Cs 405 –lab manual 2

6. Producer-Consumer Problem using Semaphores

#include <pthread.h>

#include <semaphore.h>

#include <stdlib.h>

#include <stdio.h>

#define BUFFER\_SIZE 10

int buffer[BUFFER\_SIZE];

int in = 0;

int out = 0;

sem\_t empty;

sem\_t full;

pthread\_mutex\_t mutex;

void \*producer(void \*param)

{

int item;

for (int i = 0; i < 20; i++)

{

item = rand() % 100;

sem\_wait(&empty);

pthread\_mutex\_lock(&mutex);

buffer[in] = item;

in = (in + 1) % BUFFER\_SIZE;

printf("Producer produced %d\n", item);

pthread\_mutex\_unlock(&mutex);

sem\_post(&full);

}

}

void \*consumer(void \*param)

{

int item;

for (int i = 0; i < 20; i++)

{

sem\_wait(&full);

pthread\_mutex\_lock(&mutex);

item = buffer[out];

out = (out + 1) % BUFFER\_SIZE;

printf("Consumer consumed %d\n", item);

pthread\_mutex\_unlock(&mutex);

sem\_post(&empty);

}

}

int main()

{

pthread\_t tid1, tid2;

pthread\_mutex\_init(&mutex, NULL);

sem\_init(&empty, 0, BUFFER\_SIZE);

sem\_init(&full, 0, 0);

pthread\_create(&tid1, NULL, producer, NULL);

pthread\_create(&tid2, NULL, consumer, NULL);

pthread\_join(tid1, NULL);

pthread\_join(tid2, NULL);

pthread\_mutex\_destroy(&mutex);

sem\_destroy(&empty);

sem\_destroy(&full);

return 0;

}



7. Implementation of classical inter process communication problem(Reader Writers).

#include <windows.h>

#include <stdio.h>

HANDLE mutexReadCount;

HANDLE mutexResource;

int readCount = 0;

DWORD WINAPI reader(LPVOID Param)

{

int readerId = \*((int \*)Param);

WaitForSingleObject(mutexReadCount, INFINITE);

readCount++;

if (readCount == 1)

{

WaitForSingleObject(mutexResource, INFINITE);

}

ReleaseMutex(mutexReadCount);

printf("Reader %d is reading\n", readerId);

Sleep(1000);

printf("Reader %d finished reading\n", readerId);

WaitForSingleObject(mutexReadCount, INFINITE);

readCount--;

if (readCount == 0)

{

ReleaseMutex(mutexResource);

}

ReleaseMutex(mutexReadCount);

return 0;

}

DWORD WINAPI writer(LPVOID Param)

{

int writerId = \*((int \*)Param);

printf("Writer %d is trying to write\n", writerId);

WaitForSingleObject(mutexResource, INFINITE);

printf("Writer %d is writing\n", writerId);

Sleep(2000); // Simulate writing time

printf("Writer %d finished writing\n", writerId);

ReleaseMutex(mutexResource);

return 0;

}

int main()

{

HANDLE hThreads[15];

int ids[15];

mutexReadCount = CreateMutex(NULL, FALSE, NULL);

mutexResource = CreateMutex(NULL, FALSE, NULL);

for (int i = 0; i < 10; i++)

{

ids[i] = i + 1;

hThreads[i] = CreateThread(NULL, 0, reader, &ids[i], 0, NULL);

}

for (int i = 10; i < 15; i++)

{

ids[i] = i - 9;

hThreads[i] = CreateThread(NULL, 0, writer, &ids[i], 0, NULL);

}

WaitForMultipleObjects(15, hThreads, TRUE, INFINITE);

for (int i = 0; i < 15; i++)

{

CloseHandle(hThreads[i]);

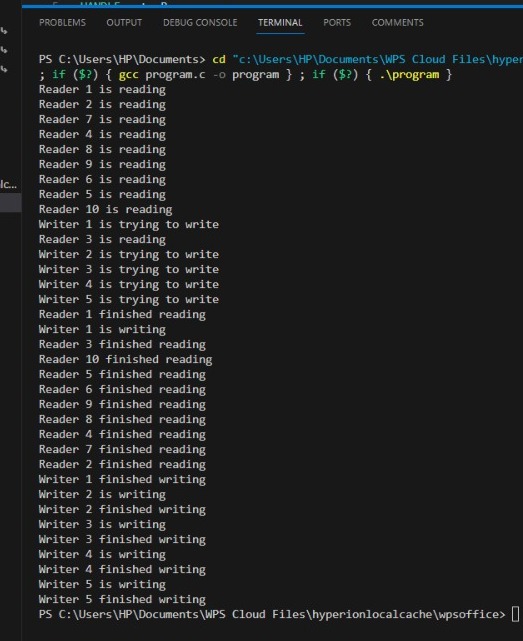
}

CloseHandle(mutexReadCount);

