
PERFORMANCE PROFILING TOOL

Minsoo Ryu

**Real-Time Computing and Communications Lab.
Hanyang University**

msryu@rtcc.hanyang.ac.kr

OUTLINE

❑ EXAMPLE SOURCE

❑ PROFILING

- PERF
- GPROF
- OPROFILE

❑ TRACING

- STRACE
- LTRACE
- FTRACE

EXAMPLE SOURCE

EXAMPLE SOURCE

□ UDP 채팅 프로그램

- Server
 - ./UDP_server [port]
- Client1 / Client2
 - ./UDP_client [IP address] [port] [user name]

EXAMPLE SOURCE

```
test@ubuntu: ~/chat
test@ubuntu:~/chat$ ./UDP_server 4000
-----
socket success
bind success
-----
<now server is waiting for client's connection>
^C
test@ubuntu:~/chat$ clear

test@ubuntu:~/chat$ ./UDP_server 4000
-----
socket success
bind success
-----
<now server is waiting for client's connection>
not exist!
add! index[0]
-----
1 client remain
[BroadcastMsg] [JYP] hi

not exist!
add! index[1]
-----
2 client remain
[BroadcastMsg] [JY] hi

already exist! index[1]
[BroadcastMsg] [JY] where R U?

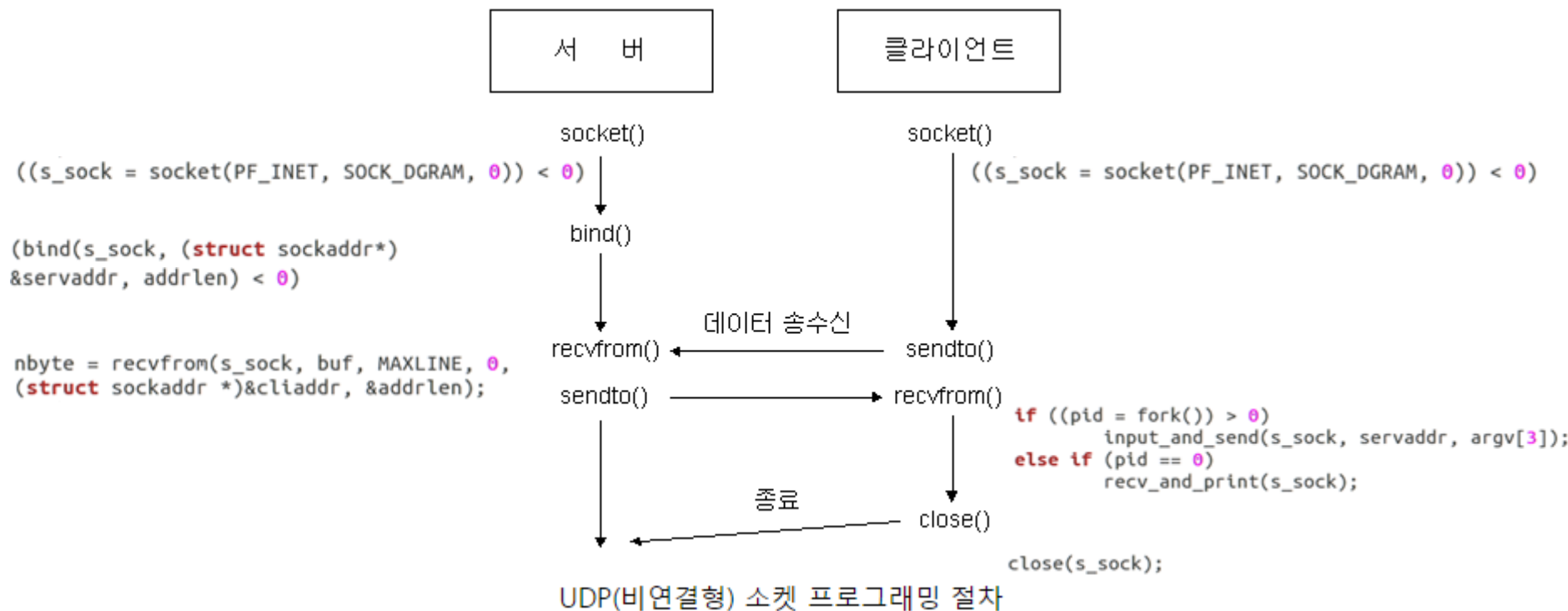
already exist! index[0]
[BroadcastMsg] [JYP] I am going to go school.

already exist! index[1]
[BroadcastMsg] [JY] can you come here?
```

```
test@ubuntu: ~/chat/client1
test@ubuntu:~/chat/client1$ ./UDP_client 192.168.177.139 4000 JYP
socket success!
hi
[JY] hi
[JY] where R U?
I am going to go school.
[JY] can you come here?
sure. where R U?
[JY] I am near school
okey I go there.
[JY] thanks. see you there
welcome see you then.
exit
good bye~
test@ubuntu:~/chat/client1$
```

```
test@ubuntu: ~/chat/client2
test@ubuntu:~/chat/client2$ ./UDP_client 192.168.177.139 4000 JY
socket success!
hi
where R U?
[JYP] I am going to go school.
can you come here?
[JYP] sure. where R U?
I am near school
[JYP] okey I go there.
thanks. see you there
[JYP] welcome see you then.
exit
good bye~
test@ubuntu:~/chat/client2$
```

