운영체제

Project #1

제출일	2018.04.16
학부	컴퓨터공학부
학번	20130398
이름	구영서
분반	
담당교수	김성조 교수님

1. Message Passing

- 전제조건 -

- 미리 제공된 Semaphores를 사용하여 'synch.h'에 Blocking Send 및 Blocking Receive를 지원하는 Message Passing 기능을 추가
- 세마포어 -> 프로세스 간에 시그널을 주고받기 위해 사용 되는 정수 값. 여기서 **프로세스는 thread_create**로 대체
- Blocking Send와 Blocking Receive는 Rendezvous라고 하며 메세지가 전달될 때까지 send와 receive가 대기한다. 즉, **보내고 receive할때까지 send는 기다리고, send할때까지 receive도 기다린다**.
- 구조체 mailbox를 만들고 msg, msg_count, _sema 로 구성되어있다. 각각 메세지, 메세지 수, 세마포어이다. 메세지에는 **input_getc()**를 통해서 사용자에게 입력받는다. 이는 문자하나만 받는 함수이다.
- 'synch.h'에 send와 receive가 만들어져 있고 'projects/msgpassing/'에서 thread_create로 send와 receive를 만들고 각 함수 안에서 무한루프를 돌면서 Rendezvous하는지 살펴본다.

- 코드 흐름과 구조 설명 -

- **msgpassingtest.c**

```
void run_message_passing_test(void)
{
      static struct mailbox mail;
       printf ("start messagepassing...\n");
       mail.msg_count = 0;
       sema_init(&mail._sema, 1); // send
       thread_create ("send", PRI_DEFAULT, send, &mail);
       thread_create ("receive", PRI_DEFAULT, receive, &mail);
}
```

thread_create가 이뤄지고 mail변수를 잃어버리는 것을 방지하기 위해서 static으로 선언했다. 가장 먼저 mail 안에 세마포어를 초기화 해준다. 그리고 먼저 보내야 받기 때문에 send를 먼저 만들어주고 receive를 만들어준다.

- **send**

```
void send(void *sema_)
{
    struct mailbox * mail = sema_;
    while(1){
      if(mail->msg_count > 0){
        if(mail->_sema.value == 0)
          sema_up(&mail->_sema);
      }
      else{
        sema_down(&mail->_sema);
        printf("send_input_wait(esc : exit)\n");
        mail->msg = input_getc();
        mail->msg_count++;
        printf("send : ");
        if( mail->msg == 27){
          printf("byebye\n");
          return;
        }
        printf("%c\n", mail->msg);
      }
    }
}
```

Thread 생성 후 send가 먼저 시작이 된다. Void * sema_ 이 파라미터로 보내진 mail의 msg_count를 검사한다. 이 값이 0 보다 크다는 것은 이미 send가 된 상황이기 때문에 제어권을 receive에게 넘겨줘야한다. **그때 세마포어의 값이 0이라면 현재 send가 제어권을 가진 상황이기 때문에 sema_up을 해준다**. msg_count가 0이라면 send를 해야하기 때문에 sema_down을 하여 제어권을 가져온다. 이때 사용자에게 input_getc()로 문자 하나를 받는다. 받고나면 msg_count가 1인 상황이기 때문에 처음에 언급했다시피 receive에게 제어권을 넘겨줘야한다.

- mail->msg == 27은 esc키의 int값이므로 종료 조건 체크를 의미한다.

