Activity 6: Process Synchronization

	ชื่อ - นามสกุล	รหัสนิสิต
1	Sakdipat Sukhaneskul	6633239021
2	Sadit Wongprayon	6633233121

วัตถุประสงค์

1. เพื่อให้นิสิตเข้าใจหลักการของ process synchronization

2. เพื่อให้นิสิตสามารถเขียนโปรแกรมใช้งาน semaphore ได้

เตรียมตัว

1. ศึกษาหลักการ semaphore ในบทที่ 6 Process Synchronization

2. ศึกษา Linux POSIX named semaphore

ความรู้พื้นฐาน

Process Synchronization เป็นองค์ประกอบที่สำคัญในการทำงานร่วมกันของ Process หรือ Thread ซึ่งเครื่องมือใน Linux จะรองรับทั้งการทำ Semaphore และ Shared Memory

Semaphore เป็นตัวแปรประเภท counter ที่แสดงถึงสถานะของทรัพยากร โดยที่ counter แบบ semaphore จะมีลักษณะพิเศษคือ Operating System จะทำการดูแลไม่ให้เกิด race condition กล่าวคือ ผู้ใช้งานสามารถมั่นใจได้ว่า ณ เวลาใดเวลาหนึ่ง ค่าใน counter จะถูกแก้ไขได้โดยเพียง Process หรือ Thread เดียวเท่านั้น

Semaphore ใน Linux มี 2 ประเภทคือ Named Semaphore สำหรับรองรับการทำงาน ระหว่างหลายๆ Process (สร้างโดยคำสั่ง **sem_open**) และ Unnamed Semaphore สำหรับรองรับการทำงานของหลาย Thread ภายใน Process เดียวกัน (สร้างโดยคำสั่ง **sem_init**)

การทำงานของ Semaphore จะขึ้นอยู่กับค่าของ counter โดย Process หรือ Thread สามารถทำการลดค่าด้วยคำสั่ง **sem_wait** หรือเพิ่มค่าด้วยคำสั่ง **sem_post** ถ้า Process หรือ Thread พยายามจะลดค่าของ Semaphore ในขณะที่มีค่าเป็นศูนย์ Process หรือ Thread นั้นจะถูก block และจะต้องรอจนกว่าค่า Semaphore จะถูกเพิ่มจนมีค่ามากกว่าศูนย์ ซึ่งจะเกิดขึ้นได้ก็ต่อเมื่อมี Process หรือ Thread อื่นมาทำการเพิ่มค่า

หาก Semaphore มีค่าตั้งต้นเป็น 1 ค่าของมันจะสลับไปมาระหว่าง 0 กับ 1 เท่านั้น เรียกว่า binary semaphore ซึ่งสามารถใช้เป็น Mutex ซึ่งมีสถานะ Lock หรือ Unlock เพื่อควบคุมการเข้า critical Section ได้ โดยให้ semaphore มีค่าเริ่มต้นเป็น 1 (Unlock) คำสั่ง sem_wait() จะลดค่าของ semaphore เป็น 0 (Lock) เมื่อเข้า critical section และคำสั่ง sem_post() จะเพิ่มค่าของ semaphore กลับเป็น 1 (Unlock) เมื่อออกจาก critical section

หาก Semaphore มีค่าตั้งต้นมากกว่า 1 จะเรียกว่า counting semaphore ซึ่งใช้สำหรับนับการใช้ทรัพยากรที่มีมากกว่า 1 ชิ้นได้

รายละเอียดของ Semaphore ศึกษาเพิ่มเติมได้จาก

https://linux.die.net/man/7/sem_overview

Callcenter Simulation

ในกิจกรรมนี้จะให้นิสิตทำการปรับปรุง source code ของโปรแกรม Callcenter Simulator โดยให้นิสิตดาวน์โหลดไฟล์ simulation.zip ใน MyCourseVille ภายหลังจากการทำ unzip จะพบไฟล์ 4 ไฟล์ได้แก่

