운영체제 과제 3

Producer-Consumer 개발 과제

이홍인

건국대학교, 소프트웨어융합학부, 소프트웨어학과

201611225

1 코드 출력

1.1 fork로 child 프로세스를 생성하여 수행시키는 프로그램

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#define COUNT 3
#define BUFFERSIZE 30 // 총 30번 데이터를 주고 받는다

void producer(FILE *pipe_write_end, FILE *index_read_end)
{
	int i, in=0;
	for (i = 1; i<=BUFFERSIZE; i++)
	{
		printf("pro : %d\n", i);
		fprintf(pipe_write_end, "%d ", i); // %d 뒤에 공백이 필요하다
		fflush(pipe_write_end); //플러시를 해줘야 데이터를 보낸다.
		while(1){ // 소비자에게서 데이터를 받을 때까지 무한 루프
			int n = fscanf(index_read_end, "%d", &in);
			if(n == 1)
				break;
		}

	}
	fclose(pipe_write_end);
```

```
	fclose(index_read_end);
	exit(0);
}

void consumer(FILE *pipe_read_end, FILE *index_write_end)
{
	int n, k, i, out=1;
	while(1)
	{
		int n = fscanf(pipe_read_end, "%d", &k);
		if(n == 1){ // 생산자의 데이터를 받을 때까지 무한루프이다
			printf("consumer got %d\n", k);
			fprintf(index_write_end, "%d ", out++);
			fflush(index_write_end); //데이터를 받으면 다시 돌려보내준다.
		}
		else{
			break;
			}
	}
	fclose(pipe_read_end);
	fclose(index_write_end);
	exit(0);
}
```

```
int main()
{
		pid_t producer_id, consumer_id;
		int pd[2], pd2[2]; // 생산자의 생산정보를 보내고, 소비자의 소비정보를 돌려보내기 위해 pipe 2개를 이용한다.
		FILE *pipe_write_end, *pipe_read_end, *index_read_end, *index_write_end;
		int in=0, out=0;

		if(pipe(pd) == -1)
				perror("Pipe fail.");
		if(pipe(pd2) == -1)
				perror("Pipe2 fail.");

		pipe_read_end = fdopen(pd[0], "r"); //생산자가 생산정보를 보내는 채널 pipe
		pipe_write_end = fdopen(pd[1], "w");
		index_read_end = fdopen(pd2[0], "r"); //소비자가 소비정보를 보내는 채널 index
		index_write_end = fdopen(pd2[1], "w");

		producer_id = fork(); // 프로세스에서 자식을 두 개 만들어 하나는 생산자로 이용한다.
		if(producer_id == 0)
		{
				fclose(pipe_read_end);
				fclose(index_write_end);
				producer(pipe_write_end, index_read_end);
		}
```

```
        consumer_id = fork(); // 다른 자식은 소비자로 이용한다.
        if(consumer_id == 0)
        {
                fclose(pipe_write_end);
                fclose(index_read_end);
                consumer(pipe_read_end, index_write_end);
        }

        fclose(pipe_read_end);
        fclose(pipe_read_end);
        wait(NULL); //생산자와 소비자가 종료하길 기다린다.
        wait(NULL);

        return 0;
}
```

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#define COUNT 3
#define BUFFERSIZE 30

void producer(FILE *pipe_write_end, FILE *index_read_end)
{
        int i, in=0;
        for (i = 1; i<=BUFFERSIZE; i++)
        {
                printf("pro : %d\n", i);
                fprintf(pipe_write_end, "%d ", i);
                fflush(pipe_write_end);
                while(1){
                        int n = fscanf(index_read_end, "%d", &in);
                        if(n == 1)
                            break;
                }

        }
        fclose(pipe_write_end);
        fclose(index_read_end);
        exit(0);
}

void consumer(FILE *pipe_read_end, FILE *index_write_end)
{
        int n, k, i, out=1;
        while(1)
        {
                int n = fscanf(pipe_read_end, "%d", &k);
                if(n == 1){
                        printf("consumer got %d\n", k);
                        fprintf(index_write_end, "%d ", out++);
                        fflush(index_write_end);
                }
                else{
                        break;
                }
        }
        fclose(pipe_read_end);
        fclose(index_write_end);
        exit(0);
}



int main()
{
        pid_t producer_id, consumer_id;
        int pd[2], pd2[2];
        FILE *pipe_write_end, *pipe_read_end, *index_read_end, *index_write_end;
        int in=0, out=0;

        if(pipe(pd) == -1)
                perror("Pipe fail.");
        if(pipe(pd2) == -1)
                perror("Pipe2 fail.");

        pipe_read_end = fdopen(pd[0], "r");
        pipe_write_end = fdopen(pd[1], "w");
        index_read_end = fdopen(pd2[0], "r");
        index_write_end = fdopen(pd2[1], "w");

        producer_id = fork();
        if(producer_id == 0)
        {
                fclose(pipe_read_end);
                fclose(index_write_end);
                producer(pipe_write_end, index_read_end);
        }

        consumer_id = fork();
        if(consumer_id == 0)
        {
                fclose(pipe_write_end);
                fclose(index_read_end);
                consumer(pipe_read_end, index_write_end);
        }

