George Suarez

CSE 460

Homework 4

1. Consider the following snapshot of a system:

Process	Allocation	Max	Available
	A B C D	A B C D	A B C D
P0	0 0 1 2	1 0 1 2	2 5 2 0
P1	1 0 0 0	1 7 5 0	
P2	1 3 5 4	2 3 5 6	
P3	0 6 3 2	0 6 5 2	
P4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm.

a. What is the content of the matrix **Need**?

Process	Max – Allocate	**Need**
P0	(1, 0, 1, 2,) – (0, 0, 1, 2) =	(1, 0, 0, 0)
P1	(1, 7, 5, 0) – (1, 0, 0, 0) =	(0, 7, 5, 0)
P2	(2, 3, 5, 6) – (1, 3, 5, 4) =	(1, 0, 0, 2)
P3	(0, 6, 5, 2) – (0, 6, 3, 2) =	(0, 0, 2, 0)
P4	(0, 6, 5, 6) – (0, 0, 1, 4) =	(0, 6, 4, 2)

b. Is the system in a safe state? Why?

To determine if the system is in a safe state, we need to check if we have enough resources to satisfy all processes because if there are not enough resources then the system is not safe since it will have a deadlock. To do this, we must check if all the numbers in the Need matrix are $\leq$ to the numbers under the Available column for a certain process. If they are, then we add Need and Available to produce a new Available for the next process.

- **P0:** All the numbers in the Need in this process (1, 0, 0, 0) is less than the corresponding numbers in Available (2, 5, 2, 0). **P0** can

now be executed, and we add Available and Allocation to produce a new Available vector for **P1** which is (0, 0, 1, 2) + (2, 5, 2, 0) = (2, 5, 3, 2).

- **P1:** Some of the numbers in Need are bigger than what is in its corresponding place in Available specifically B and C. So we skip this process for now and move on to **P2**.

- **P2:** Process **P1** was skipped so we still have the Available (2, 5, 3, 2), and the Need in this process (1, 0, 0, 2) is smaller than in the Available vector. This means that **P2** gets executed and Available and Allocation gets added to produce a new Available for **P3** which is (2, 5, 3, 2) + (1, 3, 5, 4) = (3, 8, 8, 6).

- **P3:** All the numbers in Need in this process (0, 0, 2, 0) is smaller than the corresponding numbers in Available (3, 8, 8, 6). **P3** gets executed and Available and Allocation gets added to produce the next Available for **P4** which is (3, 8, 8, 6) + (0, 6, 3, 2) = (3, 14, 11, 8).

- **P4:** All the numbers in Need in this process (0, 6, 4, 2) is smaller than the corresponding numbers in Available (3, 14, 11, 8). **P4** gets executed and Available and Allocation gets added to produce a new Available for **P1** which is (3, 14, 11, 8) + (0, 0, 1,4) = (3, 14, 12, 12).

- **Back to P1:** Now the numbers in Need (0 7 5 0) is smaller in Available (3 14 12 12). **P1** now executes which now all processes have been satisfied, and the system is in a safe state.

c. If a request from process P1 arrives for (0, 4, 2, 0), can the request be granted immediately?

If the order of execution is still the same as above (P0, P2, P3, P4, P1), then the request from process **P1** can be granted immediately since the Available resources for **P1** (3**,** 14, 12, 12) and the request of (0, 4, 2, 0) will produce a leftover of (3, 10, 10, 12).

2. Consider a swapping system in which memory consists of the following hole sizes in memory order: 16K, 14K, 4K, 20K, 18K, 7K, 9K, 12K, and 15K. Which hole is taken for successive segment requests of:

   (a) 12K
   (b) 10K
   (c) 9K

   for first fit? Now repeat the question for best fit, worst fit, and next fit.

- First Fit: This process starts at the beginning and places a process in the first hole encountered that is large enough to satisfy the request.

