2022 A Basic CS skill: ABC Winter School

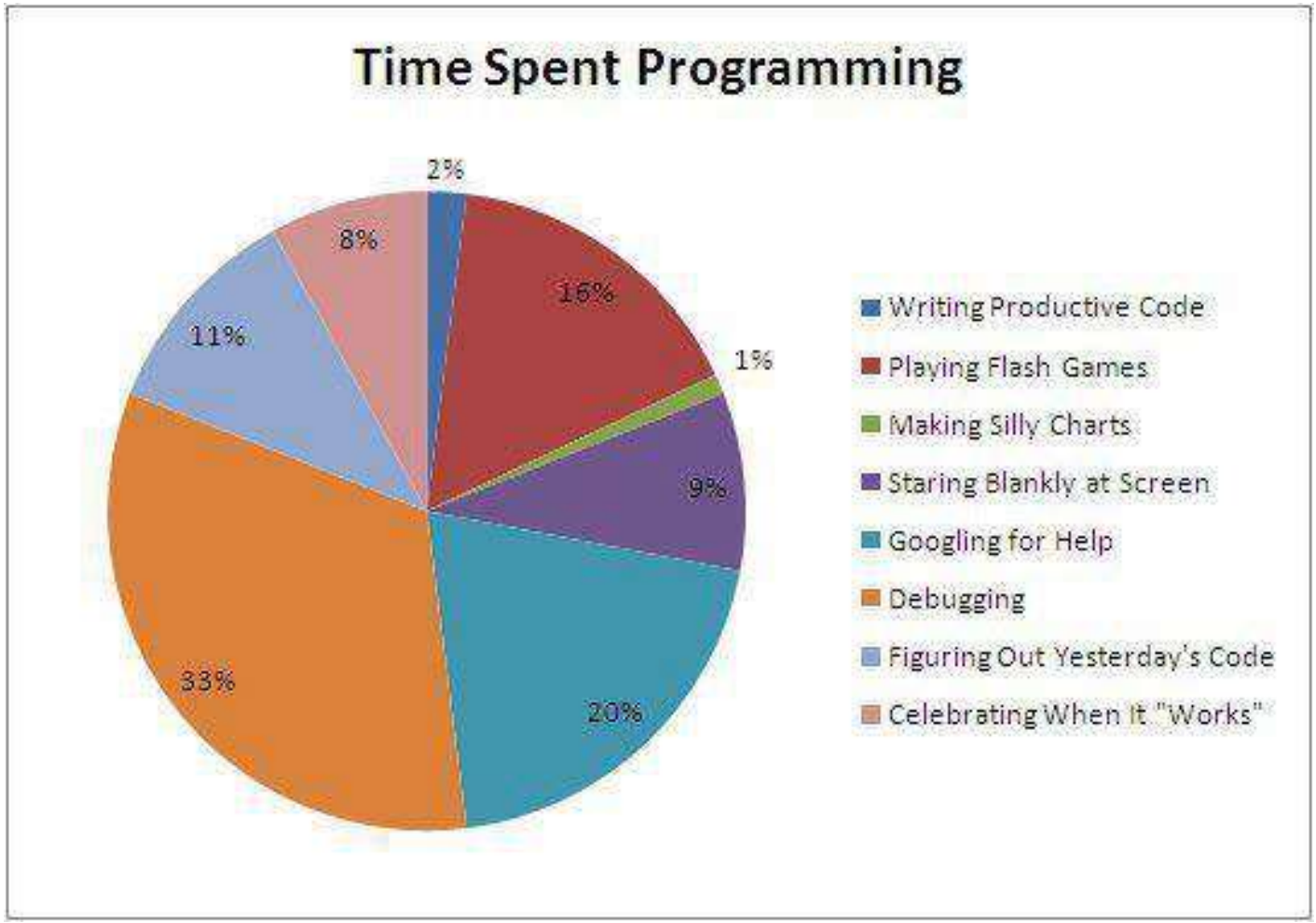
Debugging & Profiling

Team 8

2022 / 02 / 13

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Software development



Debugging

- 프로그램 내부에서 Fault, Error, Failure를 찾아내고 고치는 것을 디버깅이라고 합니다.
- Fault: 프로그램의 잘못된 결과 (잘못된 output이나 slow running 등을 의미)
- Error: 올바르지 않은 프로그램의 내부 상태
- Failure: 잘못된 코드 오타 같은 결함을 의미
- 일반적으로 프로그램 개발에서 가장 많은 시간을 차지합니다.

Print debugging

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements"

– Brian Kernighan, Unix for Beginners

- Print debugging은 가장 간단한 방법이자 많이 사용하는 방식 입니다.
- 코드 중간에 터미널에 출력을 할 수 있는 print(Python)나 printf(C언어)를 사용합니다.

