컴퓨터학부 20202920 조민혁 SamSam

-0. 리눅스 커널 및 리눅스 배포판의 이해

리눅스 배포판, 리눅스 커널, Ubuntu LTS 버전, Ubuntu 정규버전 용어를 설명하시오.

리눅스 배포판이란? 리눅스 배포판은 기본적으로 리눅스 커널을 중심으로 다양한 소프트웨어 패키지를 조합하여 사용자가 쉽게 설치하고 사용할 수 있도록 만들어진 운영 체제입니다. 리눅스는 원래 자유 소프트웨어로 배포되기 때문에, 다양한 단체나 개인이 이를 기반으로 자신만의 운영 체제를 만들 수 있습니다. 이러한 운영 체제를 '배포판'이라고 부릅니다.

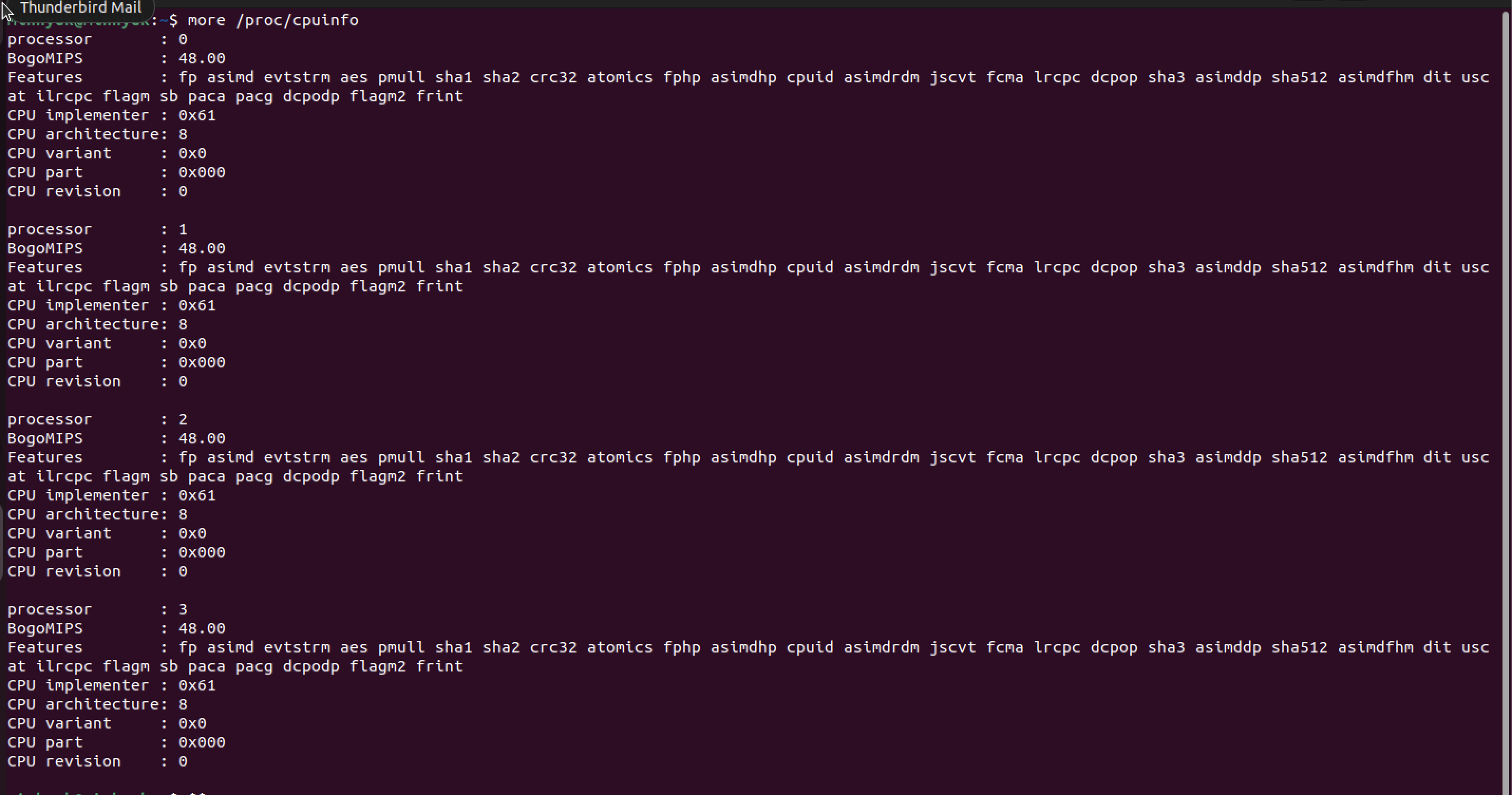
리눅스 커널이란? 리눅스 커널(Linux Kernel)은 리눅스 운영 체제의 핵심 부분으로, 하드웨어와 소프트웨어 간의 중개자 역할을 합니다. 커널은 시스템의 리소스(예: CPU, 메모리, 디스크, 네트워크 등)를 관리하고, 사용자 프로그램이 이러한 리소스에 접근할 수 있도록 해줍니다. 또한, 다양한 하드웨어 장치들을 제어하고, 멀티태스킹, 메모리 관리, 파일 시스템 관리 등의 기능을 수행합니다. 리눅스 커널은 오픈 소스 소프트웨어로, 누구나 소스 코드를 볼 수 있고, 수정하거나 배포할 수 있습니다. 리눅스 커널은 수많은 배포판(예: Ubuntu, Fedora, CentOS 등)에서 사용되며, 서버, 데스크탑, 임베디드 시스템, 스마트폰(안드로이드) 등 다양한 환경에서 동작합니다.

Ubuntu LTS 버전이란? Ubuntu LTS(Long Term Support) 버전은 우분투 운영 체제의 장기 지원 버전을 의미합니다. LTS 버전은 일반적으로 2년마다 한 번씩 출시되며, 5년 동안 보안 업데이트와 버그 수정, 하드웨어 지원이 제공됩니다. LTS 버전은 주로 안정성과 신뢰성이 중요한 서버 환경이나, 개발 환경 등에서 많이 사용됩니다. 반면, 일반 우분투 릴리스는 9개월 동안만 지원되기 때문에, 최신 기능을 빠르게 경험하고 싶은 사용자들이 주로 사용합니다.

Ubuntu 정규버전이란? 우분투(Ubuntu)에서 "정규 버전"은 일반 릴리스(또는 표준 릴리스)를 의미합니다. 우분투는 매년 4월과 10월에 새로운 버전을 출시하는데, 그중 4월에 출시되는 버전은 짝수 해에 한하여 LTS(Long Term Support) 버전으로 제공되며, 나머지 버전들은 정규 버전으로 간주됩니다.

