A quick note on `vm_fault()`

- Learn about (and trace) POSIX IPC
- Explore buffering and scheduler interactions
- Measure the probe effect
- Start to gather data for assessed *Lab Report 2*
Recall: A (kernel) programmer model for VM

Machine-independent virtual memory (VM)

- Stack
  - Read/write, grows down, anonymous object
  - "vm_map_entry"
- Heap
  - Read/write, anonymous object
- Library
  - Read/copy-on-write, named object
- Code
  - Read/copy-on-write, named object
  - "vm_space", "vm_map"
- "vm_object"
  - Anonymous swap-backed VM object
  - "vm_pager"
  - Swap pager
  - Page
  - Page
- "vm_page"
  - Shadow anonymous swap-backed VM object
  - "vm_pager"
  - Swap pager
  - Page
  - Page
- "vnode"
  - VM object
  - Vnode pager
  - Page
  - Page
- "vm_object"
  - Read/copy-on-write, named object
  - "vm_pager"
  - Swap pager
  - Page
  - Page
  - Page
- "vm_pager"
  - Vnode pager
  - Page
  - Page
- "vm_map_entry"
  - Anonymous swap-backed VM object
  - "vm_pager"
  - Swap pager
  - Page
  - Page
- "vmspace", "vm_map"

Machine-dependant physical map (PMAP)

- Physical memory
  - Physical map
  - Page-table directory
  - Page-table entry
  - Superpage
  - Code
  - Data
  - PTE
  - PDE
  - PTE
  - PTE
  - PTE

Dr Robert N. M. Watson
L41: Lab 2 - IPC
9 March 2015
The Mach VM fault handler \texttt{(vm\_fault)}

- Key goal of the Mach VM system: be as lazy as possible
  - Fill pages (with file data, zeroes, COW) on demand
  - Map pages into address spaces on demand
  - Flush TLB as infrequently as possible
- Any work avoided means reduced CPU cycles and less disk I/O
- Avoid as much work as possible when creating a mapping (e.g., \texttt{mmap()}, \texttt{execve()})
- Instead, do on-demand in the MMU trap handler, \texttt{vm\_fault()}
  - Machine-independent function drives almost all VM work
  - Input: faulting virtual address, output mapped page or signal
  - Look up object to find cached page; if none, invoke pager
  - May trigger behaviour such as zero filling or copy-on-write

- A good thing to probe with DTrace to understand VM traps
The benchmark

[guest@beaglebone ~/lab2] ./ipc-static
ipc-static [-Bqsv] [-b buffersize] [-i pipe|socket] [-t totalsize] mode

Modes (pick one - default 1thread):
- 1thread: IPC within a single thread
- 2thread: IPC between two threads in one process
- 2proc: IPC between two threads in two different processes

Optional flags:
- -B: Run in bare mode: no preparatory activities
- -i pipe|socket: Select pipe or socket for IPC (default: pipe)
- -q: Just run the benchmark, don’t print stuff out
- -s: Set send/receive socket-buffer sizes to buffersize
- -v: Provide a verbose benchmark description
- -b buffersize: Specify a buffer size (default: 131072)
- -t totalsize: Specify total I/O size (default: 16777216)

- Simple, bespoke IPC benchmark: pipes and sockets
- Statically or dynamically linked
- Adjust user and kernel buffer sizes
- Various output modes
The benchmark (2)

- Three operational modes:
  - 1thread  IPC within a single thread of a single process
  - 2thread  IPC between two threads of a single process
  - 2proc   IPC between two threads in two processes

- Adjust IPC parameters:
  - -i pipe  Use pipe() for IPC
  - -i socket Use socketpair() for IPC
  - -b bufferize Set buffer size used for each IPC system call
  - -t totalsize Set total size across all IPCs
  - -s Also set in-kernel buffer size for sockets
  - -B Suppress quiescence (whole-program tracing)

- Output flags:
  - -q Suppress all output (whole-program tracing)
  - -v Verbose output (interactive testing)
The benchmark (3)

[guest@beaglebone ~/lab2]$ ./ipc-static -v -i pipe 1thread
Benchmark configuration:
  buffersize: 131072
  totalsize: 16777216
  blockcount: 128
  mode: 1thread
  ipctype: pipe
  time: 0.033753791
485397.29 KBytes/sec

- Use verbose output
- Use pipe IPC
- Run benchmark in a single thread
- Use default buffersize of 128K, totalsize of 16M
Exploratory questions

- Baseline benchmark performance analysis:
  - How do the various benchmark configurations perform?
  - How do distributions of return values from `read()` and `write()` vary?
  - How does setting the socket-buffer size impact socket performance?
  - How much time do pipes vs. sockets spend in system calls?
  - How do context-switch rates vary across benchmark configurations?

- Probe effect and measurement decisions
  - How do various types of DTrace instrumentation affect performance – counting, logging, capturing stack traces?
  - How much difference does aggregation make for, for example, system-call counting?
  - How much can sampling be used to reduce overhead – and what is the impact on accuracy?
Experimental questions for lab report

The full lab-report assignment will be distributed during the next lab. These questions are intended to help you gather data that you will need for that lab report:

- How does changing the buffer size affect IPC performance? For sockets, consider both with, and without, the \(-s\) flag.
- How might the probe affect cause relatively different performance impacts for different IPC configurations?
This lab session

▶ Upgrade your SD Card image
  ▶ This version has fixes to FBT, stack(), and wallclocktime
  ▶ Ensure you’ve saved any scripts/data from your old card
  ▶ You will need to reinstall your SSH key on the new SD card
  ▶ Return the old card to us – we may provide future updates!

▶ Use this session to continue to build experience:
  ▶ Build and use the IPC benchmark
  ▶ Use DTrace to analyse distributions of system calls, system-call execution times, and system-call arguments and return values
  ▶ Use ministat to analyse benchmark results
  ▶ Experiment with scheduler tracing

▶ Do ask us if you have any questions or need help