L41: Lab 2 – Kernel implications of IPC

• A quick note on `vm_fault()`
• Learn about (and trace) POSIX IPC
• Explore buffering and scheduler interactions
• Measure the probe effect
• This is the first of two labs contributing to Lab Report 2:
  • Lab 2 takes an OS-centric approach
  • Lab 3 takes a microarchitecture-centric approach
• Use data from both to write the lab report
Recall: A (kernel) programmer model for VM

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

```
root@l41-beaglebone data/ipc:~ # ./ipc-static
ipc-static [-Bqv] [-b buffersize] [-i pipe|local] [-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|local` 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()` IPC
  - `-i local` Use `socketpair()` IPC
  - `-b size` Set user IPC buffer size
  - `-t size` 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)

root@l41-beaglebone ~/ipc:~ # ./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 bench mark in a single thread
• Use default buffersize of 128K, totalsize of 16M

Experimental questions for the lab report

The full lab-report assignment will be distributed during the next lab.

The following 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 – and why? For sockets, consider both with, and without, the -s flag.
• Is using multiple threads faster or slower than using multiple processes?
python-dtrace memory leak

# The benchmark has completed - stop the DTrace instrumentation

dtrace_thread.stop()
dtrace_thread.join()

dtrace_thread.consumer.__del__()  

• Memory leak in python-dtrace results in instability
• Work around by adding an explicit call to:
  dtrace_thread.consumer.__del__()  
• Thanks to James Clarke for spotting the issue

This lab session

• 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 Jupyter/Python to analyse benchmark results
• Remember to consider the hypotheses the experimental questions are exploring.
• Use the tools in the most productive way:
  • Command line DTrace for quick exploration.
  • Jupyter for data capture, visualisation and analysis.
• Do ask us if you have any questions or need help