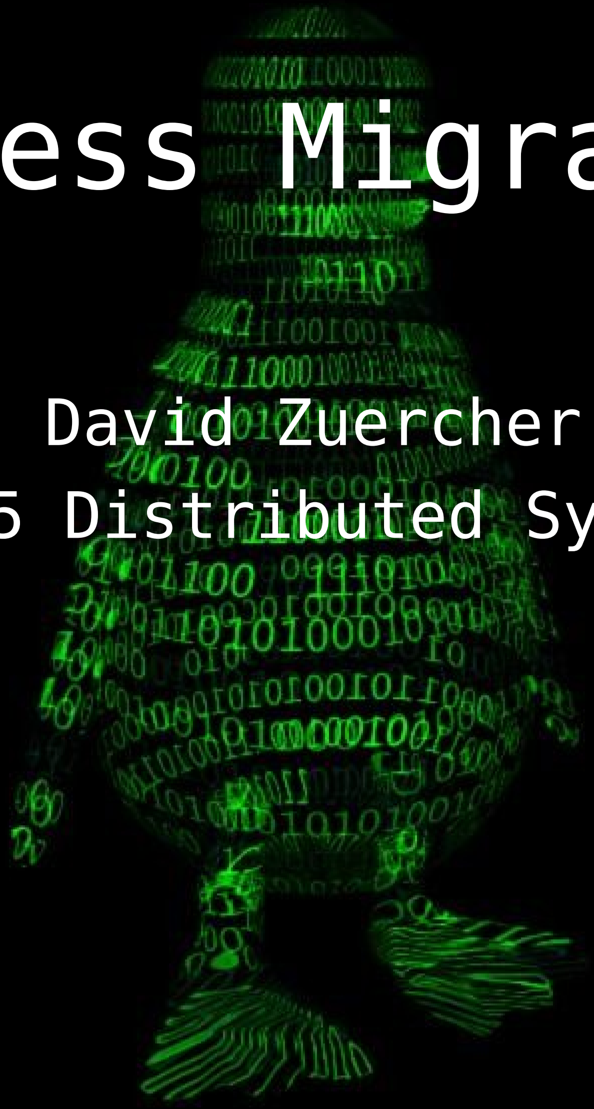
Process Migration

David Zuercher

CS555 Distributed Systems



What is Migration?

- **Mi-gra-tion**

  -The act or an instance of migrating.

  - A group migrating together.

- **Mi-gra-te**

  - Moving from one place to another.

Why Migrate?

- Some animals migrate for survival.

- A process might do the same if its environment becomes “hostile”:

  -Power outage possible or imminent.

  -The OS is angry?!

  -System becoming overburdened.

- Migration is also fun, and impressive.

Pseudo-migration

- One can’t really *migrate* a process.

- Migration implies a persistence of being (e.g. the PID won’t change).

- It is important to acknowledge migration cannot be done literally; but the realization itself is a good starting point for a solution.

A New Direction

- So, as is often the best solution, we’re going to fake it.

- Somehow quantify the state of a process so that a “clone” of it may live on at a later time.

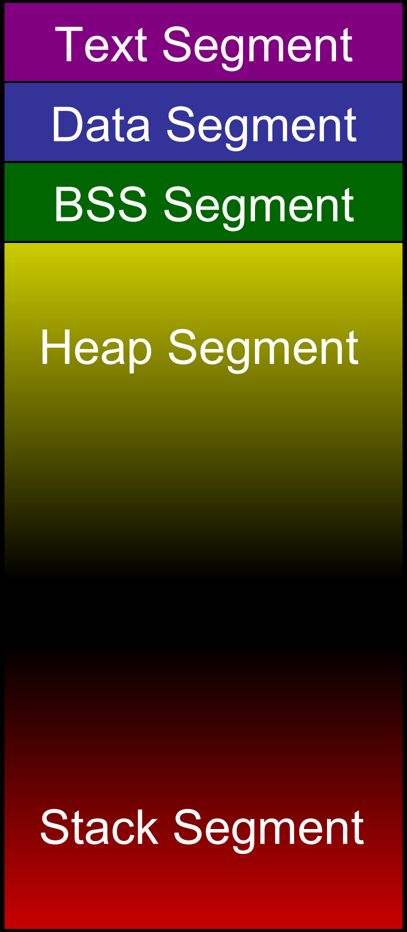
Process Cryonics

- *“Freeze”* a process in its tracks. Save its state into a file of some kind.