- makefile – สำหรับการใช้คำสั่ง make ในการ compile
- callcenter.c – เป็นโปรแกรมในส่วนของ server ที่จำลองระบบ call center ที่มีพนักงานให้บริการจำนวน n คน หรืออาจเรียกว่ามี n คู่สาย (n จะเป็นค่าที่ส่งผ่านทาง command line ไปยังตัวโปรแกรมเช่น ถ้า run ด้วยคำสั่ง callcenter 3 หมายถึงให้ทำการจำลองระบบ call center จำนวน 3 คู่สาย) โดยทำการสร้าง Named Semaphore “callcenter” พร้อมทั้งระบุค่าตั้งต้นของ Semaphore เป็น n
- customer.c – เป็นโปรแกรมที่จำลองลูกค้าหรือผู้ใช้บริการที่พยายามจะโทรเข้า callcenter โดยผู้โทรจะทำการติดต่อไปยัง callcenter เพื่อคุยกับพนักงาน (โดยการใช้ Named Semaphore ชื่อ “callcenter”) เมื่อมีพนักงานว่างมารับสายแล้ว ก็จะคุยเป็นระยะเวลาสุ่มระหว่าง 1-5 วินาที (สมมติว่าแทนเวลาจริง 1-5 นาที) หลังจากนั้นจะวางสาย โปรแกรม caller จะรอเป็นระยะเวลาสุ่มระหว่าง 1-3 วินาที ก่อนที่จะจำลองลูกค้าคนต่อไปที่จะโทรเข้า callcenter
- callcenter_rm.c – เป็นโปรแกรมที่ทำการยกเลิก Semaphore ที่ใช้ใน callcenter

ใน source code ของ callcenter.c และ customer.c ที่ได้รับจะมีรายละเอียดไม่ครบถ้วน กล่าวคือในส่วนคำสั่งที่เกี่ยวข้องกับ Semaphore ได้ถูกแทนที่ด้วย Comment ตัวอย่างเช่น ในไฟล์ customer.c บรรทัดที่ 16-18 จะมีข้อความ

```
//
// OS -- OPEN NAMED SEMAPHORE HERE
//
```

เป็นการระบุว่า ให้นำคำสั่งเกี่ยวกับการเปิด named semaphore มาแทนที่ comment นี้

สิ่งที่ต้องทำ

- ปรับปรุง source code โดยการเพิ่มคำสั่งเกี่ยวกับ Semaphore ที่เหมาะสม
- ใช้คำสั่ง make เพื่อคอมไพล์โปรแกรม ซึ่งจะได้ผลลัพธ์เป็นโปรแกรมสองโปรแกรมชื่อ callcenter และ customer
- ทำการทดสอบด้วยการรันโปรแกรม callcenter โดยมี argument เป็นจำนวนพนักงาน และรันโปรแกรม customer หลาย ๆ ครั้ง **แต่ละ customer อยู่คนละหน้าต่าง terminal กัน และให้มีจำนวน customer มากกว่าจำนวนคู่สาย**
- ตัวอย่างดังต่อไปนี้เป็นการให้ callcenter สร้างคู่สายจำนวน 2 คู่สาย และมี customer จำนวน 3 process ถ้าทำได้อย่างถูกต้อง ควรจะได้ผลลัพธ์ในลักษณะดังนี้

```
$ ./callcenter 2
Starting a call center with 2 agents.
There are 2 agents available now.
There are 2 agents available now.
There are 1 agents available now.
There are 0 agents available now.
There are 0 agents available now.
There are 0 agents available now.
There are 0 agents available now.
There are 0 agents available now.
There are 1 agents available now.
There are 0 agents available now.
There are 0 agents available now.
There are 0 agents available now.
There are 0 agents available now.
There are 0 agents available now.
There are 1 agents available now.
There are 0 agents available now.
...
```

```
$ ./customer
Starting customer
Do something else for 2 minutes

Next customer calls the call center, press ten buttons, and listens to silly music.
After waiting for 0 minutes, an agent accepts the call.  Talk for 4 minutes.
Customer ends the call.
Do something else for 1 minutes

Next customer calls the call center, press ten buttons, and listens to silly music.
After waiting for 0 minutes, an agent accepts the call.  Talk for 4 minutes.
Customer ends the call.
Do something else for 3 minutes

Next customer calls the call center, press ten buttons, and listens to silly music.
After waiting for 1 minutes, an agent accepts the call.  Talk for 4 minutes.
Customer ends the call.
Do something else for 2 minutes

Next customer calls the call center, press ten buttons, and listens to silly music.
After waiting for 1 minutes, an agent accepts the call.  Talk for 3 minutes.
Customer ends the call.
Do something else for 1 minutes
...
```

(customer อีก 2 process ให้ผลลัพธ์คล้ายๆกัน)

