EE 468 Final Exam Fall 2014 Total = 100 pts NAME: Bronson Edralin

Instructions: This is a take home exam. You may use your book, notes, homeworks and projects and their solutions. You may use Microsoft Office applications such as Word, Excel, etc or equivalent. You may also look up information on the Internet but only from wikipedia and web sites that discuss Linux or Unix system calls. You are not allowed to discuss this exam with anyone, use message boards, or any type of communication. You are to submit your final exam by into laulima by December 18, 2013 Thursday at 11:55PM.

Problems 6 and 7 have two programs to modify or upgrade: sched.c and vm.c. You can find a version of these two programs attached to this exam as a tarred and gzippeed directory. The directory is named "Final". It has subdirectories Sched and VM that contain sched.c and vm.c, respectively. The subdirectories also carry data files. When submitting, put README files in the subdirectories with instructions to compile and run. Your README file should have your name. Rename the directory "Final" to "<YourLastName>". To submit your directory of upgraded programs, tar and gzip it.

Submit your final exam solutions and programs by uploading into laulima.

Problem 1 [10 pts]. The following program will generate 100 child processes. The parent process will wait until all processes are done before printing "All child processes are done". The program is incorrect. Modify it so it works properly.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main( )
{
pid_t   pid;
int     i;

for (i = 0;  i < 100;  i++) {
        pid = fork( );
        if (pid < 0) {
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {
                printf("I am child # %d\n", i);
                return 0;
        }
        else {  /* parent process */
                wait(NULL);
                printf("All child processes are done \n");
        }
}
return 0;
} /* end of main */
```

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main( )
{
pid_t   pid;
int     i;

for (i = 0;  i < 100;  i++) {
    pid = fork( );
    if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) {
        printf("I am child # %d\n", i);
        return 0;
```

```
    }
    else {  /* parent process */
        wait(NULL);
        //printf("All child processes are done \n");
    }
}
printf("All child processes are done \n");
return 0;

} /* end of main */
```

Problem 2. Consider the following variation of the Dining Philosophers Problem called the Dining Martians Problem. There are five martians, numbered 0, 1, ..., 4, sitting around a circular table. Initially, there are ten chopsticks in the middle of the table. Each martian needs exactly three chopsticks to eat. Whenever a Martian is hungry, it starts picking up chopsticks one at a time, where the time between picking up chopsticks can be of varying delay. A hungry martian keeps picking up chopsticks (and doesn't share) until it has three, then it starts eating. When it's finished eating, it puts all its chopsticks in the middle of the table.

a. [5 pts] Give a scenario which puts the martians in a deadlock situation.

A scenario where the martians may be in a deadlock situation is when all five martians are hungry and end up taking 2 chopsticks at the same time. A martian needs 3 chopsticks to eat but all 10 chopsticks are being taken at the moment and no martian will give up their chopstick.

b. [16 pts] Show how each of the following four necessary conditions for deadlock hold in this setting.

i. Mutual exclusion

In this situation, all chopsticks are being held as nonshareable resources between the martians. A martian may make a request for a chopstick but unfortunately all requests are delayed until a chopstick gets freed.

ii. Hold and wait

All martians are holding and waiting for a chopstick to be freed so they can complete their process of eating. None of the martians will ever get to eat until one martian gets his third chopstick and complete his process of eating. Then that martian's resources (chopsticks) may be freed, but unfortunately we are in a deadlock with all martians with only 2 chopsticks each. So all martians will continue to wait and never finish because there are no free chopsticks.

iii. No preemption

No martian will give up their chopsticks so they will all continue to starve in deadlock if all 5 martians hold 2 chopsticks each with 0 free chopsticks. Preemption would have helped get them out of deadlock but unfortunately there is none.

iv. Circular wait

Each martian is waiting for the other one before them to finish eating, but a chopstick will never get freed. The circular connections or circular edges for the martians are as follows: E={(0,1),(1,2),(2,3),(3,4),(4,0)}. Each martian can be seen as the vertices V={0, 1, 2, 3, 4} for this circular graph style. For circular wait, martian 1 will wait for martian 0 to finish while martian 2 will wait for martian 1 to finish and so on.

c. [10 pts] Assume that the martians have distinct numbers for identification. Consider the following additional rule for picking up chopsticks: When there is only one chopstick left, the martian that has two chopsticks and has the lowest number gets this last chopstick. Prove or disprove the martians are deadlock free. To disprove, show a deadlock example. To prove, show which of the necessary conditions for deadlock do not hold.

Pretend each martian are assigned ids with different priorities attached to it. A higher number will have higher priority for this case. With a couple chopsticks in the middle of the table, this gives chance for martians to actually finish their process of eating. Orders may change so this scenario will prevent circular wait from happening. Priority may also be preemptive which may further help from deadlock. For example, if a martian has a high priority and it is hungry it may preempt for resources to be freed. Then it will finish its eating and free up the resources. Other martians can pick it up and start eating afterwards as well.

Problem 3. [10 pts] Consider a file system that uses inodes to represent files (Figure 12.9 on page 560 of the textbook shows an inode). Disk blocks are 2KB in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system.