EXAMPLE SOURCE



PERFORMANCE PROFILING TOOL IN LINUX

❑ EXAMPLE SOURCE

❑ PROFILING

- PERF
- GPROF
- OPROFILE

❑ TRACING

- STRACE
- LTRACE
- FTRACE

PROFILING

PERF

PERF

□ What is perf

- Performance Counters for Linux
- Profiler는 다양한 기술을 통해서 데이터 수집
 - 하드웨어 인터럽트, 코드 계측, 명령어 집합 시뮬레이션, 운영 체제 후킹, 성능 카운터
- 기본적으로는 **CPU**와 함께 제공되는 **PMU**의 도움을 받아 동작
- 사용자 레벨의 프로그램이 커널 소스에 포함되게 된 이유는 **perf**가 커널의 **ABI**와 밀접한 연관

PERF

□ Perf 설치

- **sudo apt-get install linux-tools-common**
- **sudo apt-get install linux-tools-3.19.0-25-generic**
 - Linux-cloud-tools-3.19.0.25-generic

PERF

□ Usage of part

- Perf <command> [option]

Commands	Descriptions
list	특정 컴퓨터에서 사용할 수 있는 이벤트를 나열
stat	실행된 지시 사항 및 소비된 클럭 사이클을 포함하여 일반적인 성능 이벤트의 전체 통계를 제공
top	실시간으로 성능 카운트 프로파일을 생성 및 표시
record	성능 데이터를 perf report 를 사용하여 나중에 분석할 수 있는 파일에 기록
report	파일에서 성능 데이터를 읽고 기록된 데이터를 분석
annotate	컴파일 되지 않은 코드를 간단한 주석을 통해 나타냄
diff	2개의 perf. Data 를 비교하여 나타냄

PERF

□ Usage of part

- 하드웨어 이벤트는 **modifiers**를 덧붙여서 그 적용 범위를 제한
 - U: user-level에서 발생하는 이벤트
 - k: kernel-level에서 발생하는 이벤트
 - h: hypervisor에서 발생하는 이벤트
 - H: host machine에서 발생하는 이벤트
 - G: guest machine에서 발생하는 이벤트

PERF STAT

□ Perf stat

```
test@ubuntu: ~/chat/client2
good bye~

Performance counter stats for './UDP_client 192.168.177.139 4000 y':

 5260.179993      task-clock (msec)      #    0.134 CPUs utilized
    1,592        context-switches      #    0.303 K/sec
     0          cpu-migrations        #    0.000 K/sec
    98          page-faults           #    0.019 K/sec
     0          cycles                #    0.000 GHz
     0          stalled-cycles-frontend #    0.00% frontend cycles idle
     0          stalled-cycles-backend  #    0.00% backend  cycles idle
     0          instructions           #
     0          branches               #    0.000 K/sec
     0          branch-misses          #    0.000 K/sec

 39.385122300 seconds time elapsed

test@ubuntu:~/chat/client2$
```

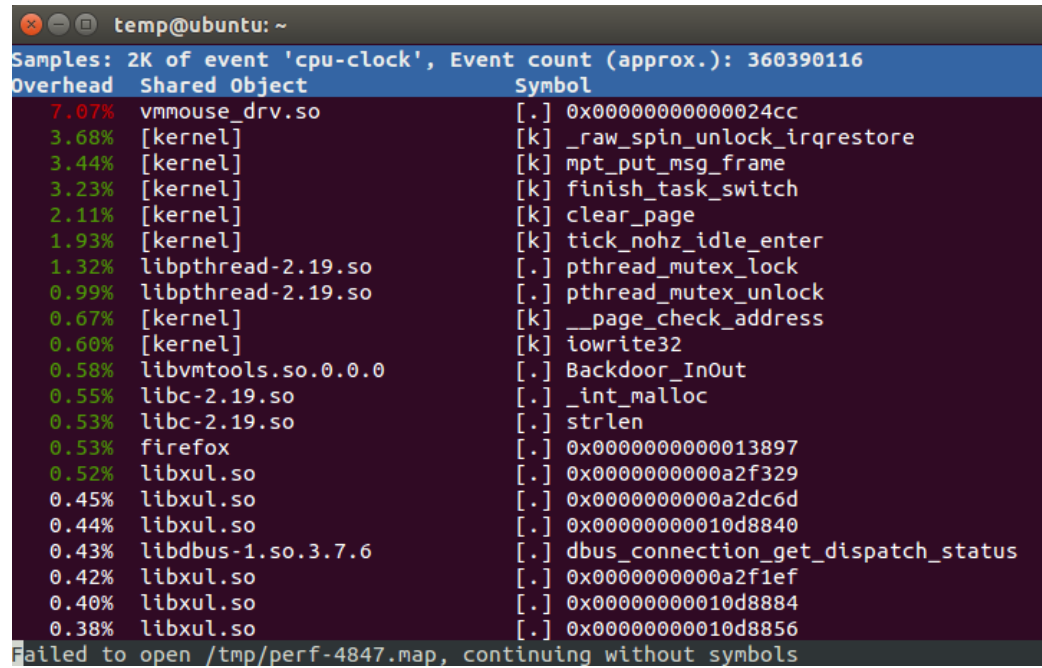
- Task-clock
 - 횟수로 5260.179993이 발생. 평균 CPU 사용률은 0.134
- Context-switches
 - 1.592회의 switch가 발생. 1초당 0.303 k/sec이 발생
- Page-faults
 - 98회의 page-faults가 발생. 1초당 0.019 k/sec이 발생