  (a) 12K
  16K – 12K = 4K

4K	14K	4K	20K	18K	7K	9K	12K	15K

  (b) 10K
  14K – 10K = 4K

4K	4K	4K	20K	18K	7K	9K	12K	15K

  (c) 9K
  20K – 9K = 11K

4K	14K	4K	11K	18K	7K	9K	12K	15K

- Best fit: Places a process in the smallest block of unallocated memory available

  (a) 12K
  12K – 12K = 0K

16K	14K	4K	20K	18K	7K	9K	0K	15K

  (b) 10K
  14K – 10K = 1K

16K	4K	4K	20K	18K	7K	9K	0K	15K

  (c) 9K
  9K – 9K = 0K

16K	4K	4K	20K	18K	7K	0K	0K	15K

- Worst fit: Places a process in the largest block of unallocated memory available.

  (a) 12K
  20K – 12K = 8K

16K	14K	4K	8K	18K	7K	9K	12K	15K

  (b) 10K
  18K – 10K = 8K

16K	14K	4K	8K	8K	7K	9K	12K	15K

  (c) 9K
  16K – 9K = 7K

7K	14K	4K	8K	8K	7K	9K	12K	15K

- Next fit: It is similar to how First Fit works, excepts it starts where it left off.

  (a) 12K
  16K – 12K = 4K

4K	14K	4K	20K	18K	7K	9K	12K	15K

  (b) 10K
  14K – 10K = 4K

4K	4K	4K	20K	18K	7K	9K	12K	15K

  (c) 9K
  20K – 9K = 11K

4K	4K	4K	11K	18K	7K	9K	12K	15K

3. Using the page table, give the physical address corresponding to each of the following virtual addresses. Explain briefly how you obtained your answers.

Physical address = (Starting address of physical frame * 1024) + offset
Offset = Virtual Address / 1024

(a) 20
Offset = (20 / 1024) = 0K + 20 = 20
20 in the virtual address space is located at the 0K-4K page which points to the 8K-12K frame in the physical memory address. Since 8K is the starting address of this frame, we get the following physical memory address:

(8 * 1024) + 20 = 8212

(b) 4100
Offset = (4100 / 1024) = 4K + 4 = 4
4100 is located at the 4K-8K page which points to the 4K-8K frame in the physical memory address. Since 4K is the starting address of this frame, we get the following physical memory address:

(4 * 1024) + 4100 = 8196

(c) 8300
Offset = (8300 / 1024) = 8K + 108 = 108
8300 is located at the 8K-12K page which points to the 24K-28K frame in the physical memory address. Since 24K is the starting address of this frame, we get the following physical memory address:

(24 * 1024) + 108 = 24684

4. In the class we mentioned briefly the readers-writers problem with **writers priority**. The problem can be solved in guarded commands as follows:

```
void writer()
{
  [writers++;]
  when ( (readers == 0) && (active_writers == 0) )[
    active_writers++;
  ]

  //write

  [writers--;  active_writers--;]
}
```

```
void reader()
{
  when ( writers == 0 ) [
```

```
    readers++;
  ]

  //read

  [readers--;]
}
```

Here *writers* represents the number of threads that are either writing or waiting to write. The variable *active_writers* represents the number of threads (0 or 1) that are currently writing.

Implement the solution using either POSIX threads or SDL threads. Again, simulate the tasks by reading from and writing to a file named *counter.txt* as in problem 4 of Homework 3.

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_thread.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <signal.h>
#include <unistd.h>
#include <iostream>
#include <fstream>

using namespace std;

SDL_bool condition = SDL_FALSE;
SDL_mutex *mutex1;
SDL_cond *readerQueue; //condition variable
SDL_cond *writerQueue; //condition variable

int readerCount = 0;
int writerCount = 0;
bool quit = false;
```

```
string fileName = "counter.txt";

int reader(void *data)
{
    while (!quit)
    {
        SDL_Delay(rand() % 3000);
        SDL_LockMutex(mutex1);
        while (!(writerCount == 0))
        {
            SDL_CondWait(readerQueue, mutex1);
        }

        readerCount++;

        SDL_UnlockMutex(mutex1);

