Lab 7: Process Synch. And Mutex(II)

1.Objective

- Understanding the deadlock and use some methods to avoid it

2.Syllabus

- Understand the concepts and principle of process deadlock
- Implement the process communication using C or C++

3.Prerequisite

- C or C++ language
- Computer which run Linux system (e.g. Ubuntu)
- Understand the concept of process deadlock

4.Concepts and Principles of Deadlock

Please refer to the chapter 6 of the text book. The main contents have been lectured in the sixth week, 11/14/2016. The slide of this chapter can be found in the course website: http://www.thinkmesh.net/ose/.

5.Experimental Contents

5.1 Producer-Consumer Problem (Semaphore)

Description: Implement a simple deadlock example. The file name is "**deadlock.c**".
Note: please modify the program to avoid deadlock. If it can run, show me your result. If not, tell me why.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

void *clean(char *name);

pthread_mutex_t brush, bucket;

int main(int argc, char *argv[])
{
	const int n = 2;
	pthread_t cleaners[n];
	char *names[] = {"Ivanon", "Petrov"};
	int i;
	pthread_mutex_init(&brush, NULL);
	pthread_mutex_init(&bucket, NULL);
	for(i = 0; i < n; i++) pthread_create(&cleaners[i], NULL, clean, (void
*)names[i]);
	for(i = 0; i < n; i++) pthread_join(cleaners[i], NULL);
	pthread_mutex_destroy(&brush);
```

```
    pthread_mutex_destroy(&bucket);
    return 0;
}

void *clean(char *name)
{
    printf("%s starts work...\n", name);
    usleep(rand() % 1000000);
    if(name[0] == 'I')
    {
        usleep(rand() % 1000000);
        pthread_mutex_lock(&bucket);
        printf("%s takes bucket...\n", name);
        usleep(rand() % 1000000);
        pthread_mutex_lock(&brush);
        printf("%s takes brush...\n", name);
    }
    else
    {
        usleep(rand() % 1000000);
        pthread_mutex_lock(&brush);
        printf("%s takes brush...\n", name);
        usleep(rand() % 1000000);
        pthread_mutex_lock(&bucket);
        printf("%s takes bucket...\n", name);
    }
    printf("%s have finished cleaning!\n", name);
    usleep(rand() % 1000000);
    pthread_mutex_unlock(&brush);
    printf("%s puts brush...\n", name);
    usleep(rand() % 1000000);
    pthread_mutex_unlock(&bucket);
    printf("%s puts bucket...\n", name);
    printf("%s have finished work!\n", name);
    pthread_exit(NULL);
}
```

Descriptions:

1. **pthread_mutex_init** : initialises the mutex referenced by mutex with attributes specified by attr.
2. **pthread_mutex_destroy** : destroy the mutex object referenced by mutex.
3. **pthread_mutex_lock** : locks the mutex object referenced by mutex
4. **pthread_mutex_unlock** : release the mutex object referenced by mutex

---

How to Run: gcc -o deadlock deadlock.c -lpthread

./deadlock

5.2 Deadlock Detection

Description: Implement the deadlock detection.The file name is "**detection.c**". **Note:** please modify the program inputing arguments, and consider detected processing and output results. If it can run, show me your result. If not, tell me why.

```
//detection.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

pthread_t sender;
pthread_t receiver;
char line[100];
char *owners[20];
char *requesters[20];
char *probes[3];
char *token;
char *procNum;
char *resource;
char *resourceOwner;
int i,status,ownersCounter,requesterCounter,fd;
bool isBlocked = false;
bool isDeadlocked = false;
void *senderThread();
void *receiverThread();
void findOwner();

int main(int argc, char *argv[])
{
	// Check arguments -> Display usage if incorrect arguments
	if (argc != 3){
		printf("Arguments: <file> <process #>\n");
		exit(1);
	}
	else{
		// Get process number argument
		procNum = argv[2];
		// Open configuration file
		FILE *file = fopen(argv[1],"r");
		if (file == 0){
			printf("Error opening file.\n");
		}else{
			// Opened file successfully.
			// Store configuration file
			while (fgets(line,100,file) != NULL){
				printf("Parsing Line: %s\n",line);
```

