|  |  |
| --- | --- |
| **Course Title** | **OS** |
| **Instructor** | **Professor.Hwang** |
| **report ID** | **HW 4** |
| **Due date** | **2016.04.19** |

|  |  |
| --- | --- |
| **Department** | 컴퓨터공학부 |
| **Student id.** | 20123432 |
| **Student name** | 홍용현 |
| **Submission date** | 2016.04.19 |

1. Dining Philosopher 문제를 세마포를 이용하여 구현하시오.

* 철학자 세명 A,B,C
* 젓가락 세개 R1,R2,R3
* 각 젓가락을 Mutual Exclusive 하게 만들기 위한 Lock L1, L2, L3
* 젓가락이 없으면 기다리는 Condition Variable C1, C2, C3
* Lock 과 Condirion Variable은 앞의 숙제처럼 리눅스 커널의 세마포로 구현
* R1,R2,R3는 앞의 숙제처럼 텍스트 화일로 구현

주어진 소스를 활용하여 구현하였다. 아래는 Hard copy

* DPa.c

#include "DP.h"

int main(){

int i;

int L1, L2, C1,C2;

if ((L1 = initsem(L1key,1)) < 0)

exit(1);

if ((L2 = initsem(L2key,1)) < 0)

exit(1);

if ((C1 = initsem(C1key,1)) < 0)

exit(1);

if ((C2 = initsem(C2key,1)) < 0)

exit(1);

for(i=0; i<100; i++){

Phil\_A(L1, L2, C1,C2);

}

return 0;

}

* DPb.c

#include "DP.h"

int main(){

int i;

int L2, L3, C2,C3;

if ((L2 = initsem(L2key,1)) < 0)

exit(1);

if ((L3 = initsem(L3key,1)) < 0)

exit(1);

if ((C2 = initsem(C2key,1)) < 0)

exit(1);

if ((C3 = initsem(C3key,1)) < 0)

exit(1);

for(i=0; i<100; i++)

Phil\_B(L2, L3, C2,C3);

return 0;

}

* DPc.c

#include "DP.h"

int main(){

int i;

int L1, L3, C1,C3;

if ((L1 = initsem(L1key,1)) < 0)

exit(1);

if ((L3 = initsem(L3key,1)) < 0)

exit(1);

if ((C1 = initsem(C1key,1)) < 0)

exit(1);

if ((C3 = initsem(C3key,1)) < 0)

exit(1);

for(i=0; i<100; i++)

Phil\_C(L3, L1, C3,C1);

return 0;

}

세 메인은 각각 다른 프로세서에서 실행된다.

* DP.h

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <time.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

#include <errno.h>

#define SEMPERM 0600

#define L1key 2012343200 // 서버에서 작업할 때는 자기 학번 등을 이용하여 다른 사람의 키와 중복되지 않게 해야 함

#define L2key L1key+1

#define L3key L1key+2

#define C1key L1key+3

#define C2key L1key+4

#define C3key L1key+5

//semaphore

typedef union semun{

int val;

struct semid\_ds \*buf; //sem\_id : semaphore set associated with a semaphore ID. There is one semaphore set per semaphore ID.

ushort \*array;

} semun;

int initsem (key\_t semkey, int n) {

int status = 0, semid;

if ( (semid = semget (semkey, 1, SEMPERM | IPC\_CREAT | IPC\_EXCL)) == -1) {

if (errno == EEXIST) //buffer = null

semid = semget (semkey, 1, 0);

}

else{

semun arg;

arg.val = n;

status = semctl(semid, 0, SETVAL, arg);

}

if (semid == -1 || status == -1) {

perror("initsem failed");

return (-1);

}

return (semid);

}

int p (int semid) {

struct sembuf p\_buf;

p\_buf.sem\_num = 0;

p\_buf.sem\_op = -1;

p\_buf.sem\_flg = SEM\_UNDO;

if (semop(semid, &p\_buf, 1) == -1) {

perror ("p(semid) failed");

exit(1);

}

return (0);

}

int v (int semid)

{

struct sembuf v\_buf;

v\_buf.sem\_num = 0;

v\_buf.sem\_op = 1;

v\_buf.sem\_flg = SEM\_UNDO;

if (semop(semid, &v\_buf, 1) == -1)

{

perror ("v(semid) failed");

exit(1);

}

return (0);

}

void Wait(int semid, int lock){

FILE \* fp = fopen("input.txt", "a+");

time\_t t;

time(&t);

fprintf(fp, "waiting %d %s", getpid(), ctime(&t) );

fclose(fp);

v(lock);

p(semid);

//printf("%d is waiting", semid);

p(lock);

}

void Signal(int semid){ //Delete

v(semid);

}

void Acquire(int mutex){

p(mutex);

}

void Release(int mutex){

v(mutex);

}

void Broadcast(int semid, int num){ //Clear

int i;

for(i=0; i<num; i++){

v(semid);

}

}

위의 세마포어와 모니터는 지난 HW3를 활용하였다. 단 Wait는 2번 항목에서 확인을 하기 위해 파일 입출력을 더했다.

int Load(int R){

int temp;

FILE \*fp;

switch (R){

case 1:

fp = fopen("R1.txt", "r");

break;

case 2:

fp = fopen("R2.txt", "r");

break;

case 3:

fp = fopen("R3.txt", "r");

break;

}

if(fp == NULL){

return 1;

}

fscanf(fp,"%d", &temp);

if(temp==1){

fclose(fp);

return 1;

}

else{

fclose(fp);

return 0;

}

}

void Store(int R, int n){

FILE \*fp;

switch (R){

case 1:

fp = fopen("R1.txt", "w+");

break;

case 2:

fp = fopen("R2.txt", "w+");

break;

case 3:

fp = fopen("R3.txt", "w+");

break;

}

fprintf(fp, "%d", n);

fclose(fp);

}

void think(){

FILE \* fp = fopen("input.txt", "a+");

fprintf(fp, "thinking\n");

fclose(fp);;

sleep(1);

}

void eating(){

FILE \* fp = fopen("input.txt", "a+");

fprintf(fp, "eating\n");

fclose(fp);;

sleep(1);

}

젓가락이 있는지 텍스트 파일로부터 확인하는 함수 Load,

젓가락을 꺼내고 넣는 Store (R1, R2, R3 파일구현) 를 구현하였다.

생각, 식사 함수는 sleep을 포함하되 2번 항목에서 확인하기 위해 파일 입출력을 더했다.

**\*파일 입출력의 경우 본문에서 Lock이 걸려있지 않으나 sol2에서 개선하였다. 자세한 건 4번 항목에서 기술**

**Take, put, 함수도 동일하다.**

void Take\_R1(int L1, int C1, int R1) {

FILE \* fp = fopen("input.txt", "a+");

time\_t t;

time(&t);

fprintf(fp, "A-taking %d %s", getpid(), ctime(&t) );

fclose(fp);;

Acquire(L1);

// R1이 1이면 젓가락이 있고 0이면 젓가락이 없어서 기다려야 함

while (Load(R1)==0) {

// print message: getpid()가 R1을 기다림

Wait(C1,L1);

// print message: getpid()가 R1을 기다리다가 깨어남

}

Store(R1,0);

// print message: getpid()가 R1을 가져옴

Release(L1);

}

void Put\_R1(int L1, int C1, int R1) {

FILE \* fp = fopen("input.txt", "a+");

time\_t t;

time(&t);

fprintf(fp, "A-put %d %s", getpid(), ctime(&t) );

fclose(fp);;

Acquire(L1);

Store(R1,1);

Signal(C1);

// print message: getpid()가 R1을 기다림

Release(L1);

}

void Phil\_A(int L1, int L2, int C1, int C2) {

int R1=1, R2=2; //chapstick

Take\_R1(L1, C1, R1);

// print message: getpid()가 생각을 시작함

think(); // sleep()으로 구현

// print message: getpid()가 생각을 멈춤

Take\_R2(L2, C2, R2);

// print message: getpid()가 먹기 시작함

eating(); // sleep()으로 구현

// print message: getpid()가 먹기를 멈춤

Put\_R1(L1, C1, R1);

Put\_R2(L2, C2, R2);

}

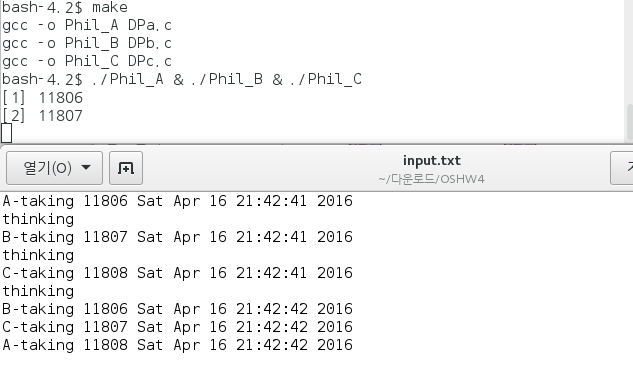
각각의 Take, Put, Phil 함수는 알고리즘이 중복되어 하나씩 Hard copy했다.

Take함수는 젓가락을 집되 없는 경우 wait한다.

Put 함수는 젓가락을 내려놓고 해당 젓가락을 얻기 위해 wait하고 있는 대기열에 signal을 보낸다.

Phil는 젓가락을 이용해 식사를 하는 과정을 나타낸다.

2. Phil\_A, Phil\_B, Phil\_C 를 별도의 프로세스로 실행하여 데드락이 걸리는 상황을 만들고 이 상황의 스냅 샷을 보이고 설명하시오.



Wait을 제한 결과이다. 반복횟수 100회, 각 sleep(1) 기준으로 하였으며 sleep 시간이 짧아질수록 데드락에 빠지기 전까지 수행 횟수가 늘어나는 경향을 보였다.

= Wait를 포함한 결과는 4번 항목에서 추가 설명하겠습니다.

3. Deadlock을 다음 두가지 방법으로 Prevention(예방)하시오.

* Circular wait 조건을 깨는 방법 – 가상대학에 solution 1으로 첨부

void Phil\_C(int L3, int L1, int C3, int C1) {

int R3=3, R1=1; //chapstick

Take\_R1(L1, C1, R1);

think(); // sleep()으로 구현

Take\_R3(L3, C3, R3);

eating();

Put\_R3(L3, C3, R3);

Put\_R1(L1, C1, R1);

}

Take\_R3와 R1의 순서를 바꿨다.

* 젓가락을 한꺼번에 가져와서 Hold & Wait 조건을 깨는 방법(Lock이 추가로 필요함)

메인함수에 for문만 남기고 모든 기능을 Phil 함수 하나로 통일하였다.

DPa.c

#include "DP.h"

int main(){

int i;

for(i=0; i<100; i++)

Phil\_A();

return 0;

}

void Put\_R1(int L1, int C1, int R1) {

Acquire(L1);

FILE \* fp = fopen("input.txt", "a+");

time\_t t;

time(&t);

fprintf(fp, "A-put %d %s", getpid(), ctime(&t) );

fclose(fp);;

Store(R1,1);

Signal(C1);

// print message: getpid()가 R1을 기다림

Release(L1);

}

sleep하는 시간을 줄이기 위해 (기존 소스는 1초 미만으로 줄이면 출력이 꼬이는 경우가 잦았음)

Lock을 파일 입출력을 포함하여 걸었다.

void Phil() {

int R1=1, R2=2,R3=3; //chapstick

int unionlock;

int L1, L2,L3, C1,C2,C3;

if ((L1 = initsem(L1key,1)) < 0)

exit(1);

if ((L2 = initsem(L2key,1)) < 0)

exit(1);

if ((L3 = initsem(L3key,1)) < 0)

exit(1);

if ((C1 = initsem(C1key,1)) < 0)

exit(1);

if ((C2 = initsem(C2key,1)) < 0)

exit(1);

if ((C3 = initsem(C3key,1)) < 0)

exit(1);

if ((unionlock = initsem(ULkey,1)) < 0)

exit(1);

//take all chapstick

Acquire(unionlock);

Take\_R1(L1, C1, R1);

Take\_R2(L2, C2, R2);

Take\_R3(L3, C3, R3);

Release(unionlock);

// print message: getpid()가 생각을 시작함

think(); // sleep()으로 구현

// print message: getpid()가 생각을 멈춤

// print message: getpid()가 먹기 시작함

eating(); // sleep()으로 구현

// print message: getpid()가 먹기를 멈춤

Put\_R1(L1, C1, R1);

Put\_R2(L2, C2, R2);

Put\_R3(L3, C3, R3);

}

void Phil\_A(){

Phil();

}

void Phil\_B(){

Phil();

}

void Phil\_C(){

Phil();

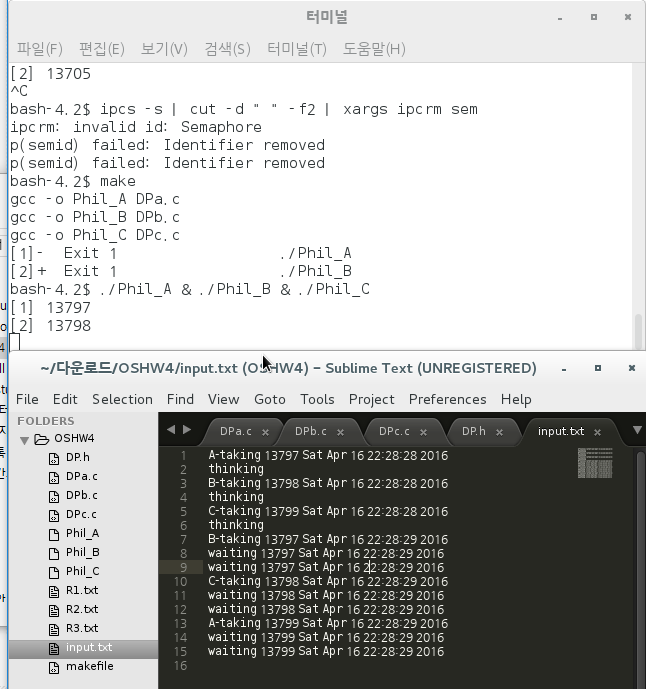
}

Phil 함수를 통합하였다.

