7 Team Project Report

<Implementation and Management of miniOS TOP Command>

Class	운영체제
Instructor	박재근 교수님
Department	전자정보공학부 it융합학과
Writer	김민석(20202894), 김영민(20201576), 안지수(20221602), 박시운(20221597)
Date	2024.05.19

Contents

5. 테스트 및 결과

- 프로젝트의 테스트 방법과 결과

- 성능 측정 결과와 분석

6. 향후 개선 사항

- 프로젝트의 한계와 개선 가능한 부분에 대한 제안

1. 서론

- 프로젝트의 목적과 배경

프로젝트 목적 : System call “top”의 구현

리눅스의 top 명령어는 리눅스 시스템의 메모리 및 프로세스 state를 빠르게 알아보기 위해 사용된다.

top 명령어 구현을 통해 리눅스 System call을 보다 깊게 이해하고, 추가 기능을 넣어 발전된 명령어 set을 설계한다.

2. 프로젝트 개요

- 프로젝트의 요약 설명

1. top 명령어 호출 시 반환되는 정보 확인
2. 해당 정보를 구현하기 위해 strace 명령어를 활용 (시스템 콜 디버깅)
3. 구현 및 출력 인터페이스 설계

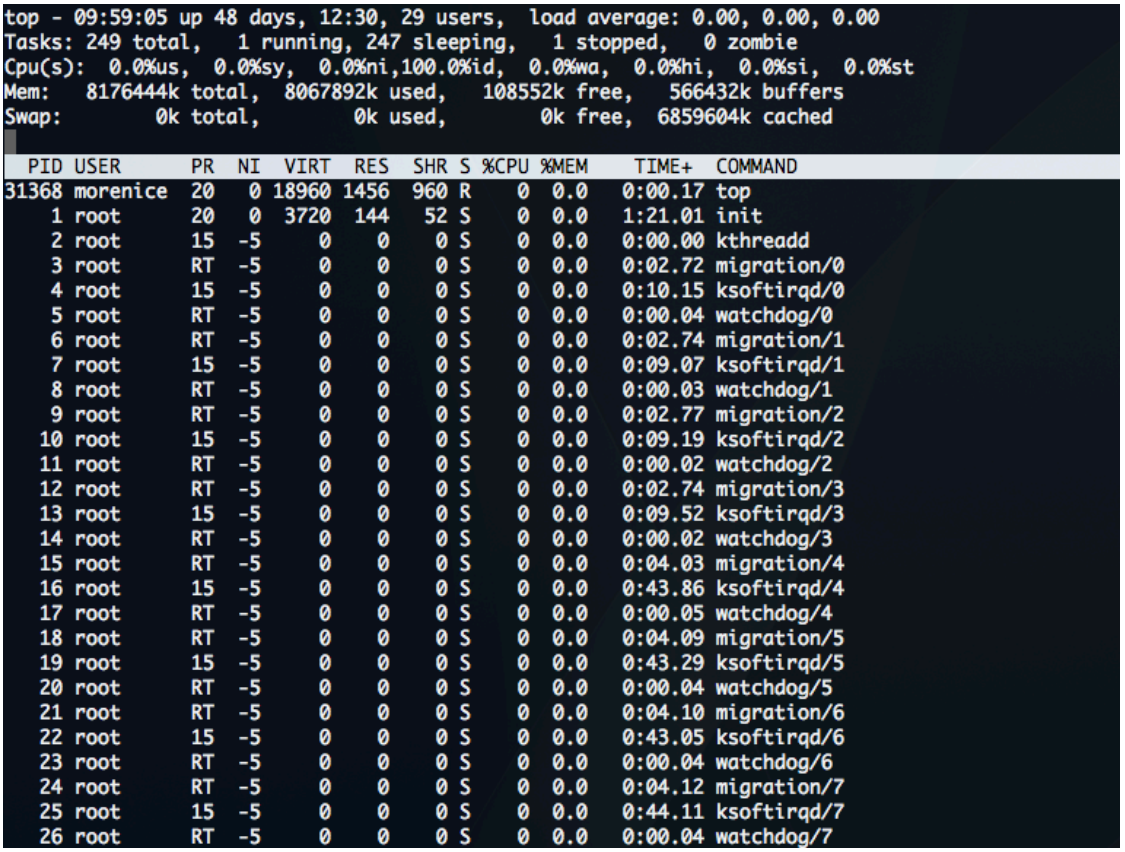
- 사용된 기술 및 도구

1. 원하는 정보를 strace 명령어를 사용해 분석. 예시로 top 명령어 1행에는 시스템 가동 시간이 나와있다. 이는 uptime 명령어로 확인 할 수 있기에, strace uptime 을 통해 uptime의 수행과정을 볼 수 있다.
2. strace 결과를 파일로 저장해 (strace -o) 프로세스가 어떤 System call을 호출하였는지 분석하였다. 다만 정보의 양이 많아 많은 기술적 문헌을 참고했다.
3. 또는 cat 명령어를 통해 표준 출력으로 읽을 수 있는 파일의 경우 터미널을 통해 직접 확인하였다.
4. 얻은 정보를 토대로, top 구현을 시작하였다. 현재 Ubuntu에서 제공하는 top의 정보들 중 구현 할 내용을 선별해 개인 분담하였다.
5. 또한, 커널 영역의 메모리 처리는 생소했기에, “디버깅을 통해 배우는 리눅스 커널의 원리”라는 책을 구매 후 스터디하였다.
6. 터미널 내 인터페이스 구현을 위해 ncurses 를 이용하였다.

