TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

2018-03-27 (25 회차)

강사: Innova Lee(이상훈)

gcccompil3r@gmail.com

학생: 정유경

ucong@naver.com

1. (C 언어 고급기술) 크기가 0 인 배열 활용하기 – adv_tech4.c

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

typedef struct
{
	int count;
	char name[20];
	int score[0]; // 크기가 0 인 배열
} FLEX;

int main(void)
{

	FLEX *p =(FLEX*)malloc(4096);
/*크기가 0 인배열에 값을 할당하는 데도 정상동작!*/
	p->score[0];
	p->score[1];
	printf("%d\n", sizeof(FLEX)); 24 바이트
	return 0;
}
```

*. 크기가 0 인 배열 선언하기

1. 크기가 0 인 배열은 독립적으로 사용할 수 없다.

2. 공용체, 구조체, 클래스 안에서 가장 마지막 멤버변수로 선언되어야 한다.

3. 또한 크기가 0 인 배열이 선언되는 공용체, 구조체, 클래스 안에서 오직 하나여야 한다.

*. malloc 많이하면 메모리 할당/해제 시간때문에 속도가 느려진다.

예제에서 나와있듯이 한번에 할당을 크게 잡고 배열처럼 쓸 수 있다.

미리 malloc 으로 할당해놓으면 커널 진입시간이 없어져서 속도가 빨라진다

*. 서버, 자료구조에서 사용한다.

*. score 가 의미하는 것은, 구조체의 끝이자 새로운 시작이다.

*. 4096 바이트를 할당하였지만, 다른 메모리 침범하지 않을 때까지 할당은 가능하다.

그렇지만 침범하는 경우 segment fault 또는 값에 에러가 날 수 있으므로 삼가자.

2. 크기가 0 인 배열을 활용하여 Queue 구현 – Queue_ans.c

*. malloc() 한번만 할것

*. Bug 가 있으니 수정해볼 것

*. 삭제한 배열요소의 인덱스들은 별도로 관리할 것

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

typedef struct
{
```

```
	int data;
	int idx; // 링크대신 인덱스를 사용
}
queue;

typedef struct
{
	int full_num; // 최대치, 몇개나 할당가능한가(값이 할당될 때마다 -1)
	int free_num; // 비어있는 것의 갯수
	int total; // 최대갯수
	int cur_idx; // 현재 가리키고 있는 인덱스(덕분에, 반복문 돌 필요가 없다)
	int free[1024]; // 삭제한 인덱스 별도관리용
	int total_free; // 별도관리하는 인덱스들이 몇개인가
	queue head[0];
}
manager;

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

	for(i = 0; i < cur_idx; i++)
		if(tmp == arr[i])
			return true;
	return false;
}

void init_data(int *data, int size) // Queue 로 넣을 데이터 초기화
{
	int i;

	for(i = 0; i < size; i++)
	{
redo:
		data[i] = rand() % 100 + 1;

		if(is_dup(data, i))
		{
			printf("%d dup! redo rand()\n", data[i]);
			goto redo;
		}
	}
}

void print_arr(int *arr, int size) // Queue 로 넣을 데이터 출력
{
	int i;

	for(i = 0; i < size; i++)
		printf("arr[%d] = %d\n", i, arr[i]);
}

void init_manager(manager *m, int alloc_size) // for 별도관리
{
	m->full_num = 0; // 할당한게 없다
```

```
	m->free_num = (alloc_size / sizeof(int) - 1029) / 2; // 비어있는 개수
/*1029 = (manager 구조체의 크기)1024*4 바이트 + 4 바이트*5 */
/*2 = 8 / sizeof(int)*/
/*alloc_size 를 왜 변경할까?*/
/*(16384-4116)/8=1533 개 요소에 값 할당가능
total=free_num(아무것도 할당되지 않았으므로)*/
	m->total = (alloc_size / sizeof(int) - 1029) / 2;
	m->cur_idx = 0;
}

