Part 02. 리눅스 시스템 프로그래밍

# Chapter 05. 표준 I/O 라이브러리(2)

## 진행 순서

< 탐색 >
Chapter 05_01 fseek
Chapter 05_02 rewind
Chapter 05_03 ftell
Chapter 05_04 fgetpos
Chapter 05_05 fsetpos
< 비우기 >
Chapter 05_06 fflush
< 에러 >
Chapter 05_07 ferror, feof, clearerr
< 파일 디스크립터 얻기 >
Chapter 05_08 fileno
<파일락 >
Chapter 05_09 flockfile, funlockfile
Chapter 05_10 실습1
Chapter 05_11 실습2

# Chapter 05_01 fseek

```
#include <stdio.h>

int fseek(FILE *stream, long offset, int whence);
```

시스템 콜 lseek과 동일한 기능을 하는 함수.

whence
SEEK_SET – 파일 위치를 offset 값으로 설정한다.
SEEK_CUR – 파일 위치를 현재 위치에서 offset만큼 더한 값으로 설정한다.
SEEK_END – 파일 위치를 끝에서 offset만큼 더한 값으로 설정한다.

성공 시 0을 리턴, 에러 시 -1 (errno) 리턴

# Chapter 05_02 rewind

```
#include <stdio.h>

void rewind (FILE *stream);
```

스트림을 시작 위치로 되돌리며 다음과 동일하다.

```
fseek(stream, 0, SEEK_SET);
```

rewind()는 fseek()과 달리 오류 지시자를 초기화한다.

```
errno = 0;
rewind(stream);
if (errno)
        /* Error */
```

# Chapter 05_03 ftell

```
#include <stdio.h>

long ftell (FILE *stream);
```

lseek()과 달리, fseek()은 갱신된 위치를 리턴하지 않는다.
위치를 파악하기 위한 용도로 ftell 함수는 현재 스트림 위치를 리턴한다.

에러가 발생하면 -1을 리턴하고, errno 적절한 값으로 설정한다.

# Chapter 05_04 fgetpos

```
#include <stdio.h>

int fgetpos (FILE *stream, fpos_t *pos);
```

fgetpos() 함수는 이식성을 위한 파일 위치 함수입니다.

성공하면 0을 리턴하고 현재 스트림 위치를 pos에 기록한다.
실패하면 -1을 리턴하고 errno를 적절히 설정한다.

fpos_t 자료 구조
fgetpos 및 fsetpos 함수에서 사용하기 위해 스트림의 파일 위치에 대한 정보를 인코딩 할 수있는 오브젝트 유형입니다.
GNU C 라이브러리에서 파일 오프셋 및 변환 상태 정보를 나타내는 내부 데이터를 포함하는 불투명 한 데이터 구조입니다.
다른 시스템에서는 내부 표현이 다를 수 있습니다.
따라서 파일 위치에 대해 고유 한 인코딩이있는 시스템을 지원하려면 fgetpos 및 fsetpos 함수를 사용하는 것이 좋습니다.
이 함수는 fpos_t 데이터 유형을 사용하여 파일 위치를 나타내며, 내부 표현은 시스템마다 다릅니다.

# Chapter 05_05 fsetpos

```
#include <stdio.h>

int fsetpos (FILE *stream, fpos_t *pos);
```

fsetpos() 함수는 이식성을 위한 파일 위치 함수입니다.

stream의 위치를 pos로 설정한다.
이는 whence가 SEEK_SET인 fseek()과 동일하게 동작한다.

성공하면 0을 반환하고, 실패하면 -1을 반환하고 errno을 적절한 값으로 설정한다.

fgetpos / fsetpos 함수는 스트림 위치를 표현하기위해 복잡한 타입을 사용하는(long 타입만으로 표현이 불가한)
비-유닉스 계열의 플랫폼을 위해 제공되는 함수이다. (이식성: 플랫폼에 따라 fpos_t 자료구조 이용)

# Chapter 05_06 fflush

```
#include <stdio.h>

int fflush (FILE *stream);
```

사용자 버퍼를 커널로 비워서 스트림에 쓴 모든 데이터가 write()를 통해 실제로 디스크에 기록되도록 만든다.
stream에 있는 쓰지 않은 데이터를 커널로 비운다.
stream이 NULL이면 프로세스의 모든 열려있는 입력 스트림이 비워진다.

성공 시 0 리턴, 실패 시 EOF 반환 (errno)

표준 IO 라이브러리 함수들은 C 라이브러리가 관리하는 버퍼를 사용하며 커널 영역이 아니라 사용자 영역에 위치한다.
프로그램은 사용자 영역에 존재하면서 시스템 콜을 사용하지 않고 사용자 코드를 실행한다.
시스템 콜은 디스크나 다른 매체에 접근할 필요가 있을 때만 호출된다. (버퍼링 – 성능 향상)

fflush()는 단지 사용자 버퍼에 있는 데이터를 커널 버퍼로 쓰기만 한다.
이는 사용자 버퍼를 사용하지 않고 write()를 직접 사용하는 효과와 동일하다.
fflush()가 데이터를 매체에 물리적으로 기록한다는 보장은 없다.
이렇게 하려면 동기식 입출력 함수 fsync() 같은 함수를 사용해야 한다.
fflush()를 통해 사용자 버퍼를 커널에 쓰고, fsync() 함수를 통해 커널 버퍼를 디스크에 기록하도록 보장한다.