- The process state is now fully serialized and static; It could be revived locally or remotely.

- *“Thaw”* the process and let it run as if nothing happened (as close as possible).

Getting Started: Process Anatomy



- Processes are “physically” composed of memory segments within a virtual address space.
- Segments are mapped one or more pages at a time.
- The heap is usually composed of many segments. The stack is just one that can grow.

The Exact Memory Mapping

In Linux, each process has its own directory under /proc.

A file called “maps” in that directory has the memory mappings for the process.

```
$> cat /proc/<PID>/maps

=Address Range    Perm  Offset        Inode      File Mapped=

08048000-08049000 r-xp 00000000 08:04 442116     /home/theprog
08049000-0804b000 rw-p 00000000 08:04 442116     /home/theprog
40000000-40012000 r-xp 00000000 08:02 128783     /lib/ld-2.2.5.so
40012000-40013000 rw-p 00012000 08:02 128783     /lib/ld-2.2.5.so
40013000-40016000 rw-p 00000000 00:00 0
4002d000-4014b000 r-xp 00000000 08:02 130798     /lib/libc-2.2.5.so
4014b000-40150000 rw-p 0011e000 08:02 130798     /lib/libc-2.2.5.so
40150000-40155000 rw-p 00000000 00:00 0
bfff3000-c0000000 rwxp ffff4000 00:00 0
```

What else needs freezing?

- Processes have a “user” area that contains the registers and some other items not accessible to the process itself.

- All variables a process uses are located in the segments mentioned before (so we don’t bother with them).

- Per-process Kernel Stack… Let’s not worry about that for now.

Its Getting Colder…

- In Linux, an utterly unrelated process may become the parent of another process (not init!).

- The new parent may “peek” and “poke” one word at a time from the new child’s memory areas.

- Register Data can be peeked and poked as well.

The Freeze Plan

- I created a *library* function:

  ```
  char *freeze_process(int PID, int ops)
  ```

- freeze_process() may be called by one process to stop and freeze another.

- The function returns the name of its primary output file, but there are two…

*.freeze & *.frore

- Two files are produced by the freezing process *.freeze & *.frore.

- *.freeze contains:

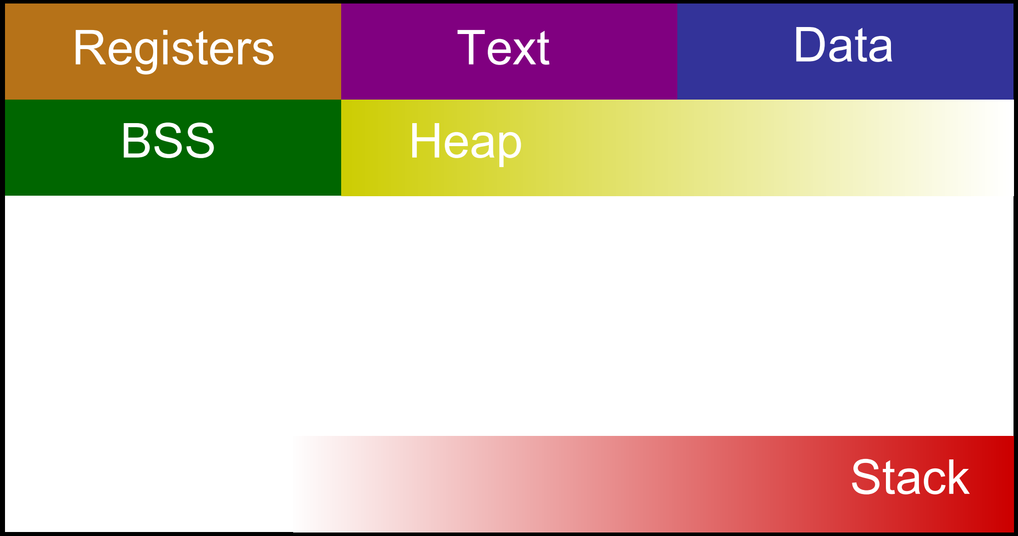
  -Register Data

  -Memory Segments

- *.frore contains:

