Managing Resources

Zachary Owen

1 Memory Manager

This is a simple program that translates a 32-bit virtual address with a 4-KB page size to a Virtual Address. The program is passed a virtual address (in decimal) on the command line and outputs the page number and offset for the given address. As an example, the program would run as follows:

```
./memman 19986
```

The program outputs:

```
The address 19986 contains:
page number = 4
offset = 3602
```

The program can be built and run using the commands:

```
make
./memman <address>
```

1.1 Functionality: main.c

This is a short program, all of the code is in main.c

1.1.1 int main (int argc, char * argv[])

This function begins by checking *argc* for a value of at least 2, ensuring that at least one argument has been provided. Any furthur arguments are ignored. After this the value is passed to the **strtol()** function and then checked to see if an overflow has occured or if something non-number like has been passed in. Both will generate appropreate error messages.

Finally the upper 20 bits are extracted and printed as the page number, and the lower 12 bits are masked to obtainthe offset. Both are printed and the program exits.

2 The Bankers Algorithm

This program is a multithreaded program that implements the bankers algorithm discussed in Section 7.5.3. Several customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied. This programming combines three separate topics:

1. multithreading
2. preventing race conditions
3. deadlock avoidance.

2.1 Functionality: bankers.c

This is the start-point of the program. It contains all initialization functions. It sets up each Mutex and all of the shared static memory used by the other files.o

2.1.1 void init_all_the_things()

This function initalizes the mutex, seeds **srand()** and finally, seeds the *maximum, allocation* and *need* arrays. During testing it was noticed that equal distribution of process lead to large amount's of unsafe requests. In order to mitigate this issue the *maximum* array is seeded with the following equation

$$pmax_{i,j} = MAX_i - \sqrt{\textbf{rand()} - MAX_i^2}$$

Where :
$pmax$: is the maximum of i resource requred by j process
MAX: is the maximum avalable to the system of resource i
rand(): is the random function

This increases the prevelence of smaller numbers. Additionally the **isqrt()** function is called instead of the standard math library square root function since the return value would always need to be floored anyway.
The function also takes the program inputs, converts them into integers and then places them into the MAX array. It also checks for overflow and improper input.

2.1.2 int main (int argc, char * argv[])

This function is the start point of the program. It checks for the correct number of input variables, then calls the **init_all_the_things()** function and then spins up the required number of threads and waits for them to join back up and then exits.

2.2 Functionality: bank.c

This file contains no functions but serves the dumping ground for all of the global variables defines and mutex that the program uses.

- *NUMBER_OF_CUSTOMERS*: Define telling the number of customer threads for main to create
- *NUMBER_OF_RESOURCES*: Define telling main how many resorces to look for from user input.
- *CUSTOMER_GOES*: How many requests should each thread make before exiting
- *HEAD_SLEEP*: How long each thread sleeps between request
- *available* : Maximum available resourse, set in main.
- *maximum* : Maximum number of resources that a process will ask for.
- *allocation* : How many of each resource is currently allocated to each process.
- *need* : Diffrence between the *maximum* and *allocation* arrays
- *MAX_RESOURCE*: Maximum amount of resources in this simulation
- *READ* : The read-write mutex.

2.3 Functionality: customer.c

This section contains the breadth of the executing code, as well as all of the functions requrired by the threads and several helper functions that get called in **main()**.

2.3.1 void* customer(void *)

This funtion is the pthread function. It's execution is mostly handled in helper functions. Its execution can be described as followed:

1. Randomly create a resource request with **make_need()**
2. Make a system request for those resources with **request_resources()**
   (a) If you get it sit on those resources for a while, release them with **release_resources()**, and then start again.
   (b) If not wait for a bit, and go through the loop again without decrementing your counter.

2.3.2 int isqrt(int)

I have no idea how this function works, It's pretty sweet though and I found it here *https://en.wikipedia.org/wiki/Integer_square_root*

2.3.3 void make_need()

A simple **for** loop that generates needs in a similar way to how they are generated in **init_all_the_things()** using the following equation:

$$request_i = MAX_i - \sqrt{\mathbf{rand()} - MAX_i^2}$$

Where :
request: is the resource requred by the process
MAX: is the maximum avalable to the system of resource i
rand(): is the random function
This increases the prevelence of smaller numbers. Additionally the **isqrt()** function is called instead of the standard math library square root function since the return value would always need to be floored anyway.

2.3.4 void print_req (int)

Helper function called in **request_resources()** just prints out

```
Request P# <#, #, #>
```

2.3.5 int request_resources(int, int *)

This function fully handles the process of aquriring resouces. The first thing it does is aquire the lock on the *READ* mutex. Then it prints a message telling the user that it is attempting to make the request using **print_req ()**. Then the process uses **alloc_req()** to allocate the resources for itself, then it runs **safety_test()**. If the system is safe, it prints a message indicating as such, prints the current state of the system using **print_state()** and finally unlocks the mutex and returns with an integer 1 on successs and 0 on failure.

2.3.6 void release_resources(int, int *)

Aquires the *READ* lock and then reclaims the calling process's resources.

2.3.7 void print_state()

Long function that prints out the number of resources each process has, its maximum allowed resouces and the currently available:

```
        Allocation      Need     Available
          A B C       A B C       A B C
P0        0 1 0       7 4 3       3 3 2
P1        3 0 2       1 2 2
P2        3 0 2       6 0 0
P3        2 1 1       0 1 1
P4        0 0 2       4 3 1
```

2.3.8 void alloc_req(int, int*)

This function adds the request matrix to the process's allocation array while subtracting them from the process's need array and the global resouce pool.

2.3.9 void free_req(int, int*)

This function subtracts the request matrix to the process's allocation array while adding them from the process's need array and the global resouce pool.

2.3.10 int saftey_test()

This fucntion performs the Banker's algorithm on the current system state.

The Banker's Algorithm is used in finding out whether or not a system is or isn't a safe state. This algorithm can be described as follows:

1. Let Work and Finish be vectors of length m and n,respectively. Initialize Work = Available and Finish[i] = false for i= 0, 1, ...,n1.
2. Find an index i such that both
   (a) Finish[i]==false
   (b) Need i $\leq$ Work If no such i exists, go to step 4
3. Work=Work+AllocationiFinish[i]=trueGo to step 2.
4. If Finish[i]== true for all i,then the system is in a safe state.This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

2.3.11 void add_arr(int* , int*)

Helper function that adds the first array to the second.

2.3.12 int can_finish(int*, int*)

Helper function that make sure that all of the values in the second matrix are greater than or equal to the first. Used in the bankers algorithm to see if a process can finish.