بسمه تعالی

گزارش تمرین عملی دوم – سوال چهارم – درس سیستم‌های عامل دکتر اسدی – نیمسال اول 405-404

نویسندگان:

1. سیداحمد موسوی‌اول – 402106648
2. عرفان تیموری – 402105813

گزارشکار.

1. تحقیق در مورد Perf:

Perf یک ابزار بسیار قدرتمند برای تحلیل کارایی (Profiling) در هسته لینوکس است. این ابزار مستقیماً از واحد پایش عملکرد (PMU) پردازنده برای جمع‌آوری داده‌های سطح پایین سخت‌افزاری مانند چرخه‌هایCPU، خطاهای کش، انشعاب‌ها یا برنچ‌های ناموفق و همچنین رویدادهای نرم‌افزاری مانند Context Switch و خطاهای صفحه (Page Fault) استفاده می‌کند. perf به توسعه‌دهندگان اجازه می‌دهد تا به سرعت گلوگاه‌های (bottlenecks) عملکردی برنامه خود را شناسایی کنند. Perf می‌تواند در چندین حالت کار کند، اما دو حالت اصلی آن عبارتند از:

1. نمونه‌برداری (Sampling): perf برنامه را در فواصل زمانی معین مثلاً هر 99 میلی‌ثانیه یا به ازای هر ۱ میلیون چرخه CPU متوقف می‌کند و بررسی می‌کند که برنامه در آن لحظه دقیقاً در کجای پشته فراخوانی (Call Stack) قرار دارد. با جمع‌آوری هزاران نمونه، یک تصویر آماری دقیق از اینکه کدام توابع بیشترین زمان CPU را مصرف می‌کنند، به دست می‌آید.

2. ردیابی (Tracing): perf می‌تواند به رویدادهای خاصی (مانند فراخوانی‌های سیستمی، خطاهای صفحه، یا رویدادهای زمان‌بندی) گوش دهد و هر بار که آن رویداد رخ می‌دهد، اطلاعاتی را ثبت کند.

دستورات رایج perf عبارتند از:

perf stat: یک خلاصه آماری از رویدادها مانند IPC (دستورالعمل بر چرخه) را برای کل اجرای برنامه نمایش می‌دهد.

perf top: شبیه به ابزار top استاندارد، اما به صورت زنده توابعی را که بیشترین بار CPU را دارند، نشان می‌دهد.

perf record, perf report: از این دو دستور در بخش عملی استفاده می‌شود و برای ضبط داده‌ها و نمایش گزارش تعاملی آن‌ها به کار می‌روند.

2. کد نابهینه (inefficient.cpp):

کد نابهینه ارائه شده (inefficient.cpp) دارای دو مشکل اساسی عملکردی است:

1. الگوریتم کند اعداد اول: تابع find_primes_slowly از یک الگوریتم بسیار کند مبتنی بر تقسیم آزمایشی استفاده می‌کند که هر عدد را به صورت جداگانه بررسی می‌کند.

2. دسترسی بد به حافظه: تابع process_data_inefficiently یک ماتریس (std::vector<std::vector<int>>) را پیمایش می‌کند. در C++، این ساختار داده به صورت سطری در حافظه ذخیره می‌شود. اما کد شما با پیمایش ستونی حلقه (j بیرونی و i داخلی) به داده‌ها دسترسی پیدا می‌کند. این امر باعث پرش‌های مداوم در حافظه شده و منجر به نرخ بالای خطای کش (Cache Miss) می‌شود که عملکرد را به شدت کاهش می‌دهد.

3. تحلیل با perf:

برای تحلیل کد به کمک ابزار perf کافیست دستورات زیر را اجرا کنیم:

```
1.  g++ -O2 -g inefficient.cpp -o inefficient
2.  sudo perf record -g ./inefficient
3.  sudo perf report
```

```
→  Desktop g++ -g -O0 -o inefficient_app inefficient.cpp
→  Desktop sudo perf record -g ./inefficient_app
Starting inefficient tasks...
Inefficient sum: 42873775000000
Total time: 29.6075 seconds
[ perf record: Woken up 48 times to write data ]
[ perf record: Captured and wrote 12.015 MB perf.data (118434 samples) ]
→  Desktop perf report
failed to open perf.data: Permission denied
→  Desktop sudo perf report
Samples: 118K of event 'cycles:P', Event count (approx.): 134592443408
  Children      Self  Command          Shared Object        Symbol
+   99.95%     0.00%  inefficient_app  inefficient_app      [.] _start
+   99.95%     0.00%  inefficient_app  libc.so.6            [.] __libc_start_main@@GLIBC_2.34
+   99.95%     0.00%  inefficient_app  libc.so.6            [.] __libc_start_call_main
+   99.95%     0.00%  inefficient_app  inefficient_app      [.] main
+   97.32%    81.12%  inefficient_app  inefficient_app      [.] process_data_inefficiently(int)
+    8.01%     6.13%  inefficient_app  inefficient_app      [.] std::vector<int, std::allocator<int> >::operator[](unsigned long)
+    7.61%     6.04%  inefficient_app  inefficient_app      [.] std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::allocator<int> > > >::operator[](un
+    4.52%     0.00%  inefficient_app  inefficient_app      [.] std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::allocator<int> > > >::vector(unsign
+    4.52%     0.00%  inefficient_app  inefficient_app      [.] std::vector<std::vector<int, std::allocator<int> >, std::allocator<std::vector<int, std::allocator<int> > > >::_M_fill_initi
+    4.52%     0.00%  inefficient_app  inefficient_app      [.] std::vector<int, std::allocator<int> >* std::__uninitialized_fill_n_a<std::vector<int, std::allocator<int> >*, unsigned long
+    4.52%     0.00%  inefficient_app  inefficient_app      [.] std::vector<int, std::allocator<int> >* std::uninitialized_fill_n<std::vector<int, std::allocator<int> >*, unsigned long, st
+    4.52%     0.00%  inefficient_app  inefficient_app      [.] std::vector<int, std::allocator<int> >* std::__uninitialized_fill_n<false>::__uninit_fill_n<std::vector<int, std::allocator<
+    4.52%     0.00%  inefficient_app  inefficient_app      [.] std::vector<int, std::allocator<int> >* std::__do_uninit_fill_n<std::vector<int, std::allocator<int> >*, unsigned long, std:
+    4.52%     0.00%  inefficient_app  inefficient_app      [.] void std::_Construct<std::vector<int, std::allocator<int> >, std::vector<int, std::allocator<int> > const&>(std::vector<int,
+    4.51%     0.00%  inefficient_app  inefficient_app      [.] std::vector<int, std::allocator<int> >::vector(std::vector<int, std::allocator<int> > const&)
+    4.33%     1.41%  inefficient_app  libc.so.6            [.] __memmove_evex_unaligned_erms
+    4.29%     0.00%  inefficient_app  inefficient_app      [.] int* std::__uninitialized_copy_a<__gnu_cxx::__normal_iterator<int const*, std::vector<int, std::allocator<int> > >, int*, in
+    4.29%     0.00%  inefficient_app  inefficient_app      [.] int* std::uninitialized_copy<__gnu_cxx::__normal_iterator<int const*, std::vector<int, std::allocator<int> > >, int*>(__gnu_
+    4.29%     0.00%  inefficient_app  inefficient_app      [.] int* std::__uninitialized_copy<true>::__uninit_copy<__gnu_cxx::__normal_iterator<int const*, std::vector<int, std::allocator
+    4.29%     0.00%  inefficient_app  inefficient_app      [.] int* std::copy<__gnu_cxx::__normal_iterator<int const*, std::vector<int, std::allocator<int> > >, int*>(__gnu_cxx::__normal_
+    4.29%     0.00%  inefficient_app  inefficient_app      [.] int* std::__copy_move_a<false, __gnu_cxx::__normal_iterator<int const*, std::vector<int, std::allocator<int> > >, int*>(__gn
+    4.29%     0.00%  inefficient_app  inefficient_app      [.] int* std::__copy_move_a1<false, int const*, int*>(int const*, int const*, int*)
+    4.29%     0.00%  inefficient_app  inefficient_app      [.] int* std::__copy_move_a2<false, int const*, int*>(int const*, int const*, int*)
+    3.86%     0.02%  inefficient_app  [kernel.kallsyms]    [k] asm_exc_page_fault
+    2.42%     0.01%  inefficient_app  [kernel.kallsyms]    [k] exc_page_fault
+    2.35%     0.02%  inefficient_app  [kernel.kallsyms]    [k] do_user_addr_fault
+    2.21%     0.04%  inefficient_app  [kernel.kallsyms]    [k] handle_mm_fault
+    2.12%     0.08%  inefficient_app  [kernel.kallsyms]    [k] __handle_mm_fault
+    2.03%     0.01%  inefficient_app  [kernel.kallsyms]    [k] handle_pte_fault
+    1.96%     0.05%  inefficient_app  [kernel.kallsyms]    [k] do_anonymous_page
+    1.36%     0.01%  inefficient_app  inefficient_app      [.] find_primes_slowly(long)
+    1.36%     1.36%  inefficient_app  inefficient_app      [.] is_prime_naive(long)
+    0.94%     0.02%  inefficient_app  [kernel.kallsyms]    [k] __mem_cgroup_charge
+    0.76%     0.00%  inefficient_app  [kernel.kallsyms]    [k] entry_SYSCALL_64_after_hwframe
+    0.75%     0.01%  inefficient_app  [kernel.kallsyms]    [k] do_syscall_64
+    0.75%     0.00%  inefficient_app  [kernel.kallsyms]    [k] x64_sys_call
```