void print_manager_info(manager *m)
{
	int i;
/* 값이 잘 들어갔는지 디버깅 용*/
	printf("m->full_num = %d\n", m→full_num); // 0
	printf("m->free_num = %d\n", m→free_num); // 1533
	printf("m->total = %d\n", m→total); // 1533
	printf("m->cur_idx = %d\n", m→cur_idx); // 0
	printf("m->total_free = %d\n", m→total_free); //0 dequeue 해야 생긴다.

	for(i = 0; i < m->total_free; i++) // free 한 배열 출력 아직 값 없다.
		printf("m->free = %d\t", m->free[i]);

	printf("\n");
}

void enqueue(manager *m, int data)
{
	m->head[m->cur_idx].data = data; // 동적할당한 배열에 난수로 data 넣는다
	m->head[m->cur_idx++].idx = m→cur_idx; //cur_idx ++
	m->free_num--; //1533-1
	m->full_num++; //0+1
}

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

	for(i = 0; i < m->full_num; i++) // full_num 개수만큼 반복문 실행
	{
		if(m->head[i].data == data) // 입력한 데이터가 있는지 검색해서 있으면
		{
			m->head[i].data = 0; // data 는 0 으로
			m->head[i - 1].idx = m→head[i].idx; // idx 인덱스를 보낸다 (안쓰니까
-1)
			m->free_num++; // 해제했으므로 증가
			m->full_num--; // 해제했으니까 감소
			m->free[m->total_free++] = i; // total_free 가 하나 증가하고
						// free 한 idx 를 별도로 저장하여 관리
		}
	}
}
```

```
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 데이터 있다면
	{
		printf("data = %d, cur_idx = %d\n", m->head[tmp].data, tmp);
		printf("idx = %d\n", m→head[tmp].idx); // 현재 idx 확인

		for(; !(m->head[tmp].data);)  // 중간에 삭제하면 data=0 이므로 제끼고 idx 를 증가
		{
			tmp = m→head[tmp].idx; // tmp 갱신
			flag = 1;
		}

		if(!flag)
			tmp = m→head[tmp].idx;// idx 갱신

		flag = 0; // flag 풀어준다 1→ 0
	}
}

// 입력과 삭제를 반복하면 중간부분이 많이 비어있을 것이다. 이를 활용하자
bool is_it_full(manager *m)
{
	if(m->full_num  <  m->cur_idx) // 할당한 개수 < 현재 인덱스 ???
		return true;	// 꽉 차있을 경우 true 반환

	return false; // 아니면 false 반환
}

/*별도관리하여  enqueue 한다*/
void enqueue_with_free(manager *m, int data)
{
	m->head[m->cur_idx - 1].idx = m->free[m->total_free – 1];
// totalfree 의 …?
	m->total_free--; // 비어있는 공간을 하나 썼으므로 1 감소
	m->head[m->free[m->total_free]].data = data; // data 넣는다
	m->head[m->free[m->total_free]].idx = m->free[m->total_free - 1];
//다음에 넣을 idx 를 free -1 하여 지정한다.
	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; // 2 의 12 승 4096 바이트
	int data[10] = {0};
	int size = sizeof(data) / sizeof(int);

	srand(time(NULL));
	init_data(data, size);
	print_arr(data, size);

/*manager 에 4096 바이트 할당*/
	manager *m = (manager *)malloc(alloc_size);
	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;
}
```