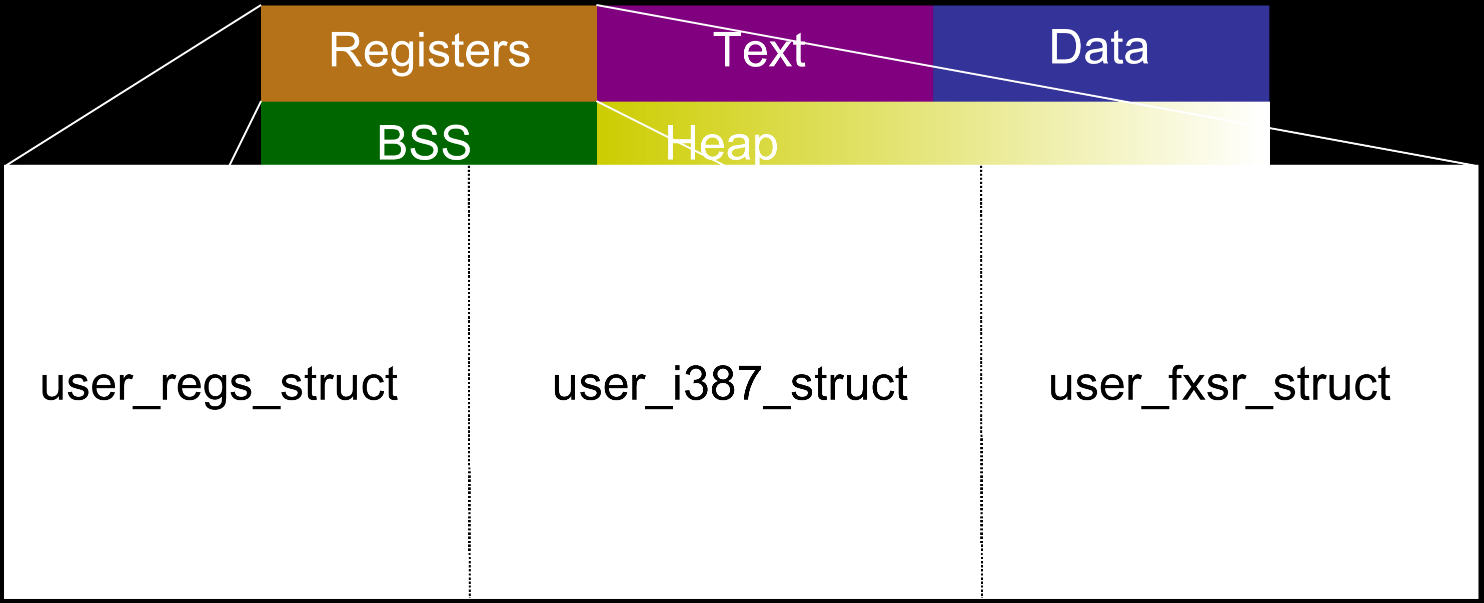
  -Memory Segment ranges & perms

*.freeze In Depth



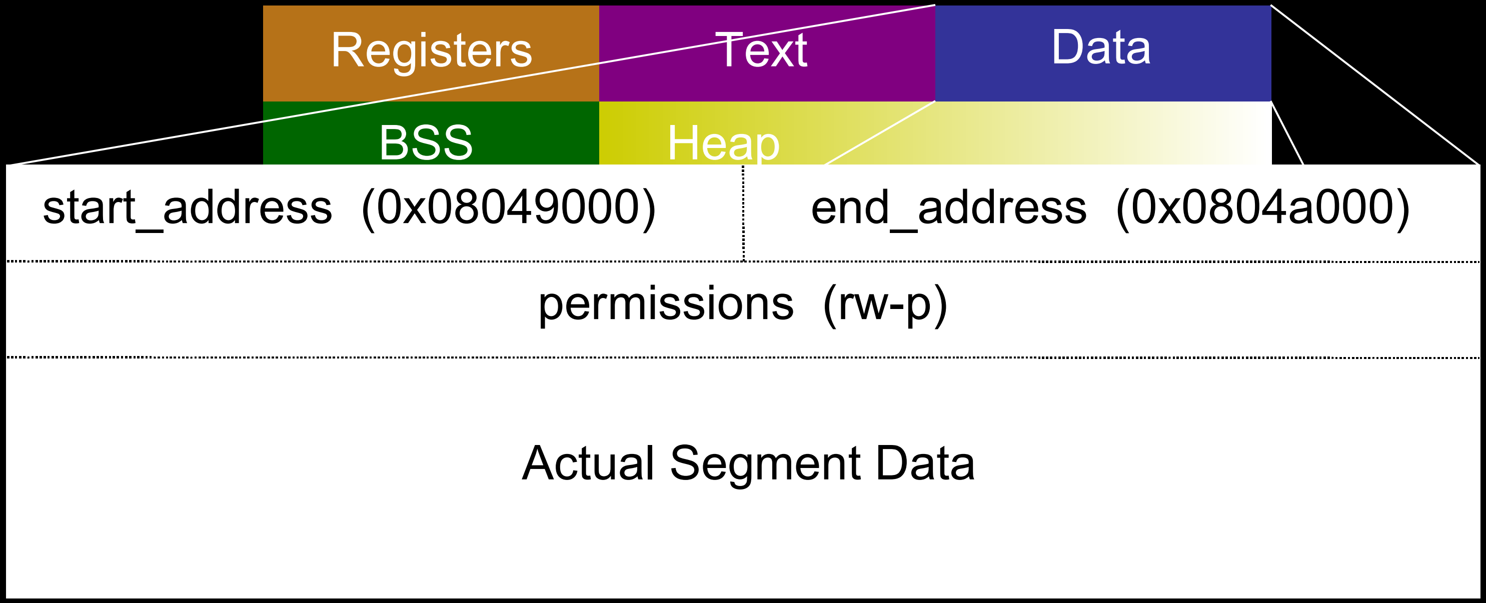
The freeze file contains as much of the process state as can be retrieved using a user-level system tool (ptrace in the case of Linux).