3. 프로젝트 설계

- 시스템 구조 개요

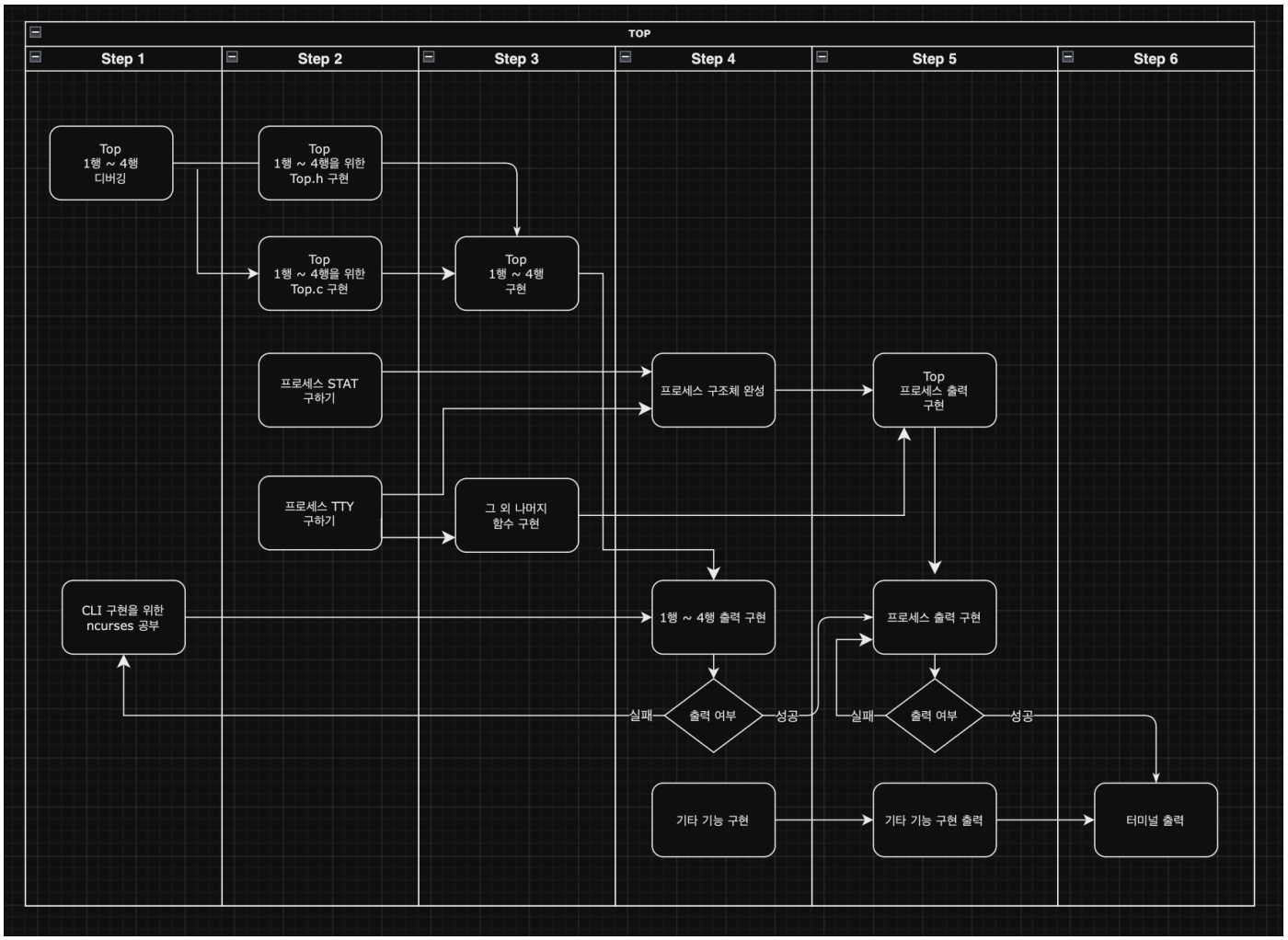
다음은 Ubuntu 내 top 명령어 실행 결과이다.



목표 구현 목록은 다음과 같다.

1. 상단 1행 ~ 4행
2. 프로세서 정보 호출 및 갱신
3. 터미널 인터페이스 구현
4. Kill, sort 구현 (미정)

- 구현된 기능의 역할 상세 설명



구현 순서는 다음과 같다.

크게 print, process, ncurse 파트로 나누었다. print는 실제 출력을 위한 코드이며, 직접적으로 불러올 수 있는 정보들을 구해 구현한다. process는 프로세스 구조체에 실제 프로세스 값들을 넣는 부분으로, 다른 부분에 비해 난이도가 높아 분리하였다. ncurse는 인터페이스 부분이며 이는 출력 코드가 완성되어야 진행 할 수 있다.

top.c / tophelp.c / top.h [가칭] 으로 분리해 진행하고 있으며, top.c에는 출력을 위한 코드가 있다.
top.h에는 include 내용과 변수 선언, 가독성을 위한 define, 프로세스 구조체 등이 있다.
tophelp.c에는 필요한 정보들을 얻기 위한 함수들이 정의되어 있다.

- 코드 구조와 주요 알고리즘 설명

현재 1행부터 4행 까지의 print 코드는 완성이 되었으나, 직접 가져올 수 없는 변수들을 위한 함수은 구현 중이다. 예를 들어 메모리 사용량 확인을 위해 /proc/meminfo에 접근해 사용 정보를 가져온다 해도, 메모리 단위 변환을 통해 출력해야 할 것이다. 이와 같이 일부 추가의 작업이 필요한 변수들은 선언만 한 채 구현해 진행했다.

<1행>

```
/*1. Uptime 가져오기*/
    time_t uptime;
    uptime = get_uptime();          //os 부팅 후 지난 시각
    char buf[BUFFER_SIZE];
    /*****  1행 UPTIME 출력    *****/
    /*2. 현재 시각 문자열 생성*/
    char nowStr[128] = { 0 };       // 현재 시각 문자열을 초기화
    time_t now = time(NULL);      // 현재 시각을 얻기
    struct tm* tmNow = localtime(&now);  // 현재 시각을 struct tm으로 변환

    // 현재 시각을 "top - HH:MM:SS " 형식으로 nowStr에 저장
    strftime(nowStr, sizeof(nowStr), "top - %H:%M:%S ", tmNow);

    /*3. Uptime 문자열 생성*/
    struct tm* tmUptime = localtime(&uptime);

    char upStr[128] = { 0 };  // uptime 문자열 초기화
    if (uptime < 60 * 60) {
        snprintf(upStr, sizeof(upStr), "%2d min", tmUptime->tm_min);
    }
    else if (uptime < 60 * 60 * 24) {
        snprintf(upStr, sizeof(upStr), "%2d:%02d", tmUptime->tm_hour, tmUptime->tm_min);
    }
    else {
        snprintf(upStr, sizeof(upStr), "%3d days, %02d:%02d", tmUptime->tm_yday, tmUptime->tm_hour,
tmUptime->tm_min);
    }

    /* 4. Load Average 가져오기 */
    FILE* loadAvgFp;
    long double loadAvg[3];
    if ((loadAvgFp = fopen(LOADAVG, "r")) == NULL) {
        fprintf(stderr, "fopen error for %s\n", LOADAVG);
        exit(1);
    }

    if (fgets(buf, BUFFER_SIZE, loadAvgFp) != NULL) {
        sscanf(buf, "%Lf%Lf%Lf", &loadAvg[0], &loadAvg[1], &loadAvg[2]);
    }
    fclose(loadAvgFp);


    /*5. 출력*/
    mvprintw(TOP_ROW, 0, "%sup %s, load average: %4.2Lf, %4.2Lf, %4.2Lf", nowStr, upStr, loadAvg[0],
loadAvg[1], loadAvg[2]);
```

이 함수 `print_top`는 시스템의 가동 시간, 현재 시각, 활성 사용자 수, 그리고 부하 평균을 화면에 출력하는 기능을 제공합니다. 각 단계는 다음과 같이 이루어집니다:

- **1**행 전체적인 코드 진행 흐름

1. **Uptime** 가져오기

- `get_uptime()` 함수는 시스템이 부팅된 이후 경과된 시간을 초 단위로 반환합니다. 이는 `time_t` 형식으로 저장됩니다.