PERF TOP

□ Perf top

- Run-time 동안 시스템을 분석
 - Top command와 비슷한 기능을 제공
- 실시간 시스템 모니터링 기능 제공

```
# perf top
```



```
temp@ubuntu: ~  
Samples: 2K of event 'cpu-clock', Event count (approx.): 360390116  
Overhead Shared Object Symbol  
7.07% vmmouse_drv.so [.] 0x00000000000024cc  
3.68% [kernel] [k] _raw_spin_unlock_irqrestore  
3.44% [kernel] [k] mpt_put_msg_frame  
3.23% [kernel] [k] finish_task_switch  
2.11% [kernel] [k] clear_page  
1.93% [kernel] [k] tick_nohz_idle_enter  
1.32% libpthread-2.19.so [.] pthread_mutex_lock  
0.99% libpthread-2.19.so [.] pthread_mutex_unlock  
0.67% [kernel] [k] __page_check_address  
0.60% [kernel] [k] iowrite32  
0.58% libvmtools.so.0.0.0 [.] Backdoor_InOut  
0.55% libc-2.19.so [.] _int_malloc  
0.53% libc-2.19.so [.] strlen  
0.53% firefox [.] 0x0000000000013897  
0.52% libxul.so [.] 0x00000000000a2f329  
0.45% libxul.so [.] 0x00000000000a2dc6d  
0.44% libxul.so [.] 0x000000000010d8840  
0.43% libdbus-1.so.3.7.6 [.] dbus_connection_get_dispatch_status  
0.42% libxul.so [.] 0x00000000000a2f1ef  
0.40% libxul.so [.] 0x000000000010d8884  
0.38% libxul.so [.] 0x000000000010d8856  
Failed to open /tmp/perf-4847.map, continuing without symbols
```

PERF RECORD

□ Perf record

- Etvents를 기록
- 기록된 데이터는 기본적으로 perf.data에 기록
 - File name 변경 가능
- 사용 방법
 - 특정 command에 대한 세부 사항 기록
 - 의심스러운 process 분석
 - 프로세스의 성능 저하 의 원인을 결정

```
# perf record [옵션] [실행파일]
```

```
# ls  
perf.data
```

```
temp@ubuntu:~$ ls  
bin Downloads perf.data repo Tizen_2.4  
Desktop examples.desktop Pictures sample Videos  
Documents Music Public Templates vim  
temp@ubuntu:~$
```


PERF REPORT

□ Perf report

- Perf.data에 저장된 내용을 보여 줌

```
# perf report
```

```
test@ubuntu: ~/chat/client2
Samples: 4K of event 'cpu-clock', Event count (approx.): 1000750000
Overhead Command Shared Object Symbol
 99.13% UDP_client UDP_client [.] slow
  0.27% UDP_client [kernel.kallsyms] [k] finish_task_switch
  0.20% UDP_client [kernel.kallsyms] [k] __do_softirq
  0.15% UDP_client [kernel.kallsyms] [k] _raw_spin_unlock_irqrestore
  0.02% UDP_client UDP_client [.] f
  0.02% UDP_client [kernel.kallsyms] [k] free_pages_and_swap_cache
  0.02% UDP_client [kernel.kallsyms] [k] mpt_put_msg_frame
  0.02% UDP_client [kernel.kallsyms] [k] n_tty_write
  0.02% UDP_client [kernel.kallsyms] [k] queue_work_on
  0.02% UDP_client [kernel.kallsyms] [k] tty_paranoia_check
  0.02% UDP_client libc-2.19.so [.] _IO_default_uflow
  0.02% UDP_client libc-2.19.so [.] _IO_fgets
  0.02% UDP_client libc-2.19.so [.] __mcount_internal
  0.02% UDP_client libc-2.19.so [.] vfprintf
Press '?' for help on key bindings
```

PERF ANNOTATE & DIFF

❑ Perf annotate

- Perf.data를 읽어서 컴파일 되지 않은 코드를 주석화하여 보여줌
- Use cases
 - Source code의 time-consuming을 확인

```
# perf anotate
```

❑ Perf diff

- 2개의 perf. Data를 읽어서 2개의 perf.data의 차이점을 보여줌
- Use cases
 - 2개의 perf.data 는 new perf.data와 old perf.data

```
# perf diff
```

과제

- ❑ Perf를 사용하여 **client1** 분석 예제 프로그램 분석
 - Stat를 사용하여 **client** 상태 확인
 - record / report 사용하여 **client** 상태 기록
 - Top을 사용하여 **client** 동작 확인
- ❑ Perf.data나 record된 파일 제출 및 캡처 사진 제출
- ❑ Stat와 top을 사용하여 확인 된 결과 캡처 사진 제출

GPROF

GPROF

□ What is gprof

- Gprof는 gcc의 binutils 패키지에 포함되어 있음
- 어느 함수에 부하가 많이 걸리는지 확인을 위하여 사용

□ Gprof 패키지 설치

- `sudo apt-get install binutils`
 - Binutils dependency: Bash, Coreutils, Diffutils, GCC, Gettext, Glibc, Grep, Make, Perl, Sed, Texinfo