        //read
        int count = -1;
        ifstream inFile;
        inFile.open(fileName.c_str());

        if (inFile.good())
        {
            inFile >> count;
            inFile.close();
        }

        SDL_LockMutex(mutex1);
        printf("\nThis is %s thread\n", (char *)data);
```

```
        if (--readerCount == 0)
        {
            SDL_CondSignal(writerQueue);
        }
        SDL_UnlockMutex(mutex1);
    }
}

int writer(void *data)
{
    while (!quit)
    {
        SDL_Delay(rand() % 3000);
        SDL_LockMutex(mutex1);
        while (!((readerCount == 0) && (writerCount == 0)))
        {
            SDL_CondWait(writerQueue, mutex1);
        }

        writerCount++;

        SDL_UnlockMutex(mutex1);

        int count = -1;
        ifstream inFile;
        inFile.open(fileName.c_str());
        if (inFile.good())
        {
            inFile >> count;
            inFile.close();
        }
```

```
        ofstream outFile;
        outFile.open(fileName.c_str());
        if (outFile.good())
        {
            outFile << count;
            outFile.close();
        }

        SDL_LockMutex(mutex1);
        printf("\nThis is %s thread %d\n", (char *)data, count);
        writerCount--; //only one writer at one time
        count++;
        SDL_CondSignal(writerQueue);
        SDL_CondBroadcast(readerQueue);
        SDL_UnlockMutex(mutex1);
    }
}

void func(int sig)
{
    quit = true;
}

const int READERS = 20;
const int WRITERS = 3;

int main()
{
    SDL_Thread *idr[20], *idw[3]; //thread identifiers
    char *readerNames[] = {(char *)"reader 1",
```

```
                        (char *)"reader 2",
                        (char *)"reader 3",
                        (char *)"reader 4",
                        (char *)"reader 5",
                        (char *)"reader 6",
                        (char *)"reader 7",
                        (char *)"reader 8",
                        (char *)"reader 9",
                        (char *)"reader 10",
                        (char *)"reader 11",
                        (char *)"reader 12",
                        (char *)"reader 13",
                        (char *)"reader 14",
                        (char *)"reader 15",
                        (char *)"reader 16",
                        (char *)"reader 17",
                        (char *)"reader 18",
                        (char *)"reader 19",
                        (char *)"reader 20"};

char *writerNames[] = {(char *)"writer 1", (char *)"writer 2", (char *)"writer 3"};

mutex1 = SDL_CreateMutex();
readerQueue = SDL_CreateCond();
writerQueue = SDL_CreateCond();

for (int i = 0; i < READERS; i++)
{
    idr[i] = SDL_CreateThread(reader, "Reader Thread", readerNames[i]);
}
```

```
    for (int i = 0; i < WRITERS; i++)
    {
        idw[i] = SDL_CreateThread(writer, "Writer Thread", writerNames[i]);
    }

    (void)signal(SIGINT, func);

    for (int i = 0; i < READERS; i++)
    {
        SDL_WaitThread(idr[i], NULL);
    }

    for (int i = 0; i < WRITERS; i++)
    {
        SDL_WaitThread(idw[i], NULL);
    }

    SDL_DestroyCond(readerQueue);
    SDL_DestroyCond(writerQueue);
    SDL_DestroyMutex(mutex1);
         return 0;
}
```

```
georgesuarez at MacBook-Pro in ~/University/CSE-460/Homework/Homework 4 on master*
$ ./sdl_readers_writers

This is writer 2 thread 12

This is reader 6 thread

This is writer 1 thread 12
```

This is reader 13 thread

This is reader 16 thread

This is reader 13 thread

This is reader 10 thread

This is reader 8 thread

This is reader 9 thread

This is reader 8 thread

This is reader 15 thread

This is reader 3 thread

This is writer 1 thread 12

This is reader 2 thread

This is reader 17 thread

This is writer 2 thread 12

This is reader 11 thread

This is reader 8 thread

This is reader 20 thread

This is reader 4 thread