* 1. /proc 파일 시스템의 이해
* (1) more / proc / cpuinfo 명령을 실행하고 processor, cores 용어를 설명하시오.

실행화면 :



processor : CPU의 각 논리적 프로세서를 식별하는 번호(ID)를 나타냅니다. 이 논리적 프로세서는 물리적 코어 또는 하이퍼스레딩 기술을 통해 생성된 가상 코어일 수 있습니다. 이는 시스템내에서 논리적 코어를 구별하기 위해 사용됩니다.

cores : 이 용어는 물리적 CPU에 포함된 실제 코어의 수를 나타냅니다. 물리적 코어는 CPU가 동시에 처리할 수 있는 독립적인 처리 단위를 의미합니다. 물리적 코어의 수를 나타내며, 이 값은 동일한 물리적 CPU 내의 각 논리적 프로세서에 대해 동일하게 나타납니다.

※ 참고 : lscpu를 사용

* (2) (학생의) 컴퓨터 시스템에 있는 프로세~~스~~서의 개수는 몇 개인지 확인하시오.

lscpu와 grep “processor” /proc/cpuinfo명령어를 통해 프로세서의 개수를 구할 수 있습니다. 이를 통해서 프로세서의 개수가 4개인 것을 확인할 수 있었습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

* (3) 각 프로세~~스~~서의 frequency는 얼마인지 확인하시오.

우선 more /proc/cpuinfo 명령어를 사용하여 확인하려고 하였습니다. 하지만 각 프로세서의 frequency는 나오지 않았습니다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

lscpu 명령어를 사용해보아도 각 프로세서의 frequency는 나오지 않았습니다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

아래는 제 리눅스의 상태이며, MAC M3 CPU를 사용한 노트북을 사용중입니다.

텍스트, 스크린샷, 폰트, 블랙이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트이(가) 표시된 사진

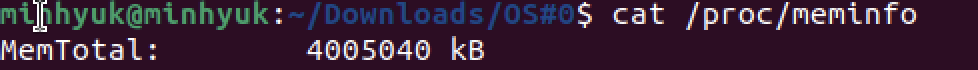
자동 생성된 설명

dmidecode -t processor | grep “Speed” 명령어를 사용하여 CPU의 frequency를 확인하였습니다.

하지만 각 프로세서의 주파수는 출력되어 나오지 않았고 CPU의 frequency만 확인할 수 있었습니다.

* (4) (학생의) 컴퓨터 시스템의 물리적 메모리 크기는 얼마인지 확인하시오.

cat /proc/meminfo로 메모리 정보를 불러왔습니다.



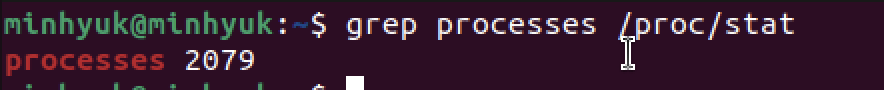
* (5) 위 물리적 메모리 중 free한 크기는 얼마인지 확인하시오.

위에서 사용한 cat /proc/meminfo 명령어를 통해 구했습니다.



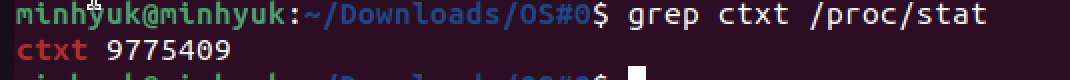
* (6) 시스템 부팅 후 fork()된 프로세스의 개수는 몇 개인지 확인하시오.

grep processes /proc/stat 명령어를 통해서 stat에서 fork()된 프로세스의 개수들을 확인할 수 있었습니다.



* (7) 부팅 후 문맥 교환(context switch)가 이루진 횟수는 얼마인지 확인하시오.

context switch를 확인하기 위해서 grep ctxt /proc/stat 명령어를 사용함으로써 /proc/stat파일에 있는 ctxt 값을 출력하였습니다.



* 2. 실행 중인 프로세스의 상태 모니터링
* (1) 주어진 cpu.c를 컴파일하고 아래 명령어를 실행하고 top 명령어 실행하시오.

$ gcc cpu.c -o cpu $ ./cpu

위의 명령어를 실행한 후 다른 터미널을 켜서 top 명령어를 수행하였습니다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

※ 참고 : 무한 루프 수행 중 다른 터미널을 열고 top 명령을 실행

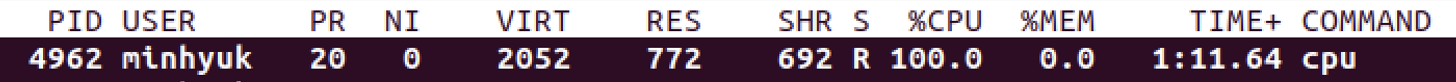
* (2) cpu 명령을 실행하는 프로세스의 PID는 무엇인가 설명하시오.

CPU 명령을 실행하는 프로세스의 PID는 "프로세스 식별자(Process Identifier)"를 의미합니다. PID는 운영 체제가 각 프로세스를 구별하고 관리하기 위해 사용하는 고유한 숫자입니다. 각 프로세스가 실행될 때 운영 체제는 그 프로세스에 PID를 할당하며, 이 번호를 통해 프로세스를 식별하고 상태를 추적할 수 있습니다. 그리고 현재 cpu 명령을 실행하는 프로세스의 PID는 4962입니다.

* (3) 위 프로세스는 얼마나 많은 CPU와 메모리를 소비하는지 확인하시오.

top 명령어를 사용하여 나온 값을 캡처하였습니다.

cpu 는 100%사용하고 있고 메모리는 0만큼 사용하고 있습니다.



* (4) 위 프로세스의 현재 상태는 무엇인지 확인하시오. 예를 들어 실행 중, 차단(block) 중, 좀비 등

R+는 Running상태에서 +로서 foreground에서 수행되는 것을 의미합니다. 즉, 현재 실행 중이면서 터미널의 전경에서 실행중이라는 의미입니다

ps aux명령어를 사용해서 현재 실행중인 프로세스들을 확인하고 거기서 현재 상태를 확인하였습니다.



* 3. 새로운 자식 프로세스를 생성하여 사용자 명령을 실행하는 방법 이해
* (1) cpu-print.c를 컴파일하고 쉘에서 아래 명령 차례로 실행하시오.

$ gcc cpu-print.c –o cpu-print

$./cpu-print

아래는 ./cpu-print를 실행하였을 때의 화면을 캡처하였습니다.

