TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

강사 : Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 : 황수정
sue100012@naver.com
25 일차 (2018. 03. 28)

목차

학습 내용 복습

- int score [0]
- Semaphore 와 Spinlock
- SHM (Shared Memory)

```
#include <stdio.h>
#include <stdlib.h>
typedef struct              // 이 코드에선 사용하지 않으니까 신경쓰지 않아도 된다.
{   int score;
     char name[20];      } ST;

typedef struct              //인덱스 0 짜리 배열 선언
{   int count;
   char name [20];
   int score [0];                  } FLEX;
```

score [0] : 인덱스가 0 이나 **실제 배열처럼 사용 가능**하다. 그렇다면 이것은 무엇을 뜻하는 것일까? 이것은 주소값이다. 이 주소가 어디를 가르키고 있는지 알려면 sizeof(FLEX)로 뽑아보면 된다. 크기는 24 로 count 와 name[20] 크기이다. 포인터는 맞으며, 의미는 이 **구조체의 끝이자 새로운 시작**이 된다.

```
int main(void)
{   int i;
    //printf("%lu₩n",sizeof(FLEX));        //FLEX 의 사이즈 출력. Score[0]을 확인하기 위해 한 것.
    FLEX *p = (FLEX *)malloc(4096);      // 메모리를 4096 으로 사용하겠다. 계속 할당받는 것이 아니고 미리
     for (i=0; i < 10000; i++)                 크게 할당 받겠다는 의미이다.
       {  p -> score[i] = i+1;
          printf("score[%d] : %d₩n",i,p->score[i]);
         //p -> score[1];
         //printf("score[1] : %d₩n", p→score[1]);            }
    return 0;               }
```

score[0]을 사용하는 이유?

위의 예제와 같이 선언하면 배열 내에 인덱스를 굳이 설정하지 않아도 원하는 만큼 동적으로 값을 넣을 수 있다. 이는 메모리에 malloc 을 많이 하면 안 좋기 때문이다. 왜 안 좋은가? 할당받는 시간이랑 해제하는 시간이 길고 느려지기 때문이다. 매번 할당받기 보다는 한 번에 크게 잡고 배열처럼 사용하는 것이 (성능에) 좋다. 어디서 많이 사용할까? 서버에서 많이 사용한다. 자료구조도 이걸 사용하면 속도가 빨라진다. get node() 할 때마다 새로운 malloc 을 할당했었기 때문이다. malloc 을 미리 세팅해놓으면 매번 새로 할당받는 것이 없어진다. **커널로 집입하는 시간이 없어진다.** 한 번에 할당받고 배열로 쓰면 할당시간이 줄어들어 빨라진다. 즉, 속도가 매우 빨라진다. 그래서 사용한다. 단, 동적할당을 다른 곳에서 사용하고 있다면 메모리 사용하는 부분이 겹쳐 segmentation fault 가 발생할 수 있다. 데이터가 넘어버리면(많으면) 밀리거나 데이터가 깨질 수도 있다. 그러니 주의해서 사용해야 한다.

위의 예제를 이용해서 큐로 풀어보자!

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct{
```

```
	int data;
	int idx;		} queue;  //idx 는 link 대신 사용한 것으로 다음 인덱스를 가르킨다.

typedef struct//구조체
{	int full_num;//최대치. 값이 할당될 때마다 하나씩 줄어든다. 몇 개나 할당할 수 있는가를 알려준다.
	int free_num;	//비어있는 것의 갯수
	int total;	//총 갯수
	int cur_idx;  //현재 내가 어떤 인덱스를 가르키고 있나. while 이나 for 가  필요 없다.
	// free idx
	int free[1024]; //별도 관리 하기 위한 배열
	int total_free;	//링크 형태가 아닌 배열로 관리함. 여기서 별도 관리가 몇 개인지 알기 위함
	queue head[0]; //총 배열
} manager;

bool is_dup(int *arr, int cur_idx)		// 중복을 허용하지 않는 것
{	int i, tmp = arr[cur_idx];

	for(i = 0; i < cur_idx; i++)	// 받은 배열과 이전 인덱스들의 데이터 값을 비교한다.
		if(tmp == arr[i])
			return true;	//같으면 1 을 다르면 0 을 반환한다.