각 하나의 Phil\_x는 젓가락을 한번에 모두 독점한 뒤 식사를 진행한다.

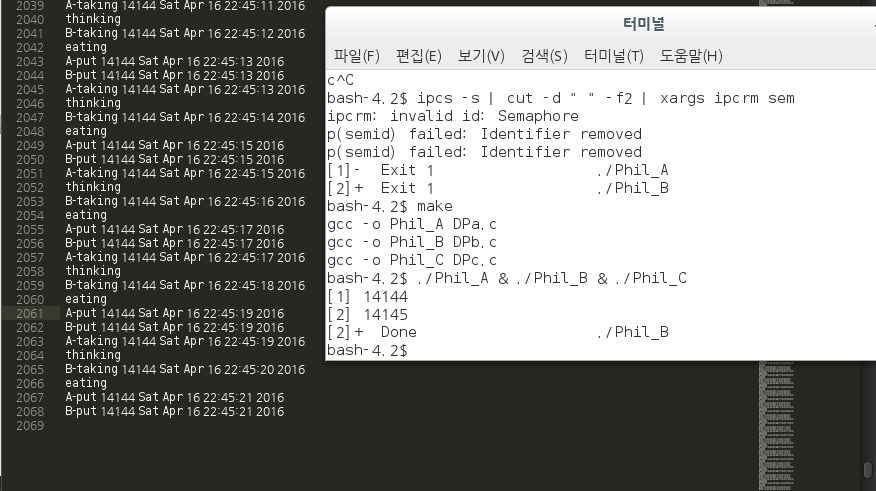
4. Deadlock이 걸리는 상태와 이 문제가 예방으로 해결된 두가지 종류의 상태를 보여주는 실행 결과를 제시하고 설명하시오.

Solution 이전의 데드락 상황 (waiting 포함 스냅샷, sleep(1))



각 프로세서들이 젓가락을 집다가 waiting하여 데드락에 빠지는 것을 볼 수 있다.

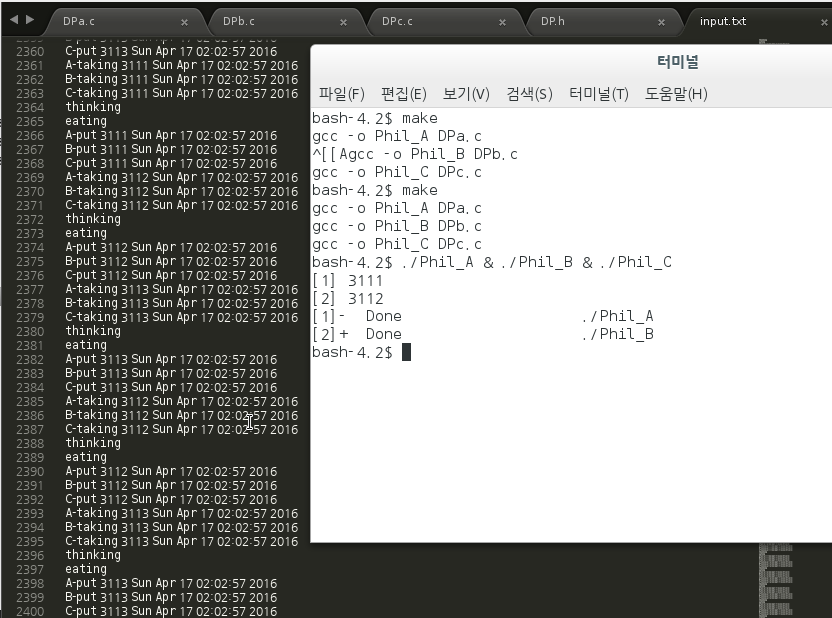
Solution First – circular wait break!



세 명 중 한 명이 젓가락을 집는 순서를 뒤집어서 순환을 깨버렸다.

Sleep(1)을 적용했기 때문에 시간이 유의미하게 흘러가 있다.

Solution Second – Hold&wait break!



한 명이 젓가락을 독점하여 Hold&wait를 깨트렸다.

Lock공유한 방식이기 때문에 독점하는 동안 다른 이가 채갈 수 없다.

또한 3개를 한 번에 가져가 다른 이가 실수로 젓가락을 채가는 것 또한 불가능하다.

총 100회 \* 3명 \* 8개 구문(taking\*3, thinking, eating, put\*3)으로 정확히 동작하였다.

Sleep(0.1)을 사용하였기 때문에 진행시간은 2초로 대폭 짧아졌다.