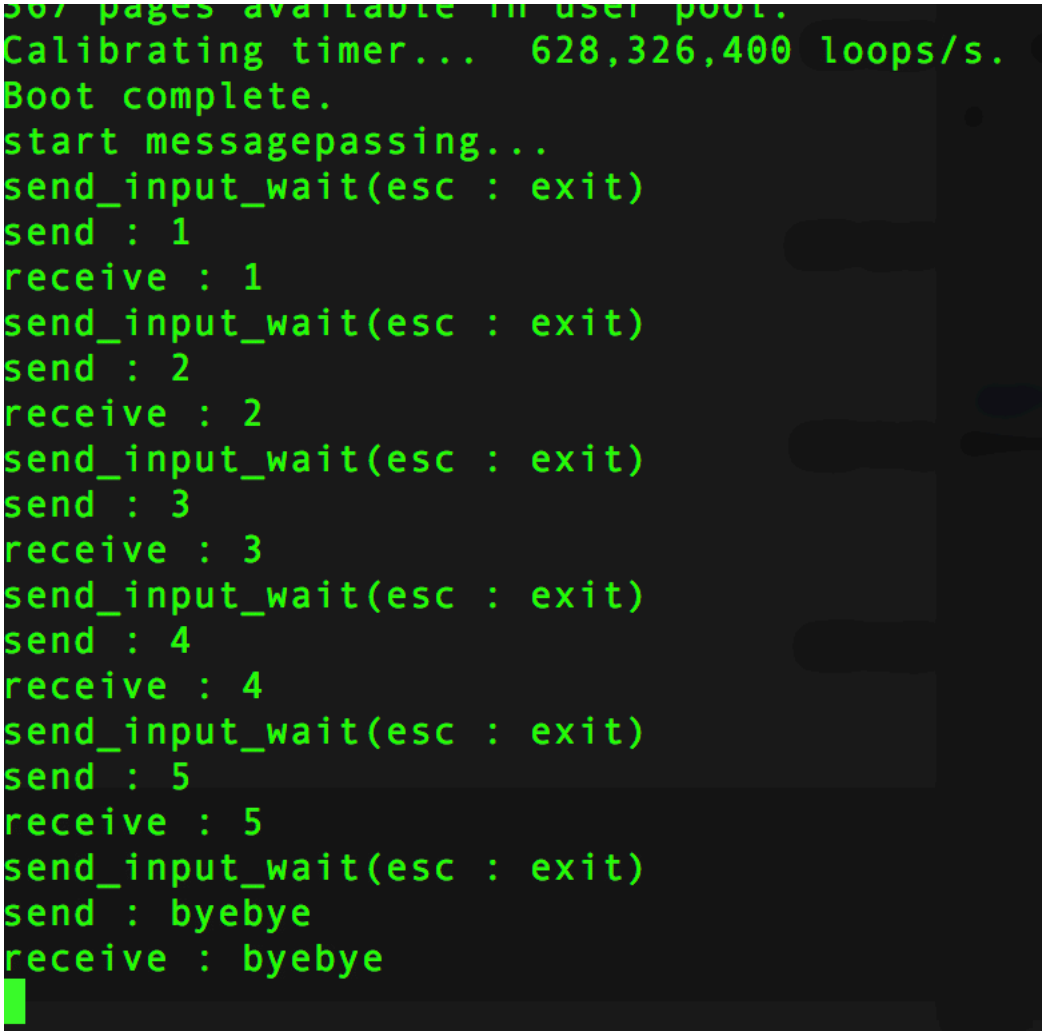
- **receive**

```
void receive(void *sema_)
{
    struct mailbox * mail = sema_;
    while(1){
      if(mail->msg_count <= 0){
        if(mail->_sema.value == 0)
          sema_up(&mail->_sema);
      }
      else{
        sema_down(&mail->_sema);
        printf("receive : ");
        if( mail->msg == 27){
          printf("byebye\n");
          return;
        }
        printf("%c\n", mail->msg);
        mail->msg_count--;

      }
    }
}
```

기본적인 구조는 send와 같다. 먼저 mail의 msg_count를 검사하고 0보다 같거나 작다는 것은 이미 receive하고 난 후 기 때문에 제어권을 send에게 넘겨주는 것이다. 그게 아니라면 제어권을 가지고 mail의 mag를 읽는다. 읽고 난 후 msg_count의 값을 빼준다. 이 때의 값이 0이라면 제어권을 send에 넘겨준다.

- 결과 -



흐름은 run_message_passing_test() 에서 시작하여 send -> receive -> send -> receive …. 이렇게 흐른다. 동시에 mail에 접근을 막기위해 이 둘은 하나의 세마포어를 가지고 Blocking Send 및 Blocking Receive을 표현했다. send가 제어권을 가지고 receive는 대기를 한다. send는 1을 사용자에게 입력받아 msg에 저장하고 msg_count를 증가시켜서 제어권을 포기한다. 그 때 receive가 바로 제어권을 쥐고 msg을 받고 msg_count을 빼준다. msg는 비어있기 때문에 다시 제어권은 send에게 간다.

- 테스트 방법은 threads폴더 안에서 ../utils/pintos messagepassing로 시작하면 된다. threads/build/ 에서 ../../utils/pintos messagepassing 해도 관계없음.

2. Crossroads

- 전제조건 -

- 모든 차량은 개별적인 thread로 동작
- 모든 차량들의 thread(**koo_alg**), 상황을 그려주는 thread(**koo_draw**)로 구성되어있다.
- 차량의 양보 규칙이 있다. **먼저 우선권은 A에서의 출발하여 X로 도착한다면 다른 차량들 중 X에서 A로 도착하는 차량의 우선권이 있다. 그 외의 차량은 모두 대기 해야한다.**
- 차량의 연산이 모두 끝 난 뒤 draw를 해야한다.
- 5개의 구조체를 선언했다. 각각의 구조체의 역할은 parsing 내용 저장하는 구조체, 기다리는 차량의 현황 저장하기 위한 구조체, 교차로 차량의 정보를 저장하는 구조체, 각 차량 쓰레드의 모든 차량들의 정보와 자신의 인덱스를 알려주기 위한 구조체, 맵 정보 등 공유 정보를 가지는 구조체 이렇게 5가지 구조체가 선언이 되어있다.
- 세마포어는 각 차량이 가지며. 맵 데이터도 세마포어를 가진다.
- 현재 thread의 차량을 **'나로 인해서'**라는 표현으로 사용 할 수 있다.

- 핵심 코드 흐름과 구조 설명 -

- **구조체 설명**

```
struct parseData {
    int numberofData; //파싱할 데이터의 수
    char ** data;     //파싱 데이터를 저장할 공간
};

struct waitingCar
{
    int * waitingCar;   //현재 차량들의 기다리는 상황 1이상이면 wait
};
```

```
struct shareData
{
    char  map_draw[7][7];           // 맵 정보
    struct semaphore * draw_sema;   // 맵 세마포어
    int endCount;                   // 차량이 종료할 때마다 하나씩 증가
    int max_index;                  // 총 차량의 수
    struct waitingCar * pwait;      // waitingCar
    bool * carInFlag;               // 차들이 멈춰야하는지 검사하는 영역에 들어왔는지를 체크
};
```

```
struct crossroadInfo
{
    char prev_state;            // 차량이 한 칸 이동할때 발자국을 지우기 위해 필요
    struct semaphore * sema;    // 차량의 세마포어
    int state;                  // 차량의 path의 값을 가져오기 위한 변수
    struct shareData * share;   // 차량은 share 데이터를 가진다.
    int from;                   // 출발장소
    int to;                     // 도착장소
    char car;                   // 자동차 이름
    int endFlag;                // 이 쓰레드가 끝났는지 체크
    bool * waitFlag;            // 나로 인해 멈춘 차들의 정보
};

struct wrapper
{
    struct crossroadInfo * info;    // 모든 차량들의 정보
    int index;                      // 나의 index
};
```

- ***run_crossroads***

```
    for (int i = 0; i < _parse.numberofData; ++i){
        wrap[i].info = _info;
        wrap[i].index = i;
        thread_create ("alg", PRI_DEFAULT, koo_alg, &wrap[i]);
     }
        thread_create ("draw", PRI_DEFAULT, koo_draw, share);
```

사용하고자 하는 변수들을 할당하고 초기화(맵 복사, 파싱 정보를 저장 등) 시켜준 뒤 차량 수(**_parse.numberofData**) 만큼 thread를 만들어 준다. 차량 thread에는 모든 차량의 정보(**_info**)와 자신의 **index**를 넣어서 파라미터로 전달한다.

- ***koo_alg***

```
void koo_alg(void *_sema){
    struct wrapper  * wrap = _sema;
    struct crossroadInfo * info = wrap->info;
    int i = wrap->index;
    int max_index = info[i].share->max_index;
    int row,col;

    while(1){
        if( info[i].endFlag == 1){
            info[i].share->endCount++;
            return;
        }
        sema_try_down(info[i].share->draw_sema);
        if(sema_try_down(info[i].sema)){
            row = path[info[i].from][info[i].to][info[i].state].row;
            col = path[info[i].from][info[i].to][info[i].state].col;

            if(info[i].state >= 0 && row != -1 && col != -1){
                if(info[i].share->map_draw[row][col] == ' ' || info[i].share->map_draw[row][col] == '-')
                {
                    waitCar(info, max_index, i, row, col);
                    clearTrail(&info[i],row,col,false);
                    info[i].state++;
                    info[i].share->map_draw[row][col] = info[i].car;
                }
            }
            else{
                clearTrail(&info[i],row,col,true);
            }
            sema_up(info[i].sema);
            sema_up(info[i].share->draw_sema);
        }
        timer_sleep(_time);
    }
}
```

info에는 모든 차량의 정보, **info[i]는 현재 thread의 차량 정보**이다. **row와 col은 현재 차량이 갈 위치**이다. endFlag가 1이면 도착을 완료한 시점으로 공유자원의 endCount를 증가시켜준다. 현재 thread가 연산을 진행하기 위해서 맵은 그려지면 안되므로 제어권을 뺏는다. 그리고 현재 thread의 sema_down을 한다. 참고로 sema_try_down()은 sema_down이 가능하다면 down시키고 true 반환 아니면 false를 반환한다.

현재 차량이 가는 곳이 갈 수 있는 ‘ ‘ or ‘-’라면 **현재 차량을 제외한 나머지 차량의 wait여부를 waitCar함수**를 통해서 알아내고 현재 차량이 갈 위치에 차량을 놓는다. 그 후 놓기전의 위치를 **clearTrail함수**를 통해서 지운다.

그리고 현재 차량의 세마포어와 맵의 세마포어를 up시킨다.

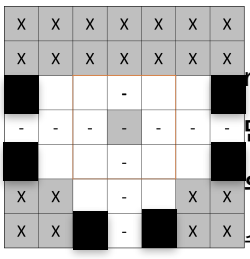
- ***waitCar와 waitAlgorithm***

```
void waitAlgorithm(struct crossroadInfo * info,int from, int to,int me, int max){
    for (int i = max - 1; i >= 0; --i)
    {
        if(me != i && info[i].state  == 2 && !(info[i].from == to &&  info[i].to == from)) {
            sema_try_down(info[i].sema);
            if(!info[me].waitFlag[i]){
                info[me].share->pwait->waitingCar[i]++;
            }
            info[me].waitFlag[i] = true;
        }
    }
}
void waitCar(struct crossroadInfo * info, int max ,int me , int row, int col){
    struct crossroadInfo _info = info[me];

    if(( _info.state <= 1 ))
        return;

    if((col >= 2 && row >= 2) && (col >= 2 && row <= 4) &&
        (col <= 4 && row >= 2) && (col <= 4 && row <= 4)){
        waitAlgorithm(info,_info.from,_info.to,me ,max);
    }
    else{
        for (int i = 0; i < max; ++i)
        {
            if(info[me].waitFlag[i]){
                if(_info.share->pwait->waitingCar[i] > 0){
                    _info.share->pwait->waitingCar[i]--;
                    info[me].waitFlag[i] = false;
                    if(_info.share->pwait->waitingCar[i] == 0){
                        sema_up(info[i].sema);
                    }
                }
            }
        }
    }
}
```

koo_alg함수에서 설명한 (**현재 차량을 제외한 나머지 차량의 wait여부를 waitCar함수)** 부분이다. 현재 state가 1이하라는 것은



이 맵에서 검은 상자 부분에 있거나 혹은 맵에 보이지도 않은 상황을 말한다. 따라서 이 부분이면 return시킨다.

