TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2018-03-21 (20회차)

강사: Innova Lee(이상훈)
gcccompil3r@gmail.com
학생: 정유경
ucong@naver.com

Quiz 2 분석

카페에 있는 50 번 문제(성적 관리 프로그램)을 개조한다.

기존에는 입력 받고 저장한 정보가 프로그램이 종료되면 없어진다.

입력한 정보를 영구히 유지할 수 있는 방식으로 만들면 더 좋지 않을까 ?

* 조건

1. 파일을 읽어서 이름 정보와 성적 정보를 가져온다.
2. 초기 구동 시 파일이 없을 수 있는데 이런 경우엔 읽어서 가져올 정보가 없다.
3. 학생 이름과 성적을 입력할 수 있도록 한다.
4. 입력된 이름과 성적은 파일에 저장되어야 한다.
5. 당연히 통계 관리도 되어야한다(평균, 표준 편차)
6. 프로그램을 종료하고 다시 켜면 파일에서 앞서 만든 정보들을 읽어와서 내용을 출력해줘야 한다.
7. 언제든 원하면 내용을 출력할 수 있는 출력함수를 만든다. [특정 버튼을 입력하면 출력이 되게 만듦]

(역시 System Call 기반으로 구현하도록 함)

1. AVL을 이용하여 구현 → best
2. Queue를 이용하여 구현

```c
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

typedef struct __queue
{
        int score;
        char *name;
        struct __queue *link;
} queue;

void disp_student_manager(int *score, char *name, int size)
{
        char *str1 = "학생 이름을 입력하시오: ";
        char *str2 = "학생 성적을 입력하시오: ";
        char tmp[32] = { 0 };
        write(1, str1, strlen(str1)); // 표준출력, 출력할 내용 "이름을 입력하시오", 길이
        read(0, name, size); // 표준입력, name에 입력을 저장, 길이
        write(1, str2, strlen(str2)); // 표준출력, 출력할 내용 "성적을 입력하시오", 길이
        read(0, tmp, sizeof(tmp)); // 표준입력, tmp에 입력을 저장, 길이
        *score = atoi(tmp); // read안에 char*가 들어가야 한다!  // char tmp[32] = { 0 }; 선언
                // *score = atoi(tmp); //Usage: int atoi(포인터)  문자→숫자 //cf. sprint : 숫자→문자열
}
void confirm_info(char *name, int score)
{
        printf("학생 이름 = %s₩n", name);
        printf("학생 성적 = %d₩n", score);
}
```

```
queue *get_queue_node(void)
{
	queue *tmp;
	tmp = (queue *)malloc(sizeof(queue));
	tmp->name = NULL;
	tmp->link = NULL;
	return tmp;
}

void enqueue(queue **head, char *name, int score)
{
	if (*head == NULL)
	{
		int len = strlen(name);
		(*head) = get_queue_node();
		(*head)->score = score;
		(*head)->name = (char *)malloc(len + 1);
		strncpy((*head)->name, name, len);
		return;
	}
	enqueue(&(*head)->link, name, score);
}

void print_queue(queue *head)
{
	queue *tmp = head;
	while (tmp)
	{
		printf("name = %s, score = %d₩n", tmp-
>name, tmp->score);
		tmp = tmp->link;
	}
}
```

```
void remove_enter(char *name)
{
        int i;                                          // cf. 문장의 경우 모든 개행을 파악해서 한칸씩 당겨야 한다
        for (i = 0; name[i]; i++)
                if (name[i] == '₩n')
                        name[i] = '₩0'; // 개행(₩n)을 널문자(₩0)로 대체한다
}
 int main(void)
 {
        int cur_len, fd, btn = 0;
        int score;
        char name[32] = { 0 }; // Slab 할당자가 32 byte 를 관리하기 때문에 성능이 빠름
        char str_score[32] = { 0 };
        char buf[64] = { 0 };
        queue *head = NULL;
        for (;;)
        {
                printf("1 번: 성적 입력, 2 번: 파일 저장, 3 번: 파일 읽기, 4 번: 종료₩n");
                scanf("%d", &btn);

switch (btn)
{
case 1:
        disp_student_manager(&score, name, sizeof(name));
        remove_enter(name); // 개행(₩n)을 널문자(₩0)로 대체한다
        confirm_info(name, score);
        enqueue(&head, name, score);
        print_queue(head);
        break;
```

Slap Allocator란?