	return false;		}
void init_data(int *data, int size)		// 중복일 경우, 다시 난수 생성
{	int i;

	for(i = 0; i < size; i++)
	{
redo:
		data[i] = rand() % 100 + 1;
		if(is_dup(data, i))		// 중복일 경우, 1 을 반환해서 난수 다시 생성하고 다시 비교한다.
		{	printf("%d dup! redo rand()₩n", data[i]);
			goto redo;					}
	}
}
void print_arr(int *arr, int size)
{	int i;
	for(i = 0; i < size; i++)
		printf("arr[%d] = %d₩n", i, arr[i]);	}

void init_manager(manager *m, int alloc_size)
{	m->full_num = 0;	//왜 0 인가? 할당한 것이 없어서
	// 12: full_num, free_num, cur_idx
```

```
	// 8: data, idx
	m->free_num = (alloc_size / sizeof(int) - 12) / 8; //12 가 아니라 1029 를 빼야한다. 구조체에 답이
                                                        있다. 1029(int)로 해놓았기 때문이다
                                                          //현재 몇개가 비어있는가를 구하기 위함이다.
                                                         //8 이 아니라 2 로 나누어야 한다.
                                        //값을 할당할 수 있는 갯수가 1533 개 들어있다. 12268/8=1533
	m->total = (alloc_size / sizeof(int) - 12) / 8;     //아무것도 할당을 안 하면 free_num 과 같다.

	m->cur_idx = 0;                         }         //아무것도 할당 안 해서 0 이다.

void print_manager_info(manager *m)          //total_free 는 dequeue 를 했을 때 활성화된다.
{	int i;

	printf("m->full_num = %d₩n", m->full_num);
	printf("m->free_num = %d₩n", m->free_num);
	printf("m->total = %d₩n", m->total);
	printf("m->cur_idx = %d₩n", m->cur_idx);
	printf("m->total_free = %d₩n", m->total_free);

	for(i = 0; i < m->total_free; i++)
		printf("m->free = %d₩t", m->free[i]);

	printf("₩n");                                                }

void enqueue(manager *m, int data)
{	m->head[m->cur_idx].data = data;
	m->head[m->cur_idx++].idx = m->cur_idx;
	m->free_num--;
	m→full_num++;                                                }

void dequeue(manager *m, int data)
{	int i;

	for(i = 0; i < m->full_num; i++)
	{	if(m->head[i].data == data)
		{	m->head[i].data = 0;
			m->head[i - 1].idx = m->head[i].idx;
			m->free_num++;
			m->full_num--;
			m->free[m->total_free++] = i;          }
	}
}
```

```
void print_queue(manager *m)
{	int i = 0;
	int flag = 0;
	int tmp = i; // m->head[i].idx;

	printf("print_queue₩n");

#if 0
	for(; !(m->head[tmp].data);)
		tmp = m->head[tmp].idx;
#endif

	while(m->head[tmp].data)	// head 에 값이 있을 경우
	{	printf("data = %d, cur_idx = %d₩n", m->head[tmp].data, tmp);
		printf("idx = %d₩n", m->head[tmp].idx);

		for(; !(m->head[tmp].data);)	//중간에 값이 없으면 pass
		{	tmp = m->head[tmp].idx;
			flag = 1;				}

		if(!flag)
			tmp = m->head[tmp].idx;

		flag = 0;
	}
}

bool is_it_full(manager *m)
{	if(m->full_num < m->cur_idx)
		return true;

	return false;			}

void enqueue_with_free(manager *m, int data)
{
	/*		m->head[i].data = 0;
			m->head[i - 1].idx = m->head[i].idx;
			m->free_num++;
			m->full_num--;
			m->free[m->total_free++] = i;
	*/
```

```
	m->head[m->cur_idx - 1].idx = m->free[m->total_free - 1];	//배열 인덱스는 0 부터 시작이므로
	m->total_free--;
	m->head[m->free[m->total_free]].data = data;		//데이터 셋팅
	m->head[m->free[m->total_free]].idx = m->free[m->total_free - 1];	//다음 인덱스 설정

	if(!(m->total_free - 1 < 0))
		m->head[m->free[m->total_free]].idx = m->free[m->total_free - 1];
	else
		printf("Need more memory₩n");

	m->free_num--;
	m->full_num++;
}

int main(void)
{	int i;
	bool is_full;
	int alloc_size = 1 << 12;
	int data[10] = {0};
	int size = sizeof(data) / sizeof(int);

	srand(time(NULL));
	init_data(data, size);
	print_arr(data, size);	//큐로 넣을 데이터 생성

	manager *m = (manager *)malloc(alloc_size); // 복잡한 구조체, 4096 바이트 할당
	init_manager(m, alloc_size);
	printf("Before Enqueue₩n");
	print_manager_info(m);

	for(i = 0; i < size; i++)
		enqueue(m, data[i]);

	printf("After Enqueue₩n");
	print_queue(m);

	dequeue(m, data[1]);

	printf("After Dequeue₩n");
	print_queue(m);

	enqueue(m, 777);
```

```
	print_manager_info(m);
	print_queue(m);

	dequeue(m, data[4]);
	dequeue(m, data[5]);
	dequeue(m, data[6]);
	enqueue(m, 333);
	print_manager_info(m);
	print_queue(m);