*.freeze In Depth(2)



ptrace() provides a way to retrieve these three structures, containing the process register data.

*.freeze In Depth(3)



Registers

Text

Data

BSS

Heap

start_address (0x08049000)

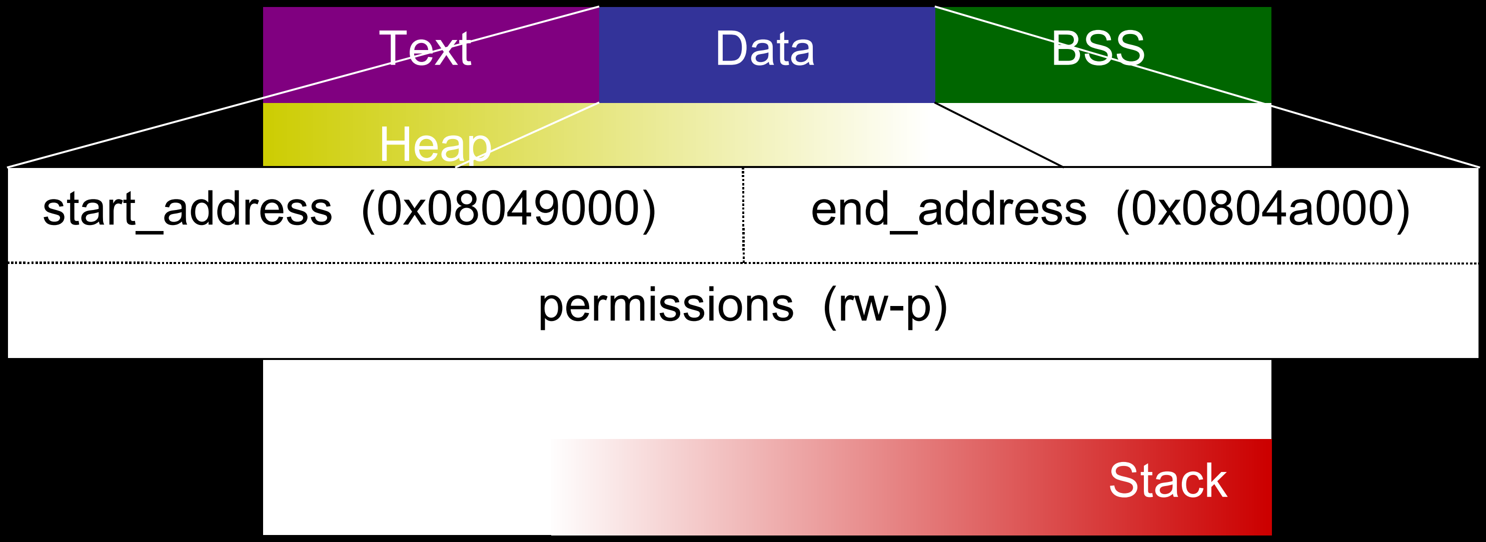
end_address (0x0804a000)