1. 문제상황

a. 우리는 30 byte 크기의 구조체를 메모리로부터 할당받기를 원한다.

b. 이때 하나의 페이지 프레임(4KB)을 30byte 구조체를 넣기 위해 할당받는것은 불필요하다.

* 4096 byte - 30byte = 4066 bytes 의 낭비가 일어난다.

c. 즉, 내부에서 불필요한 메모리 할당(4066 byte)가 발생한다.

d. 이러한 구조체를 우리는 100번 할당, 해제를 반복해야 한다. (Task 100번 생성 종료 라던지..)

* 여기서 두가지 문제가 발생함을 알 수 있다.

1) Internal Fragmentation - 내부 단편화(파편화) 문제.. 즉 4066 bytes 가 쓸모없어진다.

2) 1번 할당 해제 할때마다 페이지를 할당 후, 초기화 하고 해제하는것은 많은 시간이 걸린다.

2. 어떻게 해결할 것인가?

- 슬랩 할당자(Slab Allocator)를 도입한다.

case 2:

```
if ((fd = open("score.txt", O_CREAT | O_EXCL | O_WRONLY, 0644)) < 0)
// 없으면생성 | 있으면무시 | 읽을필요없다
fd = open("score.txt", O_RDWR | O_APPEND); // 읽기쓰기 | 이어쓰기(파일포인터(f_pos) 맨 뒤로 당긴다)
// O_APPEND : lseek(fd, SEEKCUL, SEEKEND) 와 같은기능
/* 어떤 형식으로 이름과 성적을 저장할 것인가 ?
        저장 포맷: 이름,성적₩n */
strncpy(buf, name, strlen(name));
cur_len = strlen(buf);  // ',' 넣어야 하니까 위치를 확인하자!
        //printf("cur_len = %d₩n", cur_len);
buf[cur_len] = ',';
sprint(str_score, "%d", score);  // 숫자score → 문자열str_score // 버퍼, 바꿀형식, 해당값
strncpy(&buf[cur_len + 1], str_score, strlen(str_score)); // &buf[cur_len + 1] :  뒤에 연결해준다!
buf[strlen(buf)] = '₩n'; // 저장형식을 참고하여 개행을 넣어준다
        //printf("buf = %s, buf_len = %lu₩n", buf, strlen(buf)) // %d→%lu
write(fd, buf, strlen(buf));
close(fd);
break;
```

```
case 3:
        if ((fd = open("score.txt", O_RDONLY)) > 0)  //fd = open()<0 로 쓰지 않도록 주의, fd에  참, 거짓 반환
됨
        {
                int i, backup = 0;
                read(fd, buf, sizeof(buf)); // sizeof(buf)이므로 다 들어온다. ','가 오는 곳 찾아서 해결!

                for (i = 0; buf[i]; i++)
                {
                        if (!(strncmp(&buf[i], ",", 1))) // 부정! // ',' 있는지 확인
                        {
                                strncpy(name, &buf[backup], i - backup); // 배열시작 0이니까...????
                                backup = i + 1; // 처음부터 복사하지 않도록 backup해주자
                        }
                        if (!(strncmp(&buf[i], "₩n", 1))) // 부정! // 개행 있는지 확인
                        {
                                strncpy(str_score, &buf[backup], i - backup);
                                backup = i + 1;
                                enqueue(&head, name, atoi(str_score));
                        }
                }
                print_queue(head);
        }
        else
                break;
        break;
```

```
// 이름1,성적1₩n
// 이름2,성적2₩n
// .....
// 이름n,성적n₩n
```

```
                case 4:
                        goto finish;
                        break; // 형식상

                default: // 잘못입력할 경우
                        printf("1, 2, 3, 4 중 하나 입력하시오₩n");
                        break;
                }
        }
finish:
        return 0;
}
```