CloseHandle(mutexResource);

return 0;

}



8. Implementation classical inter process communication problem (Dining\_Philosophers).

#include <windows.h>

#include <stdio.h>

#define NUM\_PHILOSOPHERS 5

HANDLE forks[NUM\_PHILOSOPHERS];

HANDLE philosophers[NUM\_PHILOSOPHERS];

DWORD WINAPI philosopher(LPVOID Param);

void take\_forks(int philosopher, int left, int right);

void put\_forks(int philosopher, int left, int right);

DWORD WINAPI philosopher(LPVOID Param)

{

    int id = \*(int \*)Param;

    int left\_fork = id;

    int right\_fork = (id + 1) % NUM\_PHILOSOPHERS;

    // Reverse order for the last philosopher to avoid deadlock

    if (id == NUM\_PHILOSOPHERS - 1)

    {

        left\_fork = right\_fork;

        right\_fork = id;

    }

    while (1)

    {

        printf("Philosopher %d is thinking.\n", id);

        Sleep(1000); // Simulate thinking

        take\_forks(id, left\_fork, right\_fork);

        printf("Philosopher %d is eating.\n", id);

        Sleep(1000); // Simulate eating

        put\_forks(id, left\_fork, right\_fork);

    }

    return 0;

}

void take\_forks(int philosopher, int left, int right)

{

    WaitForSingleObject(forks[left], INFINITE);

    printf("Philosopher %d took the left fork %d.\n", philosopher, left);

    WaitForSingleObject(forks[right], INFINITE);

    printf("Philosopher %d took the right fork %d.\n", philosopher, right);

}

void put\_forks(int philosopher, int left, int right)

{

    ReleaseMutex(forks[left]);

    ReleaseMutex(forks[right]);

    printf("Philosopher %d put down both forks %d and %d.\n", philosopher, left, right);

}

int main()

{

    int ids[NUM\_PHILOSOPHERS];

    DWORD ThreadId;

    for (int i = 0; i < NUM\_PHILOSOPHERS; i++)

    {

        forks[i] = CreateMutex(NULL, FALSE, NULL);

    }

    for (int i = 0; i < NUM\_PHILOSOPHERS; i++)

    {

        ids[i] = i;

        philosophers[i] = CreateThread(NULL, 0, philosopher, &ids[i], 0, &ThreadId);

    }

    WaitForMultipleObjects(NUM\_PHILOSOPHERS, philosophers, TRUE, INFINITE);

    for (int i = 0; i < NUM\_PHILOSOPHERS; i++)

    {

        CloseHandle(philosophers[i]);

        CloseHandle(forks[i]);

    }

    return 0;

}



9. Write a program to implement & Compare various page replacement algorithm.

#include <stdio.h>

#include <stdlib.h>

#define MAX\_FRAMES 10

#define MAX\_PAGES 50

int pages[MAX\_PAGES];

int pageFrames[MAX\_FRAMES];

int total\_pages, frames;

// Prototypes

void FIFO();

void LRU();

void OPTIMAL();

int findPage(int page, int pageFrames[], int frames)

{

    for (int i = 0; i < frames; i++)

    {

        if (pageFrames[i] == page)

        {

            return i;

        }

    }

    return -1;

}

void initialize()

{

    for (int i = 0; i < frames; i++)

    {

        pageFrames[i] = -1; // initialize page frames to -1 indicating they're empty

    }

}

void printPageFrames()

{

    for (int i = 0; i < frames; i++)

    {

        if (pageFrames[i] != -1)

        {

            printf("%d ", pageFrames[i]);

        }

        else

        {

            printf("-- ");

        }

    }

    printf("\n");

}

void FIFO()

{

    int next\_replace = 0;

    int page\_faults = 0;

    initialize();

    for (int i = 0; i < total\_pages; i++)

    {

        if (findPage(pages[i], pageFrames, frames) == -1)

        {

            pageFrames[next\_replace] = pages[i];

            next\_replace = (next\_replace + 1) % frames;

            page\_faults++;

            printPageFrames();

        }

    }

    printf("FIFO - Total Page Faults: %d\n", page\_faults);

}

void LRU()

{

    int page\_faults = 0;

    int used[MAX\_FRAMES] = {0}; // Used to keep track of usage

    initialize();

    for (int i = 0; i < total\_pages; i++)

