Xilinx Zynq FPGA,TI DSP,
MCU 기반의
프로그래밍 전문가 과정

날 짜 : 2018 . 3. 28

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 정한별
hanbulkr@gmail.com

<adv_tech4.c>

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

typedef struct
{
	int score;
	char name[20];
}ST;

// 위에는 쓰지 않는 구조체를 만든것이다.

typedef struct
{
	int count;
	char name[20];
	// 서버로 만든다. 정해진 크기를 먼저 할당해주고
	// 구조체의 끝이자 새로운 시작이다. 형체가 없는것, 주소는 있다.
	int score[0]; //인자가 0 인 것은 주소는 있지만 형체가 없는 상태이다.
}FLEX;

int main(void)
{
	// 할당받는 속도가 없어진다. 그래서빨라진다.
	// 원래는 할당받고 해제하는시간때매 느려지고 그런다.
	// 여러개를 한다고 하면 엄청 동적할당을 한다.
	// 그래서 10 만개 정도 돌리면 안되는데
	// 미리 정해주면 안좋은 점은 이미 있던 메모리를 칠 수도 있다.
	FLEX *p = (FLEX*)malloc(4096);
	int i = 0;
	p->score[0] = 0;
//	p->score[1] = 20;
//	printf("%d",sizeof(FLEX));


	for(i = 0 ;i <100 ;i++){
		p->score[i] = i+1;
		printf("%d\n",p->score[i]);
	}
	return 0;
}
```

<adv_queue.c>

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

typedef struct
{
	int data;
	int idx;
}
queue;

typedef struct
{

	int full_num; // enqueue 된 끝값
	int free_num; // 지울때 인덱스값
	int total; // 메모리 크기?
	int cur_idx; // 현재 인덱스 값
	// free idx
	int free[1024];
	int total_free;
	// 진짜 저장되고 돌릴 배열
	queue head[0];
}
manager;

bool is_dup(int *arr, int cur_idx)
{
	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)
{
	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)
{
        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;
        // 12: full_num, free_num, cur_idx
        // 8: data, idx
        // 할당할수 있는 개수는 1533 이다
        // alloc_size = (4096*4 / 4 -1029)/2
        m->free_num = (alloc_size / sizeof(int) - 1029) / 2;
        m->total = (alloc_size / sizeof(int) - 1029) / 2;
        m->cur_idx = 0;
}

void print_manager_info(manager *m)
{
        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)
	{
		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);)
		{
			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];
	m->total_free--;
	m->head[m->free[m->total_free]].data = data;
	m->head[m->free[m->total_free]].idx = m->free[m->total_free - 1];

	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;
	// 4096 을 할당한다.
	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);

	// 메모리를 할당한다.
	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;
}
```

<adv_queue.c> -ver.me

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


#define EMPTY 0
#define SIZE 10

int flag=0;


typedef struct __queue
{
	int data[0];
	// 형체가 없지만 주소가 있는 상태이다. 구조체의 끝이자 시작이 되는 부분
	int dflag[0];
}queue;

queue *get_node()
{
	queue *tmp;
	return tmp ;
}

int enqueue(queue **head, int data, int index)
{
//	queue *tmp = *head;
	// 기본 인덱스를 이용해서 값이 들어올 때마다 순서대로 값이 들어가도록 한다.
	(*head)->data[index] = data;
	return index+1;
}

void print(queue **head,int dindex)
{
	int i=0;
	int x=0;
	printf("\n");

	// 기본으로 크기를 정해준 사이즈에서 지워준 인덱스 크기만큼을 더해 준다.
	for(i = 0; i<SIZE+dindex; i++)
	{
		// 기본 인덱스와 지울때 저장한 곳이 같을 때는 넘어간다.
		if(i==(*head)->dflag[x]){
			x++;
```

```
                        continue;
                }
                printf("data[%d]=[%d]\n",i,(*head)->data[i]);

        }
}
// 지울때 구조체에서 지운 위치의 기본인덱스를 저장한다.
int dequeue(queue **head, int data, int dindex)
{
        int i=0;
        for(i =0; i<SIZE; i ++)
        {
                if((*head)->data[i] == data){
                        printf("\n%d 는 %d 번째 입니다. 지우겠습니다.\n",data,i+1);
                        (*head)->data[i] = 0;

                        (*head)->dflag[dindex]= i;
                        printf("dflag 의 갯수:%d\n",++dindex);
                        // 지울때의 dflag 에는 index i 의 값을 넣는다.
                }
        }
        return dindex;
}

int main(void)
{
        // 4096 이라는 크기를 미리 지정해 동적할당을  1 번 해준다.
        // 한번만 실행하게 한다.
        queue *head = (queue*)malloc(4096);
        int i=0;
        int index =0;
        int dindex = 0;
        int insert=3;