3. 세마포어

실행방법:gcc semlib.c sem.c → ./a.out

3-1 sem.c

```
#include "sem.h"

int main()
{
	int sid; // 세마포어 아이디
	sid=CreateSEM(0x777); // 세마포어를 생성하고
 // 셈키 777 로 세마포어 락을 걸겠다는 뜻 //셈키로 777 날아오지 않으면 세마포어 락을 풀수
없다.
	printf("before\n");
	p(sid);
	printf("Enter Critical Section\n");
	getchar();
	v(sid);
	printf("after\n");
	return 0;
}
```

3-2 semlib.c

```
#include"sem.h"
/*여러프로세스가 동시에 접근할 때 차단을 원하는 자원에 대해 세마포어를 생성한다.*/

int CreateSEM(key_t semkey){

	int status = 0, semid;
/*semid 를 생성(세마포어에 대한 채널 얻기)한다. 실패하면*/
/*IPC_CREAT | IPC_EXCL 옵션에서 키가 이미 있으므로 errno = EEXIST*/
/*IPC_CREAT(IPC 생성)만 쓰면 키에 해당하는 채널이 없는 경우 채널을 새로 생성, 채널이
존재하는 경우 기존 채널의 식별자를 반환.*/
/*각각의 채널에 접근권한을 설정할 수 있다 (SEMPERM =0777 )*/
	if(( semid = semget(semkey, 1, SEMPERM | IPC_CREAT | IPC_EXCL)) == -1)
		if(errno == EEXIST)
			semid = semget(semkey, 1, 0);
/*옵션이 0 이므로 키가 있는경우 기존채널의 ID 를 반환, 이전에 키가 없는 경우 errno=ENOENT*/
/*따라서 semkey 를 가지고 현재 세마포어의 semid 를 가지고 온다*/

	else/*채널 생성에 성공했다면*/
		status = semctl(semid, 0, SETVAL, 2);
/*세마포어 값을  2 로 설정한다. 실패하면 status 로 -1 성공하면 양수가 리턴된다*/
/*int semctl(int semid, int semnum, SETVAL, int val) */
/*semmid 로 식별되는 세마포어 set 의 semnum 번째의 배열의 값 semval 을 2 로 설정한다.*/

/*채널생성에 실패하였거나, 세마포어값 설정에 실패할 경우*/
	if(semid == -1 || status == -1)
```

```
            return -1; // -1 을 반환
      return semid;
/* 채널생성에 성공하였거나 세마포어 값을 2 로 설정하는데  성공하였다면 semID 반환*/
}


/* 아래의 p,v 연산은 패턴이므로 암기해두자!*/
/* 동작과정: p 동작 -> 자원사용 -> v 동작  */

/*세마포어의 p(감소)연산*/
int p(int semid)
{

      struct sembuf p_buf = {0, -1, SEM_UNDO};
// p_buf.sem_num , p_buf.sem_op, p_buf.sem_flg
// SEM_UNDO: 프로세스 종료시, 세마포어 설정을 원래상태로 초기화
      if(semop(semid, &p_buf, 1) == -1) // 세마포어의 값을 1 감소시키는 연산을 수행
            return -1;
      return 0;
}

/*세마포어의 v(증가) 연산*/
int v(int semid)
{
      struct sembuf p_buf = {0, 1, SEM_UNDO};
      if(semop(semid, &p_buf, 1) == -1) // 세마포어의 값을 1 증가시키는 연산을 수행
            return -1;
      return 0;
}
```

3-3 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 CraeteSEM(key_t semkey);
int p(int semid);
int v(int semid);
```

*. lock 메커니즘 2 가지 - 세마포어, 스핀락

1. 세마포어 : context switching 을 한다. Wait queue 로 빠져서 기다린다. **프로세스 대기열**이 존재한다.

대규모, 프로세스 여러개에 적용가능. 들어오는게 많을 때 성능에 부하는 걸리겠지만 데이터를 안정적으로 처리하고 싶을 때 사용한다

2. 스핀락: 스핀이란 cpu 의 점유율을 의미한다. cpu 를 지속적으로 잡고 **polling** 방식으로 계속 확인한다. 락을 걸고 돌릴 코드가 소규모 단순 간단할 때, 프로세스 여러개에 적용 불가.
굳이 context switcihng 하지 않고 (레지스터에서 메모리로 올렸다 내리는 과정을 반복하면서 클럭손실이 있다. cpu 리소스를 잡아 먹으면서 뒤의 일을 처리하지 못한다) 빨리 끝내버리겠다는 의미이다.

*. critical section(임계영역)
여러 task 들이 동시에 접근하여 정보가 꼬일 수 있는 공간을 말한다.
스레드 하나면 lock 을 걸 필요가 없지만 스레드 여러개면 꼭 lock 을 걸어주어야 한다.
(전역변수이면 critical section 일까? 아니다. 여러 스레드에 의해서 공유될 때에만 임계영역이고 이때는 lock 을 걸어주어야 한다.)
fork 는 부모자식간에 독립적인 가상메모리를 갖는다.
종속적인 가상 메모리를 갖는 경우도 있다. pthread_create()로 스레드를 만들면 메모리(힙, 데이터, 텍스트)를 공유한다. 스택의 경우 지역변수를 사용하기 위한 독립적인 공간이므로 가상메모리를 공유하지 않는다.
이때 공유하는 공간에 lock 을 걸어서 A 가 사용중일때 B 가 이 영역을 침범하지 못하도록 해야한다.

4. 공유메모리

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);  // 키값을 줘서 공유메모리 생성 shmget
int OpenSHM(long key);  // 이미 생성되어 있는 공유메모리의 shmget
SHM_t* GetPtrSHM(int shmid); // 공유메모리의 물리주소를 가리키는 포인터
int FreePtrSHM(SHM_t *shmptr);
```