Logging

- Logging은 파이썬에서 print와 같은 standard output 대신 프로그램에서 발생하는 이벤트를 추적하는 내장 모듈입니다.
  - C언어의 경우, logger.h 라이브러리가 같은 역할을 합니다.
- import logging을 통해 프로그램에서 사용할 수 있습니다.
- Logging의 큰 장점은 각 이벤트가 포맷(debug, info, warning, error, critical)에 따라 log의 레벨이 구분된다는 점입니다.

Third Party Logs

- UNIX 계열의 OS 경우, 프로그램을 실행하면서 시스템은 프로그램에서 발생한 log를 특정 장소에 기록하기도 합니다.
- 일반적으로 이러한 log들은 /var/log 위치에 기록됩니다.
- Linux 계열의 경우, systemd라는 시스템에 관련된 로그를 수집하여 /var/log/journal에 저장하는 기능이 존재합니다.
- 저장된 log는 journalctl 등의 명령어를 이용해 확인할 수 있습니다.

PDB debugger

- GCC의 GDB와 비슷한 Python 전용 debugger 입니다.
- python –m pdb [디버깅을 하고 싶은 py 파일]

```
[cs20151509@uni06 pdb_example]$ python -m pdb bubble_sort.py
> /students_home/old_students/cs20151509/ABC_program/pdb_exam
1)<module>()
-> def bubble_sort(arr):
(Pdb)
```

PDB debugger

- PDB는 내부에서 다양한 명령어를 지원합니다.

  l: 현재 line을 기준으로 11줄 정도의 코드를 보여줍니다.

  s: 현재 line을 실행합니다. (함수를 만나면 함수 내부로 진입)

  n: 현재 함수에서 다음 line을 실행합니다.

  b: break point를 설정합니다.

  p: 주어진 값을 계산하고 화면에 출력합니다.

  r: 현재 함수에서 return을 만날 때까지 실행을 진행합니다.

  q: 디버깅을 종료합니다.

참고: https://docs.python.org/ko/3.7/library/pdb.html

PDB set_trace

- PDB 모듈에 내장된 set_trace를 코드에 입력하면 미리 break point를 설정할 수 있습니다.

```
1  import pdb
2
3  def bubble_sort(array):
4      sorted = True
5      while sorted:
6          sorted = False
7          for i in range(len(array)-1):
8              if (array[i] > array[i+1]):
9                  sorted = True
10                 array[i], array[i+1] = array[i+1], array[i]
11                 pdb.set_trace()
12     return array
13
14 result = bubble_sort([3, 2, 0, 1, 4, 10])
15 print(result)
```

strace

- Application (응용 프로그램)이 사용하는 **system call**과 **signal**등을 추적하는 명령어 입니다.
- macOS의 경우 dtrace가 있습니다.

```
with1015@DESKTOP-AH4J9IJ:~$ strace -e mmap echo
mmap(NULL, 30490, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e032b000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f49e0320000
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f49dfc00000
mmap(0x7f49dffe7000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7f49dffe7000
mmap(0x7f49dffed000, 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f49dffed000
mmap(NULL, 1683056, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e008e000
mmap(NULL, 252, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e0332000
mmap(NULL, 26376, PROT_READ, MAP_SHARED, 3, 0) = 0x7f49e032b000
mmap(NULL, 23, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e032a000
mmap(NULL, 47, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e0329000
mmap(NULL, 131, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e0328000
mmap(NULL, 62, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e0327000
mmap(NULL, 34, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e0326000
mmap(NULL, 48, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e0325000
mmap(NULL, 270, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e0324000
mmap(NULL, 1516558, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49dfa8d000
mmap(NULL, 3360, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e0323000
mmap(NULL, 50, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e0322000
mmap(NULL, 199772, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f49e02ef000
```

strace

- strace [명령어] : 실행 가능한 파일 추적
- strace –e [syscall] [명령어] : 특정 system call 추적
- strace –o [file] [명령어] : file로 결과 저장
- strace –p [PID] : 실행중인 프로세스를 strace로 추적
- strace –c [명령어] : system call 통계 정보 생성
- strace –t [명령어] : strace 라인 별 시간 정보 출력
- strace –r [명령어] : 각 system call 마다 상대 시간 출력

Static Analysis

- 프로그램의 실행 없이 software를 분석하는 것.
  - Ex) 오타 검사, unused variable, type checking 등

```
with1015@DESKTOP-AH4J9IJ:~$ mypy static_test.py
static_test.py:6: error: Incompatible types in assignment (expression has type "int", variable has type "Callable[[], Any]")
static_test.py:9: error: Incompatible types in assignment (expression has type "float", variable has type "int")
static_test.py:11: error: Name 'baz' is not defined
with1015@DESKTOP-AH4J9IJ:~$ pyflakes static_test.py
static_test.py:6: redefinition of unused 'foo' from line 3
static_test.py:11: undefined name 'baz'
```

- mypy : type checking 관련 이슈를 찾아 줍니다.
- pyflakes : 오타로 인한 define 문제와 사용하지 않는 변수에 대한 이슈를 찾아줍니다.