نتیجه مورد انتظار: این گزارش به وضوح نشان می‌دهد که درصد بسیار بالایی از زمان CPU در تابع is_prime_naive که توسط find_primes_slowly فراخوانی می‌شود و همچنین در تابع process_data_inefficiently صرف شده است.

4. تحقیق در مورد FlameGraph:

FlameGraph (نمودار شعله‌ای) یک ابزار بصری‌سازی قدرتمند برای داده‌های پروفایلینگ (مانند خروجی perf) است. این نمودار پشته‌های فراخوانی (Call Stacks) را به صورت گرافیکی نمایش می‌دهد به طوری که:

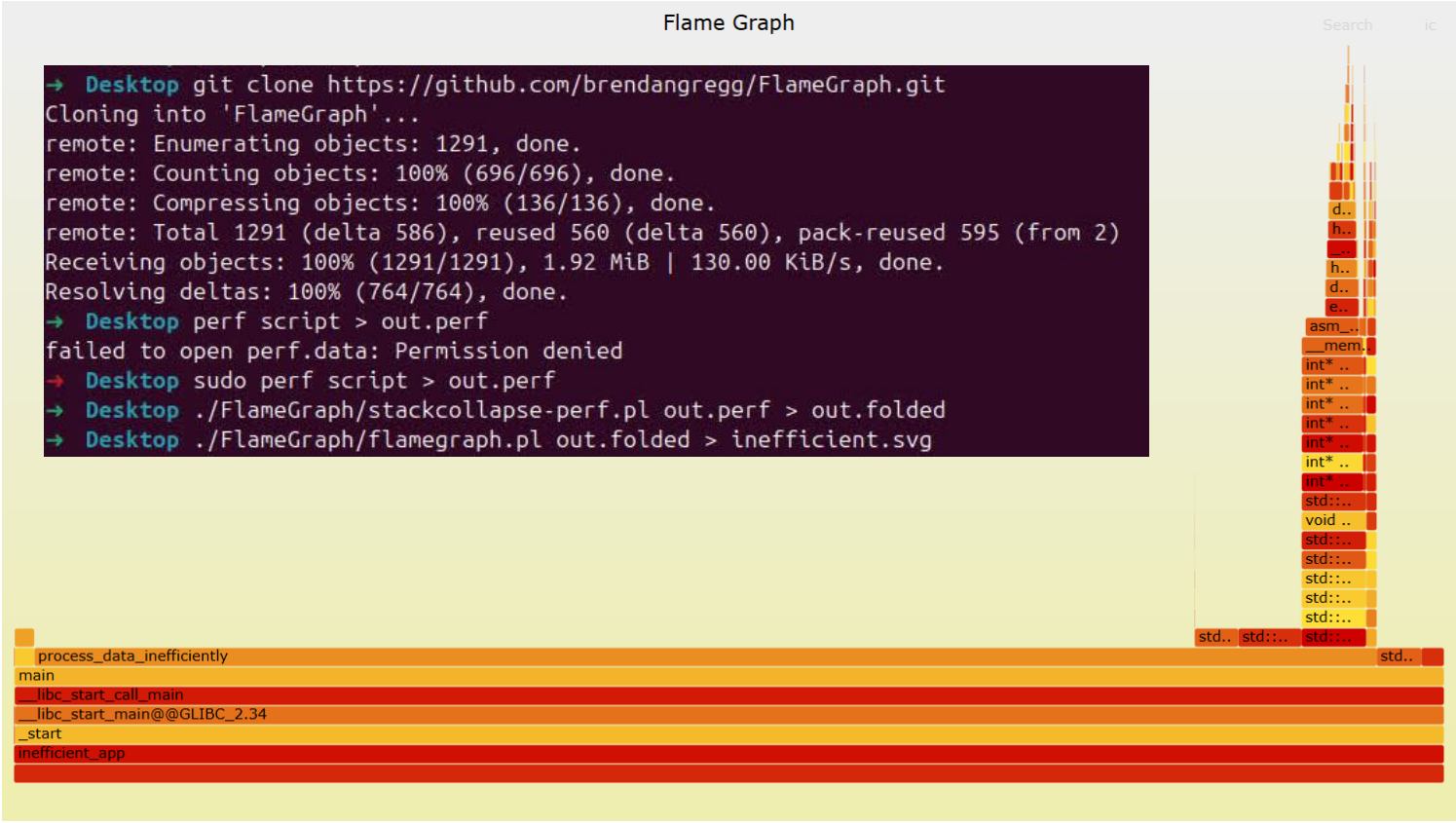
- محور Y (عمودی): عمق پشته فراخوانی را نشان می‌دهد (تابع main در پایین و توابع فراخوانی شده در بالا).
- محور X (افقی): نشان‌دهنده جمعیت نمونه‌ها است. هرچه یک بلوک پهن‌تر باشد، به این معنی است که آن تابع زمان بیشتری از CPU را به خود اختصاص داده است (یا مستقیماً یا توسط توابعی که فراخوانی کرده). این بلوک‌های پهن، گلوگاه‌های اصلی برنامه هستند. (ترتیب افقی بلوک‌ها اهمیتی ندارد).

