ใบงาน 9

วัตถุประสงค์ เพื่อศึกษาหลักการการใช้คำสั่ง make ผ่านการใช้ peterson algorithm และ ศึกษาการใช้ semaphore

กิจกรรม 1 ศึกษา http://computingblog.intakosum.net/2015/03/make.html

1.1 จาก race condition เราเรียกส่วนของโปรแกรมที่หากมีการเข้าถึงตัวแปรพร้อมกันว่า critical section

เราทราบว่า Peterson algorithm สามารถสร้างกลไกประหนึ่งการล็อคกุญแจ (enter critical section) เมื่อเข้าสู่ critical section เพื่อกันไม่ให้ โพรเซส / เทรด อื่นเข้าถึงจนกว่าจะปลดล็อค (exit critical section)

อนึ่ง Peterson มีข้อจำกัดสำหรับการล็อคระหว่าง 2 โพรเซส / เทรด เท่านั้น (algorithm ที่พัฒนาต่อจาก Peterson เพื่อล็อคระหว่าง n โพรเซส คือ bakery)

peterson.h > TRUE

```
1   #define TRUE 1
2   #define FALSE 0
3   struct Memory {
4       int  turn;
5       int flag[2];
6   };
7   void initializePeterson();
8   void removePeterson();
9   void enterCriticalSection(int i);
10  int exitCriticalSection(int i);
```

```
/* Pi */
while (true) {
  flag[i] = TRUE;
  turn = j;
  while ( flag[j] && turn == j);
  //CRITICAL SECTION
  flag[i] = FALSE;
  REMAINDER SECTION } //while
```

```
/* Pj */
while (true) {
  flag[j] = TRUE;
  turn = i;
  while ( flag[i] && turn == i);
  //CRITICAL SECTION
  flag[j] = FALSE;
  REMAINDER SECTION } //while
```

1.2 โครงสร้างของ peterson ที่ปรากฏในไฟล์ .h ส่วน .c คือ implementation ของมัน

ตามกลไกของ peterson นั้น โพรเซส i จะผ่านเข้าไปได้ต้อง flag[j] ต้องเป็น FALSE และ turn ต้องเป็น i (สังเกตว่า Pj เป็นฝ่ายอนุญาต) โดยเมื่อใช้เสร็จ โพรเซส i ประโยค flag[i] = FALSE จะเป็นการฝ่ายแจ้ง j ว่า i ไม่อยู่ใน critical section

การใช้ภาพที่แสดงฝ่าย i กับ j นั้น ในความเป็นจริง enterCriticalSection(int i) ไม่ได้เขียนเป็น function แยก แต่ต่างกันตอนเรียกค่า i กล่าวคือ โพรเซส i ส่ง **0** โพรเซส j ส่ง **1**

กล่าวโดยสรุป algorithm นี้ 2 โพรเซสจะ update ค่าพร้อมกันไม่ได้ เพราะ turn เป็น ทั้ง i และ j ในเวลาเดียวกันไม่ได้ หากจะเข้าพร้อมกัน จะมีฝ่ายหนึ่งติด while loop (และฝ่ายที่อยู่ใน critical section จะเป็นฝ่ายมาปลดด้วยการ reset Flag ของตนเอง) exitCriticalSection(int i) ก็คือการ implement flag[i] = FALSE

1.3 นักพัฒนาโปรแกรมสามารถรวม Library จาก local folder ได้ ในกรณีนี้เราใช้ perterson.h และ peterson.c ไวยากรณ์ในการ compile ได้แก่ gcc -o goodCnt petersonTest.c perterson.c โดย petersonTest ต้อง `#include "peterson.h"` (ไม่ใช่ #include <peterson.h> เพราะมันเป็น user-defined library ที่ไม่ได้อยู่ใน standard path)

1.4 unix มีเครื่องมือเพื่อให้นักพัฒนาโปรแกรมสามารถเขียน script สำหรับ compile โปรแกรมที่มี source หลายไฟล์ เรียกว่า make โดย script ไฟล์สำหรับ make จะเป็น .mk

เรียกคำเรียกหน้า : ว่า target เช่น all: testPeterson:

ส่วนที่ตามหลัง target คือ รายชื่อไฟล์ที่จะนำมาสร้าง หรือ target เช่น all คือ default target ว่าให้ไปทำที่ target testPeterson เมื่อไปถึง testPeterson: make จะทราบว่า testPeterson ต้องใช้ testPeterson.o และ perterson.o

peterson.mk

```
all: testPeterson

testPeterson: testPeterson.o peterson.o
    gcc -o testPeterson testPeterson.o peterson.o

testPeterson.o: testPeterson.c
    gcc -c testPeterson.c

peterson.o: peterson.c
    gcc -c peterson.c

clean:
    rm -rf *.o testPeterson
```