스크린샷, 직사각형, 텍스트이(가) 표시된 사진

자동 생성된 설명

※ 참고 : 무한 루프 수행 중 다른 터미널을 열고 ps 명령을 실행

* (2) cpu-print 프로세스의 부모, 즉 쉘 프로세스의 PID는 무엇인지 확인하시오? 이 pid로부터 5세대 이상의 조상 프로세스 (또는 init 프로세스)에 도달 할 때까지 거슬러 모든 조상의 PID는 무엇인지 확인하시오.

pgrep cpu-print 명령어를 사용하여 실행된 프로세스의 pid를 얻고 그 pid를 통해서 ps -o pid,ppid,cmd -p 64530이라는 명령어를 사용하여 부모프로세스의 pid와 현재 프로세스의 정보를 구하였습니다. 이를 토대로 구해진 ppid를 이용해서 계속 조상 프로세스까지 위의 과정을 반복하여 init 프로세스에 도달할 수 있었습니다.

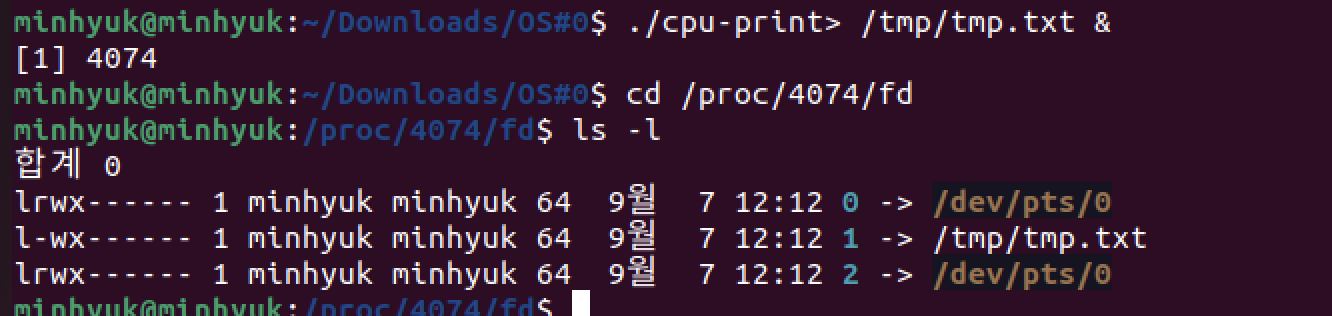
또한 pstree -s -p 64530명령어를 통해 트리구조로 이루어진 형태로 모든 조상의 PID를 구할 수 있었습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

* (3) $ ./cpu-print> /tmp/tmp.txt & 을 실행하고 새로 생성 된 프로세스의 proc 파일 시스템 정보를 확인하시오. file descriptor 0, 1 및 2 (표준 입력, 출력 및 에러)가 가리키는 위치에 주의.

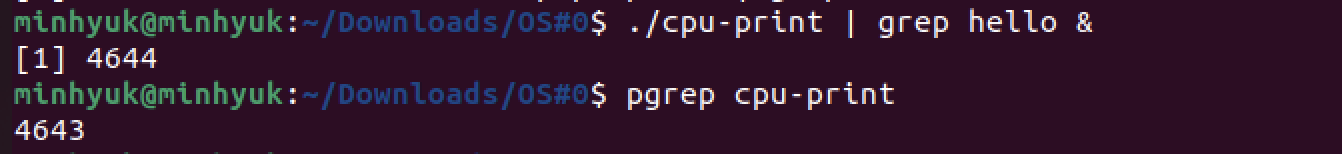
./cpu-print> /tmp/tmp.txt & 명령어는 출력을 /tmp/tmp.txt로 변경하고 백그라운드에서 수행하는 &명령어를 붙혀서 사용한 것입니다. 이를 통해 실행된 프로세스의 proc 파일 시스템 정보를 확인하기 위해서 cd /proc/4074/fd명령어를 통해 해당 디렉토리로 이동하였고 거기서 ls -l명령어를 통해 파일시스템과 관련된 상세한 정보를 얻을 수 있었습니다.



* (4) 위 이러한 내용을 토대로, 셸에서 I/O 리디렉션을 구현하는 방법을 간단하게 설명하시오.

셸에서 I/O를 리디렉션 하기 위해서 현재는 “>”을 붙혀서 출력을 리디렉션하고 , “<”을 붙혀서 입력을 리디렉션을 하고 있었습니다. 이를 구현하기 위해서는 실행하는 파일의 기본적인 입출력과 관련된 파일디스크립터 값을 변경하는 과정이 필요합니다. 이를 사용하기 위해서 내부적으로 dup2함수와 같은 과정을 거쳐서 표준 입출력의 파일 디스크립터를 변경시켜주는 형식을 사용하면 I/O 리디렉션을 구현할 수 있습니다.

(5) $ ./cpu-print | grep hello & 을 실행하고 새로 생성된 프로세스는 무엇인지 확인하시오?

$ ./cpu-print | grep hello & 명령어를 수행하였을 때는 4644라는 출력값이 나왔지만   
pgrep cpu-print명령어를 통해서 새롭게 만들어진 프로세스의 pid가 4643이라는 것을 알 수 있었습니다.

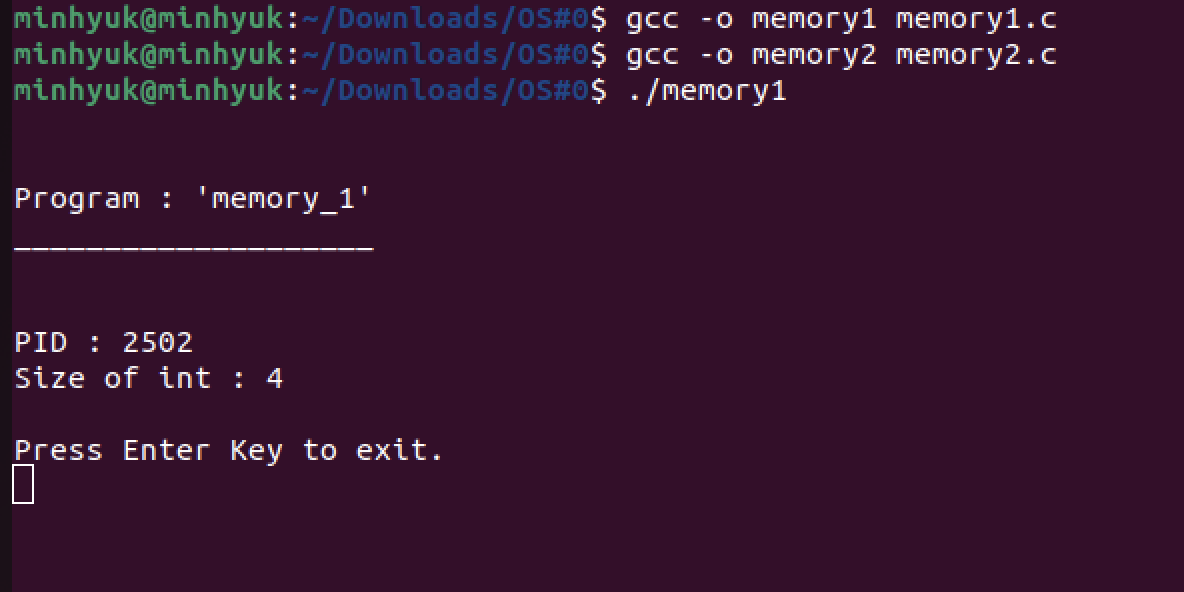


또한 ps aux명령어를 통해서 해당 프로세스에 대한 상세한 정보를 구할 수 있었습니다.