5. ساخت FlameGraph:

دستورات زیر را اجرا می‌کنیم:

```
1.  perf record -F 99 -g ./inefficient
2.  perf script > out.perf
3.  FlameGraph/stackcollapse-perf.pl out.perf > out.folded
4.  FlameGraph/flamegraph.pl out.folded > inefficient.svg
```

که در نتیجه یک فایل svg برای ما تولید می‌شود که نشان می‌دهد اجرای هر تابع چقدر طول می‌کشد (این فایل در کنار گزارش آپلود شده است).



6. بهبود بخش‌های نابهینه (کد efficient.cpp):

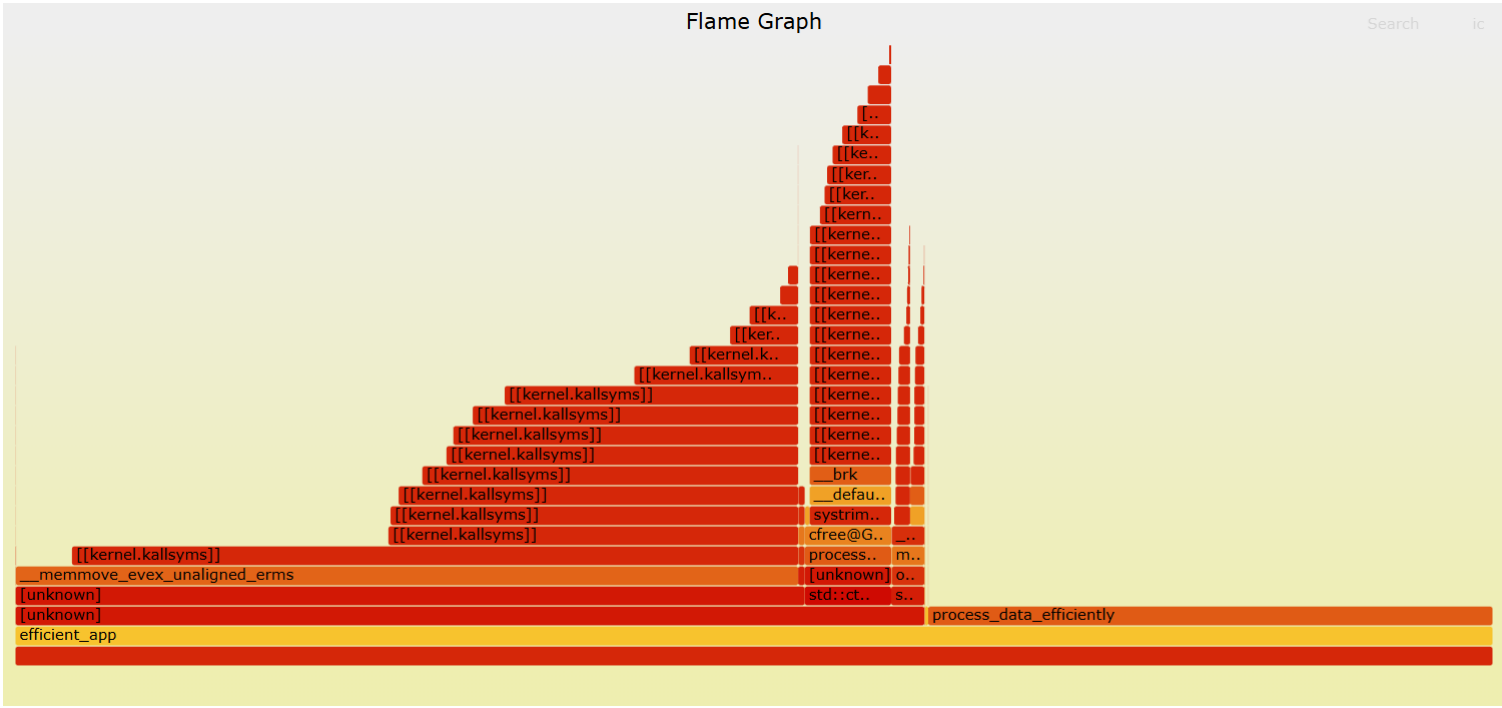
با مشاهده FlameGraph، دو گلوگاه شناسایی و در efficient.cpp به این صورت بهینه شدند:

1. بهبود دسترسی به حافظه: تابع process_data_inefficiently با process_data_efficiently جایگزین شد. در این نسخه، ترتیب حلقه‌ها برعکس شده و پیمایش به صورت سطری (حلقه i بیرونی و j داخلی) انجام می‌شود. این کار با نحوه ذخیره‌سازی داده‌ها در حافظه مطابقت کامل دارد، از مجاورت کش (Cache Locality) به خوبی بهره می‌برد و خطاهای کش را تقریبا به صفر می‌رساند.

2. بهبود اعداد اول: تابع find_primes_slowly با find_primes_efficiently جایگزین شد. این تابع جدید از الگوریتم غربال اراتوستن استفاده می‌کند. این الگوریتم به جای بررسی تک‌تک اعداد، مضارب اعداد اول را علامت می‌زند و به طور چشمگیری سریع‌تر است.

7. نمایش نتیجه بهینه‌سازی:

با تکرار مراحل ۳ و ۵ برای کد efficient.cpp، یک فایل efficient.svg جدید تولید می‌شود (این فایل در کنار گزارش آپلود شده است).



نتیجه مورد انتظار: با مقایسه efficient.svg با inefficient.svg، به وضوح دیده می‌شود که برج‌های مربوط به دو تابع بهینه‌شده بسیار باریک‌تر شده‌اند. این نشان می‌دهد که آنها اکنون بخش بسیار ناچیزی از کل زمان اجرای برنامه را به خود اختصاص می‌دهند و بهینه‌سازی موفقیت‌آمیز بوده است.

8. تحقیق در مورد Tracy:

Tracy یک ابزار پروفایلینگ و بصری‌سازی آنی (real-time) و بسیار کم‌سربار (low-overhead) است که به طور خاص برای C++ طراحی شده است. برخلاف perf که یک نمونه‌بردار (sampler) است، Tracy یک ابزار دقیق است. این به آن معناست که باید به صورت دستی، با استفاده از ماکروهای ساده‌ای (مانند ;ZoneScoped)، ناحیه‌های مورد نظر خود را در کد مشخص کنیم. Tracy سپس به برنامه در حال اجرا متصل شده و به صورت زنده، خط زمانی اجرای توابع در تمام رشته‌ها، قفل‌ها (locks)، تخصیص‌های حافظه و ... را با دقت بسیار بالا (سطح نانوثانیه) نمایش می‌دهد.

9. تحلیل با Tracy:

برای تحلیل دقیق‌تر، کدها با Tracy ابزار دقیق می‌شوند:

1. آماده‌سازی: کتابخانه Tracy به پروژه اضافه شده و هدر Tracy.hpp در فایل‌های cpp. گنجانده می‌شود.

2. نشانه‌گذاری: ماکروی ;ZoneScoped در ابتدای توابع ...find_primes_ و ...process_data_ (در هر دو فایل) قرار داده می‌شود.

3. اجرا: سرور Tracy (GUI) اجرا شده و سپس برنامه‌های کامپایل شده با فلگ TRACY_ENABLE اجرا می‌شوند.

نتایج و خروجی اجرا کردن کد غیربهینه و بهینه به صورت زیر است: