Xilinx Zynq FPGA, TI DSP, MCU 기반의 프로그래밍 및 회로 설계 전문가 과정

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 장성환
redmk1025@gmail.com

ADVANCED C 구현 방법

* 2.c

```
#include <stdio.h>
#include <stdlib.h>
typedef struct{
	int score;
	char name[20];
}ST;
typedef struct{
	int count; //4
	char name[20]; //20
	int score[0]; //
}FLEX;
int main(void){
	int i;
	FLEX *p = (FLEX *)malloc(4096);
	ST *s = (ST *)malloc(4096);
	for(i=1;i<=100;i++){
	p-> score[i]=i;
	printf("%d\n",p->score[i]);
	}
	for(i=1;i<=10000;i++){
	p-> score[i]=i;
//	  printf("%d\n",p->score[i]);
	}
	return 0;
}
//속도 때문에 쓰는 것이다. score 는 이 구조체의 끝이 어딘가를 나타낸다.
//즉 다음 구조체의 시작 시점을 나타낼 수 있다.
```

```
* 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;
	int free_num;
	int total;
	int cur_idx;
	// free idx
	int free[1024];
	int total_free;
	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
	m->free_num = (alloc_size / sizeof(int) - 12) / 8;
	m->total = (alloc_size / sizeof(int) - 12) / 8;
	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;
        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);

        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;
}
```

OS LOCK 매커니즘

1.semaphore
대기열이 존재

2.spinlock
cpu 를 지속적으로 갖고있다. (polling)

두개의 차이점은 ?
스핀락은 여러개에 적용이 불가능하다. 세마피어는 프로세스 여러개 적용이 가능하다.

화장실 3 개 빈곳 하나 들어가면 빨간불 빨간불을 락이라고 한다.
다 락인 경우에 다른 화장실을 찾아 가는게 세마피어

스핀락은 계속 나올때 까지 문 두들기고 있다.

세마포어에서 대기한다는 것은 waitqueue 로 간다는 것이다.
wq 간다는 것은 컨텍트스위칭을 한다는 것이고, 컨텍스위칭의 비용은 크다.
왜냐하면 하드웨어 계층 구조상 느려터진 메모리로 옮기고 다시 복원해야 하기 때문이다. 저장하고 복원하는 과정에서 클럭손실이 발생한다.

대규모 는 세마포어가 좋고 단순 간단한 것은 스핀락이 좋다.
단순 간단한 것은 빨리 끝내고 빠지는게 낫다.
대규모는 처리하는데 붙잡고 있으면 나머지 부분이 처리가 안되기 때문에
성능이 약간 떨어지더라도 같이 처리하는게 좋기 때문이다.

세마피어를 에서 크리티컬섹션 (임계영역)이 매우 중요하다.
프로세서를 만드는 것이 fork()만 있을까 ?
fork()는 종속관계가 아예 없이 메모리를 새롭게 생성하였다.

쓰레드(thread)는 task_struct 를 만들게 된다. 이 쓰레드는 종속적이다.

p_thread 로 만들게 되면 완전히 메모리를 공유하게 된다.

만약 전역변수 cnt=0 가 있고 쓰레드 a 는 더하기 b 는 빼기라고 할 경우
a 로 가면 cnt =1 b 로가면 cntt=0 이렇게 가는걸 우리가 원하는 정상적인 상황이라고 한다면 실제로는 a b 반복 할수록 우리가 원하는 연산값이 나오지 않는다.
ex) 1, 2, 1, 2, 3.....

b 가 -- 하려는 순간 컨텍스트 스위치가 발생하여 넘어가는 순간
한쪽이 씹히게 된다.

Using lock 락을 쓸 경우에는
a 가 사용중일때 컨텍스트 스위칭이 발생하여 b 가 들어 가려고 해도 못들어 가게 된다. 즉, a 가 처리되기 전까지 b 가 들어갈 수 없다.
a 가 처리를 끝내고 락을 해제했을 때, b 가 진입하여 처리를 한다.
이렇게 세마코어와 스핀락을 쓰는 이유이다.

critical section 이 뭐지 그럼 ?

여러 task 들이 동시에 접근해서 정보가 꼬일 수 있는 구간을 크리티컬 섹션이라고 한다.

데이터 하나가 여러 프로세서가 사용할 때, 그것을 크리티컬 섹션이라고 한다.
(꼬일 수 있다.)

코드 구현은 어셈블리어로 되어있다. (따라서 어셈을 모르면 운영체제 못만진다.)

* sem.c

```
#include "sem.h"

int main(void){
   int sid;
   sid = CreateSEM(0x777); //sid 는 세마코어 아이디.
//0x777 세마코어의 권한
   printf("before\n");
   p(sid);
   printf("Enter Critical Section\n"); //세마코어 1 증가 하므로 그냥 써준것
   getchar(); //엔터 입력 받음
   v(sid);
   printf("after\n");
   return 0;
}
```

헤더 파일과 여러 소스를 가진 프로그램을 컴파일 하는 방법은
gcc [p1.c] [p2.c]와 같은 방법을 사용하면 된다.
세마포어는 공유자원을 동시에 여러 프로세스가 사용하는것을 막고, 오직 하나의 프로세스만 사용할 수 있도록 보장해 준다.

==========

* 함수 분석

semget() 세마포어 생성 및 접근함수
프로토 타입은 int semget(key_t key, int nsems, int semflg)이다.
세머포어 식별번호 key 와 세마포어 갯수(접근제한 하려는 자원의 갯수) 동작옵션을 인자로 전달하여 리턴값은 실패시 -1 또는 새로만들어진 세마포어 식별자 또는 키와 일치하는 세마포어 식별자를 리턴한다.

semflag 는 다음과 같다.
IPC_CREAT
- 키에 해당하는 공유 세마포어가 없다면 새로 생성, 있다면 무시
IPC_EXCL
- 세마포어가 이미 존재한다면 실패 반환하며 세마포어에 접근 할 수 없다.

---

semctl() 세마포어 제어 함수
프로토 타입은 int semctl(int semid, int semnum, int cmd, union semun arg)

세마포어 식별자와 세마포어 집합 내 세마포어 위치, 제어명령, cmd 에 따라 설정 또는 값을 구하는 변수를 인자로 전달하여 리턴값은 성공시 0 실패시 -1 을 리턴한다.

cmd 종류

*semlib.c

```
#include "sem.h"

int CreateSEM(key_t semkey){//셈키는 0777 이라는 걸로 나는 세마코어 락을 걸겠다는 소리. (따라서 777 이 와야 풀 수 있다.)
   int status = 0;
   int semid;
   if((semid = semget(semkey, 1, SEMPERM | IPC_CREAT | IPC_EXCL)) == 1){ //세마코어 권한을 주고 IPC (interprocesscommunication) 프로세서간 통신
// IPC EXCL 해당 키값으로 세마코어가 있으면 무시해라.
         if(errno == EEXIST){ //세마코어가 존재 한다면
            semid = semget(semkey, 1, 0);
//셈겟으로 세마코어 존재하는 것을 가져오겠다는 것이다.(만들어진 세마코어 아이디를 가져옴)
         }
         else{ //존재 하지 않는다면
            status = semctl(semid, 0, SETVAL, 2);
//현재 세마코어 아이디를 가운데에 있는 것으로 세팅해 준다는 것 즉 0 으로 세팅
//그리고 sid 값을 리턴 받았음
         }
   }
   if(semid == -1 || status == -1){
         return -1;
   }
   return semid;
}

int p(int semid){
   struct sembuf p_buf = {0, -1, SEM_UNDO};
```

cmd	cmd 내용
GETVAL	세마포어의 현재 값을 구한다.
GETPID	세마포어에 가장 최근에 접근했던 프로세스의 프로세스 ID를 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 설명
cmd 에 따라 달라지며, 설정 또는 값을 구하는 변수이다.

```
union semun{
   int                  val;
   struct   semid_ds   *buf;
   unsigned short int  *arrary;
}
```

---

semop() 세마포어의 값을 변경하는 함수
세마포어를 사용하기 위해서 먼저 세마포어를 1 감소시키고, 사용후에는 1 증가시키는데 이렇게 세마포어 값을 증감하는 것을 요청하는 함수이다.

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

세마포어 식별자와 세마포어 값을 위한 설정값, 그리고 변경하려는 세마포어 갯수를 인자로 전달하여 리턴값은 실패시 -1 성공시 0 을 리턴한다.
세마포어 설정 값인

```
//SEM_UNDO 는 만약 세마코어가 종료가 되면 다시 세마코어를 원래 값으로 초기화 시키라는 것이다. (초기값이 0 이엇으므로 0 으로 간다)
   if(semop(semid, &p_buf, 1) == -1){
//semop 는 세마코어값을 1 증가시키라는 것이다. 연산 실패시 -1 리턴
          return -1; //오류발생
   }
   return 0;//정상처리
}

int v(int semid){
   struct sembuf p_buf = {0, 1, SEM_UNDO}; // 다른점은 1 이라는것 (빼기)
   if(semop(semid, &p_buf, 1) == -1){ //세마코어에 1 을 빼겠다는 것
          return -1;
   }
   return 0;
}
```

```
struct sembuf{
 short sem_num; // 세마포어 번호
 short sem_op; // 세마포어 증감 값
 short sem_flg; // 옵션
}
```

옵션의 종류는 다음과 같다.

sem_flg	sem_flg 내용
IPC_NOWAIT	호출 즉시 실행하지 못했을 경우 기다리지 않고 실패로 바로 복귀합니다.
SEM_UNDO	프로세스가 종료되면 시스템에서 세마포어 설저을 원래 상태로 되돌립니다. 그러므로 보통 이 옵션을 사용합니다.

인수 unsigned nsops 의 경우

unsigned nsops 변경하려는 세마포어 개수로 변경하려는 세마포어 개수가 여러 개일 때 사용합니다.
예를 들어 변경하려는 세마포어 개수가 한 개라면

```
struct sembuf sbuf = { 0, -1, 0};
semop( semmop, &sbuf, 1);
```

이렇게 값이 1이 되지만 한번에 2개 이상의 세마포어를 변경한다면,

```
struct sembuf pbuf[2]  = {
    { 0,  1, 0} ,  // 첫번째 세마포어 값을 1 증가
    { 1, -1, 0}    // 두번째 세마포어 값을 1 감소
   }
semop( semmop, pbuf, 2);
```

*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);
```

==========

*코드 분석

*****쉐어드 메모리는 ?

메모리를 공유한다는것

어떤 메모리를 공유하나 ?

페이지를 공유한다. 따라서 물리메모리를 공유한다는 것이다.

페이지 프레임과 페이지는 다르다.
실제 다루는 것은 페이지 프레임이며
페이지는 이런것을 처리하는 SW 이다.

센드는 물리메모리에 쓰기

recv 는 물리메모리에서 읽기를 한다.

어떤 프로세스를 만들고 정보를 공유하려고 한다면,
이와 같은 코드를 써야한다.

(즉, 속도 각가속도 등등 계산값을 구하는 프로세서에서 모터 제어 프로세서에 넘겨주기 위해서 쓴다.)

*** IPC 는 프로세서간의 정보를 공유하기 위해서 사용한다. ***

차량일 경우, 메인 제어기가 영상처리 프로세스 모터제어로 보내는 통신 프로세스 등등이 돌아가게 된다.
영상처리 후 어떤 제어 값을 모터제어로 보내기 위해여 프로세서간의 통신이 필요하다. IPC 를 이용하여 이를 구현할 수 있다.

*send.c

```
#include "shm.h"

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

   mid = OpenSHM(0x888);//특정한 구간에 페이지 프레임을 얻고 아이디값을 줌

   p = GetPtrSHM(mid); //가상메모리 이기 때문에 실제 물리 메모리를 얻어오기 위함

   getchar();
   strcpy(p->name, "ninano"); //p 네임에 ninano
   p->score = 93; //p 스코어에 93

   FreePtrSHM(p); //아이디값을 전달하는 이유는 개나소나 접근하면 안되기때

   return 0;
}
```

컴파일 방법은
gcc -o send shmlib.c send.c
gcc -o recv shmlib.c recv.c
터미널 두개 띄우고 ./send 이후에 ./recv
==========
* 함수 분석
shmget() 공유 메모리 생성 함수
프로토 타입은 int shmget(key_t key, size_t size, int shmflg)
주어진 인자 key 를 식별번호로 하는 공유메모리 공간 할당을 커널에 요청한다.
성공적으로 공유메모리 공간을 할당하게 되면, 공유 메모리를 가리키는 식별자 (shmid)를 리턴한다.

생성될 공유 메모리의 공간 크기는 size 를 통하여 byte 단위로 지정이 가능하다.

공간의 할당은 shmflg 가 IPC_PRIVATE 거나 key 를 가지는 공유 메모리 영역이 존재하지 않거나 IPC_CREAT 가 지정되었을 경우 이루어진다.

shmflg 종류
IPC_CREAT
- 새로운 영역을 할당 만약 이 값이 없다면, shmget 은 key 로 이미 생성된 접근 가능한 공유 메모리 영역이 있는지 확인하고, 이에 대한 식별자를 리턴
IPC_EXCL
- IPC_EXCL 은 CREAT 와 함께 사용하며 공유 메모리 영역이 이미 존재하면 에러를 리턴한다.
---

*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);
}

SHM_t *GetPtrSHM(int shmid){
   return (SHM_t*)shmat(shmid, (char*)0, 0); //아이디 가지고 장소를 찾는다.
// 맨 첫번째 주소부터 찾겠다는 것.
}

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

shmat() 공유 메모리를 프로세스에 첨부하는 함수
프로토 타입은 void* shmat(int shmid, const void* shmaddr, int shmflg)
주어진 인자 shmid 를 식별번호로 하여 첨부되는 어드레스 주소 shmaddr (일반적으로 NULL 지정) 동작옵션에 따라서 공유메모리 주소를 리턴한다.

같은 메모리 영역을 공유하기 위해서는 공유 메모리를 생성한 후에 프로세스에 자신의 영역에 첨부를 한 뒤에 마치 자신의 메모리를 사용하듯 사용한다.
즉, 공유 메모리를 사용하기 위해서는 공유 메모리 생성 후 , 이 메모리가 필요한 프로세스는 필요할 때 마다 자신의 프로세스에 첨부를 한 후에 다른 메모리를 사용하듯 사용하면 된다.

shmflg 종류
SHM_RDONLY
- 공유메모리를 읽기 전용으로 한다.

---

shmdt() 프로세스에 첨부된 공유 메모리를 프로세스에서 분리
프로토 타입은 void* shmaddr(const void* shmaddr)
분리할 공유 메모리 주소를 넣는다. 실패시 -1 리턴하고 성공시 0 이며 공유 메모리 분리 성공을 나타낸다.

프로세스에서 공유 메모리를 사용하다가 필요가 없어지면 프로세스에서 제거하는 함수이다.

==========

* 코드 분석

SHM_t 라는 구조체 (24 바이트 크기)를 선언하였다.
해당 구조체를 0x888 이라는 특수한 키값을 통하여 shmget()함수를 통하여
해당 키값을 가진 24 바이트 크기의 공유 메모리를 생성하였다.
리턴값으로 이 공유 메모리를 가리키는 식별자를 리턴하였다. (mid)

*recv.c

```
#include "shm.h"

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

   mid = CreateSHM(0x888); //메모리 주소값 얻음
   p = GetPtrSHM(mid); //

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

   FreePtrSHM(p);
   return 0;
}
```

이 mid 값을 shmat()함수를 통하여 해당 식별자에 해당하는 공유 메모리의 위치를 리턴한다. (메모리의 주소값 전달)

따라서 두개의 다른 프로세서들이 같은 공유 메모리(구조체 SHM_t)를 사용 할 수 있다.

또한 작업을 마무리 한 후에,

shmdt() 함수를 통하여 분리할 공유 메모리 주소 p 를 인자로 전달하여
해당 공유 메모리를 분리하게 된다.

* 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;  //p 는 이러한 구조체의 공간을 만든 것임.

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