permissions (rw-p)

Actual Segment Data

The remainder of the memory segments have much the same form, addresses followed by permissions and then the actual data of the segment.

*.frore In Depth



Text

Data

BSS

Heap

start_address (0x08049000)

end_address (0x0804a000)

permissions (rw-p)

Stack

*.frore is a template for the process. It contains everything *.freeze contains except the registers and the actual segment data.

Thoughts on Freezing

- Freezing a process and letting it slip away into digital oblivion is easy once we know how to get access to the necessary information.

- As intended, *.freeze & *.frore are portable across Linux systems with similar architecture (x86).

- Now, how do we bring it back to life…?

The Big Thaw

- How do the files produced by freezing a process become a process of their own?

- In Linux, the only way to make a *new* process is with fork() or clone().

- Replacing the context of a running program is the job of exec(), but it takes an executable image, not our cryo-files.

A New Executable?

- Could a new executable be built, using the saved data, that would produce a running clone of our frozen program.

- Unfortunately no. An ELF or COFF executable cannot specify the size or contents of the stack, nor duplicate the new mappings of the heap at any given address.

A New Executable? (2)

- Furthermore, if the program had a BSS segment, an executable cannot be set up to fill the contents of such a segment (the Kernel allocates clean pages for BSS).

- Also, as we see later signal handlers and file descriptors cannot be handled this way.

A New Executable?(3)

- Perhaps a loader could be written with process revival specifically in mind. That is another project entirely, not implemented here.

Seen “The Exorcist”?

- The idea is to “possess” an existing process, replacing its execution context with our process.

- What kind of process makes a good candidate for possession?

  -One with an *identical* memory mapping.

  -One with at *least* the same stack depth.

  -More features are needed, but these prove the point.

The Mapping Conundrum

- Can a process change the memory mapping of another process? Not by any means… changes to a process’s memory mapping must be voluntary or performed by the kernel.

- The ability to change a child's memory mapping would be a great addition to the ptrace() system call!!

Voluntary Mapping

- Our “host” process has to change itself, via mmap() and other similar calls.
- The catch… How can a process shrink or expand its own text segment?
- It might possibly be done at user level, but not easily.
- *Divine* assistance is necessary… the Kernel.

/dev/overseer

- A kernel module has the power to re-map a process on-the-fly.

- Using Kernel functions, a process could be remapped to serve as a suitable host for possession.

- The host process must simply write the contents of the *.frore file to the overseer module at it will be re-mapped.

It’s Getting Warmer…

- A new *library* function is created:

```
int thaw_process(int PID, int ops)
```

- Using the overseer module and ptrace(), a *new* process is created with the same execution context as the original frozen process.
- freeze_process & thaw_process are a big step but only a small part of migrating more complicated processes.

What Escaped Freezing?

- Unfortunately this simple freeze does not handle saving such things as:

  -open file descriptors

  -installed signal handlers

  -pending timers

  -relationships to other processes

  -many others…

Enter OVERSEER

- Everything the simple freeze missed was installed via a system call (like open(), signal(), …).

- In Linux, system calls can be intercepted with ptrace().

- With ptrace(), the entry and exit of each system call can be inspected, then the appropriate action taken by the parent process.

OVERSEER

- As its name suggests, the overseer will monitor every aspect of another process that is necessary for freezing.

- Overseer builds lists of installed signal handlers, open file descriptors (with their associated files) child processes, and some other items.

Intercepting System Calls

- With ptrace(), delivery of a signal to the parent can be arranged when its child enters or exits a system call (the child gets stopped by the Kernel).

- The parent must look in the child's registers to see *which* system call caused the signal, and *what* its arguments were.

- Inspection of the registers is also necessary to determine the return value of the system call.

System Calls in Linux

- System calls are requests by user processes for services from the Kernel.

- How is a system call actually made?

  -The appropriate syscall num is loaded into EAX.

  -The arguments are loaded into EBX, ECX, EDX, …

  -Interrupt 0x80 is invoked, transferring control to the Kernel.

*.frost

- Information OVERSEER gathers throughout the monitoring process is written into a *.frost file at the time of freezing.

- For *.frost to be complete, the process must obviously be monitored from the beginning of its execution.

OVERSEER (File Handling)

- Calls like open() and dup() cause new entries to the file descriptor (FD) list.

- Calls like read() and lseek() adjust the offset for the entries in the FD list.

- And so on…

OVERSEER (Signal Handling)

- The installation of signal handlers is the passing of a certain struct to the Kernel. If intercepted and saved, the signal handler could be installed in the host later on.

OVERSEER (Process Family)

- The fork() system call introduces an interesting twist to the overseer program.
- This migration implementation allows for communication between parent and child.
- When a fork() is encountered, the overseer forks as well, and its child monitors the forked process…

OVERSEER (Process Family)

- When migration is not involved, parent/child communication is usually not a big deal.

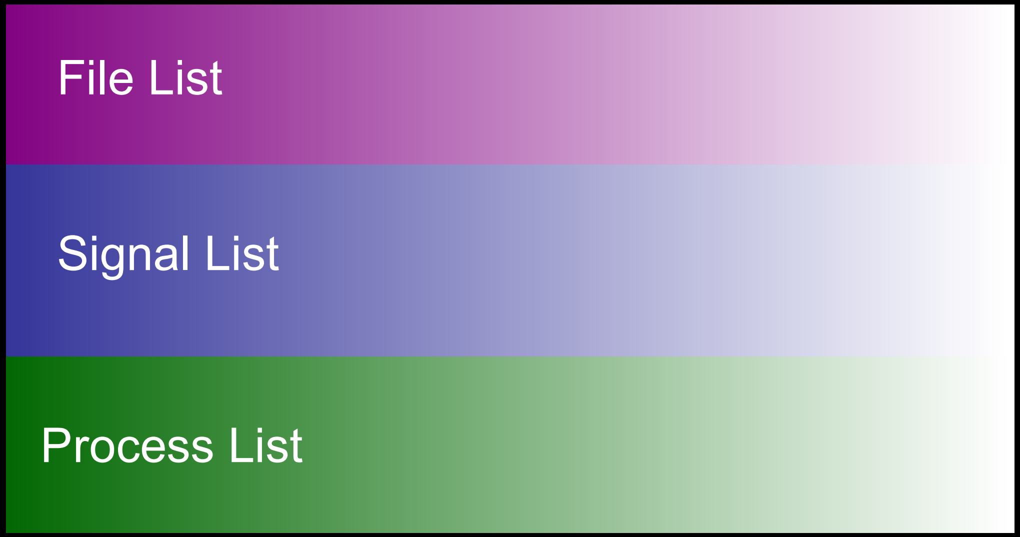
- How can this communication continue on the destination machine?

  -PIDs are different.

  -calls such as getpid() and getppid() will NOT return the same values.

  -static storage of PIDs (by the processes) will be wrong.

*.frore In Depth



File List

Signal List

Process List

*.frore contains the structures and other things necessary to get these “services” reinitialized.

Demo Time

- An example process “theprog” is going to be migrated.
- Add a fork() or two…
- Look at some code maybe…
- Some requests maybe…

Future Expansion

- There is a lot that even the mighty OVERSEER does not handle quite yet.

  -Multi-threaded processes.

  -Shared memory (in the same family).

  -Pipes/FIFOs (in the same family).

  -Sockets (set up a proxy… ugly).

Any Questions?

Please be gentle!!!

The End

Now migrate yourself home...