GPROF

□ Gprof의 동작방식

- 함수 호출 시 진입에서 종료할 때 까지의 시간을 기록 하여, 이 정보에 대한 통계를 제공하는 방식
 - 프로그램 수행시간 동안 각 함수의 호출 횟수와 함수 호출
 - 각 함수가 호출될 때마다 횟수를 기록
 - ✓ -pg: 시간함수(mcount)를 넣어주는 옵션

GPROF

□ 내부 원리

- 타이머
 - 0 ms마다 PC 조사
 - Settimer 를 main이전에 호출하여 SIGPROF 시그널 발생
 - 시그널 핸들러에서 PC 카운터 증가
- Enter/exit 후킹
 - 함수 호출 전에 mcount 함수 호출
 - 호출 전의 PC와 호출 후의 PC를 이용 콜 그래프 정보 작성, 정확한 함수 호출 횟수 기록

GPROF

□ Gprof profile 종류

▪ Flat profile

- 함수 별로 사용하는 **CPU** 시간 / 호출 횟수를 보여 줌
- 수집한 전체 **profiling** 정보의 간단한 요약
- 성능을 높이기 위해 어떤 함수를 수정해야 할 지 나타냄
 - ✓ 프로그램이 각 함수에서 보낸 시간과 함수가 실행된 횟수 표시

▪ Call Graph

- 어떤 함수 호출을 없애거나 다른 효율적인 함수로 대체할지 제안
- 함수들간의 관계를 드러내고, 감춰진 버그를 알려주기도 함
- 호출 그래프를 확인 후, 특정 코드 경로를 최적화 가능
- 각 함수마다 해당 함수를 호출한 상위 함수, 호출된 횟수, 서브루틴에서 소비한 시간과 같은 세부 내역을 표시

GPROF

□ Gprof 저장 및 실행

```
# gcc -o a.out a.c -pg
```

- 시간함수를 일일이 넣어주는 옵션

```
# ./a.out
```

```
# ls  
gmon.out
```

```
# gprof a.out gmon.out
```

■ 추가옵션

- -l 옵션 : 행 별 소요 시간
- -l -A -x : 소스코드 출력, 행 별 소요 시간
- -F 함수 : 특정 함수의 콜 그래프 출력

GPROF

□ Gprof flat profile

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
99.99	16.28	16.28	5000	0.00	0.00	slow
0.00	16.28	0.00	1	0.00	3.26	f
0.00	16.28	0.00	1	0.00	13.02	g

- Slow 함수가 거의 대부분의 시간을 사용
- F 함수는 1만 호출하지만, 호출당 평균 3.26밀리 초 사용
- G 함수는 1번만 호출하지만, 호출당 평균 13.02 밀리 초 사용

GPROF

□ Gprof 내용

- % time: 전체 실행 시간에서 해당 함수 실행에 걸린 시간의 백분율
 - 보는 시각 차이, 옵션으로 제외한 함수 등 여러 이유로 총합은 100%가 안됨
- Cumulative seconds: 함수와 이전 함수의 실행 시간을 합산한 값
- Self seconds: 해당 함수 단독 실행에 걸린 시간
 - 목록을 정렬하는 1차 키
- Calls: 함수를 profiling 할 경우, 함수 호출 횟수
 - Profiling하지 않는 함수일 경우 비어있음

GPROF

□ Gprof 내용

- Self ms/call: 함수를 profiling 할 경우, 호출당 평균 실행 시간 실행 시간은 해당 함수만 고려
 - Profiling하지 않는 함수일 경우 비어 있음
- Total ms/call: 함수를 profiling 할 경우, 호출당 평균 실행 시간 실행 시간은 해당 함수와 함수에서 호출한 모든 함수까지 고려
 - Profiling하지 않는 함수는 비어 있음
- Name: 함수 이름
 - 목록을 정렬하는 2차 키
 - 괄호 안에 있는 색인은 gprof 목록에서 함수 위치를 표시
 - ✓ 출력 시 gprof 목록에 나타낼 순서 지시

GPROF

□ Gprof call_graph profile

```
Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.06% of 16.28 seconds

index % time    self  children  called      name
-----
[1]   100.0    3.26    0.00    1000/5000    f [4]
        13.02    0.00    4000/5000    g [3]
        16.28    0.00    5000         slow [1]
-----
[2]   100.0    0.00    16.28         <spontaneous>
        0.00    13.02         1/1    main [2]
        0.00    3.26         1/1    g [3]
        0.00         1/1    f [4]
-----
[3]    80.0    0.00    13.02         1/1    main [2]
        0.00    13.02         1    g [3]
        13.02    0.00    4000/5000    slow [1]
-----
[4]    20.0    0.00    3.26         1/1    main [2]
        0.00    3.26         1    f [4]
        3.26    0.00    1000/5000    slow [1]
-----
```

□ 현재 함수 행에서 각 필드의 의미

- **Index:** 각 항목에 할당된 고유 번호
 - 색인 번호는 숫자로 취급해 정렬
 - 함수를 찾기 쉽도록 함수 이름 옆에도 표시
- **% time:** 해당 함수와 자식 함수에서 보낸 총 시간 백분율
 - 보는 시각 차이, 옵션으로 제외한 함수 등 여러 이유로 총합은 100%가 안됨
- **Self:** 해당 함수에서 보낸 시간
- **Children:** 해당 함수가 호출된 횟수
 - 함수가 자신을 재귀적으로 호출하는 경우, 비재귀적 호출 횟수 + 재귀적 호출 횟수로 표시
- **Name:** 해당 함수 이름
 - 함수 이름 뒤에 색인 번호가 표시
 - 함수가 사이클에 속할 경우, 이름과 색인 번호 사이에 사이클 번호가 표시

□ 자식 함수 행에서 각 필드의 의미

- **Self:** 현재 함수가 자식 함수를 호출해 자식 함수에서 보낸 총 시간
- **Children:** 현재 함수가 자식 함수를 호출해 자식 함수의 자식에서 보낸 총 시간
- **Called:** 현재 함수가 자식 함수를 호출한 횟수/자식 함수가 호출된 총 횟수 자식 함수가 호출된 총 횟수는 재귀적 호출 횟수 미포함
- **Name:** 자식 함수 이름. 자식 함수 색인 번호가 이름 옆에 표시. 자식이 사이클에 속할 경우, 이름과 색인 번호 사이에 사이클 번호가 표시

□ 부모 함수 행에서 각 필드의 의미

- **Self:** 부모 함수가 현재 함수를 호출해 현재 함수에서 보낸 총 시간
- **Children:** 부모 함수가 현재 함수를 호출해 현재 함수의 자식 함수에서 보낸 총 시간
- **Called:** 부모 함수가 현재 함수를 호출한 횟수/현재 함수가 호출된 총 횟수 현재 함수가 호출된 총 횟수는 재귀적 호출 횟수 미포함
- **Name:** 부모 함수 이름
 - 부모 함수 색인 번호가 이름 옆에 표시
 - 부모가 사이클에 속할 경우, 이름과 색인 번호 사이에 사이클 번호가 표시 부모 함수 결정이 불가능하다면 <spontaneous>라는 단어가 표시되고, 나머지 필드는 공란

GPROF

□ Gprof 다른 기능

- Gprof를 사용하면 상세한 소스 목록, 줄 단위 **profiling** 정보를 얻음
 - 어느 부분을 최적화할지 결정할 때, **gprof** 정보가 유용
- Gprof를 사용하여 소스코드를 한 줄씩 살펴보며 비효율적인 부분을 탐색 가능
 - 줄 단위 **profiling**과 **flat profile**을 사용하면 코드가 어떤 경로로 가장 자주 실행되는지 확인 가능
 - 상세한 소스 목록을 사용하여 함수 호출에서 반복문과 분기문을 살펴보면서 어떤 반복문이 가장 많이 실행되는지 어떤 분기를 가장 많이 사용하는지 확인 가능
- 최적의 성능을 얻기 위해 코드를 세밀히 조절할 때 유용

OPROFILE

과제

- ❑ **Gprof를 사용하여 Client2를 분석**
 - **Client2를 flat profile 이용하여 분석**
 - **Flat profile로 분석된 client2를 간단히 설명**
- ❑ **Gprof를 사용하여 확인한 flat profile 캡처 후 제출**
- ❑ **Flat profile로 분석한 내용 추가**

OPROFILE

□ What is Oprofile

▪ Oprofile

- OProfile은 Linux 시스템 전반에 걸친 프로파일러
 - ✓ Profiling은 백그라운드에서 사용자가 모르게 진행되며, 아무때나 프로파일 데이터를 얻을 수 있음

▪ Oprofile

- 실행중인 프로세스의 **CPU Usage**를 profiling
- 실행중인 프로세스의 어떤 함수가 **CPU** 자원을 많이 사용하는지를 식별
- 특정 함수가 많은 자원을 사용할 경우, 프로그램은 그 함수 콜을 줄일 수 있는 방법으로 재설계될 수 있음
- **CPU** 사용률이 낮은 장비에서는 별로 쓸모가 없음
 - ✓ CPU 사용률 낮은 장비: I/O 바운딩 서버, Mutex 대기
- **Database** 개발자에게 자주 사용

OPROFILE

- Oprofile은 작업 부하가 낮은, 시스템 전체 성능 감시 도구
 - ✓ 시스템 상 실행 가능 프로그램들에 대한 정보
 - ✓ 사용된 메모리, L2 캐쉬 요청 수, 전송된 인터럽트 숫자
- Oprofile은 대체 하드웨어(타이머)를 사용하여 카운터가 인터럽트를 발생할 때마다 성능과 관련된 데이터 샘플을 수집
 - ✓ 데이터 샘플은 주기적으로 디스크에 기록
 - ✓ 기록된 데이터를 사용하여 시스템 수준 성능과 응용 프로그램 수준 성능에 대한 리포트를 생성

OPROFILE

□ oprofile 설치

- 1.
 - sudo apt-get install libiberty-dev
 - sudo apt-get install oprofile
- 2.
 - wget <http://prdownloads.sourceforge.net/oprofile/oprofile-1.1.0.tar.gz>
 - tar xvfz oprofile-1.1.0.tar.gz
 - cd oprofile-1.1.0
 - sudo ./configure --with-kernel-support
 - Make install

OPROFILE

□ Usage of part

▪ Oprofile

Commancds	Descriptions
opcontrol	어떤 데이터를 수집할 것인지 설정
op_help	시스템 프로세서에 사용 가능한 작업과 각 작업에 대한 간략한 설명을 보여줌
op_merge	동일한 실행 프로그램에서 수집한 여러 샘플을 하나로 병합
op_to_source	만일 응용 프로그램이 디버깅 심볼을 사용하여 컴파일된 경우 실행 프로그램에 주석 추가된 소스를 생성
oprofiled	데몬으로 실행되어 샘플 데이터를 디스크에 주기적으로 기록
oprofpp	프로파일 데이터를 검색
op_import	외부 이진 형식에서 시스템의 원시 형식으로 변환

OPROFILE

□ Oprofile 설정

- Oprofile을 실행하기 위해서는 먼저 설정 필요
 - 최소한 커널을 감시하도록 선택
- Opcontrol을 사용하여 Oprofile을 설정
 - /root/.oprofile/daemonrc: opcontrol 명령이 실행된 후 파일 저장소
- Oprofile 초기화
 - Opcontrol -init
 - Opcontrol -deinit
 - Modprobe oprofile timer=1
 - Opcontrol -start
 - ✓ Opcontrol -stop

OPROFILE

□ Oprofile 설정

- Oprofile 시작과 정지
 - Opcontrol --start: Oprofile을 사용하여 시스템 감시를 시작하시려면 루트로 로그인 하신 후 명령어 실행
 - /root/.oprofile/daemonrc에 저장
 - Opcontrol --shutdown
 - ✓ OProfile 데몬인 oprofiled가 시작
 - ✓ 이 데몬은 주기적으로 샘플 데이터를 /var/lib/oprofile/samples/ 디렉토리에 기록
 - ✓ 데몬의 로그 파일은 /var/lib/oprofile/oprofiled.log에 저장
 - ✓ 만일 다른 설정 옵션을 사용하여 oprofile을 재 시작하시면, 이전 세션의 샘플 파일은 자동으로 /var/lib/oprofile/samples/session-*N* 디렉토리에 백업 저장
 - 여기서 *N*은 이전에 백업된 세션 숫자에 1을 더한 숫자

PERFORMANCE PROFILING TOOL IN LINUX

❑ EXAMPLE SOURCE

❑ PROFILING

- PERF
- GPROF
- OPROFILE

❑ TRACING

- STRACE
- LTRACE
- FTRACE

TRACING

STRACE & LTARCE

STRACE

□ System call 과 signal 추적

- 특정 프로세스와 자식 프로세스의 system call과 프로세스에 전달되는 signal을 추적할 수 있는 명령 행 도구
- 디버깅 할 때뿐 아니라 애플리케이션 동작하는 내부 원리나 구조를 파악하는 데에도 많은 도움
 - Kernel의 시스템 콜을 추적함으로 확인

□ Strace 패키지 설치

- `sudo apt-get install strace`

STRACE

□ Strace

▪ option

option	Descriptions
-c	각각의 시스템 콜에 대한 시간, 콜 개수, 에러를 카운트하고 프로그램 탈 출시 보고
-f	추적 중인 프로세스가 fork 한 자식 프로세스들을 추적
-r	각 시스템 콜에 대한 엔트리 상의 관련 타임스탬프를 출력
-t	각 라인에 시간을 출력
-T	시스템 콜에 소요된 시간을 출력
-e	추적할 이벤트가 어떤 것인지 그것을 어떻게 추적할 것인지를 변경하는 한정 표현식
-p	PID 프로세스 아이디를 지정
-s	Strsize 출력할 수 있는 최대 문자열 크기를 지정(기본값 32)
-S	특정 컬럼을 정렬(-c와 함께 사용)

❑ Strace client 확인 결과

```
test@ubuntu:~/chat/client$ strace ./UDP_client 192.168.177.139 4000 y
execve("./UDP_client", [". /UDP_client", "192.168.177.139", "4000", "y"], [/ 61 vars *]) = 0
brk(0) = 0xee1000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fcdacc36000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=111999, ...}) = 0
mmap(NULL, 111999, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fcdacc1a000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0\>\0\1\0\0\0\320\37\2\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1840928, ...}) = 0
mmap(NULL, 3949248, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fcdac651000
mprotect(0x7fcdac80c000, 2093056, PROT_NONE) = 0
mmap(0x7fcdaca0b000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1ba000) = 0x7fcdaca0b000
mmap(0x7fcdaca11000, 17088, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fcdaca11000
close(3) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fcdacc19000
Amazon 192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fcdacc17000
arch_prctl(ARCH_SET_FS, 0x7fcdacc17740) = 0
mprotect(0x7fcdaca0b000, 16384, PROT_READ) = 0
mprotect(0x601000, 4096, PROT_READ) = 0
mprotect(0x7fcdacc38000, 4096, PROT_READ) = 0
munmap(0x7fcdacc1a000, 111999) = 0
brk(0) = 0xee1000
brk(0xf02000) = 0xf02000
rt_sigaction(SIGPROF, {0x7fcdac74d8c0, ~[]}, SA_RESTORER|SA_RESTART, 0x7fcdac687d40}, {SIG_DFL, [], 0}, 8) = 0
setitimer(ITIMER_PROF, {it_interval={0, 10000}, it_value={0, 10000}}, {it_interval={0, 0}, it_value={0, 0}}) = 0
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 26), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fcdacc35000
write(1, "socket success!\n", 16socket success!
) = 16
clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7fcdacc17a10) = 4171
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 26), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fcdacc34000
read(0, hi
"hi\n", 1024) = 3
```