    {

        int found = findPage(pages[i], pageFrames, frames);

        if (found == -1)

        { // Page fault

            int lru = 0;

            for (int j = 1; j < frames; j++)

            {

                if (used[j] < used[lru])

                {

                    lru = j;

                }

            }

            pageFrames[lru] = pages[i];

            page\_faults++;

            printPageFrames();

        }

        for (int k = 0; k < frames; k++)

        {

            used[k]++; // Increment the use time of all frames

        }

        used[found] = 0; // Reset the frame that was just referenced

    }

    printf("LRU - Total Page Faults: %d\n", page\_faults);

}

void OPTIMAL()

{

    int page\_faults = 0;

    initialize();

    for (int i = 0; i < total\_pages; i++)

    {

        if (findPage(pages[i], pageFrames, frames) == -1)

        { // Page fault

            int opt\_index = -1, farthest = i;

            for (int j = 0; j < frames; j++)

            {

                int k;

                for (k = i + 1; k < total\_pages; k++)

                {

                    if (pageFrames[j] == pages[k])

                    {

                        if (k > farthest)

                        {

                            farthest = k;

                            opt\_index = j;

                        }

                        break;

                    }

                }

                if (k == total\_pages)

                { // Not found in the future

                    opt\_index = j;

                    break;

                }

            }

            if (opt\_index == -1)

                opt\_index = 0; // If all are equally likely

            pageFrames[opt\_index] = pages[i];

            page\_faults++;

            printPageFrames();

        }

    }

    printf("Optimal - Total Page Faults: %d\n", page\_faults);

}

int main()

{

    printf("Enter number of frames: ");

    scanf("%d", &frames);

    printf("Enter number of pages: ");

    scanf("%d", &total\_pages);

    printf("Enter page reference string:\n");

    for (int i = 0; i < total\_pages; i++)

    {

        scanf("%d", &pages[i]);

    }

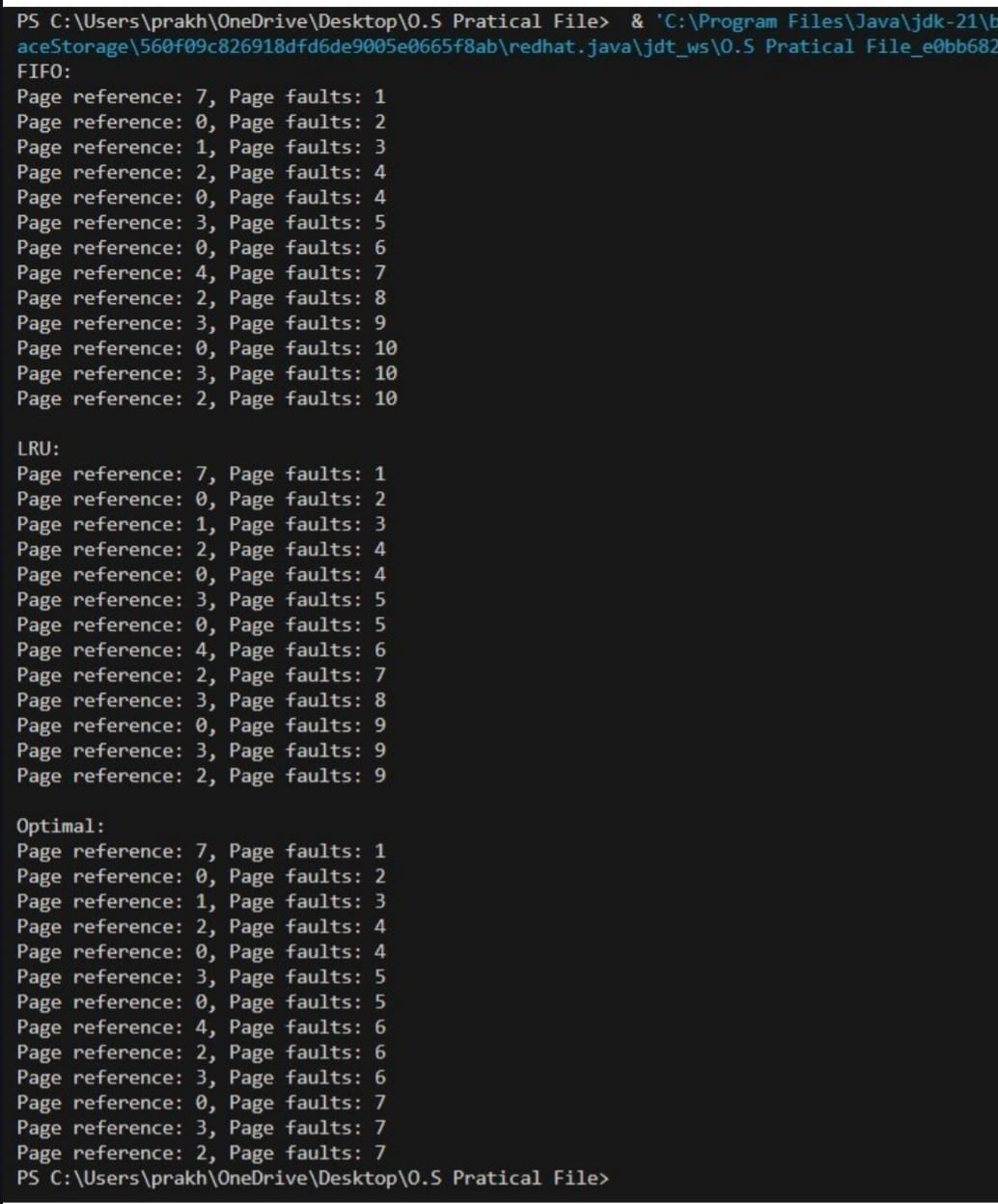
    FIFO();