4644 프로세스 또한 생성되어서 실행되고 있는 것을 알 수 있었습니다.

* 4. 프로세스의 가상 메모리와 실제 메모리 비교
* (1) 주어진 두 프로그램 memory1.c 및 memory2.c를 컴파일하고 실행하시오.

memory1, memory2를 각각 gcc -o memory1 memory1.c , gcc -o memory2 memory2.c 명령어를 통해서 컴파일하였고 터미널 창을 각각 열어 ./memory1과 ./memory2를 실행하였습니다.

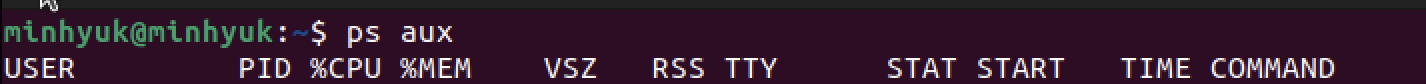


텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

※ 참고 : 두 프로그램 모두 메모리에 큰 배열을 할당함. 한 프로그램은 할당한 배열에 접근하나 다른 프로그램은 할당한 배열에 접근하지 않음. 두 프로그램 모두 종료하기 전에 일시 중지하여 메모리 사용량을 검사(ps 명령어) 할 수 있음.

* (2) 각 프로세스의 가상메모리 크기와 실메모리 크기를 캡쳐해서 비교하고, 주어진 프로그램 소스 코드와 ps 명령어를 통해 확인한 내용을 자세하게 설명하시오.



기본적으로 ps aux는 위와 같은 순서대로 결과값을 출력합니다. 이때 VSZ란 가상메모리의 크기이고, RSS는 실제 메모리의 크기입니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

이를 통해서 ps aux | grep memory1로 ps aux에서 memory1이 있는 것에 대한 정보를 구한 것과 ps aux | grep memory2를 통해 구한 memory2에 대한 정보들을 통해서 각각 memory1 프로세스와 memory2 프로세스가 memory1은 가상메모리를 5938kb, 실제메모리를 1088kb 사용하고 있고, memory2가 가상메모리를 5972kb, 실제메모리를 3360kb를 사용하는 것을 알 수 있었습니다. 이를 통해서 memory1은 소스코드 상에서 1000000 \* sizeof(int) 크기만큼의 가상메모리를 할당하여 실제로 사용하지 않아도 가상 메모리로 할당되게 하는 것인데, memory1은 실제로 코드 상에서 배열에 접근하지 않음으로 실제 메모리 사용량이 가상 메모리보다 작다는 것을 알 수 있습니다. 이에 반해 memory2도 소스코드 상에서 1000000 \* sizeof(int) 크기만큼의 가상메모리를 할당하였지만,

for(i=0;i<ARRAY\_SIZE/2;i++)

{

array[i] = 10;

}

for(i=1;i<ARRAY\_SIZE/2;i++)

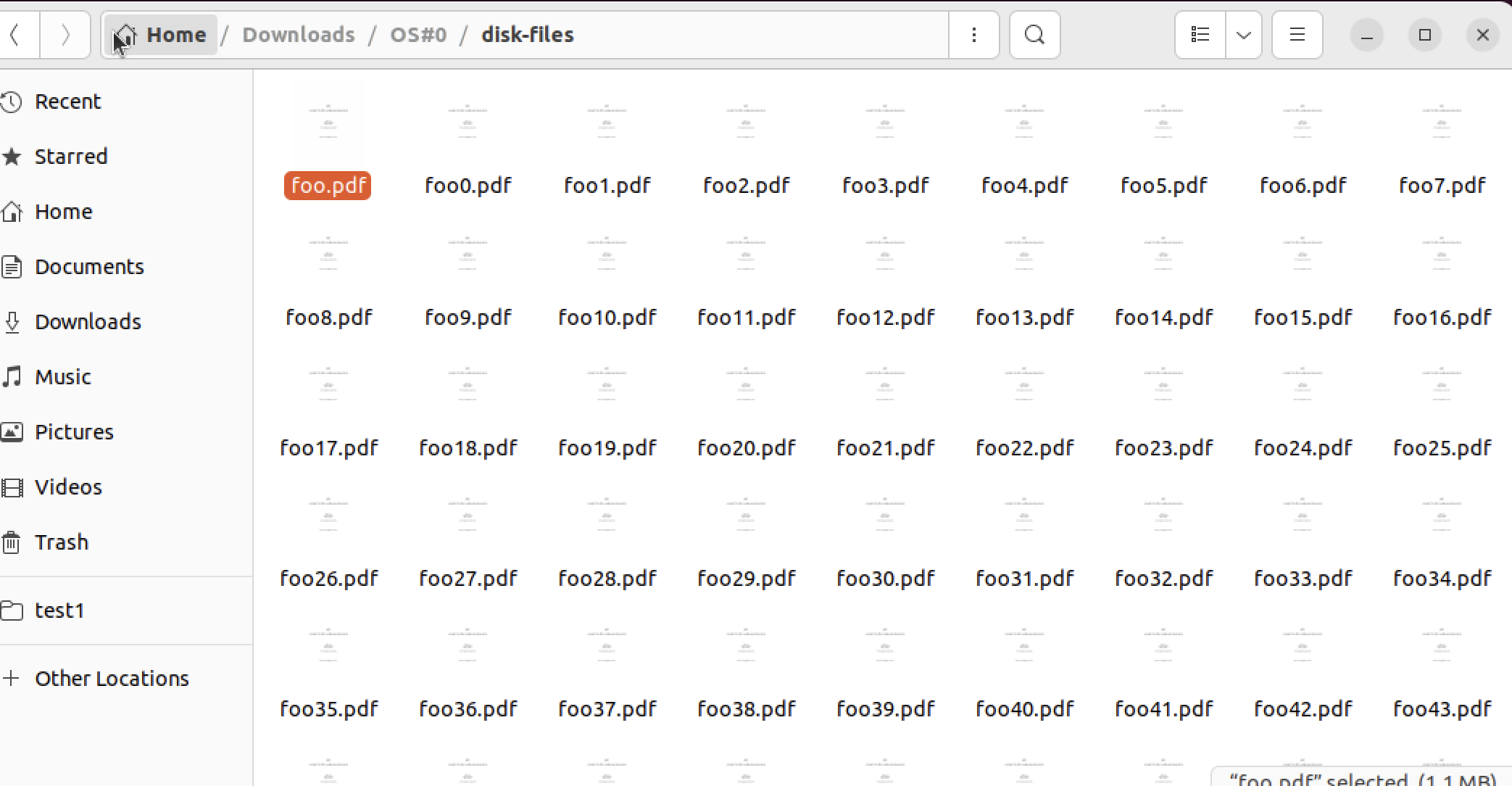
{

array[i] = array[i-1]+25;

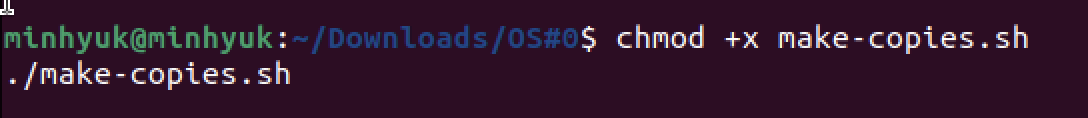
}

와 같은 형식으로 실제로 배열에 접근하고 있기 때문에, 사용하는 실제 메모리가 memory1보다 크다는 것을 확인할 수 있습니다. 이처럼 배열을 선언하고 사용하지 않으면 가상으로 메모리는 할당되어지고 실제로 사용되는 메모리의 크기는 그러한 배열을 사용할 때 할당된다는 것을 알 수 있습니다. ps aux 명령어를 사용하여 VSZ와 RSS를 확인하여 가상메모리와 실제메모리를 비교하여 메모리가 제대로 할당되어서 사용되었는지 확인할 수 있습니다.

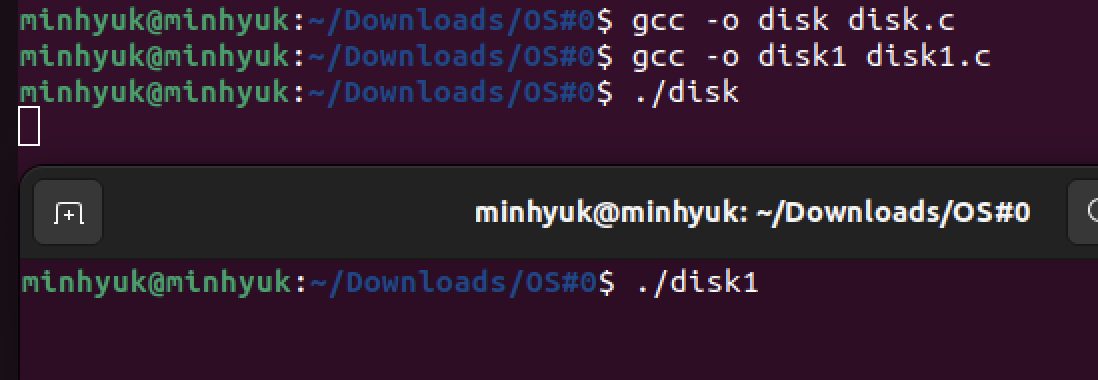
* 5. 프로세스가 오픈한 파일 그리고 디스크
* (1) 주어진 disk.c 및 disk1.c 프로그램을 컴파일하고 실행하시오.



disk-files 디렉토리를 만들어 foo.pdf 파일을 옮겼습니다.



그 후 위의 chmod +x make-copies.sh로 make-copies.sh파일의 실행권한을 추가하고 make-copies.sh파일을 실행하였습니다. 그렇게 실행하여 foo.pdf파일의 5000개를 얻을 수 있었습니다.

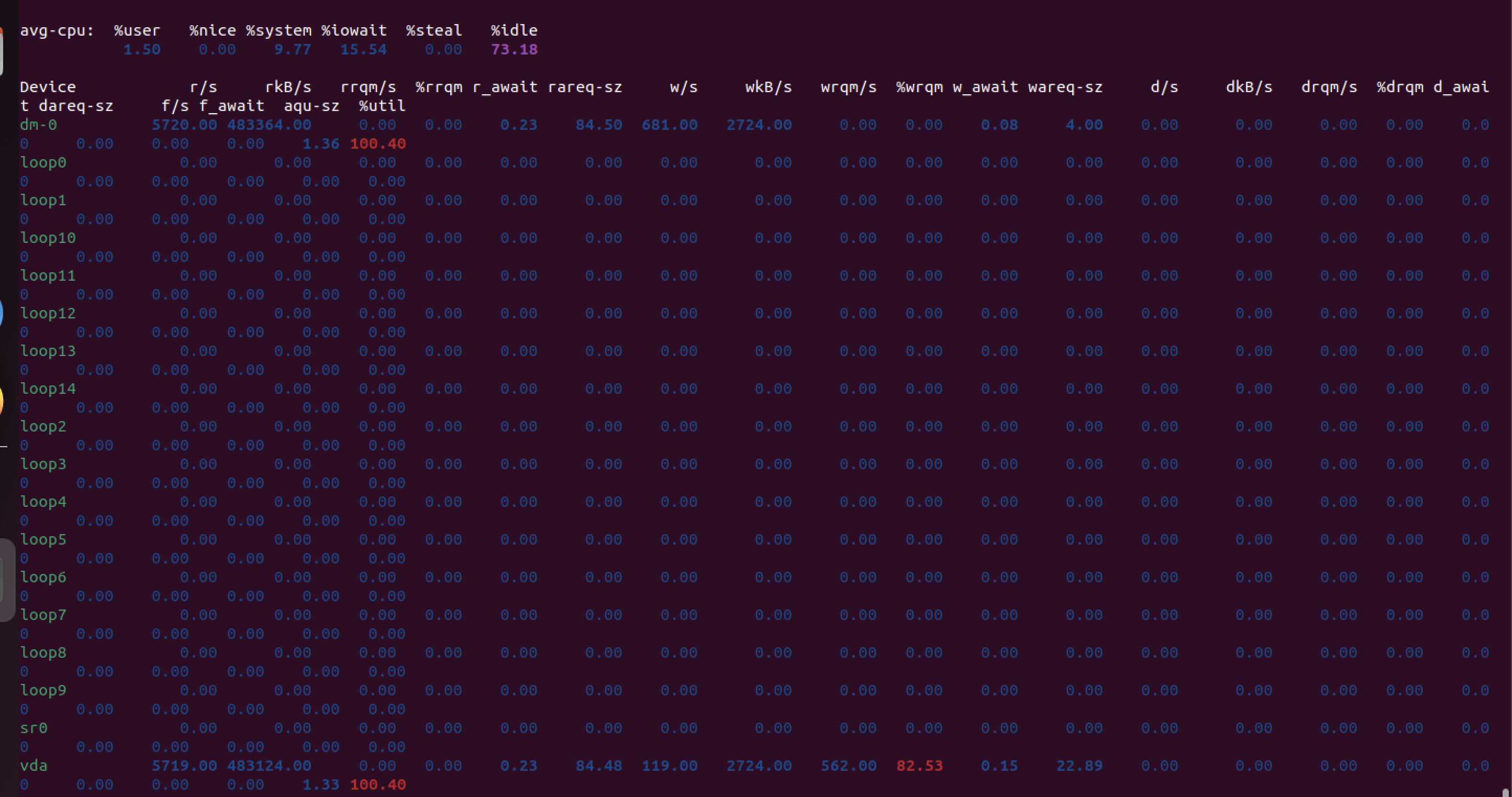


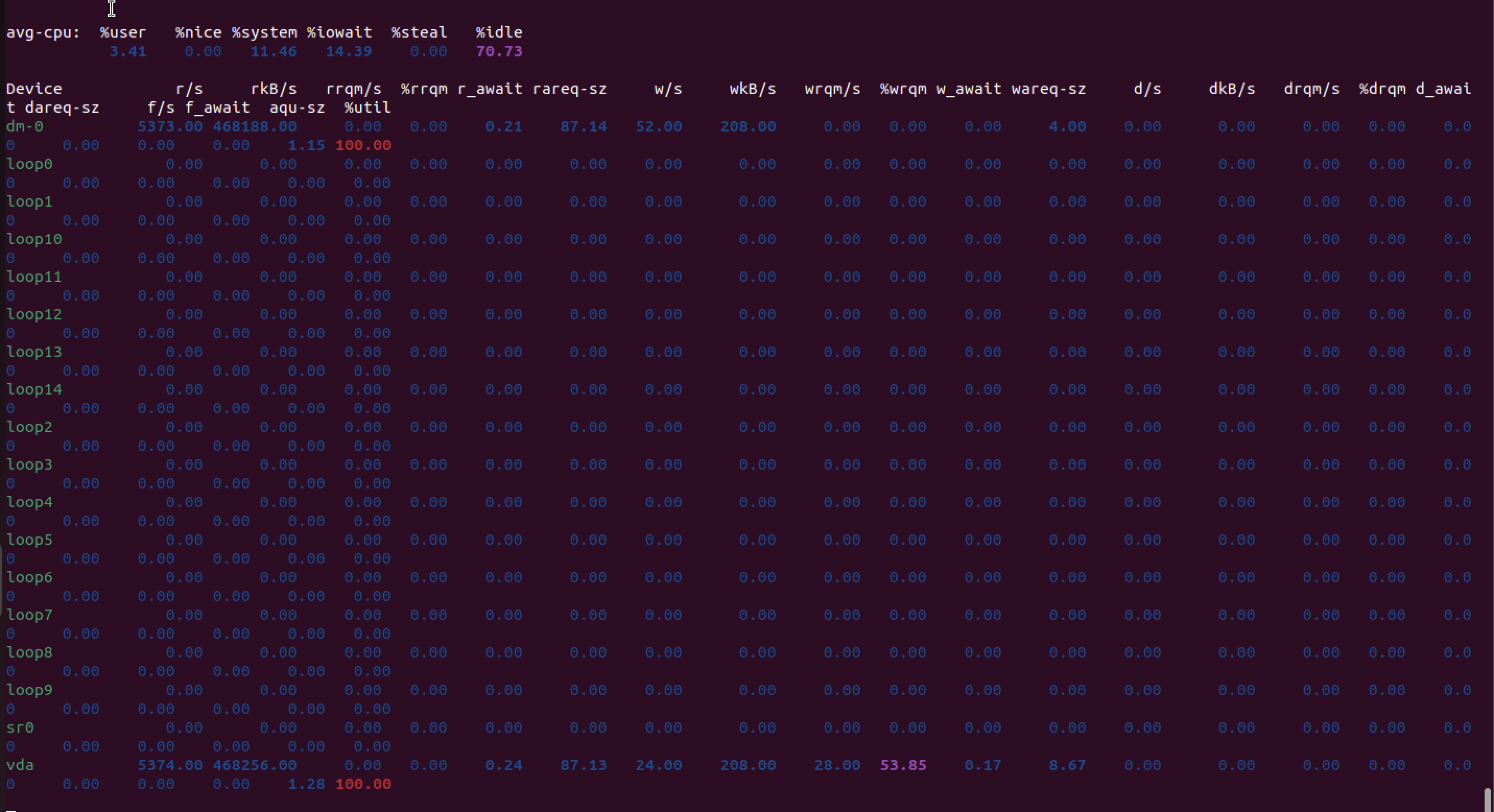
이후 gcc -o disk disk.c, gcc -o disk1 disk1.c를 통해서 컴파일 후 ./disk와 ./disk1 명령어를 통해 disk와 disk1을 실행하였습니다.

※ 참고 : 주어진 프로그램은 디스크에서 여러 파일을 읽음. 먼저 다음과 같이 파일을 생성해야 함. disk-files 디렉토리를 만들고 그 디렉토리에 foo.pdf 파일을 넣음. 주어진 make-copies.sh를 사용하여 해당 폴더에 다른 파일 이름으로 동일한 파일을 5000개 복사. disk 및 disk1 프로그램은 생성된 파일을 읽음.