LTRACE

□ 공유 라이브러리 추적

- 디버깅 할 때뿐 아니라 애플리케이션 동작하는 내부 원리나 구조를 파악하는 데에도 많은 도움
 - User mode의 라이브러리를 추적함으로 확인

□ Ltrace 패키지 설치

- `sudo apt-get install ltrace`

LTRACE

□ Ltrace

- option

option	Descriptions
-u username	Username의 권한으로 명령 실행하면서 라이브러리 추적
-tt	타임 스탬프 표시
-T	실지 함수호출 시간 표시
-p PID	이미 실행중인 프로세스 추적
-o	추적 결과 파일에 저장
-e	[라이브러리 함수명][라이브러리 파일명](+/-)

LTRACE

□ Ltrace client 확인 결과

```
^Ctest@ubuntu:~/chat/client1$ ltrace ./UDP_client 192.168.177.139 4000 j
__libc_start_main(0x400bad, 4, 0x7ffe5fcfb848, 0x400f40 <unfinished ...>
__monstartup(0x400a80, 0x400fe5, 0x7ffe5fcfb870, 0) = 0
__cxa_atexit(0x400970, 0, 0, -1) = 0
mcount(4, 0x7ffe5fcfb848, 0x7ffe5fcfb870, 1) = 0x400bad
socket(2, 2, 0) = 3
puts("socket success!"socket success!
) = 16
bzero(0x7ffe5fcfb730, 16) = <void>
inet_pton(2, 0x7ffe5fcfd355, 0x7ffe5fcfb734, 0) = 1
atoi(0x7ffe5fcfd365, 9, 0x7ffe5fcfb734, 0) = 4000
htons(4000, 0, 10, 0) = 0xa00f
fork() = 4242
mcount(3, 0x8bb1a8c0a00f0002, 0, 0x7ffe5fcfd36a) = 3
sprintf("[j] ", "[%s] ", "j") = 4
strlen("[j] ") = 4
fgets(hi
"hi\n", 1000, 0x7f07220e8640) = 0x7ffe5fcfaef0
strcat("[j] ", "hi\n") = "[j] hi\n"
strlen("[j] hi\n") = 7
sendto(3, 0x7ffe5fcfb2e0, 7, 0) = 7
strstr("hi\n", "exit") = nil
fgets(exit
"exit\n", 1000, 0x7f07220e8640) = 0x7ffe5fcfaef0
strcat("[j] ", "exit\n") = "[j] exit\n"
strlen("[j] exit\n") = 9
sendto(3, 0x7ffe5fcfb2e0, 9, 0) = 9
strstr("exit\n", "exit") = "exit\n"
puts("good bye~"good bye~
) = 10
close(3) = 0
__mcleanup(0, 0, 64, 0x7f07220e8eb0) = 1
+++ exited (status 0) +++
```

과제

- ❑ 서버의 동작 방식을 **strace**와 **ltrace**를 통해서 확인
 - **Kernel**의 **system call**이 동작하는 방식을 확인
 - **User**의 라이브러리가 동작하는 방식을 확인
- ❑ **Strace**를 사용한 확인한 캡처 화면과 **system call**을 추적하여 서버가 동작하는 방식을 간단히 설명
- ❑ **Ltrace**를 사용한 확인한 캡처 화면과 라이브러리를 추적하여 서버가 동작하는 방식을 간단히 설명

- ❑ **Ltrace**를 사용하여 **Client1**과 **Client2**를 비교 화면 캡처
 - **Client1**과 **Client2**의 차이점 간단히 설명

FRACE

FTRACE

□ What is ftrace

- 리눅스 커널의 Tracing을 위한 커널의 디버깅 컴포넌트
 - 특정 기간 동안 리눅스 커널의 내부 함수 동작을 트레이싱 하기 위한 목적으로 사용
- 애플리케이션이 실행되는 동안에 리눅스 커널 내부에서 어떠한 일들이 발생하는지를 분석
- 커널 레벨에서 동작하는 커널 함수들의 흐름을 원자단위로 분석이 가능
- 원하는 커널 함수들만을 필터링하여 분석할 수 도 있어 디버깅으로 인한 오버헤드 비용을 최소화 함

FTRACE

□ Ftrace 동작 원리

- GCC 컴파일러의 프로파일링 옵션(-pg)으로 빌드된 커널을 이용하여 커널 내부 함수들을 tracing하는 구조
 - -pg 스위치에 의해서 만들어 지는 각 커널 함수의 시작점에 mcount 루틴을 통해 동작
- Ftrace는 동일한 코드를 실행하는 다른 CPU에 대해 염려하지 않고 자유롭게 수정이 될 수 있도록 하기 위해서 kstop_machine을 호출
 - CONFIG_STOP_MACHINE=y , ./include/linux/stop_machine.h
 - 싱글 코어처럼 수행하는 Machine

FTRACE

□ Plugins

- Current_tracer을 이용해 Enable/Disable 가능
- 셋팅되어 있는 플러그인의 열람은 “#> cat current_tracer”