```

두 개의 파이프를 생성하고, fork()를 이용해 두 자식 프로세스를 생성한다. 자식 프로세스 중 생산자는 첫번째 파이프를 통하여 소비자에게 생산한 데이터를 전한다. 소비자 프로세스는 두번째 파이프를 이용하여, 생산자 프로세스가 생산 할 때까지 무한루프에서 파일입력이 성공하길 기다리다가, 생산자가 생산하면 이를 읽어 출력하고 다음 생산자에게 신호를 전달한다. 이를 기다리던 생산자는 다음 생산에 진입하고 모든 과정은 마지막 생산까지 반복된다. 이 때 생산과 소비가 반드시 번갈아가며 진행되어야 하는 구조이다.

1.2 pthread로 쓰레드를 생성하여 수행시키는 프로그램

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

#define NUM_THREAD 2

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* producer(void *pData);
void* consumer(void *pData);
int buffer[100]; // 생산자 소비자 사이 데이터 버퍼, 소비자가 소비 할 때 혹은 생산자가 생산할 때 락을 이용해 버퍼를 잠근다.
int count = 0, in = -1, out = -1;  // 생산자는 10이상

int main(void)
{
        int num = 0;
        int rc =0, i = 0;
```

```
	int result[NUM_THREAD];
	pthread_t threads[NUM_THREAD];
	pthread_create(&threads[0], NULL, producer, NULL); // 전역변수를 공유한다.
	pthread_create(&threads[1], NULL, consumer, NULL);
	for (i=0; i<NUM_THREAD; i++)
		rc = pthread_join(threads[i], NULL);
		if(rc!=0) {
			printf("Error in thread[%d] : %d\n", i, rc);
			exit(1);
		}
	return 0;
}
void* producer(void *pData)
{
	int i;
	for (i=0; i<30; i++)
	{
		while(count == 30);
		pthread_mutex_lock(&mutex); // 크리티컬 섹션의 보호를 위해 락을 얻는다.
		in++;
		in %= 10; ; //버퍼를 참조하는 인덱스를 설정한다. 순환형 버퍼.
		buffer[in] = i
		count++;
		printf("Pro : %d\n", (buffer[in]+1));
		sleep(0.1);
		pthread_mutex_unlock(&mutex);
```

```
        }
}
void* consumer(void *pData)
{
        int index, i;
        for (i=0; i<30; i++)
        {
                while(count == 0);
                pthread_mutex_lock(&mutex);
                out++;
                out %= 10;
                index = buffer[out];
                count--;
                printf("Consumer : %d\n", (index+1));
                sleep(0.1);
                pthread_mutex_unlock(&mutex);
        }
}
```

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

#define NUM_THREAD 2
#define SIZE 10
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void* producer(void *pData);
void* consumer(void *pData);
int buffer[100];
int count = 0, in = -1, out = -1;

int main(void)
{
    int num = 0;
    int rc =0, i = 0;
    int result[NUM_THREAD];
    pthread_t threads[NUM_THREAD];
    pthread_create(&threads[0], NULL, producer, NULL);
    pthread_create(&threads[1], NULL, consumer, NULL);
    for (i=0; i<NUM_THREAD; i++)
        rc = pthread_join(threads[i], NULL);
        if(rc!=0) {
            printf("Error in thread[%d] : %d\n", i, rc);
            exit(1);
        }
    return 0;
}
void* producer(void *pData)
{
    int i;
    for (i=0; i<30; i++)
    {
        while(count == 10);
        pthread_mutex_lock(&mutex);
        in++;
        in %= 10;
        buffer[in] = i;
        count++;
        printf("Pro : %d\n", (buffer[in]+1));
        sleep(0.1);
        pthread_mutex_unlock(&mutex);
    }
}
void* consumer(void *pData)
{
    int index, i;
    for (i=0; i<30; i++)
    {
        while(count == 0);
        pthread_mutex_lock(&mutex);
        out++;
        out %= 10;
        index = buffer[out];
        count--;
        printf("Consumer : %d\n", (index+1));
        sleep(0.1);
        pthread_mutex_unlock(&mutex);
    }
}
```