명령어 ps, tail

- **명령어 ps**

ps -ef | grep bash | grep -v grep | awk '{print $2}’

- 현재 실행되고 있는 프로세스
- bash 찾기(찾는 bash, 현재 구동중인 bash)
- grep 프로세스를 제외
- PID(프로세스 ID)알아내기

- **명령어 tail**

-tail -c 20 [파일명] : -c 문자 수

-Tail –n 20 [파일명] : -n 라인 수

1. dup.c

fd=open("a.txt",O_WRONLY|O_CREAT|O_TRUNC,0644);

// WR, 없으면 생성해라, 갱신해라, 권한 0644

// 기본값이 3이다...파일 처음 오픈 시 파일 식별자의 초기값은 3...? 확인필요

.

close(1); // 표준출력이 닫힌다

dup(fd);

//그 상태에서 dup 한다. 방금 종료된 1을 복사한다 fd가 1의 역할을 한다

// printf는 모니터에 뿌리는데 그걸 1이 받아간다.

//fd 3인 a.txt로 연결된다

printf("출력될까???₩n");

// a.txt로 출력된다!!!!

2. dup_2.c

```
fd= open("a.txt", O_RDONLY); // a.txt 연다

close(0); // 표준입력 닫는다

dup(fd);     // dup가 하는 일은 윗줄 0번을 fd가 대체하도록
             // 키보드로 받는 입력을 fd로 받겠다
             // 즉, 파일 자체가 0번을 대체한다. (입력 자체가 파일)
gets(buff);  //문자열 입력을 받는 함수 gets()
             //키보드 입력 받을 수 없다. Gets 무시되고 printf 출력된다!

printf("출력될까?₩n"); // 출력된다

//gets받는방법
printf("%s",buff);  // 추가하면, buff가 출력된다
                    // 파일 내용이 출력된다(a.txt)...?
```

3. lseek.c

```
int main(int argc, char *argv[])
char ch='a';
int fd=open(argv[1], O_WRONLY|O_CREAT|O_TRUNC,0644);

lseek(fd,512-1,SEEK_SET); // fd에서 512-1만큼 오프셋을 주어서 f_pos 를 set한다
                          // 즉, 512번째에 f_pos SEEK_SET하고 다음내용인 'a'를 512번지의 1byte 에 기록한다.

write(fd,&ch,4); // fd에 쓴다 &ch에서 1바이트를 가져와서

close(fd); // open한 fd를 닫는다
```

실행: ./a.out mbr.txt

확인: xxd mbr.txt

MBR이란?

- MBR이란 하드디스크로 부팅하기위한 정보와 파티션 분할 정보 부팅에 사용되는 실제 에 대한 정보가 저장된 곳으로 하드 디스크로 들어오는 관문이 되는 곳이다.
- 하드 디스크의 첫번째 정보 저장공간(첫번째 sector)에 저장되어 있다. 이 첫번째 sector는 512Byte 이며, 512바이트 중에서 처음 446바이트는 운영체제를 읽어 들이기 위한 준비단계 코드가 들어 있으며 나머지 64바이트에 파티션에 대한 정보를 들어 있다.
- 2바이트는 MBR이 맞는지 확인하기 값(Magic Number)으로 기록 → 이를 보고 윈도우 또는 리눅스 전용 부팅디스크임을 인식한다.

여러 개의 프로세스가 동시에 실행될 수 있는 이유?!

아주 빠른 속도로 여러 프로세스들이 제어권을 넘겨주면서 CPU를 사용한다면 우리는 느끼지 못하는 순간에 모든 작업이 완료된다.

1. 정말 동시일까?! ← 싱글코어
2. 병렬처리 ← 멀티코어

결론, context switiching을 통한 multitasking이 실현된다

컴퓨터 구조론 핵심

CPU는 한순간에 한가지의 연산만 수행한다!