□ Events

- Event 디렉토리의 enable 파일을 이용해 Enable/Disable 가능
- 셋팅되어 있는 이벤트의 열람은 cat set_events을 통해 가능

□ Ftrace trace plugins 기능

- function: 임의의 기간동안 커널 내부 함수의 호출관계를 분석
- function_graph: 그래프 형식의 함수 관계 분석
- wakeup, wakeup_dl, wakeup_rt: wake up latency 분석
- mmiotrace: 메모리 맵의 I/O에 대한 분석
- irqsoff: interrupt latency 분석
- nop: 디버깅 기능 off

FTRACE

□ Function tracer 예제

- Ftrace 의 기본 설정은 모든 커널 함수들을 trace
 - 1. `cat set_ftrace_filter`
 - ✓ ##### all functions enabled ##### : 모든 함수 trace 하도록 설정
 - 2. `echo CommonTraceWorkHandler > set_ftrace_filter`
 - ✓ CommonTraceWorkHandler 은 함수만을 trace 하도록 설정
 - 3. `cat current_tracer`
 - ✓ nop: 현재 tracer 세팅 안되어 있음
 - 4. `echo function > current_tracer`
 - ✓ 함수 trace 하도록 설정
 - 5. `cat trace | head -15`

FTRACE

□ Ftrace tracer 예제

- 시스템의 응답속도에 영향을 미칠 것들에 대한 trace
- Wakeup, wakeup_rt, irqsoff, preemptoff, preemptirqsoff 를 trace 가능
- Irqsoff: irq 가 disable 되어 있던 시간을 출력
 - irq disable 되어 있는 시간이 많으면, irq 응답이 느려지는 것이므로, 사용자 App 의 응답속도에 영향 줌
 - 1. echo irqsoff > current_tracer
 - 2. cat trace | head -20
- Preemptoff: 선점 기능이 disable 되어 있던 시간을 출력
 - 1. echo preemptoff > current_tracer
 - 2. cat trace | head -20

FTRACE

□ Ftrace Event Tracer 예제

- 커널의 서브시스템 (block, ext4, schedule, IRQ, workqueue 등)에 대한 event 를 분석
- Tracepoint 로 사전 후킹 작업한 커널함수만 트레이싱 가능
 - Tracepoint 로 컴파일 시에만 트레이싱 가능
- 위의 다른 tracer 들 plugins과는 구현 방식이 다름
- Scsi 버스 이벤트에 trace 를 걸어, 외장하드 연결했을 때 예제
 - 1. `echo 1 > events/scsi/enable`
 - 2. `cat set_event`
 - 3. `cat trace | head -20`

FTRACE

□ Ftrace Stack Tracer 예제

- Stack Tracer 는 현재 커널 스택의 함수 호출 구조 확인 (Backtrace)
 - 1. `echo 1 > /proc/sys/kernel/stack_tracer_enabled -> stack tracer on`
 - 2. `cat stack_trace`

FTRACE

□ Ftrace function_graph tracer 예제

- Ftrace 로 커널 함수 호출 in/out 확인하기
- Function_graph tracer 와 function tracer 의 차이점
 - Function_graph 는 in 뿐만 아니라 out 까지 기록
 - 해당 함수의 수행시간도 파악
 - 함수 호출 depth를 확인
 - 1. echo function_graph > current_tracer
 - 2. cat trace | head -15

FTRACE

□ Trace-cmd

- Trace-cmd 는 커널의 **ftrace**을 위한 사용자 공간의 콘솔기반 명령 인터페이스
- Echo명령만으로 **tracing**을 하는 것이 불편하고 어렵기 때문에
- 콘솔모드에서 일일이 많은 명령들을 입력해야 하는 불편함 해소
- 많은 명령들을 외워야 하는 사용성 문제 해결

FTRACE

□ Trace-cmd

- Trace-cmd 설치 방법
 - `sudo apt-get install trace-cmd`
- Trace-cmd 미사용
 - `#> mount -t debugfs nodev /sys/kernel/debug`
`#> cd /sys/kernel/debug/tracing`
`#> echo function ./current_tracer`
`#> echo 1 > tracing_on`
`#> ls /system/`
`#> echo 0 > tracing_on`
- Trace-cmd 사용시
 - `#>/sdcard/trace-cmd record -p function ls /system`

FTRACE

□ Trace-cmd list

- Trace-cmd list -o
 - Trace-cmd record -O option
- Trace-cmd list -p
 - 사용가능한 plugins
- Trace-cmd list -e
 - 사용가능한 events

과제

- ❑ 특정 시간 동안의 **OS** 스케줄링 **Latency**를 **Tracing**하고 출력 결과를 캡처하여 제출
 - **Hint:** 특정 시간 동안의 **OS** 스케줄링 **Latency**ghkrdls 이벤트
✓ sched_wakeup
- ❑ 인터럽트가 **disable**되는 구간에 대해 디버깅하고 출력 결과를 캡처하여 제출
 - **Hint:** 인터럽트가 **disable**되는 구간에 대한 디버깅 플러그인
✓ irqsoff -d

제출 방법

□ 제출 방법

- 워드나 한글로 작성하여 메일에 첨부
- 문서 제목에 학번과 이름을 적을 것

□ E-mail

- jypark@rtcc.hanyang.ac.kr

□ 마감일

- 다음 실습 수업시간 전까지

수고하셨습니다.