p_thread()를 이용한 두 개의 스레드를 생성한다. 두 스레드가 공유할 데이터 영역에 생산-소비 전달을 담당하는 공유 버퍼와, 그 버퍼 내부에 생산된 개수에 대한 카운터 변수를 뮤텍스를 이용해 보호한다. 두 스레드 중 생산자는 스레드 내부에서 총 30번까지 째의 값을 생산해 내며, 그 과정에서 버퍼에 10번이상의

생산이 존재할 경우 소비자의 소비를 기다린다. 이와 마찬가지로 소비자는 30번 진행하며 0번 이하의 소비에 이러를 시 더이상 소비하지 않고 생산자의 생산을 기다린다.

2 프로그램 수행 장면 캡쳐

1.1 fork로 child 프로세스를 생성하여 수행시키는 프로그램 (왼쪽)

1.2 pthread로 쓰레드를 생성하여 수행시키는 프로그램 (오른쪽)

```
theorist@thesktop:~/c$ ./oscs
pro : 1
consumer got 1
pro : 2
consumer got 2
pro : 3
consumer got 3
pro : 4
consumer got 4
pro : 5
consumer got 5
pro : 6
consumer got 6
pro : 7
consumer got 7
pro : 8
consumer got 8
pro : 9
consumer got 9
pro : 10
consumer got 10
pro : 11
consumer got 11
pro : 12
consumer got 12
pro : 13
consumer got 13
pro : 14
consumer got 14
pro : 15
consumer got 15
pro : 16
consumer got 16
pro : 17
consumer got 17
pro : 18
consumer got 18
pro : 19
consumer got 19
pro : 20
consumer got 20
pro : 21
consumer got 21
pro : 22
consumer got 22
pro : 23
consumer got 23
pro : 24
consumer got 24
pro : 25
consumer got 25
pro : 26
consumer got 26
pro : 27
consumer got 27
pro : 28
consumer got 28
pro : 29
consumer got 29
pro : 30
consumer got 30
```

```
theorist@thesktop:~/c$ ./prosumer
Pro : 1
Pro : 2
Pro : 3
Pro : 4
Pro : 5
Pro : 6
Pro : 7
Pro : 8
Pro : 9
Pro : 10
Consumer : 1
Consumer : 2
Consumer : 3
Consumer : 4
Consumer : 5
Consumer : 6
Consumer : 7
Consumer : 8
Consumer : 9
Consumer : 10
Pro : 11
Pro : 12
Pro : 13
Pro : 14
Pro : 15
Pro : 16
Pro : 17
Pro : 18
Pro : 19
Pro : 20
Consumer : 11
Consumer : 12
Consumer : 13
Consumer : 14
Consumer : 15
Consumer : 16
Consumer : 17
Consumer : 18
Consumer : 19
Consumer : 20
Pro : 21
Pro : 22
Pro : 23
Pro : 24
Pro : 25
Pro : 26
Pro : 27
Pro : 28
Pro : 29
Pro : 30
Consumer : 21
Consumer : 22
Consumer : 23
Consumer : 24
Consumer : 25
Consumer : 26
Consumer : 27
Consumer : 28
Consumer : 29
Consumer : 30
```

3 요점정리

1.1 fork로 child 프로세스를 생성하여 수행시키는 프로그램

1. fork에 대하여

fork 함수를 통해 메인스레드에서 자식을 두 개를 생성해 파이프를 통해 두 자식 프로세스 간에 통신을 하는 방식을 알게 되었다. 하나의 자식을 생성해 부모와 자식 프로세스의 행동을 달리하는 구조보다 두 개의 자식을 생성하는 것이 코드 이해가 더 직관적으로 느껴졌다.

