TI DSP,MCU 및 Xilinux Zynq FPGA

프로그래밍 전문가 과정

이름	문지희
학생 이메일	mjh8127@naver.com
날짜	2018/3/28
수업일수	25 일차
담당강사	Innova Lee(이상훈)
강사 이메일	gcccompil3r@gmail.com

목차

1. index 0 개 짜리 배열

1. index 0개 짜리 배열

~소스코드

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

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

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

int main(void)
{
	int i;
	FLEX *p=(FLEX*)malloc(4096);
	//p->score[0];
	//p->score[1];
	for(i=0;i<10000;i++)
	{
		p->score[i]=i+1;
```

```
                printf("%d₩n",p->score[i]);
        }
        return 0;
}
```

~결과
컴파일 에러가 나지 않는다.

malloc 을 많이 하면 할당받는 시간과 해제하는시간이 너무 길고 느려진다.
위의 예제처럼 한번에 크게 잡고 배열처럼 사용하면 성능이 좋아진다.
한번에 크게할당하니까 커널로 진입해야하는 시간이 없고, s
core 는 이 구조체의 끝이 어디인가, 구조체 이후의 새로운 시작점이 어디인가를 알 수 있어 구조체의 끝이자 새로운 시작이라 볼 수 있다.

하지만 이를 사용할 때 데이터가 넘어버리면 밀리거나 데이터 깨질 수 있으니 조심해야한다.

2. index 0개 짜리를 이용한 queue 구현

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

typedef struct
{
        int data;
        int idx;//link 대신 다음에 가리키는 인덱스
}
queue;
```

```
typedef struct
{
	int full_num;//최대치. 값이 할당될 때마다 값이 줄어든다.
	int free_num;//비어있는 것의 갯수
	int total;//총 갯수
	int cur_idx;//현재 내가 어떤 인덱스를 가리키고있나
	// free idx
	int free[1024];//별도관리를 위한 배열
	int total_free;//별도로 관리하는 것이 몇개인지
	queue head[0];//배열의 인덱스?, 배열에 들어가는 것들
}
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; //받은 배열과 이전 인덱스들의 데이터 값을 비교해서 같으면 1

	return false; //다르면 0 반환
}

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))//같을 경우
                    {
                              printf("%d dup! redo rand()₩n", data[i]);
                              goto redo;//redo 로 가서 다시 난수 생성하고 이전 인덱스들 데이터 값 비교
                    }
          }
}

void print_arr(int *arr, int size)
{
          int i;

          for(i = 0; i < size; i++)//10개의 배열 출력
                    printf("arr[%d] = %d₩n", i, arr[i]);
}

void init_manager(manager *m, int alloc_size)
{
          m->full_num = 0;//할당한게 아무것도 없어서 0
          // 12: full_num, free_num, cur_idx
          // 8: data, idx
          m->free_num = (alloc_size / sizeof(int) - 12) / 2;//현재 몇개가 비어있는지 넣을 것
                                                                  // 12268/8=1533이 실질적으로 사용할 수
있는 공간
          m->total = (alloc_size / sizeof(int) - 12) / 8;//1533
                                                                  //아무것도 할당안해서 free_num 이랑 같음
          m->cur_idx = 0;//아무것도 할당안해서 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");
}//total_free 는 dequeue를 했을 때 활성화됨

void enqueue(manager *m, int data)
{
	m->head[m->cur_idx].data = data;
	m->head[m->cur_idx++].idx = m->cur_idx;
		//인덱스 1 증가
	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;//데이터를 0으로 만든다
			m->head[i - 1].idx = m->head[i].idx;
			m->free_num++;//해제했으니 1증가
			m->full_num--;//해제했으니 1감소
			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 데이터 값이 있다면
	{
		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 제낌
```

```
                {
                        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)
                //total_num 으로 바꾸기
                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];
	//1빼이유
	//
	m->total_free--;//배열의 인덱스가 0부터시작하니까 1빼야한
	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];//다음에 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;//true 나 false 를 담는 변수 설정
	int alloc_size = 1 << 12;
	int data[10] = {0};
	int size = sizeof(data) / sizeof(int); //10
	//queue 로 집어넣을 데이터 생성한 것

	srand(time(NULL));
	init_data(data, size);//배열에 값 넣기
	print_arr(data, size);//배열 값 출력

	manager *m = (manager *)malloc(alloc_size);//4096바이트 할당(10^12)
	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);//끝까지 데이터가 차있는지

	print_manager_info(m);
	print_queue(m);

	return 0;
}
```

2. Semaphore

-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.

-critical section(임계영역)

: 여러 task 를 동시에 접근해서 정보가 꼬일 수 있는 공간이다. 쓰레드가 여러개에 프로세스로 있으면 critical section 이 되어 lock 을 걸어야한다.

1) semaphore

lock 이 풀릴 때 까지 접근하지 못해서 lock 이 없는 곳을 찾아다니며 lock 이 없는 곳이 없을 때 프로세스 대기열이 있다.
프로세 스 대기열이 있다 라는 것은 wait queue 로 빠진다는 것이고 context switching 를 해야하는데 이를 하려면 비용이 크다.
(하드웨어 레지스터 를 메모리로 옮겼다가 다시 복원해서 클럭손실이 있음.)

2) spinlock

cpu 점유율이 지속적으로 잡고있다.(polling), 가볍고 빠르게 처리할 수 있는 프로세스들에게 spinlock 이 걸린다.
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 여러개 적용이 가능하다.
단순하고 간단할 땐 spinlock 이 좋고, Context switching 을 하는 것 처럼 여러 프로세스를 실행해야 할 때에는 semaphore lock 형식이 좋다. semaphore 는 크고 대규모의 프로세스를 처리해야할 때 사용된다.

1. sem

-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;//semaphore 아이디
        sid=CreateSEM(0x777);//semaphore 의 권한
        printf("before₩n");
        p(sid);
        printf("Enter Critical Section₩n");//p 를 통해 semaphore 의 값을 1 증가시킨후 출력
        getchar();
        v(sid);
        printf("after₩n");
```

```
	return 0;
}
```

-semlib.c

```
#include "sem.h"

int CreateSEM(key_t semkey)//0x777로 semaphore lock 을 걸겠다. 777로 풀 수 있음
{
	int status=0,semid;
	if((semid=semget(semkey,1,SEMPERM|IPC_CREAT|IPC_EXCL))==1) //권한을 준다 | 프로세스간 통신 생성| semaphore 있으면 씹어라
	{
		if(errno==EEXIST)//존재한다면
		{
			semid=semget(semkey,1,0);//semaphore 존재하는 것을 가져온다.
		}
	}
	else
	{
		status=semctl(semid,0,SETVAL,2);//SETVAL : semaphore 아이디를 0으로 만들어줌(semid 와 SETVAL 사이에 있는 숫자로), sid 값 리턴받음
	}
	if(semid==-1||status==-1)
		return -1;
	return semid;
}

int p(int semid)
{//더하기
	struct sembuf p_buf={0,-1,SEM_UNDO};//SEM_UNDO : semaphore 가 종료되면 다시 semaphore 를 원래 값으로 초기화 시켜라(0으로 되돌려라)
```

```
        if(semop(semid,&p_buf,1) == -1)//semop : semaphore 값을 1 증가시켜라
                return -1;//정상적으로 처리되지 않으면 -1반환
        return 0;//정상적으로 처리되면 0반환
}

int v(int semid)
{//빼기

        struct sembuf p_buf={0,1,SEM_UNDO};
        if(semop(semid,&p_buf,1) == -1)//뺄셈할 때 적용
                return -1;
        return 0;

}
```

~결과

```
before
Enter Critical Section

after
```

semop( ) :

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

인자 → semget()호출에 의해 반환된 키 값, 세마퍼 집합에서 수행 될 동작 배열을 가리키는 포인터, nsops 가 배열 안에 있는 동작의 갯수이다.

→ system call

성공시 0반환 실패시 -1 반환

```
struct sembuf {
        ushort   sem_num;       /* 배열안에서의 세마퍼 인덱스 */
        short    sem_op;        /* 세마퍼 동작 */
        short    sem_flg;       /* 동작 플래그 */
};
```

sem_num

다루고자 하는 세마퍼의 번호

sem_op

수행할 동작 (양수,음수, 또는 0)

음수일 때 세마퍼로 부터 그 값을 뺀다. 세마퍼가 접근을 감지하고 통제하는 자원들을 사용함을 의미한다.

양수일 때 그 값이 더해진다. 세마퍼에게 리소스를 돌려준다는 것을 의미

sem_flg

동작 플래그

2. shm

-shared memory(공유메모리)
: 한개의 메모리를 여러 장치(프로세서)가 공동으로 사용하는 형태의 기억장치 또는 다중처리 시스템을 사용할 때 여러 프로세서가 하나의 기억장치를 공유하여 사용하는 것을 의미한다.
물리 메모리를 공유한다.

-shm.h

```
#include "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.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); //물리메모리 식별자 반환
}

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

int FreePtrSHM(SHM_t *shmptr)
{
	return shmdt((char*)shmptr);//성공시 1,실패시 0 반환
}
```

-send.c

```
#include "shm.h"

int main(void)
{

	 int mid;
```

```
        SHM_t *p; //shared memory
        mid=OpenSHM(0x888);//메모리아이디, 페이지프레임의 아이디값을 얻음(권한을 얻었다)

        p=GetPtrSHM(mid);//진짜 쉐어드메모리의 포인터 값을 얻음, 공유메모리의 물리주소를 얻음

        getchar();

        strcpy(p->name,"아무개");
        p->score = 93;

        FreePtrSHM(p);//shared memory 해제, 시간이 걸린다.

        return 0;

}//물리메모리에 write
```

-recv.c

```
#include "shm.h"

int main(void)
{
        int mid;
        SHM_t *p;
        mid=CreateSHM(0x888);//쉐어드메모리 생성 0x888로 생성
        p=GetPtrSHM(mid);//물리메모리의 주소 얻음

        getchar();

        printf("이름 : [%s], 점수 : [%d]₩n ", p->name, p->score);//아무개랑 93받음

        FreePtrSHM(p);
```

```
        return 0;
}

//ipc 는 프로세스간에 정보를 공유하기 위해 사용한다.
```

~결과

```
 xeno@xeno-NH:~/proj/0328$ ./recv

이름 : [아무개], 점수 : [93]

IPC 통신을 이용하여 프로세스간 데이터를 공유할 수 있음을 확인했다.
```