2. 현재 시각 문자열 생성
   - `time(NULL)`을 사용하여 현재 시각을 `time_t` 형식으로 가져옵니다.
   - `localtime(&now)`를 통해 현재 시각을 `struct tm` 구조체로 변환합니다.
   - `strftime()` 함수를 사용하여 현재 시각을 `"top - HH:MM:SS "` 형식의 문자열로 변환하여 `nowStr`에 저장합니다.
3. **Uptime** 문자열 생성
   - `localtime(&uptime)`을 사용하여 업타임을 `struct tm` 구조체로 변환합니다.
   - 업타임을 다양한 형식으로 문자열 `upStr`에 저장합니다
     - 업타임이 **1**시간 미만일 경우 분**(minute)** 단위로 표현합니다.
     - 업타임이 **1**시간 이상**, 1**일 미만일 경우 `HH:MM` 형식으로 표현합니다.
     - 업타임이 **1**일 이상일 경우 `days, HH:MM` 형식으로 표현합니다.
4. **Load Average** 가져오기
   - `fopen(LOADAVG, "r")`을 사용하여 `/proc/loadavg` 파일을 읽기 모드로 엽니다. 이 파일에는 최근의 시스템 로드 애버리지 값이 저장되어 있습니다.
   - `fgets(buf, BUFFER_SIZE, loadAvgFp)`를 사용하여 파일에서 한 줄을 읽어옵니다.
   - `sscanf(buf, "%Lf%Lf%Lf", &loadAvg[0], &loadAvg[1], &loadAvg[2])`를 사용하여 읽어온 줄에서 세 개의 로드 애버리지 값을 파싱하여 `loadAvg` 배열에 저장합니다.
   - 파일을 닫습니다.
5. 출력
   - `mvprintw()` 함수를 사용하여 터미널의 `TOP_ROW` 행에 다음과 같은 형식으로 정보를 출력합니다**:**
     - `nowStr`**:** 현재 시각을 나타내는 문자열 **(**형식**:** `top - HH:MM:SS`**)**
     - `upStr`**:** 업타임을 나타내는 문자열
     - `loadAvg`**:** 최근 **1, 5, 15**분 동안의 로드 애버리지 값들 **(**`%4.2Lf` 형식으로 출력**)**

<2행>

```
#define TASK_ROW 1        //task 출력할 행

myProc procList[PROCESS_MAX];
int procCnt = 0;

// 파일이 성공적으로 열린 경우에만 아래의 코드 실행
void calculate_and_print_stats() {
unsigned int total = 0, running = 0, sleeping = 0, stopped = 0, zombie = 0, uninterrupti
ble_sleep=0, tracedORstopped=0;
total = procCnt;
for(int i = 0; i < procCnt; i++){
  if(!strcmp(procList[i].stat, "R")) //실행 중인 프로세스
    running++;
  else if(!strcmp(procList[i].stat, "D")) //불가피하게 대기 중인 프로세스
    uninterruptible_sleep++;
  else if(!strcmp(procList[i].stat, "S")) //대기 중인 프로세스
    sleeping++;
  else if(!strcmp(procList[i].stat, "T")) //정지된 프로세스
    stopped++;
  else if(!strcmp(procList[i].stat, "t")) //추적 중인 프로세스 또는 멈춤 상태인 프로세스
    tracedORstopped++;
  else if(!strcmp(procList[i].stat, "Z")) //좀비 상태인 프로세스
    zombie++;
  }


mvprintw(TASK_ROW, 0, "Tasks:  %4u total,  %4u running, %4u uninterruptible_sleep, %4u s
leeping,  %4u stopped, %4u tracedORstopped, %4u zombie", total, running, uninterruptible
_sleep, sleeping, stopped, tracedORstopped, zombie);
}
```

- **2**행 전체적인 코드 진행 흐름

1. 프로세스 상태 변수 초기화
    - 여러 상태별 프로세스 수를 저장하기 위한 변수들을 선언하고 초기화합니다.
    - **total** 변수는 전체 프로세스 수를 저장하며, **procCnt**는 시스템에서 현재 실행 중인 총 프로세스 수를 나타냅니다.
2. 프로세스 상태 카운팅
    - **for** 루프를 통해 모든 프로세스를 순회하며 각 프로세스의 상태를 확인합니다.
    - **procList[i].stat**는 각 프로세스의 상태를 나타내는 문자열입니다
    - 각 상태별 프로세스 수를 해당 변수에 증가시킵니다.
3. 프로세스 상태 출력
    - **mvprintw** 함수를 사용하여 터미널의 **TASK_ROW** 행에 프로세스 상태 정보를 출력합니다.

4. **Hertz** 값 가져오기 및 **uptime** 초기화
    - `sysconf(_SC_CLK_TCK)`을 사용하여 시스템의 **Hertz** 값을 가져옵니다. 이 값은 초당 컨텍스트 스위칭**(context switching)**의 횟수를 나타냅니다.
    - `char buffer[BUFFER_SIZE]`는 데이터를 읽어올 때 사용되는 버퍼입니다.
    - `uptime = get_uptime()`을 통해 시스템이 부팅된 이후 경과된 시간을 초 단위로 가져옵니다.

<3행>

- **3**행 전체적인 코드 진행 흐름
- 메모리 정보 읽기 및 출력
    a. 파일포인터 열기

```
char* MEMptr;
unsigned long memTotal, memFree, memUsed, memAvailable, buffers, cached;
FILE* meminfoFP;

if ((meminfoFP = fopen(MEMINFO, "r")) == NULL) {
    fprintf(stderr, "MEM 사용 관련 파일 ( %s ) 을 읽어오는데 실패했습니다.\n", MEMINFO);
    exit(1);
}
```

        - `meminfoFP`는 `/proc/meminfo` 파일을 열기 위한 파일 포인터입니다.
        - `fopen` 함수로 파일을 열고, 실패 시 에러 메시지를 출력하고 프로그램을 종료합니다.

    b. 메모리 정보 파싱

```
memset(buffer, '\0', BUFFER_SIZE);
fgets(buffer, BUFFER_SIZE, meminfoFP);

MEMptr = buffer;
while (!isdigit(*MEMptr))
{
    MEMptr++;
}
sscanf(MEMptr, "%lu", &memTotal);
```

        - 파일에서 한 줄씩 읽어와서 `buffer`에 저장합니다.

- **buffer**에서 숫자가 아닌 문자를 건너뛰고, 숫자가 시작하는 위치에서 **sscanf**로 값을 읽어옵니다.
- 이 과정을 총 다섯 번 반복하여 **memTotal**, **memFree**, **memAvailable**, **buffers**, **cached** 값을 읽어옵니다.

c. 사용 메모리 계산 및 출력

```
memUsed = memTotal - memFree - buffers - cached;
mvprintw(MEM_ROW, 0, "Kib Mem : %8lu total,  %8lu free,  %8lu used,  %8lu buff/cache"
         , memTotal, memFree, memUsed, buffers + cached);
fclose(meminfoFP);
```

- 사용된 메모리는 **memTotal - memFree - buffers - cached**로 계산합니다.
- **mvprintw** 함수를 사용하여 메모리 정보를 터미널에 출력합니다.
- 파일 포인터를 닫습니다.

<4행>

```
char* CPUptr;

    FILE* cpuStatFP;
    if ((cpuStatFP = fopen(CPUSTAT, "r")) == NULL) {
        fprintf(stderr, "CPU 사용 관련 파일 ( %s ) 을 읽어오는데 실패했습니다.\n", CPUSTAT);
        exit(1);
    }
    memset(buffer, '\0', BUFFER_SIZE);
    fgets(buffer, BUFFER_SIZE, cpuStatFP); // CPU 정보를 읽어오기
    fclose(cpuStatFP);

    CPUptr = buffer;
    while (!isdigit(*CPUptr)) CPUptr++; // isdigit으로 숫자 찾기

    long double ticks[CPUTicks] = { 0.0, };
    sscanf(CPUptr, "%Lf %Lf %Lf %Lf %Lf %Lf %Lf %Lf",
           &ticks[0], &ticks[1], &ticks[2], &ticks[3], &ticks[4], &ticks[5], &ticks[6], &ticks[7]);

    unsigned long nowTicks = 0;
    long double results[CPUTicks] = { 0.0, };

    if (beforeUptime == 0) {
        nowTicks = uptime * hertz;
        for (int i = 0; i < CPUTicks; i++) {
            results[i] = ticks[i];
        }
    }
    else {
        nowTicks = (uptime - beforeUptime) * hertz;
        for (int i = 0; i < CPUTicks; i++) {
            results[i] = ticks[i] - beforeTicks[i];
        }
    }

    for (int i = 0; i < CPUTicks; i++) {
        results[i] = (results[i] / nowTicks) * 100;
        if (isnan(results[i]) || isinf(results[i])) {
            results[i] = 0;
        }
    }

    mvprintw(CPU_ROW, 0, "%%Cpu(s):  %4.1Lf us, %4.1Lf sy, %4.1Lf ni, %4.1Lf id, %4.1Lf wa, %4.1Lf hi,
%4.1Lf si, %4.1Lf st",
             results[0], results[1], results[2], results[3], results[4], results[5], results[6],
results[7]);

    beforeUptime = uptime;
    for (int i = 0; i < CPUTicks; i++)
        beforeTicks[i] = ticks[i];
```

- **4**행 전체적인 코드 진행 흐름

1. 변수 및 파일 포인터 초기화
    - `char* CPUptr;` - **CPU** 정보 문자열을 가리킬 포인터.
    - `FILE* cpuStatFP;` - `/proc/stat` 파일을 가리킬 파일 포인터. `CPUSTAT`는 이 파일의 경로를 나타냅니다.
    - `fopen`을 통해 `cpuStatFP`에 `/proc/stat` 파일을 열고 파일 포인터를 할당합니다. 파일을 여는데 실패하면 오류 메시지를 출력하고 프로그램을 종료합니다.
    - `memset`을 통해 버퍼를 초기화합니다.
    - `fgets`를 통해 파일에서 한 줄을 읽어옵니다.
    - `fclose`를 통해 파일을 닫습니다.
2. **CPU** 사용 정보 파싱
    - `CPUptr = buffer;` - `CPUptr`가 버퍼의 시작을 가리키도록 설정합니다.
    - `while (!isdigit(*CPUptr)) CPUptr++;` - 숫자가 나올 때까지 `CPUptr`를 이동시킵니다.
    - `sscanf`를 통해 숫자 값을 읽어 `ticks` 배열에 저장합니다. 이 배열은 각 **CPU** 사용 지표 **(user, system, idle** 등**)**를 나타냅니다.
3. **CPU** 사용량 계산
    - `unsigned long nowTicks = 0;` - 현재 틱 수를 저장할 변수.
    - `long double results[CPUTicks] = { 0.0, };` - **CPU** 사용량 비율을 저장할 배열.
    - `if (beforeUptime == 0)` - 첫 번째 호출일 경우:
        - `nowTicks`를 `uptime * hertz`로 계산합니다.
        - 각 `ticks` 값을 `results`에 그대로 복사합니다.
    - `else` - 두 번째 호출 이후:
        - `nowTicks`를 `(uptime - beforeUptime) * hertz`로 계산합니다.
        - 현재 `ticks`에서 이전 `ticks`를 빼서 사용량을 계산합니다.
    - `for` 루프를 통해 `results` 배열의 값을 틱 수로 나누고 **100**을 곱해 백분율로 변환합니다. **NaN**이나 무한대 값이 나오면 **0**으로 설정합니다.
4. **CPU** 사용량 출력 및 이전 값 저장
    - `mvprintw`를 통해 `results` 배열의 값을 화면에 출력합니다. 각 값은 각각 **user, system, nice, idle, wait, hardware interrupt, software interrupt, steal** 시간을 나타냅니다.
    - `beforeUptime`에 현재 `uptime`을 저장합니다.
    - `beforeTicks`에 현재 `ticks` 값을 저장합니다.

위 코드를 통해 **CPU** 사용량을 읽고 계산하여 **3**행에 출력하는 역할을 합니다.

<5행>

- **5**행 전체적인 코드 진행 흐름
- 프로세스 정보 읽기 및 출력
    a. 열 너비 계산

```
int columnWidth[COLUMN_CNT] = {strlen(PID_STR), strlen(PR_STR), strlen(NI_STR), strlen(VIRT_STR),
strlen(RES_STR), strlen(SHR_STR), strlen(S_STR), strlen(CPU_STR), strlen(MEM_STR), strlen(TIME_P_STR),
strlen(COMMAND_STR)};

for(int i = 0; i < procCnt; i++){
    sprintf(buf, "%lu", procList[i].pid);
    if(columnWidth[PID_IDX] < strlen(buf))
        columnWidth[PID_IDX] = strlen(buf);
}

// 반복문을 통해 각 열의 최대 길이를 계산합니다 (PR, NI, VIRT, RES, SHR, S, CPU, MEM, TIME, COMMAND 열에 대해 각각
수행)
```

- 각 열의 너비를 저장할 배열을 초기화합니다.
- 각 프로세스의 속성을 읽어와서 열 너비를 계산합니다. 예를 들어, **PID**의 최대 길이를 계산하기 위해 모든 프로세스의 **PID** 길이를 확인합니다.

b. 열 시작 위치 계산

```
int startX[COLUMN_CNT] = {0, };

int startCol = 0, endCol = 0;
int maxCmd = -1;

if(col >= COLUMN_CNT - 1){
    startCol = COMMAND_IDX;
    endCol = COLUMN_CNT;
    maxCmd = COLS;
}
else{
    int i;
    for(i = col + 1; i < COLUMN_CNT; i++){
        startX[i] = columnWidth[i-1] + 2 + startX[i-1];
        if(startX[i] >= COLS){
            endCol = i;
            break;
        }
    }
    startCol = col;
    if(i == COLUMN_CNT){
        endCol = COLUMN_CNT;
        maxCmd = COLS - startX[COMMAND_IDX];
    }
}
```

- **startX** 배열은 각 열의 시작 **x**좌표를 저장합니다.
- 사용자가 선택한 열(**col**)에 따라 시작 및 종료 열을 설정합니다.
- 열 너비와 터미널 너비(**COLS**)를 고려하여 각 열의 시작 위치를 계산합니다.

<6행>

- **6**행 전체적인 코드 진행 흐름

1. 컬럼의 초기 폭 설정

```
int columnWidth[COLUMN_CNT] = {
    strlen(PID_STR), strlen(PR_STR), strlen(NI_STR),
    strlen(VIRT_STR), strlen(RES_STR), strlen(SHR_STR), strlen(S_STR),
    strlen(CPU_STR), strlen(MEM_STR), strlen(TIME_P_STR),
    strlen(COMMAND_STR)
};
```

- **columnWidth** 배열은 각 열의 초기 폭을 저장합니다. 열의 폭은 각 열 이름의 길이로 설정됩니다.

2. 각 열의 최대 폭 계산

```
for(int i = 0; i < procCnt; i++){ /* PID 최대 길이 저장 */ ... }
for(int i = 0; i < procCnt; i++){ /* PR 최대 길이 저장 */ ... }
...
```

- 각 열에 대해 반복문을 사용하여 해당 열에 들어갈 값의 최대 길이를 계산합니다. 예를 들어, **PID** 열의 경우 모든 프로세스의 **PID** 값을 문자열로 변환하여 그 길이를 확인하고, 그 중 가장 긴 값을 **columnWidth[PID_IDX]**에 저장합니다. 이를 통해 각 열의 폭이 해당 열의 모든 값을 표시할 수 있을 만큼 충분히 넓어지도록 합니다.

3. 각 열의 시작 위치 계산

```
int startX[COLUMN_CNT] = {0, }; /* 각 column의 시작 x좌표 */
int startCol = 0, endCol = 0;
int maxCmd = -1; /* COMMAND 출력 가능한 최대 길이 */
```

- **startX** 배열은 각 열의 시작 위치를 저장합니다. **startCol**과 **endCol**은 표시할 열의 범위를 나타내며, **maxCmd**는 **COMMAND** 열의 최대 출력 가능 길이를 저장합니다.

4. **COMMAND** 열만 출력하는 경우 처리

```
if(col >= COLUMN_CNT - 1){ /* COMMAND COLUMN만 출력하는 경우 */
    startCol = COMMAND_IDX;
    endCol = COLUMN_CNT;
    maxCmd = COLS;
}
```

- 사용자가 우측 화살표를 많이 눌러서 **COMMAND** 열만 출력하려는 경우, **startCol**을 **COMMAND** 열로 설정하고, **maxCmd**를 터미널 너비만큼 설정합니다.

5. 각 열의 시작 위치 및 출력 범위 계산

```
else{
    int i;
    for(i = col + 1; i < COLUMN_CNT; i++){
        startX[i] = columnWidth[i-1] + 2 + startX[i-1];
        if(startX[i] >= COLS){ /* COLUMN의 시작이 이미 터미널 너비 초과한 경우 */
            endCol = i;
            break;
        }
    }
    startCol = col;
    if(i == COLUMN_CNT){
        endCol = COLUMN_CNT; /* COLUMN 전부 출력하는 경우 */
        maxCmd = COLS - startX[COMMAND_IDX]; /* COMMAND 최대 출력 길이 */
    }
}
```

- **col**이 **COMMAND** 열보다 작은 경우, **startX** 배열을 사용하여 각 열의 시작 위치를 계산합니다. 만약 시작 위치가 터미널 너비를 초과하면 해당 열 이후의 열은 출력하지 않도록 **endCol**을 설정합니다. 모든 열이 출력 가능하면 **endCol**을 **COLUMN_CNT**로 설정하고, **maxCmd**를 **COMMAND** 열의 최대 출력 길이로 설정합니다.

tophelp.c 분석

top.c 수행을 도와주는 코드로 `get_procPath,add_proc_list,erase_proc_list,erase_proc` 등과 같은 함수들을 포함하지만 코드가 길고 보고서 길이상 가장 중요한 함수인 `add_proc_list`에 대해서만 보고서에 추가하였습니다.

<add_proc_list>

[알고리즘]

경로 접근성 확인: 주어진 경로와 그 하위 경로가 접근 가능한지 확인합니다.

기본 프로세스 정보 수집: /proc/[pid]/stat 파일을 읽어 프로세스의 기본 정보를 수집합니다.

사용자 정보 수집: 프로세스 소유자의 UID를 통해 사용자 이름을 얻습니다.

CPU 사용량 계산: 프로세스의 CPU 사용 시간을 계산합니다.

메모리 사용량 수집: /proc/[pid]/status 파일을 읽어 프로세스의 메모리 사용 정보를 수집합니다.

TTY 정보 수집: 프로세스의 터미널(TTY) 정보를 얻습니다.

우선순위 및 nice 값 수집: 프로세스의 우선순위 및 nice 값을 수집합니다.

상태 정보 수집: 프로세스의 상태 정보를 수집합니다.

시작 시간 계산: 프로세스의 시작 시간을 포맷팅합니다.

CPU 시간 계산: 프로세스가 사용한 총 CPU 시간을 계산하고 포맷팅합니다.

명령어 및 전체 명령어 수집: 프로세스의 실행 명령어와 전체 명령어 라인을 수집합니다.

프로세스 목록에 추가: 수집한 정보를 구조체에 저장하고 전역 프로세스 목록에 추가합니다.

이 함수는 /proc 파일 시스템을 통해 프로세스 정보를 수집하고, 이를 구조체에 저장하여 프로세스 목록을 관리합니다. 이 과정은 파일 접근성 확인, 기본 프로세스 정보 수집, CPU 및 메모리 사용량 계산, 상태 정보 수집, 시작 시간 및 CPU 시간 계산, 명령어 수집 등 다양한 단계를 포함합니다. 수집된 정보는 전역 프로세스 목록에 저장됩니다.

[코드]

- 주요 함수 **add_proc_list** 전체적인 코드 설명

1. 파일 접근 및 초기화

```
void add_proc_list(char path[1024], unsigned long cpuTimeTable[8192]) {
    if (access(path, R_OK) < 0) {
        fprintf(stderr, "access error for %s\n", path);
        return;
    }
    myProc proc;
    erase_proc(&proc);
```

- 함수는 **path**와 **cpuTimeTable**을 인자로 받습니다.
- 주어진 **path**에 접근 가능한지 확인하고, 접근할 수 없으면 에러 메시지를 출력하고 반환합니다.
- **myProc** 구조체를 초기화하는 **erase_proc** 함수를 호출합니다.

2. 프로세스 **ID(PID)** 읽기

```
    char statPath[1024];
    strcpy(statPath, path);
    strcat(statPath, "/stat");

    if (access(statPath, R_OK) < 0) {
        fprintf(stderr, "access error for %s\n", statPath);
        return;
    }
    FILE* statFp;
    if ((statFp = fopen(statPath, "r")) == NULL) {
        sleep(1);
        return;
    }

    char statToken[32][32];
    memset(statToken, '\0', 32 * 32);
    for (int i = 0; i < 32; i++)
        fscanf(statFp, "%s", statToken[i]);
    fclose(statFp);

    proc.pid = (long)atoi(statToken[0]); // pid 획득
```

- **/stat** 파일 경로를 만들고 접근 가능한지 확인합니다.
- 파일을 열고, 열 수 없는 경우 **1**초간 기다린 후 반환합니다.
- **stat** 파일에서 데이터를 읽어와 **statToken** 배열에 저장합니다.
- 프로세스 **ID(PID)**를 **statToken[0]**에서 읽어와 **proc.pid**에 저장합니다.

3. 사용자명 읽기

```
    char statusPath[1024];
    strcpy(statusPath, path);
    strcat(statusPath, "/status");

    if (access(statusPath, R_OK) < 0) {
        fprintf(stderr, "access error for %s\n", statusPath);
        return;
    }
    FILE* statusFp;
    if ((statusFp = fopen(statusPath, "r")) == NULL) {
        sleep(1);
        return;
    }

    char line[256];
    while (fgets(line, sizeof(line), statusFp)) {
        if (strncmp(line, "Uid:", 4) == 0) {
            unsigned long uid;
            sscanf(line, "Uid:\t%lu", &uid);
            proc.uid = uid;
            struct passwd* pw = getpwuid(uid);
            if (pw) {
                strncpy(proc.user, pw->pw_name, UNAME_LEN - 1);
                proc.user[UNAME_LEN - 1] = '\0';
            } else {
                strncpy(proc.user, "unknown", UNAME_LEN - 1);
                proc.user[UNAME_LEN - 1] = '\0';
            }
            break;
        }
    }
    fclose(statusFp);
```

- **`/status`** 파일 경로를 만들고 접근 가능한지 확인합니다.
- 파일을 열고, 열 수 없는 경우 **1**초간 기다린 후 반환합니다.
- **`status`** 파일에서 **"Uid:"**로 시작하는 줄을 찾아 **UID**를 읽어옵니다.
- **UID**를 통해 사용자명을 얻고 **`proc.user`**에 저장합니다. 만약 사용자명을 찾을 수 없으면 **"unknown"**으로 설정합니다.

4. **CPU** 사용률 계산

```
    unsigned long utime = (unsigned long)atoi(statToken[13]);
    unsigned long stime = (unsigned long)atoi(statToken[14]);
    unsigned long startTime = (unsigned long)atoi(statToken[21]);
    unsigned long long totalTime = utime + stime;
    time_t uptime = get_uptime();
    unsigned long long Hertz = sysconf(_SC_CLK_TCK);

    long double seconds = uptime - (startTime / Hertz);
    long double cpuUsage = 100.0 * ((totalTime / Hertz) / seconds);

    if (cpuUsage < 0 || cpuUsage > 100) {
        proc.cpu = 0;
    } else {
        proc.cpu = roundl(cpuUsage * 100) / 100.0;
    }
```

- `statToken`에서 **utime, stime, startTime**을 읽어옵니다.
- 시스템 업타임과 **Hertz** 값을 이용하여 **CPU** 사용 시간을 계산합니다.
- **CPU** 사용률을 계산하고, **0**에서 **100** 사이의 값으로 반올림하여 `proc.cpu`에 저장합니다.

5. 메모리 사용률 계산

```
    unsigned long vsz = 0, rss = 0, shr = 0, vmLck = 0;
    if ((statusFp = fopen(statusPath, "r")) != NULL) {
        while (fgets(line, sizeof(line), statusFp)) {
            if (strncmp(line, "VmSize:", 7) == 0) {
                sscanf(line, "VmSize:\t%lu", &vsz);
            }
            else if (strncmp(line, "VmRSS:", 6) == 0) {
                sscanf(line, "VmRSS:\t%lu", &rss);
            }
            else if (strncmp(line, "RssShmem:", 9) == 0) {
                sscanf(line, "RssShmem:\t%lu", &shr);
            }
            else if (strncmp(line, "VmLck:", 6) == 0) {
                sscanf(line, "VmLck:\t%lu", &vmLck);
            }
        }
        fclose(statusFp);
    }

    proc.vsz = vsz;
    proc.rss = rss;
    proc.shr = shr;

    long double memUsage = (long double)rss / 1024 * 100.0;
    if (memUsage < 0 || memUsage > 100) {
        proc.mem = 0;
    } else {
        proc.mem = roundl(memUsage * 100) / 100.0;
    }
```

- `status` 파일을 다시 열어 메모리 사용량**(VmSize, VmRSS, RssShmem, VmLck)**을 읽어옵니다.

- 메모리 사용량을 계산하고, **0**에서 **100** 사이의 값으로 반올림하여 **proc.mem**에 저장합니다.

6. 우선순위 및 니스 값, 시작 시간 및 실행 시간, 명령어 읽기

```
    unsigned long vsz = 0, rss = 0, shr = 0, vmLck = 0;
    if ((statusFp = fopen(statusPath, "r")) != NULL) {
        while (fgets(line, sizeof(line), statusFp)) {
            if (strncmp(line, "VmSize:", 7) == 0) {
                sscanf(line, "VmSize:\t%lu", &vsz);
            }
            else if (strncmp(line, "VmRSS:", 6) == 0) {
                sscanf(line, "VmRSS:\t%lu", &rss);
            }
            else if (strncmp(line, "RssShmem:", 9) == 0) {
                sscanf(line, "RssShmem:\t%lu", &shr);
            }
            else if (strncmp(line, "VmLck:", 6) == 0) {
                sscanf(line, "VmLck:\t%lu", &vmLck);
            }
        }
        fclose(statusFp);
    }

    proc.vsz = vsz;
    proc.rss = rss;
    proc.shr = shr;

    long double memUsage = (long double)rss / 1024 * 100.0;
    if (memUsage < 0 || memUsage > 100) {
        proc.mem = 0;
    } else {
        proc.mem = roundl(memUsage * 100) / 100.0;
    }
```

- **statToken**에서 우선순위**(priority)**와 니스**(nice)** 값을 읽어옵니다.
- 프로세스의 시작 시간을 계산하여 적절한 형식으로 **proc.start**에 저장합니다.
- 프로세스의 실행 시간을 계산하여 **proc.time**에 저장합니다.
- 명령어를 읽어와 **proc.cmd**에 저장합니다. 명령어의 마지막 **')'** 문자를 제거합니다.

7. 프로세스 목록에 추가
   - **proc** 구조체를 **procList**에 추가하고, 프로세스 개수를 증가시킵니다.

5. 테스트 및 결과

1) top 실행

```
top - 03:14:09 up  4:44, load average: 3.09, 1.59, 0.94
Tasks:   209 total,     2 running,     0 uninterruptible_sleep,  152 sleeping,     0 stopped,    0 tracedORstopped,    0 zombie
Kib Mem :  4001444 total,    378140 free,   2191176 used,   1432128 buff/cache
%Cpu(s):  24.0 us,  0.0 sy, 30.7 ni, 126.0 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st

  PID    PR   NI      VIRT     RES    SHR  S             %CPU  MEM  TIME+  COMMAND
29122    20    0  11683520  346392  16240             S  50.9  0.0   0:54  firefox
 1552    20    0   4802856  563068  28012             S   9.9  0.0  57:28  gnome-shell
29014    20    0    713764   71280   6788             S   1.7  0.0   0:04  gedit
  444    20    0     25928   14020      0             S   1.2  0.0   6:49  systemd-resolve
29385    20    0   2417160   91436    960             S   1.1  0.0   0:01  WebExtensions
 1973    20    0    153532    3468      0             S   0.5  0.0   3:06  VBoxClient
24704    20    0   1025168   95612   6812             S   0.4  0.0   0:28  nautilus
 1926    20    0    236564   83616  21748             S   0.3  0.0   1:58  Xwayland
28484    20    0         0       0      0             I   0.3  0.0   0:05  kworker/u4:3-ext4-rsv-conversio
28249    20    0         0       0      0             I   0.2  0.0   0:06  kworker/u4:2-ext4-rsv-conversio
  442    20    0     14828    6912      0             S   0.1  0.0   0:45  systemd-oomd
28837    20    0         0       0      0             I   0.1  0.0   0:01  kworker/0:0-cgroup_destroy
   15    20    0         0       0      0             R   0.1  0.0   0:24  rcu_preempt
20099    20    0   1037940  177040    184             S   0.1  0.0   0:09  snap-store
 1742    20    0    315096   11624      0             S   0.1  0.0   0:22  ibus-daemon
 2009    20    0   1392832  113328   1536             S   0.1  0.0   0:17  xdg-desktop-por
26716    20    0   2732492   63572   7284             S   0.1  0.0   0:02  gjs
   34    20    0         0       0      0             S   0.0  0.0   0:14  kcompactd0
20319    20    0    461492   88304      0             S   0.0  0.0   0:05  fwupd
    1    20    0    166980   12136      0             S   0.0  0.0   0:09  systemd
  593    20    0   1393376   26828      0             S   0.0  0.0   0:11  snapd
  955    20    0    362468    3852      0             S   0.0  0.0   0:09  VBoxService
 1148    20    0    447088   43216      0             S   0.0  0.0   0:11  packagekitd
 1390     9  -11   1684636   21680    376             S   0.0  0.0   0:09  pulseaudio
   23    20    0         0       0      0             S   0.0  0.0   0:06  ksoftirqd/1
   48    20    0         0       0      0             S   0.0  0.0   0:07  kswapd0
   96     0  -20         0       0      0             I   0.0  0.0   0:06  kworker/0:1H-kblockd
  217    19   -1     65176   16936      4             S   0.0  0.0   0:07  systemd-journal
  572    20    0     11112    7040      0             S   0.0  0.0   0:06  dbus-daemon
 1406    20    0     17492   13312      0             S   0.0  0.0   0:08  dbus-daemon
 1485    39   19    721680   27368      0             S   0.0  0.0   0:07  tracker-miner-f
 2015    20    0    163604    7424      0             S   0.0  0.0   0:06  ibus-engine-sim
   14    20    0         0       0      0             S   0.0  0.0   0:05  ksoftirqd/0
   22  -100    0         0       0      0             S   0.0  0.0   0:02  migration/1
   47     0  -20         0       0      0             I   0.0  0.0   0:05  kworker/1:1H-kblockd
  173    20    0         0       0      0             S   0.0  0.0   0:03  jbd2/dm-0-8
  574    20    0    408708   18868      0             S   0.0  0.0   0:04  NetworkManager
  585    20    0    242976   11528      0             S   0.0  0.0   0:02  polkitd
 1381    20    0     26704   18540      0             S   0.0  0.0   0:05  systemd
 1623    20    0    315204    8192      0             S   0.0  0.0   0:04  gvfs-afc-volume
 1743    20    0    991880   24012    204             S   0.0  0.0   0:03  gsd-color
```