#if 1
	// 강제로 꽉찼다 가정하고 free 공간을 활용 해보자!
	is_full = true;
#endif

	//if(is_it_full(m))
	if(is_full)
		enqueue_with_free(m, 3333);

	print_manager_info(m);
	print_queue(m);

	return 0;
}
```

Semaphore 와 Spinlock

(sem.h)

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<errno.h>

#define SEMPERM 0777

int CreateSEM(key_t semkey);
int p(int semid);
int v(int semid);
```

(sem.c)

```
#include"sem.h"
int main(void)
{       int sid;
        sid = CreateSEM(0x777);         //semaphore 의 권한
        printf("before₩n");
        p(sid);
        printf("Enter Critical Section₩n"); ;//p 를 통해 semaphore 의 값을 1 증가시킨 후 출력
        getchar();                      // 잠깐대기하기위해문자하나받음
        v(sid);
        printf("after₩n");

        return 0;       }
```

(semlib.c)

```
#include"sem.h"
int CreateSEM(key_t semkey)
{       int status = 0, semid;
        if(( semid = semget(semkey, 1, SEMPERM | IPC_CREAT | IPC_EXCL)) == 1)
                if(errno == EEXIST)
                        semid = semget(semkey, 1, 0);
        else
                status = semctl(semid, 0, SETVAL, 2);

        if(semid == -1 || status == -1)
                return -1;

        return semid;                   }

int p(int semid)
{       struct sembuf p_buf = {0, -1, SEM_UNDO};
        if(semop(semid, &p_buf, 1) == -1)
                return -1;
        return 0;                       }

int v(int semid)
{       struct sembuf p_buf = {0, 1, SEM_UNDO};
        if(semop(semid, &p_buf, 1) == -1)
                return -1;
        return 0;                       }
```

결과 gcc semlib.c sem.c 로 컴파일 후에 실행시킨다.

```
before
Enter Critical Section
```

After

OS 에서 lock 메커니즘

프로세스가 동작 될 때 다른 프로세스가 동작하는 프로세스에 접근하여 간섭할 수 있다면 그 프로세스는 꼬여 데이터가 엉망이 될 수 있기 때문에 한 프로세스가 동작할 때 다른 프로세스가 접근하지 못하도록 하는 것이 lock 메커니즘이다.

- Semaphore

  lock 이 풀릴 때까지 접근하지 못해서 lock 이 없는 곳을 찾아 다니며 lock 이 없는 곳이 없을 때 프로세스 대기열에 있다. '**프로세스 대기열이 있다.**' 라는 것은 wait queue 로 빠진다는 것이고 context switching 를 한다는 것이다. CPU 리소스를 잡아 먹고 있어 그 동안 다른 일을 할 수 없다. 또 이를 하려면 비용이 크다. 하드웨어 레지스터를 메모리로 옮겼다가 다시 복원해서 클락 손실이 있기 때문이다. 성능을 일부 희생하더라고 데이터를 모두 처리하겠다는 뜻이다. 그래서 단순 간단한 작업보다는 대규모에서 사용한다.

- spinlock

  cpu 점유율이 지속적으로 잡고 있다. 다른 프로세스는 접근 불가이다. Lock 이 풀릴 때까지 배회한다. polling 으로 지속적 확인을 한다고 생각하면 된다. 가볍고 빠르게 처리할 수 있는 프로세스들에게 spinlock 이 걸린다. 즉, semaphore 와 다르게 단순 간단한 곳에 사용된다. 빨리 끝내는 것이 이득일 때 사용된다.

  polling : 컴퓨터나 단말 제어 장치에서 여러 개의 단말 장치에 대하여 차례로 송신 요구의 유무를 문의하고 요구가 있을 경우 그 단말 장치에 송신을 시작하도록 명령하며, 없을 때에는 다음 단말 장치에 문의하는 전송 제어 방식.