```
                    // If the line read contains owns, store it in owners array
                    if (strstr(line,"owns") != NULL){

                        token = strtok(line, " ");
                        while (token != NULL){
                            // Don't need to store 'owns'
                            if (!strcmp(token,"owns")){
                                token = strtok(NULL, " ");
                                continue;
                            }
                            // Allocate space and copy token into array
                            owners[ownersCounter] = malloc(strlen(token) + 1);
                            strcpy(owners[ownersCounter], token);
                            token = strtok(NULL, " ");
                            ownersCounter++;
                        }
                    }else{
                        // We have a request line, see if it is my own proces, block if
it is
                        if (strstr(line,procNum)){

                            token = strtok(line, " ");
                            while (token != NULL){
                                // Don't need to store 'requests'
                                if (!strcmp(token,"requests")){
                                    token = strtok(NULL, " ");
                                    continue;
                                }
                                // Allocate space and copy token into array
                                requesters[requesterCounter] = malloc(strlen(token)
+ 1);

                                strcpy(requesters[requesterCounter], token);
                                token = strtok(NULL, " ");
                                requesterCounter++;
                            }
                            isBlocked = true;
                        }

                    }
                }
                printf("**** Done reading configuration ****\n");
                fclose(file);

                // If process is blocked, find the owner of resource i'm blocked on and
form probe
                if (isBlocked){
                    findOwner();
                }
            }
        }
```

```
	// Create sender thread
	if ((status = pthread_create(&sender,NULL,senderThread,NULL)) != 0){
		fprintf (stderr, "Error creating thread - Status: %d: %s\n", status,
strerror(status));
		exit (1);
	}

	// Create receiver thread
	if ((status = pthread_create(&receiver,NULL,receiverThread,NULL)) != 0){
		fprintf (stderr, "Error creating thread - Status: %d: %s\n", status,
strerror(status));
		exit (1);
	}

	// Main thread - loop while not deadlocked
	while (!isDeadlocked) {
		// do nothing while not deadlocked
	}

	printf("**** System is deadlocked ****\n");
	return 1;
}

/*************************************************************
  * Sending thread, sends probes every 10 seconds.
  *************************************************************/
void *senderThread(){
	while (isBlocked){
		// Sends a Probe to the process owning the resource it is blocked on
		fd = open(resourceOwner, O_WRONLY);
		printf("%s is Writing %s:%s:%s\n",procNum,probes[0], probes[1],
probes[2]);
		write(fd, probes[0], sizeof(char) * 2);
		write(fd, probes[1], sizeof(char) * 2);
		write(fd, probes[2], sizeof(char) * 2);
		close(fd);
		printf("%s wrote to the pipe.\n",procNum);
		sleep(10);
	}
	return NULL;
}

/*************************************************************
  * Receiving thread, constantly looking for a probe
  * that wants to contact the process.
  *************************************************************/
void *receiverThread(){
	while (1){
		// Listen for probe - Look for any pipe with a name equivalent to process
number
```

```
            // this means there is a process that wishes to contact me
            fd = open(procNum, O_RDONLY);
            if (fd < 0){
                // There is no pipe with my name. Noone wants to contact me
            }else{
                if (isBlocked){
                    read(fd,probes[0],sizeof(char) * 2);
                    read(fd,probes[1],sizeof(char) * 2);
                    read(fd,probes[2],sizeof(char) * 2);
                    printf("%s has
read %s:%s:%s\n",procNum,probes[0],probes[1],probes[2]);
                    // Check probe blocked and receiving process. If same =
deadlocked
                    if (!strcmp(probes[0],probes[2])){
                        printf("%s detected deadlock because %s
= %s\n",procNum,probes[0],probes[2]);
                        close (fd);
                        unlink (procNum);
                        isDeadlocked = true;
                    }
                    strcpy(probes[1],procNum);
                    strcpy(probes[2],resourceOwner);
                }else{
                    printf("%s is not blocked and received a probe.
discarding.\n",procNum);
                }
            }
            sleep(5);
        }
        return NULL;
}