*. 메모리(페이지프레임, 즉 물리메모리)를 공유한다
(물리메모리의 최소단위는 페이지프레임, 페이지 프레임을 관리하기 위한 SW 가 '페이지')

*. 공유메모리 생성하고 CreatSHM 한쪽은 쓰고 한쪽은 읽는다
→ send 는 물리메모리에 쓴다 (recv: 물리메모리에서 읽는다)
→ 그래서 IPC_CREAT 붙어있다

4-1 send.c

```
#include "shm.h"
```

```
int main()
{
        int mid; // (int shmid) 공유 메모리를 생성할 때 만들어진 공유 메모리의 ID
        SHM_t *p; // 쉐어드 메모리 타입의 포인터 // 즉 공유하고자 하는 메모리 번지
        mid = OpenSHM(0x888);  //공유 메모리를 읽기 위한 key 변수 0x888 또는 0x777 아무거나
이용가능
// shmget 의 리턴값: shmid 즉 공유된 페이지 주소, 물리메모리의 페이지 프레임을 얻는다.
        p=GetPtrSHM(mid); // 공유메모리의 물리주소를 가리키는 포인터를 생성
//즉, 물리주소를  p 에 얻어온다

        getchar(); // 대기한다.
        strcpy(p→name,"김모모");  // SHM_t 구조체로 공간 만들어서 strcpy()
        p->score = 93;
        FreePtrSHM(p); // 사용이 끝났으면 해제한다. 실제 물리메모리에 쓰여진 값이 삭제되는
것은 아니다
          return 0;
}
```

*. fork 할때에는 프로세스는 가상메모리 주소를 공유할 수 없었다.
주소값으로 다른 프로세스에 접근하면 세그먼트 폴트 났었는데 여기서는 p 로 접근이 가능하다
IPC 를 사용하였기 때문!!
따라서 프로세스간 정보공유를 하려면 IPC 또는 SHM 을 이용한다.

4-2 recv.c

```
#include "shm.h"

int main()
{
        int mid;
        SHM_t*p;
        mid = CreateSHM(0x888);
         p=GetPtrSHM(mid); // 아무나 접근할 수 없도록 위에서 얻어진 메모리 id 를 받아서
//실제 물리 메모리를 가리키는 포인터를 얻어온다 // 생성된 공유메모리를 프로세스에 연결한다

        getchar(); // 대기한다.
        printf("이름: [%s], 점수:[%d]\n", p->name, p->score);

        FreePtrSHM(p);

        return 0;
}
```

*. shmat() 함수 : 공유메모리가 할당된 주소를 찾는다! mid 이용

4-3 shmlib.c 쉐어드 메모리 라이브러리

```
#include "shm.h"
```

```
int CreateSHM(long key) // 공유메모리 생성
{// 키값 만들고,
        return shmget(key,  sizeof(SHM_t)   , IPC_CREAT | 0777);
// shmget 의 리턴값: shmid 즉 공유된 페이지 주소, 물리메모리의 페이지 프레임을 얻는다.
}

int OpenSHM(long key)
{

        return shmget(key, sizeof(SHM_t),0); // 공유메모리를 읽기위한 키값, 공유메모리의
크기
SHM_t 의 크기로 설정
}

// 생성된 공유 메모리를 프로세스에 연결한다(shmat)
SHM_t *GetPtrSHM(int shmid)
{
        return (SHM_t*) shmat(shmid,(char*)0,0);
// 식별자, 메모리가 붙을 주소(0 이므로 커널이 명시), 0 이므로 공유메모리는 읽기 쓰기
가능모드로 열림
}

int FreePtrSHM(SHM_t *shmptr)
{// 다른 프로세스에 연결된 공유메모리공간의 사용을 끝낸 후 프로세스와 공유메모리 공간의
연결을 끊는다
        return shmdt((char*)shmptr);
}
```