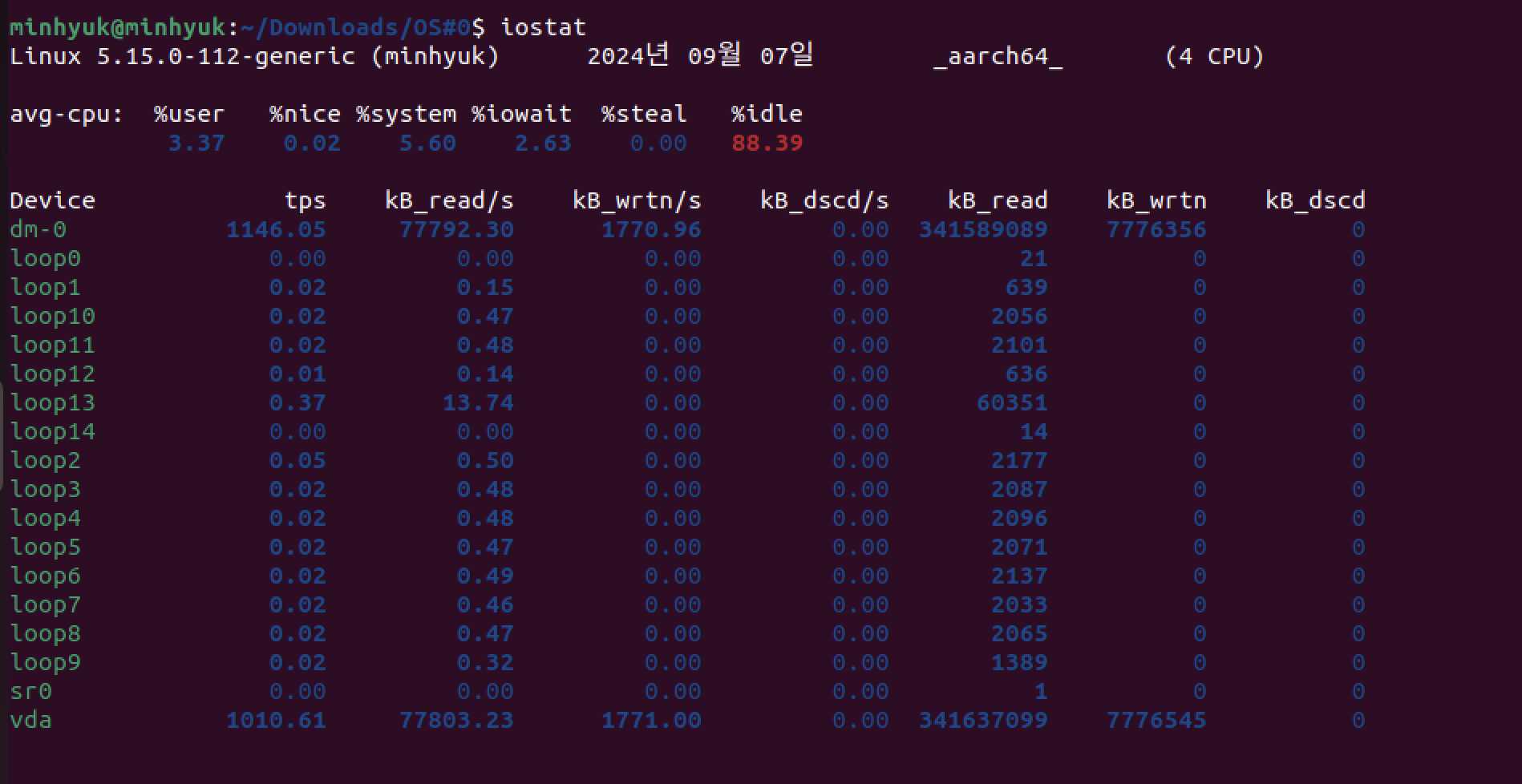
* (2) disk.c 및 disk1.c 의 실행 파일 프로그램이 실행되는 동안 디스크 사용률을 측정하고, 주어진 프로그램 소스 코드와 iostat 명령어를 통해 확인한 내용을 자세하게 비교 설명하시오.

우선 iostat -x 1 명령어를 사용하여 1초마다 상세한 디스크 정보를 불러올 수 있도록 하였습니다. 그 후 시간이 지난 후 같은 명령어를 수행해보았고, 마지막으로 각각에 대하여 iostat명령어를 사용해보았습니다.