# Chapter 05_07 ferror, feof, clearerr

```
#include <stdio.h>

int ferror (FILE *stream);
int feof (FILE *stream);
void clearerr (FILE *stream);
```

fread()나 이와 유사한 함수를 사용할 때 에러가 밸생했거나 EOF에 도달했는지 판단하기 위해 해당 스트림의 상태를 확인하기 위해 제공되는 함수
ferror()는 스트림에 에러 지시자가 설정되었는지 검사한다.
해당 스트림에 에러 지시자가 설정되어 있을 경우 0이 아닌값을 리턴하며, 그렇지 않은 경우 0을 리턴한다.

feof()는 해당 스트림에 EOF 지시자가 설정되어 있는지 검사한다.
EOF 지시자는 파일 끝에 도달하면 표준 입출력 인터페이스에서 설정한다.
해당 스트림에 EOF 지시자가 설정되어 있을 경우 0이 아닌 값을 리턴하며, 그렇지 않은 경우 0을 리턴한다.

clearerr() 함수는 스트림에서 에러 지시자와 EOF 지시자를 초기화한다.

# Chapter 05_08 fileno

```
#include <stdio.h>

int fileno (FILE *stream);
```

스트림에서 파일 디스크립터를 구하려면 fileno()을 이용한다.
만약 표준 I/O 라이브러리를 이용해 개발 도중 시스템 콜과 대응하는 표준 I/O 함수가 없을 경우,
파일 디스크립터를 얻어 시스템 콜을 수행할 수 있다면 유용할 것이다.

표준 I/O 함수와 시스템 콜을 섞어서 사용하는 것은 권장하지 않는다.

# Chapter 05_09 flockfile, funlockfile

```
#include <stdio.h>

void flockfile (FILE *stream);
void funlockfile (FILE *stream);
```

flockfile() 함수는 stream의 락이 해제될 때까지 기다린 다음 락 카운터를 올리고 락을 얻은 다음 스레드가 stream을 소유하도록 만든 후에 반환한다.

funlockfile() 함수는 stream과 연관된 락 카운터를 하나 줄인다.
만일 락 카운터가 0이 되면 현재 스레드는 stream의 소유권을 포기해서 다른 스레드가 락을 얻을 수 있도록 한다.

# Chapter 05_10 실습1

```
#include <stdio.h>
#include <unistd.h>

int main()
{
        FILE *fp;
        long pos;
        int i;

        fp = fopen("./test.file", "w+");
        if (fp == NULL) {
                perror("fopen error: ");
                return -1;
        }

        fprintf(fp, "Hello World!\n");

        /* ftell, rewind test */
        pos = ftell(fp);
        if (pos == -1) {
                perror("ftell error: ");
                return -1;
        }
        printf("current pos is %ld\n", pos);

        printf("move pos to start\n");
        rewind(fp);
        pos = ftell(fp);
        if (pos == -1) {
                perror("ftell error: ");
                return -1;
        }
        printf("current pos is %ld\n", pos);

        fclose(fp);

        /* fflush test */
        for (i = 0; i < 5; i++) {
                fprintf(stdout, ".");
                sleep(1);
        }
        printf("\n");

        for (i = 0; i < 5; i++) {
                fprintf(stdout, ".");
                fflush(stdout);
                sleep(1);
        }
        printf("\n");

        return 0;
}
```

# Chapter 05_10 실습1

```
[root@localhost ch12]# gcc -g stdio_example2.c -o stdio_example2
[root@localhost ch12]# ./stdio_example2
current pos is 13
move pos to start
current pos is 0
.....
.....
[root@localhost ch12]# ls
stdio_example2  stdio_example2.c  test.file
```

1초 간격으로 for문을 돌면서 콤마를 찍는 첫 번째 루프는 실제로 5초 이후에 “.....” 5개가 한번에 출력되며, fflush() 이용한 두 번째 루프는 1초 간격으로 콤마를 하나씩 찍는것을 확인할 수 있다.

# Chapter 05_11 실습2

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *mod_file_thread(void *fpp)
{
	FILE *fp = (FILE *)fpp;
	char buf[1024];
	printf("<thread start>\n");

	flockfile(fp);
	printf("lock!, data file contents:\n");
	while (fgets(buf, sizeof(buf), fp)) {
		printf("%s", buf);
	}
	printf("now add data \"Hello world!\\n\"\n");
	sprintf(buf, "Hello world!\n");
	if (fputs(buf, fp) == EOF) {
		perror("fputs error: ");
		pthread_exit(NULL);
	}
	sleep(3);
	printf("finish, unlock!\n");
	funlockfile(fp);

	pthread_exit(NULL);
}
```

```c
int main(int argc, char *argv[])
{
	FILE *fp;
	pthread_t th1, th2;

	if (argc < 2) {
		printf("Usage: %s <data-file>\n", argv[0]);
		return -1;
	}

	fp = fopen(argv[1], "r+");
	if (!fp) {
		perror("fopen error: ");
		return -1;
	}

	pthread_create(&th1, NULL, mod_file_thread, (void *)fp);
	pthread_create(&th2, NULL, mod_file_thread, (void *)fp);

	pthread_join(th1, NULL);
	pthread_join(th2, NULL);

	return 0;
}
```

# Chapter 05_11 실습2

```
[root@localhost ch12]# gcc -g flock_example.c -o flock_example -lpthread

[root@localhost ch12]# cat test.file
Hello World!

[root@localhost ch12]# ./flock_example test.file
<thread start>
lock!, data file contents:
Hello World!
now add data "Hello world!\n"
<thread start>
finish, unlock!
lock!, data file contents:
now add data "Hello world!\n"
finish, unlock!

[root@localhost ch12]# cat test.file
Hello World!
Hello world!
Hello world!
```

동일한 파일 포인터를 참조하는 동일한 역할을 하는 스레드 2개가 동시에 실행되지만 파일락으로 인하여 첫 번째 스레드가 unlock이 된 이후에야 두 번째 스레드가 lock을 잡고 작업을 수행한다.