2. pipe에 대하여

pipe를 포함한 응용이 처음이었다. 파이프를 이용하기 위해서는 길이 2인 정수배열을 인자로 전달해야 한다. 파이프는 단방향이며, 따라서 배열의 첫 요소가 파이프의 읽는 종단, 두번째 요소가 파이프의 쓰는 종단이다. 파이프는 생성에 실패시 5가지의 에러를 제공하며 반환값이 -1이다. 0이어야만 성공이다. 성공적으로 반환된 파이프는 특수한 형태의 파일이므로 이에 대한 파일객체를 확보하는 것이 가능했으며, 따라서 그에 대하여 read/write/fgets/fputs/fscanf/fprintf 등의 다양한 파일함수가 지원되었다.

3.Acknowledge 방식의 producer consumer 해결 방식

네트워크를 공부하며 stop-and-wait 방식의 메시지 전송 - 메시지에 대한 응답 방식으로 파이프 IPC 설계하는 방법을 착안해냈다. 그다지 효율적인 방식이 아닌 것도 알게 되었다. 메시지 전송에 통한 IPC는 대체적으로 메모리 공유에 의한 IPC보다 비효율적일 거라는 짐작을 하게 되었다.

1.2 pthread로 쓰레드를 생성하여 수행시키는 프로그램

1. pthread에 대하여

pthread는 POSIX 표준 스레드 API이다. pthread_create, pthread_self, pthread_join, pthread_detach, pthread_exit, pthread_kill등 다양한 스레드 관련 함수들과 관련되어 있다. 경량 프로세스라고도 불리는

스레드 개념을 반영해서 함수형태로 실행되며, 모든 스레드에서 공동데이터영역, 즉 전역변수에 대한 접근이 가능했다.

스레드를 생성하는 pthread_create 문법은, 스레드번호, 스레드 특성, 스레드가 실행할 함수, 그 함수에 전달할 인자로 구성되어있으며, 성공시 0을 리턴한다. 스레드의 자원을 회수할 때 까지 기다리기 위해 pthread_join이 이용되며, 기다릴 스레드번호, 그 스레드 종료시 종료코드를 전달해야한다.

2. mutex와 접근제어에 대하여

POSIX 스레드에서 임계영역의 공유자원에 접근을 동기화 하기 위해서 오직 하나의 스레드만 접근할 수 있도록 하였다. 자바의 lock과 monitor가 있다면 pthread에는 mutex, 세마포어가 있다. (이쪽이 원조이겠지만.) 오직 뮤텍스를 얻은 스레드만 해당 구역의 코드를 수행하고, 나머지 코드는 해당 뮤텍스를 얻기 위해 대기하며 경쟁하게되는 구조가 비슷하다. pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock등의 함수 사용방식은 유사하나 fast, recursive, error checking 뮤텍스 타입이 존재하는 차이점이 있다. 정적 뮤텍스를 생성하기 위한 매크로 PTHREAD_MUTEX_INITIALIZER도 활용을 이용하면 보다 성능이 좋다고 한다..

3.producer - consumer 모델에 대하여

스레딩 스케줄링의 기본 문제인 생산자-소비자 문제를 해결하는 수준이 다양함을 알게되었다. 이전에 생산자-소비자 모델이란 단지 생산자와 소비자의 다른 속도를 적절히 억제하여 충돌을 방지하면 충분하다고 생각했다. 하지만 운영체제 차원에서는 단지 소비자가 없는 생산자가 생산하지 않았는데 소비하거나하는 수준의 충돌을 방지하는 해결도 있지만, 생산자가 최대한 생산할 수 있는 개수를 설정하거나, 더 나아가 실행시간을 더욱 앞당기거나 반응성을 높히는 해결도 존재한다. 이번에 생산자가 최대 10개의 pending 생산을 수행하게 하였는데 이를 통해 1개의 생산만 하는 것보다 효율을 높일 수 있고, 30개를 한번에 생산하고 30개를 소비하는 것보다는 병렬성을 높일수 있었다.