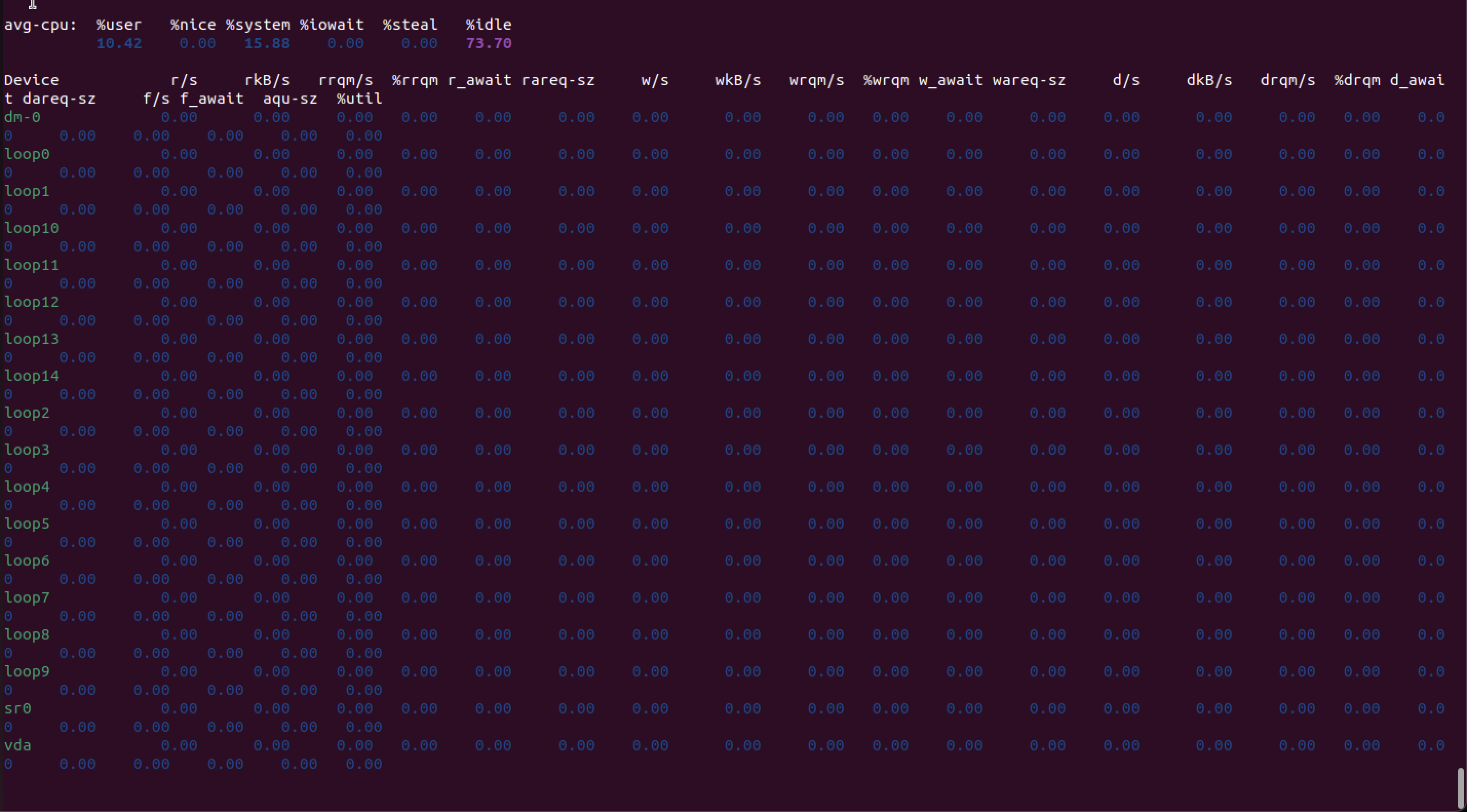


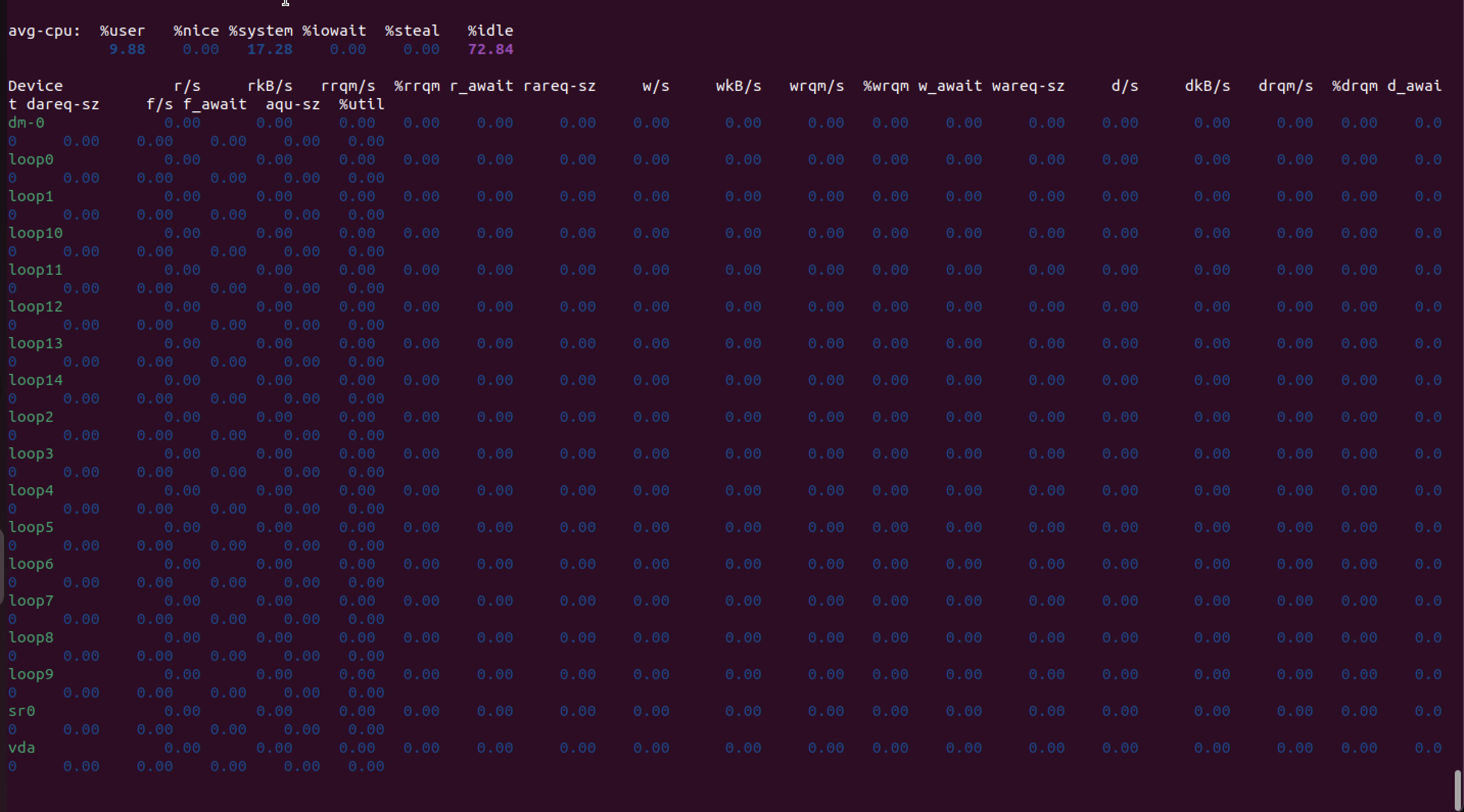


위의 사진은 iostat -x 1 명령어를 ./disk를 사용하였을 때 수행한 경우입니다. 두번째 사진은 첫번째에서 시간이 지난 후 수행한 경우입니다.



위는 disk에 iostat 명령어를 수행한 경우입니다.





위의 사진은 iostat -x 1 명령어를 ./disk1를 사용하였을 때 수행한 경우입니다. 두번째 사진은 첫번째에서 시간이 지난 후 수행한 경우입니다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

또한 위는 disk1에 iostat 명령어를 사용하였을 경우입니다.

명령어를 출력하였을 때 나온 결과물들의 항목들을 토대로 각각 비교해보았습니다.

I/O 대기 시간 (%iowait):

disk.c: 파일을 무작위로 선택하여 접근하므로, 디스크의 캐시 적중률이 낮을 수 있고, 이로 인해 I/O 대기 시간이 변동이 큽니다.

disk1.c: 동일한 파일을 반복해서 읽기 때문에 캐시 적중률이 높을 수 있기에 I/O 대기 시간이 상대적으로 일정한 것을 볼 수 있습니다.

초당 읽기/쓰기 데이터 양 (kB\_read/s, kB\_wrtn/s):

disk.c: 무작위 파일 접근으로 인해 읽기/쓰기 데이터 양의 변화가 큽니다.

disk1.c: 고정된 파일 접근으로 인해 읽기 데이터 양이 더 일정한 것을 볼 수 있었습니다.

디스크 유휴 시간 (%idle):

disk.c: 파일을 무작위로 접근하므로 디스크 유휴 시간이 감소합니다.

disk1.c: 파일 접근 패턴이 일정하므로 디스크 유휴 시간이 더 일정합니다.

초당 I/O 요청 수 (tps):

disk.c: 파일을 무작위로 접근하므로, I/O 요청이 랜덤하게 발생하여 tps 값이 변동합니다.

disk1.c: 고정된 파일에 대한 요청이 반복되므로, tps 값이 더 일정합니다.

결론: disk.c는 무작위로 파일에 접근하기 때문에 iostat명령어를 사용하였을 때 iowait, idle, tps와 같은 지표들이 변동적으로 나오게 되는 것을 확인할 수 있었습니다. 또한 disk1.c는 특정 파일(foo0.pdf)에 대한 반복적인 접근을 하기에 iostat 명령어를 사용하였을 때 모든 지표들이 조금 더 일정하고 예측가능한 것을 확인해 볼 수 있었습니다.