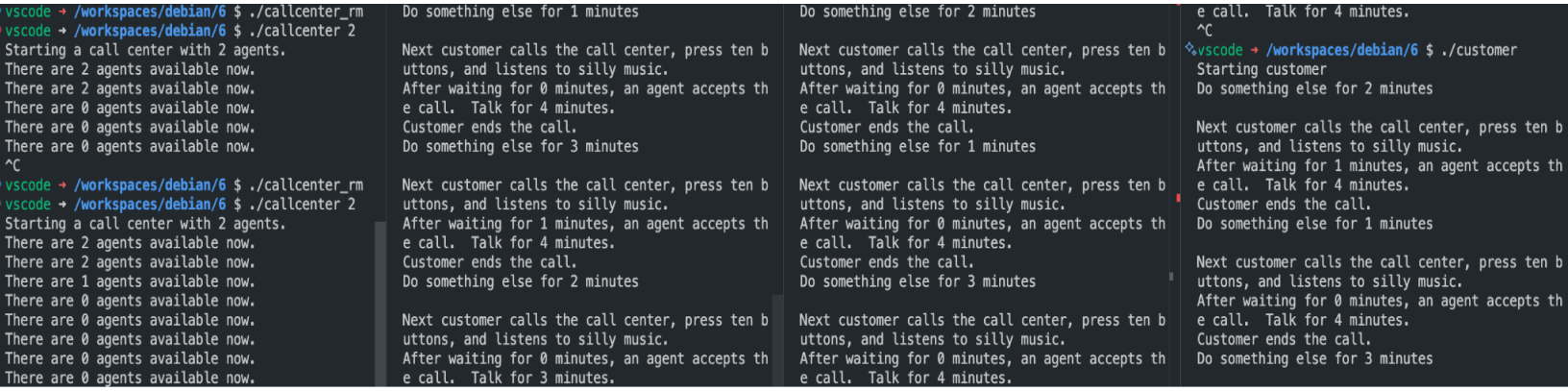
ให้ capture หน้าจอผลลัพธ์เก็บไว้

- ถ้าต้องการ reset ค่าของ semaphore ให้รันโปรแกรม ./callcenter_rm

<u>สิ่งที่ต้องส่งใน courseville</u>

1) souce code ที่ได้แก้แล้ว
2) ภาพหน้าจอผลลัพธ์

จะใส่สิ่งที่ต้องส่งโดยเพิ่มลงในไฟล์นี้ หรือส่งเป็นไฟล์แยกต่างหากก็ได้



source code

callcenter.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <semaphore.h>

int main(int argc, char **argv) {
   int num_agents = 2;
   if(argc > 1)
       num_agents = atoi(argv[1]);
   printf("Starting a call center with %d agents.\n", num_agents);

   //
   // OS -- CRAETE NAMED SEMAPHORE HERE
   sem_t *sem = sem_open("/callcenter", O_CREAT, 0644, num_agents);
   if (sem == SEM_FAILED) {
       perror("sem_open");
       exit(1);
   }

   int semval;
   while(1) {
       //
       // OS -- PLACE CURRENT VALUE OF SEMAPHORE IN 'semval' HERE
       //
       sem_getvalue(sem, &semval);

       printf("There are %d agents available now.\n", semval);
       sleep(3);
   }
}
```

customer.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <time.h>
#include <semaphore.h>

int rand_int(int n)
{
   // Generate random integer number between [1, n]
   int r = rand();
   return (r % n) + 1;
}
int main(int argc, char **argv)
{
   printf("Starting customer\n");
   //
   // OS -- OPEN NAMED SEMAPHORE HERE
   //
   sem_t *sem = sem_open("/callcenter", 0);
   if (sem == SEM_FAILED) {
       perror("sem_open");
       exit(1);
   }


   while (1)
   {
       // Customer will sleep between 1-3 seconds before placing the next phone call
       int sleep_time = rand_int(3);
       printf("Do something else for %d minutes\n\n", sleep_time);
       sleep(sleep_time);
       printf("Next customer calls the call center, press ten buttons, and listens to silly
music.\n");
       time_t t0 = time(NULL);
       // Wait for an agent

       //
       // OS -- LOCK SEMAPHORE HERE
       //
       sem_wait(sem);

       time_t t = time(NULL) - t0;
       // An agent accepts the call, using it for 3-5 seconds.
       int call_time = rand_int(3)+2;
```

```
        printf("After waiting for %ld minutes, an agent accepts the call.  Talk for %d
minutes.\n", t, call_time);
        sleep(call_time);
        // Customer hangs up the phone

        //
        // OS -- UNLOCK SEMAPHORE HERE
        //
        sem_post(sem);

        printf("Customer ends the call.\n");
    }
}
```