บรรทัดต่อจาก target line ใช้ tab เสมอ ไม่งั้น make จะไม่ทำงาน คือไวยากรณ์สำหรับการ compile

option -c ของ gcc -c คือ compile ให้เป็น .o (object ไฟล์ ไม่ใช่ binary ไฟล์(ซึ่งเป็น platform dependent)) .o นอกจากจะอำนวยเรื่อง platform dependent แล้ว ยังเป็นการให้ผู้พัฒนาสามารถ distribute code โดยไม่ต้องเผย source code

(ต่อ) make เป็นเครื่องมือที่เหมือนกับ apt ปัจจุบัน กล่าวคือ เป็น installer ตัวอย่างวิธีการเรียก เป็นดังนี้

make -f makeFile.mk (ให้ทำตั้งแต่ all ซึ่งไปไม่ถึง clean)

หรือ make -f makeFile.mk clean (เรียกเฉพาะ target clean)

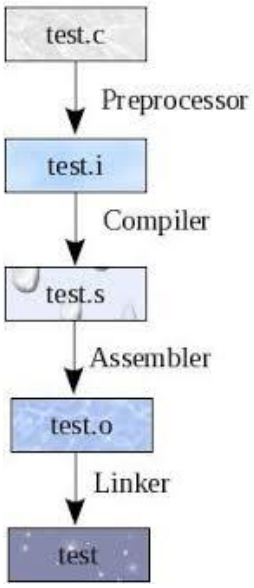
อนึ่ง หากใช้ .o นอกจากจะปกป้อง source code ยังร่นเวลาจาก การ compile .c เป็น binary ด้วย เพราะจะแปลงจาก .o เป็น binary ไม่ใช่ตั้งแต่ .c

นอกจากนี้ make จะดู timestamp ให้ กล่าวคือ หากพบว่า .o หรือ .c ไม่มีการเปลี่ยนแปลง code (timestamp ไม่เปลี่ยน make ก็จะแจ้งเราว่าไม่ต้องเสียเวลา compile ใดๆซ้ำของเก่าให้เสียเวลา (นึกภาพมี source ไฟล์เยอะสุดๆ)

ศึกษาเพิ่มเติม

https://computingblog.intakosum.net/2015/03/make.html

ติดตั้ง make สั่ง sudo apt install make



https://meenakshi02.wordpress.com/2012/08/22/c-programmes-compilation-process/

```c
int main(int  argc, char *argv[]) {
  int   shmID;  int status;
  int  *count;
  pid_t  pid;
  int NITER = 100;

  shmID = shmget(IPC_PRIVATE,
            sizeof(struct Memory), IPC_CREAT | 0666);

  count = (int *) shmat(shmID, (void *)0, 0);

  *count = 5;

  //initialize Peterson's algorithm shared memory
  initializePeterson();

  pid = fork();
  if (pid == 0) {
       for (int i = 0; i < NITER; i++)
            childProcess(count);
       exit(0);
  }
  for (int i = 0; i < NITER; i++)
       parentProcess(count);
  wait(&status);
  printf("The final value of count is %d\n", *count);

  //remove shared memory for Peterson's algorithm
  removePeterson();
  shmdt((void *) count);
  shmctl(shmID, IPC_RMID, NULL);
  exit(0);
}
void  parentProcess(int *count) {
     enterCriticalSection(0);
     int temp = *count;  temp++;
     /*sleep(rand() % 3); */ *count = temp;
     /* A */
}
void  childProcess(int *count) {
     /* B */
     int temp = *count;  temp--;
     /*sleep(rand() % 3); */ *count = temp;
     exitCriticalSection(1);
}
//for compilation see 1.3 gcc -o q1 f1.c f2.c
```

```c
#include  <stdio.h>
#include  <stdlib.h>
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>
#include <string.h>
#include <unistd.h>
#include "peterson.h" //**
#include <wait.h>
void  childProcess(int *);
void parentProcess(int *);
```

peterson.h > TRUE

```c
#define TRUE 1
#define FALSE 0
struct Memory {
    int  turn;
    int flag[2];
};
void initializePeterson();
void removePeterson();
void enterCriticalSection(int i);
int exitCriticalSection(int i);
```

1.5 โปรแกรม Lab9_q1.c เป็นดังนี้ parent จะคอยแย่ง increment ระหว่างที่ child จะคอยแบ่ยง decrement ค่า count คนละ NITER รอบ เนื่องจากเราประสานจังหวะการ increment และ decrement ด้วย peterson ทำให้โปรแกรมทำงานถูกต้อง

พึงสังเกตว่าเราอาจใช้ *count++ หรือ *count—เลยหรือไม่ คำตอบคือไม่ เพราะในหน่วยประมวลผลนั้น ยังต้อง load(&decode) execute and restore อยู่ กล่าวคือ increment operator มีขั้นตอนสั้นกว่า ดังนั้นการจำลองแบบ Load, Execute, and Restore เห็นผลการขาด mutual exclution ดีกว่า เรียกคุณสมบัติว่าขั้นตอนย่อยต้องเสมือนไม่ถูกขัดระว่างขั้นตอนย่อยต่างๆว่า atomicity