/*********************************************************
  * Called when the process is blocking. Find the owner
  * of the resource the process is blocked on, and forms
  * the probe.
  *********************************************************/
void findOwner(){
        printf("***** %s is blocking *****\n",procNum);
        // Search through requesters. Find the resource i'm blocked on
        while (1){
            if (!strcmp(requesters[i],procNum)){
                resource = requesters[i+1];
                break;
            }
            i++;
        }
        i = 0;
        // Search through owners. Find the owner of the resource i'm blocked on
        while (1){
```

```
            if (!strcmp(owners[i],resource)){
                resourceOwner = owners[i-1];
                break;
            }
            i++;
        }
        // Create pipe/file
        mkfifo(resourceOwner,0666);
        printf("%s created the pipe %s\n",procNum,resourceOwner);
        // Form probe for blocked process
        probes[0] = malloc(strlen(procNum) + 1);
        strcpy(probes[0], procNum);
        probes[1] = malloc(strlen(procNum) +1);
        strcpy(probes[1], procNum);
        probes[2] = malloc(strlen(resourceOwner) + 1);
        strcpy(probes[2], resourceOwner);
        i = 0;
}
```

//state1.data

P1 owns r2
P1 requests r1
P2 owns r1
P2 owns r3
P2 requests r4
P3 owns r4

//state2.data

P1 owns r2
P1 requests r1
P2 owns r1
P2 owns r3
P2 requests r4
P3 owns r4
P3 requests r2

Function interpret:

1. **pthread_create:** create a new thread.

How to Run: gcc -o detection detection.c -lpthread
./detection state1.data 3

5.3 Banker Algorithm

Description: The Banker's algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra.

Implement the program.The file name is "**banker.c**". **Note:** please consider and modify resources. If it can run, show me your result. If not, tell me why.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define resourceTypeQuan 3
#define processQuan 5

int i = 0;//Switch on C99 mode or we cannot initialize variable in for loop
int j = 0;
pthread_mutex_t mutex;//mutex lock for access to global variable

int initResourceVector [resourceTypeQuan];
//available, max, allocation, need
int availResourceVector [resourceTypeQuan];
int allocMatrix [processQuan][resourceTypeQuan] =
{{1,1,0},{1,3,0},{0,0,2},{0,1,1},{0,2,0}};
int maxMatrix [processQuan][resourceTypeQuan] =
{{5,5,5},{3,3,6},{3,5,3},{7,1,4},{7,2,2}};
int needMatrix [processQuan][resourceTypeQuan];

int requestResource(int processID,int requestVector[]);
int releaseResource(int processID,int releaseVector[]);
int ifGreaterThanNeed(int processID,int requestVector[]);
int ifEnoughToRelease(int processID,int releaseVector[]);
int ifInSafeMode();
int ifEnoughToAlloc();
void printNeedMatrix();
void printAllocMatrix();
void printAvailable();
void printReqOrRelVector(int vec[]);
void *customer(void* customerID);

int main(int argc, char const *argv[])
{
    if(argc != resourceTypeQuan + 1)
    {
        printf("Quantity of parameter is not correct.\n");
        return -1;
    }
    for(i = 0; i < resourceTypeQuan; i++)
    {
```

```
            initResourceVector[i] = atoi(argv[i+1]);//argv[0] is name of program
            availResourceVector[i] = initResourceVector[i];
        }

        //initialize needMatrix
        for (i = 0; i < processQuan; ++i)
        {
            for (j = 0; j < resourceTypeQuan; ++j)
            {
                needMatrix[i][j] = maxMatrix[i][j] - allocMatrix[i][j];
            }
        }

        printf("Available resources vector is:\n");
        printAvailable();

        printf("Initial allocation matrix is:\n");
        printAllocMatrix();

        printf("Initial need matrix is:\n");
        printNeedMatrix();

        pthread_mutex_init(&mutex,NULL);//declared at head of code
        pthread_attr_t attrDefault;
        pthread_attr_init(&attrDefault);
        pthread_t *tid = malloc(sizeof(pthread_t) * processQuan);
        int *pid = malloc(sizeof(int) * processQuan);//customer's ID, for banker's
algorithm, not pthread
        //initialize pid and create threads
        for(i = 0; i < processQuan; i++)
        {
            *(pid + i) = i;
            pthread_create((tid+i), &attrDefault, customer, (pid+i));
        }
        //join threads
        for(i = 0; i < processQuan; i++)
        {
            pthread_join(*(tid+i),NULL);
        }
        return 0;
}

void *customer(void* customerID)
{
```

```
int processID = *(int*)customerID;

while(1)
{