Profiling

- 프로파일링 혹은 성능 분석이라고도 합니다.
- 최적화 문제와 연결되기 때문에 시스템 분야에서 매우매우매우매우 중요합니다.
- 일반적으로 프로그램을 사용할 때, CPU의 utilization이나 memory 사용량 같은 컴퓨터의 resource가 얼마나 사용되는지를 분석합니다.
- 프로그램 자체에서 사용자가 코드를 통해 Profiling을 하기도 하지만 일반적으로 외부의 tool에 도움을 받아 하는 경우가 많습니다.

Time 측정

- Linux에는 time이라는 명령어를 통해 프로그램이 실행되고 종료될 때 걸리는 시간을 측정할 수 있습니다.
- time [실행할 명령] 으로 사용합니다.

```
[cs20151509@uni06 pdb_example]$ time python bubble_sort.py
Traceback (most recent call last):
  File "bubble_sort.py", line 10, in <module>
    print(bubble_sort([4,2,1,8,7,6]))
  File "bubble_sort.py", line 5, in bubble_sort
    if arr[j] > arr[j+1]:
IndexError: list index out of range

real    0m0.051s
user    0m0.033s
sys     0m0.017s
```

Real : 프로그램 시작에서 종료까지 모든 요소들이 실행되는데 걸린 시간

User : CPU가 프로그램에서 유저 코드를 실행하는데 사용한 시간

Sys: CPU가 프로그램에서 커널 코드 (시스템 코드)를 실행하는데 사용한 시간

Time 측정

- 앞서 time 명령어는 프로그램 전체를 측정하는데 사용됩니다.
- 만약 프로그램 내부에서 어떤 하나의 명령어를 실행하는데 걸리는 시간을 측정하고 싶다면, 각 언어에 존재하는 time library를 사용해야 합니다.
- 예를 들면, Python에는 time이라는 모듈이 존재합니다.

```
1 import time
2
3 start = time.time()
4 print("Hello World!")
5 end = time.time()
6
7 print(end - start)
```

Print 함수가 실행되는데 얼마나 시간이 걸리는지를 측정합니다.

cProflie

- Python에서 사용되는 CPU Profiler 입니다.
- CPU를 통해 프로그램의 어떤 부분이 얼마나 자주, 오래 사용되었는지 분석할 수 있도록 통계를 보여줍니다.
- python –m cProfile [프로파일링을 하고 싶은 py 파일]
- 참고: https://docs.python.org/ko/3/library/profile.html

```
[cs20151509@uni06 ABC_program]$ python -m cProfile time_profiling.py
Hello World!
3.79085540771e-05
         4 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 time_profiling.py:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
        2    0.000    0.000    0.000    0.000 {time.time}
```

memory_profiler

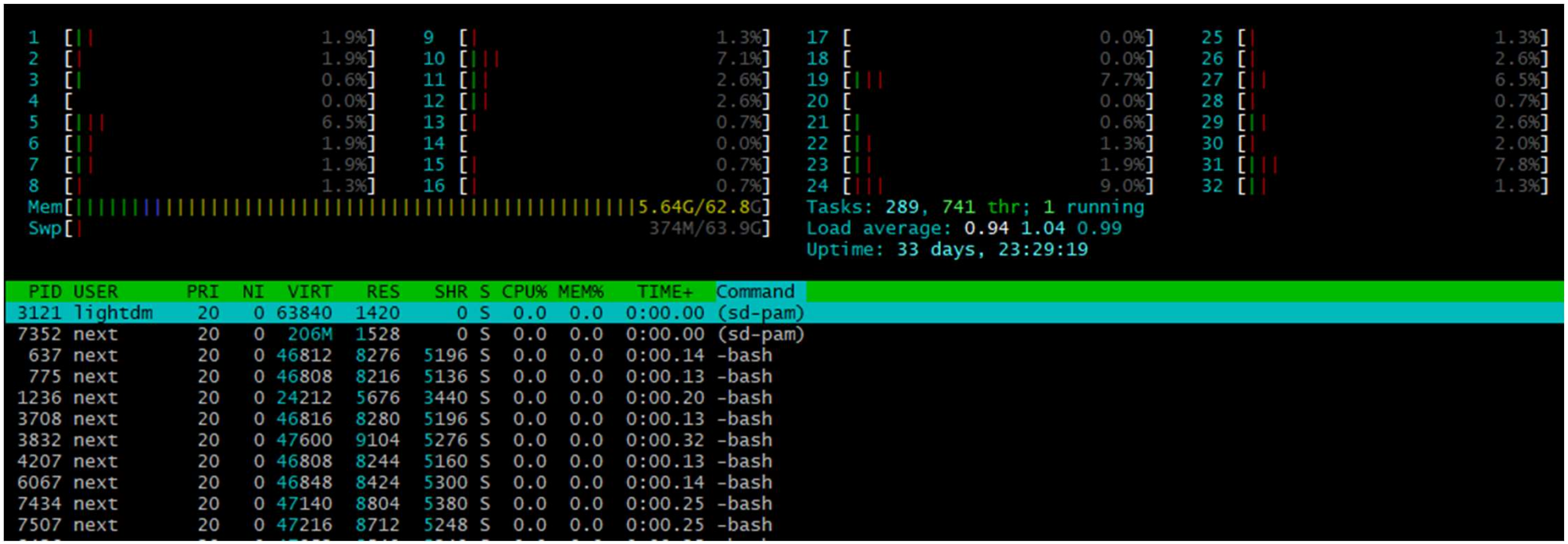
- pip install memory_profiler 명령어로 설치 가능합니다.
- Decorator(@)를 통해 profiling을 원하는 함수를 선택 가능합니다.

```
[cs20151509@uni06 pdb_example]$ python -m memory_profiler memory_test.py
Filename: memory_test.py

Line #    Mem usage    Increment   Line Contents
================================================
     1   24.492 MiB   24.492 MiB   @profile
     2                             def test_func():
     3   32.125 MiB    7.633 MiB       a = [1] * (10 ** 6)
     4  184.715 MiB  152.590 MiB       b = [2] * (2 * 10 ** 7)
     5   32.125 MiB    0.000 MiB       del b
     6   32.125 MiB    0.000 MiB       return a
```

top, free, df + 그 외

- top (htop) : 프로세스별로 CPU 사용량과 할당중인 코어에 대해 볼 수 있습니다.



top, free, df + 그 외

- free : memory 사용량과 swap memory 사용량에 대해 볼 수 있습니다.

```
next@tino120:~$ free
              total        used        free      shared  buff/cache   available
Mem:       65835092     5182276     2263680      215004    58389136    59187068
Swap:      66974716      382464    66592252
```

- df : disk storage 사용량에 대해 볼 수 있습니다.

```
next@tino120:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev             32G     0   32G   0% /dev
tmpfs           6.3G   27M  6.3G   1% /run
/dev/sda1       172G  123G   41G  76% /
tmpfs            32G  230M   32G   1% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs            32G     0   32G   0% /sys/fs/cgroup
/dev/sdf1       939G  874G   18G  99% /ssd_ckpt
/dev/sdc1       235G  155G   68G  70% /ssd_dataset
/dev/sdg1       3.6T  2.8T  684G  81% /hetpipe_ckpt
/dev/sdd1       917G  675G  196G  78% /next
tmpfs           6.3G   32K  6.3G   1% /run/user/108
overlay         917G  675G  196G  78% /next/docker/ov
overlay         917G  675G  196G  78% /next/docker/ov
overlay         917G  675G  196G  78% /next/docker/ov
overlay         917G  675G  196G  78% /next/docker/ov
tmpfs           6.3G     0  6.3G   0% /run/user/1000
/dev/sde1       917G  215G  656G  25% /ssd_dataset2
```

top, free, df + 그 외

- nvidia-smi : GPU 사용량에 대해 볼 수 있습니다. (주로 딥러닝에 사용)

```
next@tino120:~$ nvidia-smi
Sun Feb 13 14:43:56 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.39       Driver Version: 418.39       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  TITAN RTX           Off  | 00000000:02:00.0 Off |                  N/A |
| 41%   25C    P8     2W / 280W |      0MiB / 24190MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  TITAN RTX           Off  | 00000000:03:00.0 Off |                  N/A |
| 40%   27C    P8     8W / 280W |      0MiB / 24190MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   2  TITAN RTX           Off  | 00000000:82:00.0 Off |                  N/A |
| 41%   26C    P8     1W / 280W |      0MiB / 24190MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   3  TITAN RTX           Off  | 00000000:83:00.0 Off |                  N/A |
| 41%   27C    P8     6W / 280W |      0MiB / 24190MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Activity

- 오늘 배운 디버깅, 프로파일러를 사용해봅시다.

Thank you


UNIST
ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY
2009