    LRU();

    OPTIMAL();

    return 0;

}



10. Write a program to implement & Compare various Disk & Drum scheduling Algorithms

#include <stdio.h>

#include <stdlib.h>

#define MAX\_FRAMES

#define MAX\_PAGES

int pages[MAX\_PAGES];

int pageFrames[MAX\_FRAMES];

int total\_pages, frames;

// Prototypes

void FIFO();

void LRU();

void OPTIMAL();

int findPage(int page, int pageFrames[], int frames);

int findPage(int page, int pageFrames[], int frames)

{

for (int i = 0; i < frames; i++)

{

if (pageFrames[i] == page)

{

return i;

}

}

return -1;

}

void initialize()

{

for (int i = 0; i < frames; i++)

{

pageFrames[i] = -1; // initialize page frames to -1 indicating they're empty

}

}

void printPageFrames()

{

for (int i = 0; i < frames; i++)

{

if (pageFrames[i] != -1)

{

printf("%d ", pageFrames[i]);

}

else

{

printf("-- ");

}

}

printf("\n");

}

void FIFO()

{

int next\_replace = 0;

int page\_faults = 0;

initialize();

for (int i = 0; i < total\_pages; i++)

{

if (findPage(pages[i], pageFrames, frames) == -1)

{

pageFrames[next\_replace] = pages[i];

next\_replace = (next\_replace + 1) % frames;

page\_faults++;

printPageFrames();

}

}

printf("FIFO - Total Page Faults: %d\n", page\_faults);

}

void LRU()

{

int page\_faults = 0;

int used[MAX\_FRAMES] = {0}; // Used to keep track of usage

initialize();

for (int i = 0; i < total\_pages; i++)

{

int found = findPage(pages[i], pageFrames, frames);

if (found == -1)

{ // Page fault

int lru = 0;

for (int j = 1; j < frames; j++)

{

if (used[j] < used[lru])

{

lru = j;

}

}

pageFrames[lru] = pages[i];

page\_faults++;

printPageFrames();

}

for (int k = 0; k < frames; k++)

{

used[k]++; // Increment the use time of all frames

}

if (found != -1)

{

used[found] = 0; // Reset the frame that was just referenced

}

}

printf("LRU - Total Page Faults: %d\n", page\_faults);

}

void OPTIMAL()

{

int page\_faults = 0;

initialize();

for (int i = 0; i < total\_pages; i++)

{

if (findPage(pages[i], pageFrames, frames) == -1)

{ // Page fault

int opt\_index = -1, farthest = i;

for (int j = 0; j < frames; j++)

{

int k;

for (k = i + 1; k < total\_pages; k++)

{

if (pageFrames[j] == pages[k])

{

if (k > farthest)

{

farthest = k;

opt\_index = j;

}

break;

}

}

if (k == total\_pages)

{ // Not found in the future

opt\_index = j;

break;

}

}

if (opt\_index == -1)

{

opt\_index = 0; // If all are equally likely

}

pageFrames[opt\_index] = pages[i];

page\_faults++;

printPageFrames();

}

}

printf("Optimal - Total Page Faults: %d\n", page\_faults);

}

int main()

{

printf("Enter number of frames: ");

scanf("%d", &frames);

printf("Enter number of pages: ");

scanf("%d", &total\_pages);

printf("Enter page reference string:\n");

for (int i = 0; i < total\_pages; i++)