//      srand(time(NULL));
        for(i=0;i<SIZE; i++)
                index = enqueue(&head, rand()%20+1,index);
        print(&head,dindex);
        dindex = dequeue(&head,18,dindex);
        print(&head,dindex);
        printf("\n%d 를 집어 넣겠습니다.\n",insert);
        index = enqueue(&head, insert, index);
        print(&head,dindex);

        dindex = dequeue(&head,10,dindex);
        print(&head,dindex);

        return 0;
}
```

<sem.c>

```
#include "sem.h"

int main(void)
{
	int sid;
	// 키 값을 지정한다.
	sid = CreateSEM(0x777);

	printf("before\n");
	p(sid);
	printf("Enter Critical Section\n");
	getchar();
	v(sid);
	printf("after\n");
	return 0;
}
```

<semlib.c>

```
#include "sem.h"
// 키값을 받아온다.
int CreateSEM(key_t semkey)
{
	int status = 0,semid;

	// IPC = inter process communication 프로세스 여러개가 공유를 한다.
	// EXCL : 해당 키값으로 세마포어가 있으면 씹어라
	if((semid = semget(semkey, 1,SEMPERM |IPC_CREAT|IPC_EXCL)) == 1){

		if(errno == EEXIST){
			// 현재 있는 세마 포어를 가져오겠다.
			semid = semget(semkey, 1, 0);
		}
		else
			;
	}
	else{
			// SETVAL 하면 세마포어를 0 로 하겠다.
			status = semctl(semid,0,SETVAL,2);
	}
	if(semid == -1 || status == -1){
		return -1;
	}
	else
		;

	return semid;
```

```
}

int p(int semid)
{
	// 세마포어르 원래 값으로 돌려라 0 으로 돌아감.
	struct sembuf p_buf = {0,-1,SEM_UNDO};
	// 세마포어 값을 1 증가시켜라.. 윗줄과 아랫줄 합치는게 세마포어 1 증가.
	if(semop(semid,&p_buf,1)== -1)
		return -1;
	return 0;
}


int v(int semid)
{
	// 이것은 뺄셈 할때 이렇게 쓴다. 0,1 들어간거.UNDO 프로세스가 종료가될때 초기화를 시킨다.
	struct sembuf p_buf = {0,1, SEM_UNDO};
	// 세마포어에 있는 CNT 값이 -1 을 하고 리턴은 잘되면 0 이다.
	if(semop(semid, &p_buf,1) == -1)
		return -1;
	return 0;
}
```

<sem.h>

```
#ifndef __SEM_H__
#define __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);

#endif
```

setctl(int semid , int semnum, int cmd , union semun arg)

int semid　　시스템에서 세머포어를 식별하는 집합 번호

int semnum　　세마포어 집합 내에서의 세마포어 위치

Int cmd

cmd	cmd 내용
GETVAL	세마포어의 현재 값을 구한다.
GETPID	세마포어에 가장 최근에 접근했던 프로세스의 프로세스 ID 를 구한다.
GETNCNT	세마포어 값이 증가하기를 기다리는 프로세스의 개수
GETZCNT	세마포어 값이 0 이 되기를 기다리는 프로세스의 개수
GETALL	세마포어 집합의 모든 세마포어 값을 구한다.
SETVAL	세마포어 값을 설정
SETALL	세마퍼어 집합의 모든 세마포어 값을 설정
IPC_STAT	세마포어의 정보를 구한다.
IPC_SET	세마포어의 소유권과 접근 허가를 설정
IPC_RMID	세마포어 집합을 삭제

union semun arg　　CMD 에 따라 달라지며, 설정 또는 값을 구하는 변수

int semop (int semid, struct sembuf *sops, unsigned nsops);

첫번째 아규먼트는 semget() 호출에 의해 반환된 키값이다.
두번째 아규먼트(sops)는 세마퍼 집합에서 수행될 *동작* 배열을 가리키는 포인터이고
세번째 아규먼트(nsops)는 배열안에 있는 동작의 갯수이다.

int semget (key_t key, int nsems, int semflg)

key_t key 시스템에서 세머포어를 식별하는 집합 번호
int nsems 세마포어 집합 내의 세마포어 개수로 접급 제한하려는 자원의 개수

semflg	옵션 내용
IPC_CREATE	key 에 해당하는 공유 세머포어가 없다면 새로 생성한다. 만약있다면 무시하며 생성을 위해 접근 권한을 지정해 주어야 한다.
IPC_EXCL	세머포어가 이미 있다면 실패로 반환하며 세머포어에 접근하지 못한다. 이 옵션이 없어야 기존 세마포어를 사용할 수 있다.

반환
-1　　실패
-1 이외　새로 만들어진 세마포어 식별자 또는 key 와 일치하는 세마포어 식별자

<recv.c>