	//request random number of resources
	sleep(1);
	int requestVector[resourceTypeQuan];

	//Because i is global variable, we should lock from here
	//lock mutex for accessing global variable and printf
	pthread_mutex_lock(&mutex);
	//initialize requestVector
	for(i = 0; i < resourceTypeQuan; i++)
	{
		if(needMatrix[processID][i] != 0)
		{
			requestVector[i] = rand() % needMatrix[processID][i];
		}
		else
		{
			requestVector[i] = 0;
		}
	}


	printf("Customer %d is trying to request resources:\n",processID);
	printReqOrRelVector(requestVector);
	//requestResource() will still return -1 when it fail and return 0 when
succeed in allocate, like textbook says
	//altough I put the error message output part in the requestResource
function
	requestResource(processID,requestVector);
	//unlock
	pthread_mutex_unlock(&mutex);

	//release random number of resources
	sleep(1);
	int releaseVector[resourceTypeQuan];
	//Because i is global variable, we should lock from here
	//lock mutex for accessing global variable and printf
	pthread_mutex_lock(&mutex);
	//initialize releaseVector
	for(i = 0; i < resourceTypeQuan; i++)
	{
```

```
                if(allocMatrix[processID][i] != 0)
                {
                        releaseVector[i] = rand() % allocMatrix[processID][i];
                }
                else
                {
                        releaseVector[i] = 0;
                }
        }
        printf("Customer %d is trying to release resources:\n",processID);
        printReqOrRelVector(releaseVector);
        //releaseResource() will still return -1 when it fail and return 0 when
succeed in allocate, like textbook says
        //altough I put the error message output part in the releaseResource
function
        releaseResource(processID,releaseVector);
        //unlock
        pthread_mutex_unlock(&mutex);
    }
}

int requestResource(int processID,int requestVector[])
{
    //whether request number of resources is greater than needed
    if (ifGreaterThanNeed(processID,requestVector) == -1)
    {
        printf("requested resources is bigger than needed.\n");
        return -1;
    }
    printf("Requested resources are not more than needed.\nPretend to allocate...\n");

    //whether request number of resources is greater than needed
    if(ifEnoughToAlloc(requestVector) == -1)
    {
        printf("There is not enough resources for this process.\n");
        return -1;
    }

    //pretend allocated
    for (i = 0; i < resourceTypeQuan; ++i)
    {
        needMatrix[processID][i] -= requestVector[i];
        allocMatrix[processID][i] += requestVector[i];
        availResourceVector[i] -= requestVector[i];
```

```
        }
        printf("Checking if it is still safe...\n");

        //check if still in safe status
        if (ifInSafeMode() == 0)
        {
                printf("Safe. Allocated successfully.\nNow available resources vector
is:\n");
                printAvailable();
                printf("Now allocated matrix is:\n");
                printAllocMatrix();
                printf("Now need matrix is:\n");
                printNeedMatrix();
                return 0;
        }
        else
        {
                printf("It is not safe. Rolling back.\n");
                for (i = 0; i < resourceTypeQuan; ++i)
                {
                        needMatrix[processID][i] += requestVector[i];
                        allocMatrix[processID][i] -= requestVector[i];
                        availResourceVector[i] += requestVector[i];
                }
                printf("Rolled back successfully.\n");
                return -1;
        }
}

int releaseResource(int processID,int releaseVector[])
{
        if(ifEnoughToRelease(processID,releaseVector) == -1)
        {
                printf("The process do not own enough resources to release.\n");
                return -1;
        }

        //enough to release
        for(i = 0; i < resourceTypeQuan; i++)
        {
                allocMatrix[processID][i] -= releaseVector[i];
                needMatrix[processID][i] += releaseVector[i];
                availResourceVector[i] += releaseVector[i];
        }
```

```
        printf("Release successfully.\nNow available resources vector is:\n");
        printAvailable();
        printf("Now allocated matrix is:\n");
        printAllocMatrix();
        printf("Now need matrix is:\n");
        printNeedMatrix();
        return 0;
}

int ifEnoughToRelease(int processID,int releaseVector[])
{
        for (i = 0; i < resourceTypeQuan; ++i)
        {