※ semaphore 와 spinlock 의 차이점 ※

Spinlock 은 여러 프로세스에 적용할 수 없고, semaphore 는 프로세스에 여러 개 적용이 가능하다. 단순하고 간단할 땐 spinlock 이 좋고, Context switching 을 하는 것처럼 여러 프로세스를 실행해야 할 때에는 semaphore lock 형식이 좋다. semaphore 는 크고 대규모의 프로세스를 처리해야 할 때 사용된다. 같은 프로그램 안에서 두 개를 다 쓸 수 있다.

critical section(임계영역) : **여러 task 를 동시에 접근해서 정보가 꼬일 수 있는 구간**이다. 그래서 안전하게 사용하려면 Semaphored 와 spinlock 으로 lock 을 걸어서 다른정보가 진입하지 못하도록 해야한다. 쓰레드(프로세스가 독립적인 것과 다르게 쓰레드는 종속적이다. 메모리를 완전히 공유한다.)가 여러 개에 프로세스로 있으면 critical section 이 되어 lock 을 걸어야 한다.

SHM (Shared Memory)

(shm.h)

```
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include <errno.h>
typedef struct
{   char name[20];
    int score;                } SHM_t;

  int CreateSHM(long key);
  int OpenSHM(long key);
  SHM_t *GetPtrSHM(int shmid);
  int FreePtrSHM(SHM_t *shmptr);
```

(shmlib.c)

```
#include "shm.h"
int CreateSHM(long key)
{  return shmget(key, sizeof(SHM_t), IPC_CREAT |0777);  }

int OpenSHM(long key)
{   return shmget(key, sizeof(SHM_t), 0);  } //물리메모리 식별자(포인터) 반환

SHM_t *GetPtrSHM(int shmid)
{   return (SHM_t*)shmat(shmid, (char*)0, 0);    }

int FreePtrSHM(SHM_t *shmptr)
{  return shmdt((char*)shmptr);  }
```

(send.c)

```
#include "shm.h"
int main(void)
{  int mid;
    SHM_t *p; //공유하고자 하는 메모리 shared memory

  mid = OpenSHM(0x888); //메모리아이디. 페이지프레임의 아이디값을 얻음. 권한을 얻음.

  p = GetPtrSHM(mid); //진짜 쉐어드메모리의 포인터 값을 얻음. 공유메모리의 물리주소를 얻음.
  getchar();
  strcpy(p->name, "아무개");
  p->score = 93;
  FreePtrSHM(p);
  return 0;                }
```

(recv.c)

```
#include "shm.h"
int main(void)
```

```
{  int mid;
  SHM_t *p;

  mid = CreateSHM(0x888);

  p = GetPtrSHM(mid);
  getchar();
  printf("이름 : [%s], 점수 : [%d]₩n", p->name, p->score);
  FreePtrSHM(p);
  return 0;                    }
```

결과 gcc -o shmlib.c send.c > gcc –o shmlib.c recv.c 후에 터미널 두 개를 띄우고 아래와 같이 실행한다.

```
sue100012@sue100012-Z20NH-AS51B5U:~/project/3_28$ ./send

sue100012@sue100012-Z20NH-AS51B5U:~/project/3_28$ ./recv

이름 : [아무개], 점수 : [93]
```

- shared memory(공유메모리)
  한 개의 메모리를 여러 장치(프로세서)가 공동으로 사용하는 형태의 기억장치 또는 다중처리 시스템을 사용할 때 **여러 프로세서가 하나의 기억장치를 공유하여 사용하는 것**을 의미한다. 물리 메모리를 공유한다.

Send 에서는 물리 메모리에 write 를 하는 것이고 Recv 에서는 물리 메모리에서 read 를 하는 역할 을 한다. IPC 통신을 위해 SHM 은 정말 중요하다. 프로세스간 정보를 공유하기 위해서 반드시 사용 해야 한다. 이 방식으로 페이지프레임 아이디 값을 가져와 물리 메모리 주소에 직접 값을 넣을 수 있다.

- IPC
  프로세스 간의 성보 공유를 하기 위해서 사용한다. A 프로세스와 B 프로세스가 각각 존재한다고 했을 때, A 프로세스와 B 프로세스는 원래 서로 공유를 할 순 없다. 하지만, IPC 를 사용하면 shared memory 를 통하여 프로세스간에 정보들을 공유할 수 있게 된다. 개념으로 말하자면, 메모리 상에 특정 공간을 잡아 놓고 해당공간에 서로 접근할 수 있는 권한을 만든다. 그리고 그 공간에 정보들을 넣으면 공유가 되어 접근 권한을 가진 프로세스들이 메모리에 접근 가능하며 읽고 쓰고 할 수 있게 된다.