만약 주황색 테두리 안으로 현재 차량이 들어온다면 이 차량이 먼저 진입했기 때문에 충돌을 방지하기 위하여 특정 차량들은 잠시 기다려야한다. 이 차량이 이 테두리 안으로 들어온다면 waitAlgorithm함수를 호출한다. 벗어 나기 전까지 계속 들어오는 이유는 현재 차량이 나가면서 뒤차가 다시 기다려야 하는 상황이 올 수 있으므로 벗어나기 전까지 계속 waitAlgorithm함수를 호출하여 기다림을 업데이트한다.

waitAlgorithm함수를 설명하기 앞서 **테두리 밖으로 빠져나온다면 기다렸던 차량들을 다시 진행시켜야한다.** 그 작업이 else 문에서 진행한다. **info[me].waitFlag는 나(현재 thread차량)로 인해서 기다린 차들의 목록**이다 info[me].waitFlag[3]이 true라면 3번차량이 나로 인해서 기다리는 것이다. 이 차량의 기다림 수치를 빼준다. **바로 세마포어를 up시키지 않는 이유**는 주황색 테두리 안으로 차량이 들어왔고 기다릴 필요가 없는 차량이 같이 들어왔을 때 중복으로 다른 차량을 기다리게 해야할 수 있기 때문이다. 예를 들어 a와 d라는 차량이 c라는 차량은 2의 값으로 기다리게 할 수 있고 a와 d차량이 도착하면 그때 c

의 waitingCar 값이 2에서 0으로 되어 sema_up 해줘야하기 때문에 바로 sema_up 시키지 않는다.

다음은 waitAlgorithm함수이다. 차량이 주황색 테두리 안으로 들어왔을 때 접근하게 된다. 접근 차량으로 인하여 기다려야 하는 차들을 sema_try_down(sema_try_down 쓴 이유는 위에 설명 한 것처럼 중복으로 wait시킬 수 있으므로) 시킨다. **전제조건의 양보 규칙을 적용하였다. 나로 인해 기다리게 되는 차들을 나의 정보(info[me].waitFlag)에 true로 저장하고 share의 waitingCar의 기다림 수치를 더해준다.**

- ***koo_draw와 allFree***

```
int _time = 100;

void koo_draw(void *_sema){
    struct shareData * share = _sema;
    while(1){
        if(share->endCount >= share->max_index){
            printf("byebye\n");
            allFree();
            return;
        }
        sema_try_down(share->draw_sema);
        for (int i = 0; i < 7; ++i)
        {
            for (int j = 0; j < 7; ++j)
            {
                printf("%c", share->map_draw[i][j]);
            }
            printf("\n");
        }
        puts("-----------------------------------------");
        puts("-----------------------------------------");
        sema_up(share->draw_sema);
        timer_sleep(_time);
    }
}
void allFree(){
    int max = share->max_index;
    free(share->carInFlag);
    free(share->draw_sema);
    free(share->pwait->waitingCar);
    free(share->pwait);
    free(share);
    free(wrap);
    for (int i = 0; i < max; ++i)
    {
        free(_info[i].sema);
        free(_info[i].waitFlag);
    }
    free(_info);
}
```

간단하게 설명하고자 한다. **koo_draw는 상수 맵 데이터를 copy해서 담은 share의 맵에 koo_alg의 연산이 진행되고 그 정보들을 koo_draw에서 뿌려준다.**

allFree는 할당한 메모리들을 해제해주는 작업을 진행한다.

- 결과 -

스크린샷의 양이 너무 많아서 따로 첨부하지 않음

- 테스트 방법은 threads폴더 안에서 ../utils/pintos crossroads aAC:bBA:cBC:dCA:eAB:fAB:gCB 로 시작하면 된다. threads/build/ 에서 ../../utils/pintos crossroads aAC:bBA:cBC:dCA:eAB:fAB:gCB 해도 관계없음.

- crossroads aAC:bBC:cBA:dCB:eAC:fAC:gCA

- crossroads aAC:bBA:cBC:dCA:eAB:fAC:gCB

다음 인자로 테스트 했음.