                if (releaseVector[i] <= allocMatrix[processID][i])
                {
                        continue;
                }
                else
                {
                        return -1;
                }
        }
        return 0;
}
int ifGreaterThanNeed(int processID,int requestVector[])
{
        for (i = 0; i < resourceTypeQuan; ++i)
        {
                if (requestVector[i] <= needMatrix[processID][i])
                {
                        continue;
                }
                else
                {
                        return -1;
                }
        }
        return 0;
}

int ifEnoughToAlloc(int requestVector[])
{
        //first element of requestVector is processID
        for (i = 0; i < resourceTypeQuan; ++i)
```

```
        {
            if (requestVector[i] <= availResourceVector[i])
            {
                continue;
            }
            else
            {
                return -1;
            }
        }
        return 0;
}

void printNeedMatrix()
{
        for (i = 0; i < processQuan; ++i)
        {
            printf("{ ");
            for (j = 0; j < resourceTypeQuan; ++j)
            {
                printf("%d, ", needMatrix[i][j]);
            }
            printf("}\n");
        }
        return;
}

void printAllocMatrix()
{
        for (i = 0; i < processQuan; ++i)
        {
            printf("{ ");
            for (j = 0; j < resourceTypeQuan; ++j)
            {
                printf("%d, ", allocMatrix[i][j]);
            }
            printf("}\n");
        }
        return;
}

void printAvailable()
{
        for (i = 0; i < resourceTypeQuan; ++i)
```

```
	{
		printf("%d, ",availResourceVector[i]);
	}
	printf("\n");
	return;
}

void printReqOrRelVector(int vec[])
{
	for (i = 0; i < resourceTypeQuan; ++i)
	{
		printf("%d, ",vec[i]);
	}
	printf("\n");
	return;
}
int ifInSafeMode()
{
	int ifFinish[processQuan] = {0};//there is no bool type in old C
	int work[resourceTypeQuan];//temporary available resources vector
	for(i = 0; i < resourceTypeQuan; i++)
	{
		work[i] = availResourceVector[i];
	}
	int k;
	for(i = 0; i < processQuan; i++)
	{
		if (ifFinish[i] == 0)
		{
			for(j = 0; j < resourceTypeQuan; j++)
			{
				if(needMatrix[i][j] <= work[j])
				{
					if(j == resourceTypeQuan - 1)//means we checked whole
vector, so this process can execute
					{
						ifFinish[i] = 1;
						for (k = 0; k < resourceTypeQuan; ++k)
						{
							work[k] += allocMatrix[i][k];
							//execute and release resources
						}
						//if we break here, it will not check all process, so we
should reset i to let it check from beginning
```

```
                                    //If we cannot find any runnable process from beginning
to the end in i loop, we can determine that
                                    //there is no any runnable process, but we cannot know
if we do not reset i.
                                    i = -1;//at the end of this loop, i++, so -1++ = 0
                                    break;//in loop j, break to loop i and check next
runnable process
                               }
                               else//not finished checking all resource, but this kind
resources is enough
                               {
                                    continue;
                               }
                          }
                          else//resources not enough, break to loop i for next process
                          {
                               //because there is no change happened, so we do not need to
reset i in this condition.
                               break;
                          }
                     }
               }
               else
               {
                    continue;
               }
          }
          //there are two condition if we finish loop i
          //1. there is no process can run in this condition.
          //2. all processes are runned, which means it is in safe status.
          for(i = 0; i < processQuan; i++)
          {
               if (ifFinish[i] == 0)
               {
                    //not all processes are runned, so it is condition 1.
                    return -1;
               }
               else
               {
                    continue;
               }
          }
          //finished loop, so it is condition 2
          return 0;
```

}

How to Run: gcc -o banker banker.c -lpthread
./banker 5 5 5

6.Conclusion

In this chapter, three experiments have been complete. They have shown how to avoid the processes deadlock.