```
#include "shm.h"

int main(void)
{
	int mid;
	SHM_t *p;

	// 쉐어드 메모리를 만들어준다. 키를 저렇게 받아서 만듬.
	// 읽어옴.
	mid = CreateSHM(0x888);
	p= GetPtrSHM(mid);

	// 엔터
	getchar();
	// 프린트를 한다.
	printf("이름 : [%s],점수 : [%d]\n", p->name, p->score);
	// 공간을 해제 한다.
	FreePtrSHM(p);
	return 0;
}
```

<send.c>

```
#include "shm.h"

int main(void)
{

	// 메모리 아이디
	int mid;
	// 쉐어드 메모리
	SHM_t *p;

	mid = OpenSHM(0x888);
	//쉐어드 메모리의 값을 얻어와라.
	p = GetPtrSHM(mid);

	//엔터를 치면
	getchar();
	// 아무개를 얻어옴 구조체의 공간을 가져온거임.
	strcpy(p->name,"아무개");
	//
	p->score = 93;
	// 쉐어드 메모리 다 썼으니 해제 한다.
	FreePtrSHM(p);
	return 0;
}
```

<shmlib.c>

```
#include "shm.h"

int CreateSHM(long key)
{
	//	0777 의 권한을 준다.
	return shmget(key, sizeof(SHM_t), IPC_CREAT | 0777);
}
int OpenSHM(long key)
{
	return shmget(key, sizeof(SHM_t),0);
	// 나는 이포인터를 쉐어드 메모리로 지정할 것이다.
	// 그럼 공유된 페이지 주소를 얻는다. 물리주소에 있는...
}
SHM_t *GetPtrSHM(int shmid)
{
	// at 는 공유메모리의 장소라고 생각하면 된다. 공유메로리 물리주소.
	// id 값을 가지고 찾는다. 맨 첫번째 주소부터 찾겠다.
	return (SHM_t *)shmat(shmid, (char *)0,0);
}
int FreePtrSHM(SHM_t *shmptr)
{
	return shmdt((char*)shmptr);
}
```

<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);
```

<Shared Memory>

공유 메모리는 단어 뜻에서 알 수 있듯 하나의 프로세스에서가 아니라 **여러 프로세스가 함께 사용하는 메모리**를 말합니다. **이 공유 메모리를 이용하면 프로세스끼리 통신**을 할 수 있으며, **같은 데이터를 공유**할 수 있습니다.

이렇게 같은 메모리 영역을 공유하기 위해서는 **공유 메모리를 생성한 후에 프로세스의 자신의 영역에 첨부**를 한 후에 **마치 자신의 메모리를 사용하듯 사용**합니다.

즉, 공유 메모리를 사용하기 위해서는 **공유 메모리를 생성한 후**에, 이 **메모리가 필요한 프로세스는 필요할 때마다 자신의 프로세스에 첨부를 한 후**에 **자신의 메모리를 사용하듯 사용**하면 되겠습니다.

int shmget(key_t key, int size, int shmflg);

shmget() 함수는 공유 메모리를 생성합니다.

key_t key　　공유 메모리를 구별하는 식별 번호
int size　　공유 메모리 크기

shmflg	옵션 내용
IPC_CREATE	**key 에 해당하는 공유 메모리가 없다면 새로 생성**한다. **만약있다면 무시**하며 **생성을 위해 접근 권한을 지정**해 주어야 한다.
IPC_EXCL	공유 메모리가 이미 있다면 실패로 반환하며 공유 메모리에 접근하지 못한다. **이 옵션이 없어야 기존 공유 메모리에 접근할 수 있다.**

반환
-1　　실패
-1 이외　공유 메모리 생성 성공, 공유 메모리 식별자

void *shmat(int shmid, const void *shmaddr, int shmflg)

shmat() 함수는 공유 메모리를 마치 프로세스의 몸 안으로 첨부합니다.

int shmid　　공유 메모리를 구별하는 식별 번호
void *shmaddr　　첨부되는 어드레스 주소. 일반적으로 NULL 을 지정

shmflg	옵션 내용
SHM_RDONLY	공유 메모리를 읽기 전용으로
SHM_RND	shmaddr 이 NULL 이 아닌 경우일 때만 사용되며, shmaddr 을 반올림하여 메모리 페이지 경계에 맞춘다.

반환
(void *)　실패
-1
이외　프로세스에 첨부된 프로세스에서의 공유 메모리 주소

int shmdt(const void *shmaddr)

shmdt() 함수는 프로세스에 첨부된 공유 메모리를 프로세스에서 분리합니다.

void *shmaddr 분리할 공유 메모리 주소

반환
-1　실패
0　공유 메모리 분리 성공