***. shmget** : shmget 은 커널에 **공유메모리 공간을 요청**하기 위해 호출하는 시스템 호출 함수이다. key 는 바로 위에서 설명했듯이 고유의 공유메모리임을 알려주기 위해서 사용된다. shmget 을 이용해서 새로운 공유메모리 영역을 생성하거나 기존에 만들어져있던 공유메모리 영역을 참조할수 있다성공하면 **shmid** 를 반환, 실패하면 -1 을 반환한다.

- key_t key : 첫번째 아규먼트는 여러개의 공유메모리중 **원하는 공유메모리에 접근하기 위한 Key 값**이다. 이 Key 값은 커널에 의해서 관리되며, Key 값을 통해서 선택적인 공유메모리에의 접근이 가능하다

- int size : 두번째 아규먼트는 **공유메모리 의 최소크기** 이다. 새로운 공유메모리를 생성하고자 한다면 크기를 명시해주어야 한다. **존재하는 메모리를 참조한다면 크기는 0** 으로 명시한다.

- int shmflg : 3 번째 아규먼트는 공유메모리의 접근권한과, 생성방식을 명시하기 위해서 사용한다. 아규먼트의 생성방식을 지정하기 위해서 IPC_CREAT 와 IPC_EXCL 을 사용할수 있다.

1. IPC_CREAT : 새로운 영역을 할당한다. 만약 이 값이 사용되지 않았다면, shmget()은 **key** 로 이미 생성된 접근 가능한 공유메모리 영역이 있는지 확인하고 이에 대한 식별자를 되돌려줄 것이다.
2. IPC_EXCL : IPC_CREAT 와 함께 사용하며 공유메모리 영역이 이미 존재하면 에러를 리턴한다.
3. mode_flags(하위 9bit)

접근 권한의 지정을 위해서 사용한다. 실행권한은 사용하지 않는다.
만약 공유 메모리 영역이 이미 존재한다면 접근권한은 수정된다.

***. shmat : 생성된 공유 메모리를 프로세스에 연결**

일단 공유메모리 공간을 생성했으면, 우리는 공유메모리에 접근할수 있는 int 형의 "식별자" 를 얻게 된다. 우리는 이 식별자를 shmat 를 이용해서 지금의 프로세스가 공유메모리를 사용가능하도록 "덧붙임" 작업을 해주어야 한다. 첫번째 아규먼트는 **shmget 을 이용해서 얻어낸 식별자 번호**이며, 두번째 아규먼트는 메모리가 붙을 주소를 명시하기 위해 사용하는데, **0 을 사용할경우 커널이 메모리가 붙을 주소를 명시**하게 된다. 특별한 사항이 없다면 0 을 사용하도록 한다. 세번째 아규먼트를 이용해서, 우리는 **해당 공유메모리에 대한 "읽기전용", "읽기/쓰기가능" 모드**로 열수 있는데, SHM_RDONLY 를 지정할경우 읽기 전용으로, 아무값도 지정하지 않을경우 "읽기/쓰기 가능" 모드로 열리게 된다.

shmat() 함수

```
#include <sys/types.h>
#include <sys/shm.h>
void xshmat(int shmid, const void xshmaddr, int shmflg);
```

4-4 shm.h

```
#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);
}

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

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

***. shmdt**

프로세스가 더이상 공유메모리를 사용할필요가 없을경우 **프로세스와 공유메모리를 분리** 하기 위해서 사용한다. 이 함수를 호출할 경우 단지 현재 프로세스와 공유메모리를 분리시킬뿐이지, **공유메모리 내용을 삭제하지는 않는다**는 점을 기억해야 한다. 공유메모리를 커널상에서 삭제 시키길 원한다면 shmctl 같은 함수를 이용해야 한다.