“CPU클럭이 2.2~2.4Ghz”

- 1클럭: 명령어 하나(파이프라인 1개) 수행 시 걸리는 시간.
- 20억개 명령어를 1초에 수행한다.

Context switching by OS

- 프로세스마다 각각의 task_struct가 생성된다. Task_struct 하위의 tread_struct와 thread_union에 레지스터 정보가 저장된다. 즉, Context가 바로 범용 레지스터이다.

- 프로세스들이 CPU를 얻으려고 경쟁하는 과정에서 context switching이 일어난다.

이때 운영체제가 고려하는 것이 우선순위이다.

- Run Queue에 현재 동작중인 프로세스가 있다. 이때, 유일한 SW인터럽트인 시스템 콜이 일어나면 운영체제는 작업중이던 프로세스를 Wait Queue에 대기시킨다. 커널에서 인터럽트를 처리하고 나서 Wait Queue에 있는 프로세스들의 우선순위를 비교하여 프로세스를 다시 Run Queue로 복귀시킨다.
- Context switching할때마다 파이프라인이 깨진다.

Blocking vs Non-Blocking

Blocking

반드시 순차적으로 이루어져야 하는 상황에 사용

운영체제의 스케줄링은 프로세스의 우선순위와 관련한것이고 컴파일러의 스케줄링은 명령어 우선순위와 관련된다. 컴파일러는 명령어의 위치를 바꾸어 파이프라인의 지연을 막는데 이를 인스트럭션 스케줄링이라고 한다. 이때 작업순서가 깨질 수 있다. 이를 해결하기 위해 Memory barrier를 사용한다.

예를들면, 운영체제 부팅 시 하드웨어 초기화가 꼭 먼저 이루어져야 하는 경우가 있다. 또는 read( )시 입력할 때까지 제어권을 넘겨주지 않는 경우도 있다.

Non-Blocking

다수의 빠른 통신이 이루어져야 하는 상황에 사용

유연하게 대처가능

4. blocking.c (myfifo.c)

```
int fd, ret;
char buf[1024];
mkfifo("myfifo"); // 파이프 만드는 함수!! 파이프란, 데이터를 연결하는 통로!
fd = open("myfifo", O_RDWR);

for(;;)
{
        ret = read(0, buf, sizeof(buf));
        buf[ret-1] = 0;
        printf("Keyboard Input : [%s]₩n", buf);

        read(fd, buf, sizeof(buf));  // ret 즉, 써준 만큼 받는다
        buf[ret -1] = 0;
        printf("pipe Input : [%s]₩n", buf);
}
```

mkfifo myfifo → ./a.out → 새 터미널: cat > myfifo

5. nonblock.c

```
fd = open("myfifo", O_RDWR);
fcntl(0, F_SETFL, O_NONBLOCK);  // blocking에서 바뀐 부분 //자주 쓰이는 루틴이니 암기하자!
fcntl(fd, F_SETFL, O_NONBLOCK); // SETFL: 파일권한을 Nonblocking으로 셋팅하여 read( )>0 이면 바로 넘긴다.

if((ret=read(0,buf,sizeof(buf))) > 0)  // 상황1
 buf[ret-1] =0; // 마지막을 널문자로 바꿔준다
printf("Keyboard Input :[%s}₩n", buf);

if ((ret = read(fd, buf, sizeof(buf))) >0)  // 상황2
 buf[ret-1] =0;
printf("Pipe Input : [%s]₩n", buf); }
```

Block Device vs Character Device

- **B**lock Device

특정 단위로 움직인다.

물리메모리의 최소단위는 2의 12승(4096byte, 4KB)이다. 따라서 HDD도 4KB로 동작한다

DRAM의 경우도 블록디바이스이다.

순서가 필요없다. ex. Call, jmp

- **C**haracter Device

순서가 지켜져야 한다

ex. 키보드, 모니터, vga(비디오 드라이버)