คำสั่ง ข้อ 1

1.1 ตอบ /* A */ 1.2 ตอบ /* B */

1.3 เขียน Lab9_q1.mk แบบสมบูรณ์ (ต้องแปลงเป็น .o ด้วย) ให้ได้โปรแกรมชื่อ goodCnt

กิจกรรมที่ 2 posix semaphore

```c
#include <stdio.h>      //for printf
#include <pthread.h>
#include <stdlib.h>     //for exit
#include <semaphore.h> //posix semaphore

#define NITER 100000 //100 x 1,000

pthread_attr_t attr[2];
pthread_t tid[2];
sem_t mySemaphore;
int cnt;
void * Count(void* a) {
  int i;
  for (i = 0; i < NITER; i++) {
    /* C */
    cnt++;
    sem_post(&mySemaphore);
  }
  void* lastSeen = malloc(sizeof(int));
  // *lastSeen now refers to an actual
  // piece of memory
  if (pthread_self() == tid[0]) {
    *(int *)lastSeen = cnt;
    printf("thr %lu exits. = ",
              pthread_self());
    printf(" lastSeen = %d\n",
              *(int *)lastSeen );
    pthread_exit( (void*)lastSeen );
  } //if tid[0]
}
int main() {
  /* D */
  pthread_attr_init(&attr[0]);
  pthread_attr_init(&attr[1]);
  pthread_create(&tid[0],&attr[0],
                  Count,NULL);
  pthread_create(&tid[1],&attr[1],
                  Count,NULL);
  void* returnVal;
  pthread_join(tid[0], &returnVal);
  int x = *(int*)returnVal;
  printf("Last cnt from tid[0] is %d\n",
                  x);

  pthread_join(tid[1],NULL);
  //NULL if not expecting return

  printf("final cnt = %d\n",cnt);
  sem_destroy(&mySemaphore);
  return 0;
}
```

ทำทั้งข้อ 1 และ 2 กำหนดส่ง (TBA)

2.1 โปรแกรม Lab10_q2.c เป็นดังนี้ tid[0] และ tid[1] จะแย่งกัน increment ค่า cnt threadละ หนึ่งแสนรอบ (เช่นเคยว่าเราคาดหวังการใช้ semaphore ซึ่งเป็น เครื่องมือพื้นฐาน ในภาษา c เพื่อใช้ในการจัดจังหวะ จะทำให้ได้คำตอบที่ถูกต้อง) หลักการของ semaphore ชัดเจนกว่า peterson ตรงที่มันเหมือนจำนวนกุญแจสำหรับไขเข้าห้องภาระกิจส่วนตัว ดังนั้นไม่ว่าใครล็อตที่บรรทัดที่ 160 ได้ อีกฝ่ายจะเข้าไม่ได้ (แทนที่จะให้ผู้เรียกคุมค่า i,j เหมือน peterson

อนึ่ง peterson ที่ใช้ข้อที่แล้ว ผู้พัฒนาได้ชี้แจงข้อมูลเพิ่มเติมว่ารองรับเฉพาะระบบที่เป็น single core ซึ่งไม่ใช่ wsl ที่เป็น multicore ดังนั้น อาจพบคำตอบที่พลาดได้ (เห็นความสำคัญของการใช้เครื่องมือที่พัฒนาตามเทคโนโลยีไหมครับ)

2.2 posix semaphore มี 4 function ดังนี้

2.2.1 int sem_init(sem_t *sem, int pshared, unsigned int value);
2.2.2 int sem_wait(sem_t *sem);
2.2.3 int sem_post(sem_t *sem);
2.2.4 int sem_destroy(sem_t *sem);

sem_init() มีไว้ instance ของ semaphore

sem_wait() คือการ acquire คือการลดค่าใน instance ของ semaphore (หากค่านั้นน้อยกว่า 0 thread ที่เรียก sem_wait() จะเข้า block state (ปล่อย resource ที่ตนครอบครอง เพื่อให้ระบบมีโอกาสทำงานต่อเพื่อเลี่ยง deadlock)

sem_post() คือการ release คือการเพิ่มค่าของ semaphore เพื่อไป wake thread ใดๆที่ติด block อยู่ (ปัจจุบันมีเครื่องมือที่ใหม่กว่า semaphore อีก)

2.3 สังเกต prototype ของ semaphore ว่ารับ pointer ดังนั้นตอนเรียกจริงต้องกำกับด้วย & เพื่อส่ง address ของ semaphore ไป (บรรทัดที่ 180 และ 470) ประเด็นคือ paremeter ตัวที่ 2 และ 3 ของ sem_init() คืออะไร คำตอบคือ pshared = 0 คือ shared semaphore ระหว่าง thread ส่วน value คือค่าเริ่มต้นของ semaphore (เท่าไหร่ดีสำหรับโจทย์นี้)

คำสั่ง ข้อ 2

2.1 ตอบ /* C */

2.2 ตอบ /* D */

2.3 เขียน output