2) PID에서 enter 키를 누를 경우 정렬

```
top - 03:14:33 up  4:44, load average: 2.04, 1.46, 0.91
Tasks:   209 total,     1 running,     0 uninterruptible_sleep,  152 sleeping,     0 stopped,    0 tracedORstopped,    0 zombie
Kib Mem :  4001444 total,    373408 free,   2195832 used,   1432204 buff/cache
%Cpu(s):  19.0 us,  0.0 sy, 27.3 ni, 134.0 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st

  PID    PR   NI      VIRT     RES    SHR  S             %CPU  MEM  TIME+  COMMAND
    1    20    0    166980   12136      0             S   0.0  0.0   0:09  systemd
    2    20    0         0       0      0             S   0.0  0.0   0:00  kthreadd
    3     0  -20         0       0      0             I   0.0  0.0   0:00  rcu_gp
    4     0  -20         0       0      0             I   0.0  0.0   0:00  rcu_par_gp
    5     0  -20         0       0      0             I   0.0  0.0   0:00  slub_flushwq
    6     0  -20         0       0      0             I   0.0  0.0   0:00  netns
    8     0  -20         0       0      0             I   0.0  0.0   0:00  kworker/0:0H-events_highpri
   10     0  -20         0       0      0             I   0.0  0.0   0:00  mm_percpu_wq
   11    20    0         0       0      0             I   0.0  0.0   0:00  rcu_tasks_kthread
   12    20    0         0       0      0             I   0.0  0.0   0:00  rcu_tasks_rude_kthread
   13    20    0         0       0      0             I   0.0  0.0   0:00  rcu_tasks_trace_kthread
   14    20    0         0       0      0             S   0.0  0.0   0:05  ksoftirqd/0
   15    20    0         0       0      0             I   0.1  0.0   0:24  rcu_preempt
   16  -100    0         0       0      0             S   0.0  0.0   0:01  migration/0
   17   -51    0         0       0      0             S   0.0  0.0   0:00  idle_inject/0
   19    20    0         0       0      0             S   0.0  0.0   0:00  cpuhp/0
   20    20    0         0       0      0             S   0.0  0.0   0:00  cpuhp/1
   21   -51    0         0       0      0             S   0.0  0.0   0:00  idle_inject/1
   22  -100    0         0       0      0             S   0.0  0.0   0:02  migration/1
   23    20    0         0       0      0             S   0.0  0.0   0:06  ksoftirqd/1
   26    20    0         0       0      0             S   0.0  0.0   0:00  kdevtmpfs
   27     0  -20         0       0      0             I   0.0  0.0   0:00  inet_frag_wq
   28    20    0         0       0      0             S   0.0  0.0   0:00  kauditd
   29    20    0         0       0      0             S   0.0  0.0   0:00  khungtaskd
   31    20    0         0       0      0             S   0.0  0.0   0:00  oom_reaper
   33     0  -20         0       0      0             I   0.0  0.0   0:00  writeback
   34    20    0         0       0      0             S   0.0  0.0   0:14  kcompactd0
   35    25    5         0       0      0             S   0.0  0.0   0:00  ksmd
   36    39   19         0       0      0             S   0.0  0.0   0:00  khugepaged
   37     0  -20         0       0      0             I   0.0  0.0   0:00  kintegrityd
   38     0  -20         0       0      0             I   0.0  0.0   0:00  kblockd
   39     0  -20         0       0      0             I   0.0  0.0   0:00  blkcg_punt_bio
   40     0  -20         0       0      0             I   0.0  0.0   0:00  tpm_dev_wq
   41     0  -20         0       0      0             I   0.0  0.0   0:00  ata_sff
   42     0  -20         0       0      0             I   0.0  0.0   0:00  md
   43     0  -20         0       0      0             I   0.0  0.0   0:00  edac-poller
   44     0  -20         0       0      0             I   0.0  0.0   0:00  devfreq_wq
   45   -51    0         0       0      0             S   0.0  0.0   0:00  watchdogd
   47     0  -20         0       0      0             I   0.0  0.0   0:05  kworker/1:1H-kblockd
   48    20    0         0       0      0             S   0.0  0.0   0:07  kswapd0
   49    20    0         0       0      0             S   0.0  0.0   0:00  ecryptfs-kthread
```

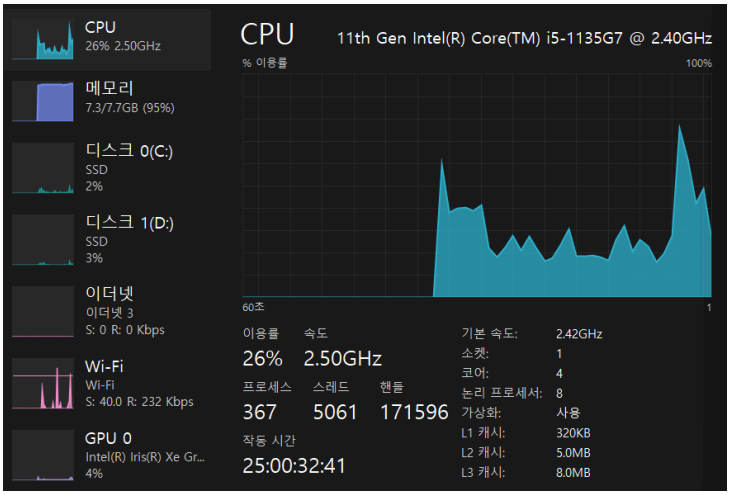
3) %CPU에서 enter 키를 누를 경우

```
PID: 1
User: root
CPU Usage: 0.03
Memory Usage: 0.00
Command:
Time Slice: 2
Time Passed: 2
Press 'q' to quit.
```

```
PID: 4
User: root
CPU Usage: 0.00
Memory Usage: 0.00
Command:
Time Slice: 2
Time Passed: 12
Press 'q' to quit.
```

6. 향후 개선 사항

Linux에서 top을 구현하면 RAM의 메모리 영역과 디스크를 메모리처럼 이용하는 Swap 메모리가 나타나지만 특정 프로세스의 파일을 읽어와서 가상 메모리 크기, 실제 메모리 사용량, 공유 메모리 및 메모리 사용 비율을 구현하기엔 한계가 있었다. CPU 사용률 계산한 부분과 같이 특정 파일의 경로를 불러오기까지는 성공하였지만 이를 메모리 정보가 없는 경우와 있는 경우를 나누어 처리하는 부분에서 논리적으로 코드를 구현하기가 어려웠다. 조건부로 구현할 수록 고려해야할 조건들이 계속해서 생겨나 오류가 해결되지 않았다. 또, 기존의 top은 uid를 구할 수 있지만 사용자의 이름을 얻어 user명을 획득하는 과정에 어려움이 있었다. user명이 긴 경우 버퍼 크기에 한계가 있어 오버플로가 발생하는 점, 사용자의 정보를 얻을 때 실패하여 NULL을 반환하는 점 등을 고려하여 구현하기가 까다로웠다.



구현한 top을 이용하여 모니터링할 수 있는 도구로 개선할 수 있다. 예를 들어 window의 작업 관리자를 보면 위의 그림과 같이 동적인 그래프로 사용률을 확인할 수 있다. 이처럼 top을 JavaScript 기반의

라이브러리들을 이용하여 웹 인터페이스에서 동적 그래프를 이용하여 보다 직관적인 시각화를 제공하는 방법으로 향후 개선해보고자 한다.