{

scanf("%d", &pages[i]);

}

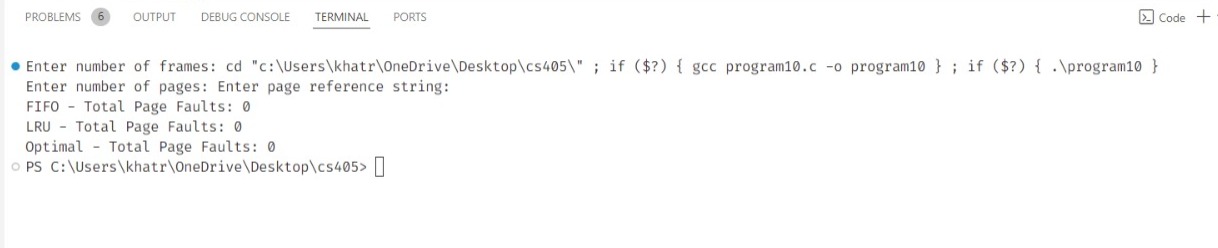
FIFO();

LRU();

OPTIMAL();

return 0;

}



11. Write a program to implement Banker’s algorithms.

#include <stdio.h>

#include <stdlib.h>

int main()

{

int n, m, i, j, k;

n = 5; // Number of processes

m = 3; // Number of resources

int alloc[5][3] = {{0, 1, 0}, // P0 // Allocation Matrix {2, 0, 0}, // P1

{3, 0, 2}, // P2

{2, 1, 1}, // P3

{0, 0, 2}}; // P4

int max[5][3] = {{7, 5, 3}, // P0 // Maximum Demand Matrix {3, 2, 2}, // P1

{9, 0, 2}, // P2

{2, 2, 2}, // P3

{4, 3, 3}}; // P4

int avail[3] = {3, 3, 2}; // Available Resources

int f[n], ans[n], ind = 0;

for (k = 0; k < n; k++)

{

f[k] = 0;

}

int need[n][m];

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

{

for (j = 0; j < m; j++)

need[i][j] = max[i][j] - alloc[i][j];

}

int y = 0;

for (k = 0; k < 5; k++)

{

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

{

if (f[i] == 0)

{

int flag = 0;

for (j = 0; j < m; j++)

{

if (need[i][j] > avail[j])

{

flag = 1;

break;

}

}

if (flag == 0)

{

ans[ind++] = i;

for (y = 0; y < m; y++)

avail[y] += alloc[i][y];

f[i] = 1;

}

}

}

}

printf("Following is the SAFE Sequence\n");

for (i = 0; i < n - 1; i++)

printf(" P%d ->", ans[i]);

printf(" P%d", ans[n - 1]);

return 0;

}



12. Write a program to implement Remote Procedure Call(RPC)

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

int main()

{

int client\_sock;

struct sockaddr\_in server\_addr;

int buffer[2], result;

ssize\_t sent\_bytes, recv\_bytes;

// Create socket

client\_sock = socket(AF\_INET, SOCK\_STREAM, 0);

if (client\_sock < 0)

{

perror("[-]Socket error");

exit(EXIT\_FAILURE);

}

printf("[+]Client socket created.\n");

// Configure settings

memset(&server\_addr, 0, sizeof(server\_addr));

server\_addr.sin\_family = AF\_INET;

server\_addr.sin\_port = htons(7799);

server\_addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

// Connect to server

if (connect(client\_sock, (struct sockaddr \*)&server\_addr, sizeof(server\_addr)) < 0)

{

perror("[-]Connect error");

close(client\_sock);

exit(EXIT\_FAILURE);

}

printf("[+]Connected to server.\n");

// Send data

buffer[0] = 10; // a

buffer[1] = 20; // b

sent\_bytes = send(client\_sock, buffer, sizeof(buffer), 0);

if (sent\_bytes < 0)

{

perror("[-]Send error");

close(client\_sock);

exit(EXIT\_FAILURE);

}

printf("[+]Data sent.\n");

// Receive result

recv\_bytes = recv(client\_sock, &result, sizeof(result), 0);

if (recv\_bytes < 0)

{

perror("[-]Receive error");

close(client\_sock);

exit(EXIT\_FAILURE);

}

else if (recv\_bytes == 0)

{

printf("[-]Server closed the connection unexpectedly.\n");

close(client\_sock);

exit(EXIT\_FAILURE);

}

printf("[+]Result received: %d\n", result);

// Close the socket

close(client\_sock);

printf("[+]Connection closed.\n");

return 0;

}



­­­­­­­­­­­­­­­­­­­­­­