Given:
Disk blocks = 2kB in size
Pointer to Disk block require 4B
12 Direct Disk Blocks
1 Single Indirect Disk Block
1 Double Indirect Disk Block
1 Triple Indirect Disk Block

$$\frac{\left(2KB * \frac{1024B}{1KB}\right)}{4B} = \frac{2048B}{4B} = 512$$

12 Direct Disk Blocks: $12 * 2kB = 24kB$
1 Single Disk Block: $512 * 2KB = 1024KB$
1 Double Disk Block: $512^2 * 2KB = 524288KB$
1 Triple Disk Block: $512^3 * 2KB = 2.68435 * 10^8 KB$
Total = $24KB + 1024KB + 524288KB + 2.68435 * 10^8 KB = 2.68961 * 10^8 KB = 256.5GB$
TOTAL=256.5GB

Problem 4 [9 pts]. Consider the following segment table

Segment	Base	Length
0	1000	256
1	2000	100
2	3000	500
3	4000	200
4	5000	200

For the following logical addresses, indicate if it is valid and if it is valid, indicate is its physical address?

a. 1, 500:

At Segment 1:
2000 + 500 = 2500
It is invalid

b. 3, 120:

At Segment 3:
4000 + 120 = 4120
It is valid at 4120

c. 2, 128:

At Segment 2:
3000 + 128 = 3128
It is valid at 3128

Problem 5 (20 pts). Find program sched.c which simulates a demand paging system with three frames for three replacement algorithms: FIFO, least recently used (LRU), and optimal. It gets a reference string from a data file scheddata (note that the contents of this file can be changed by the user). The format of the reference string is explained in the comments in the program. The algorithms are implemented by the functions fifo( ), lru( ), and optimal( ). Currently, only optimal( ) works. Fix the other two functions so they work too. Successful implementation of fifo( ) and lru( ) is each worth 10 points

```
/*
 *  EE 468 Final Exam Problem
 *
 *  This program simulates a dynamic page system with 3 pages for three
 *  replacement algorithms:  FIFO, least recently used (LRU), and
 *  optimal.  The list of address references come from a data
 *  file named "scheddata".  The first line in the data file
 *  is the size of the list, e.g., if the line has the integer 20
 *  then the list has 20 integer entries.  The subsequent lines
 *  have the references.  The program will then output the
 *  entries to verify that the data was loaded properly.  Then
 *  it will determine the hits and misses for each of the algorithms.
 *  The current program has optimal implemented.  But FIFO and LRU
 *  are not implemented properly.  There are functions fifo() and lru()
 *  for FIFO and LRU but they are presently just stubs.  You must
 *  implement them.
 *
 *  Note:  the prob6data file that is in this directory is the same
 *  as the one in textbook problem 9.21.
 */


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

#define MAXREF    40  /* Max number of references */
#define NUMPAGES  3   /* Number of pages */

void dispOutcome(int numref, int ref[], int outcome[]);
void fifo(int numref, int ref[], int outcome[]);
void lru(int numref, int ref[], int outcome[]);
void optimal(int numref, int ref[], int outcome[]);

int main()
{

FILE * pFile;

int i;

int numref;       /* Number of references */
int ref[MAXREF];  /* References */
int refvalue;
```

```
/*
 * The following array is used to store the outcome of a
 * replacement algorithm.  Outcome[k] is the result of the
 * kth reference.  If Outcome[k] = 0 then it's a miss (page
 * fault); and if Outcome[k] = 1 then it's a hit.
 */
int outcome[MAXREF]; /* Outcome of an algorithm */

/*
 * Load references
 */
pFile = fopen("scheddata","r");
if (pFile == NULL) {
   printf("Didn't open prob7data file\n");
   return 0;
}
fscanf(pFile, "%d", &numref);
printf("numref = %d\n",numref);
if (numref > MAXREF) {
   printf("Number of references %d is too large\n",numref);
}
for (i=0; i<numref; i++) { /* Read all the reference values */
   fscanf(pFile," %d ", &refvalue);
   ref[i] = refvalue;
}

printf("\n");  /* Print out references to verify */
printf("References that were loaded\n");
printf("Number of references = %d\n",numref);
for (i=0; i<numref; i++) {
   printf("Reference[%d] = %d\n",i,ref[i]);
}

printf("\nFIFO outcomes\n");
fifo(numref, ref, outcome);
dispOutcome(numref, ref, outcome);

printf("\nLRU outcomes\n");
lru(numref, ref, outcome);
dispOutcome(numref, ref, outcome);

printf("\nOptimal outcomes\n");
optimal(numref, ref, outcome);
dispOutcome(numref, ref, outcome);

}

void dispOutcome(int numref, int ref[], int outcome[])
{
int i;

for (i=0; i<numref; i++) {
   printf("Ref[%d] = %d outcome = ",i,ref[i]);
   if (outcome[i] == 0) printf("Miss\n");
```

```
   else if (outcome[i] == 1) printf("Hit\n");
   else printf("invalid outcome\n");
}
}

/*
 * FIFO replacement algorithm.  It doesn't work and
 * you have to fix it.
 */
void fifo(int numref, int ref[], int outcome[])
{

int pagecount = 0; /* Number of stored pages */
int page[NUMPAGES]; /* References in the stored pages */

int i, j, k, n;
int nextref[NUMPAGES]; /* When next replacement occurs */
int swap;

for (i=0; i<numref; i++) {

   /* Check if reference i is in the stored pages */
   outcome[i] = 0;
   for (k=0; k<pagecount; k++) {
      if (ref[i] == page[k]) {
         outcome[i] = 1; // A hit occured
         break;
      }
   }

   /* If it's a miss then store the new page */
   if (outcome[i] == 0) {

      /* If there is empty space then store it there */
      if (pagecount < NUMPAGES) {
         page[pagecount] = ref[i];
         pagecount++;
      }
      else { /* Otherwise, you must find the page to replace */

         for (j=0; j<pagecount; j++) {
           page[j] = page[j+1];
         }

         /* Replace with ref[i] */
         page[2] = ref[i];
      }
   }
   //printf("Ref[i] is %d, Outcome[i] is: %d\n",ref[i],outcome[i]);
   //printf("page[0]=%d, page[1]=%d, page[2]=%d\n\n",page[0],page[1],page[2]);

}
```

```
}

/*
 * Least recently used replacement algorithm.  It doesn't work
 * and you have to fix it.
 */
void lru(int numref, int ref[], int outcome[])
{

int pagecount = 0; /* Number of stored pages */
int page[NUMPAGES]; /* References in the stored pages */

int i, j, k, n;
int nextref[NUMPAGES]; /* When next replacement occurs */
int swap;

for (i=0; i<numref; i++) {

   /* Check if reference i is in the stored pages */
   outcome[i] = 0;
   for (k=0; k<pagecount; k++) {
      if (ref[i] == page[k]) {
         outcome[i] = 1; // A hit occured
         swap = page[k];
         break;
      }
   }
    if (outcome[i] == 1) {
         for (j=0; j<pagecount; j++) {
           page[j] = page[j+1];
         }
         /* Replace with ref[i] */
         page[2] = swap;
   }
   /* If it's a miss then store the new page */
   if (outcome[i] == 0) {

      /* If there is empty space then store it there */
      if (pagecount < NUMPAGES) {
         page[pagecount] = ref[i];
         pagecount++;
      }
      else { /* Otherwise, you must find the page to replace */

         for (j=0; j<pagecount; j++) {
           page[j] = page[j+1];
         }

         /* Replace with ref[i] */
         page[2] = ref[i];
      }
   }
   //printf("Ref[i] is %d, Outcome[i] is: %d\n",ref[i],outcome[i]);
   //printf("page[0]=%d, page[1]=%d, page[2]=%d\n\n",page[0],page[1],page[2]);
```

```
}


}

/*
 * Optimal replacement algorithm
 *
 */
void optimal(int numref, int ref[], int outcome[])
{

int pagecount = 0; /* Number of stored pages */
int page[NUMPAGES]; /* References in the stored pages */

int i, j, k, n;
int nextref[NUMPAGES]; /* When next replacement occurs */
int bestpageindex;
int bestpagevalue;

for (i=0; i<numref; i++) {

   /* Check if reference i is in the stored pages */
   outcome[i] = 0;
   for (k=0; k<pagecount; k++) {
      if (ref[i] == page[k]) {
         outcome[i] = 1;
         break;
      }
   }

   /* If it's a miss then store the new page */
   if (outcome[i] == 0) {

      /* If there is empty space then store it there */
      if (pagecount < NUMPAGES) {
         page[pagecount] = ref[i];
         pagecount++;
      }
      else { /* Otherwise, you must find the page to replace */

         /* Find the next replacements for each page j*/
         for (j=0; j<pagecount; j++) {
            for (n=i+1; n<numref; n++) {
               if (page[j] == ref[n]) break;
            }
            nextref[j] = n;
         }

         /* Choose the best page to replace */
         bestpageindex = 0;
         bestpagevalue = 0;
         for (j=0; j<pagecount; j++) {
```

```
            if (nextref[j] > bestpagevalue) {
               bestpageindex = j;
               bestpagevalue = nextref[j];
            }
         }

         /* Replace with ref[i] */
         page[bestpageindex] = ref[i];
      }
   }
}
}
```

Problem 6 [20 pts]. This problem is essentially the Programming Project in Chapter 9 of the textbook on pages 458 through 461. Read these pages as well as the following:
We will assume that both the physical and virtual memories have size 2^{16} which is 65,536. Other specifics include the following:

- 256 entries in the page table
- Page and frame size of 256 bytes
- 16 entries in the TLB
- TLB uses the least recently used (LRU) replacement policy
- 256 frames

Initially, all the data is stored on disk (e.g., a backing store) and the physical memory has empty frames. Frames are filled starting from frame 0 and then filling consecutive unfilled frames. To keep track of which unfilled frame to fill next, we only need a pointer that is initialized to point to frame 0. Then as each frame is filled the pointer increments by one.

To simplify the scenario, we will assume that the page table is not part of physical memory. An example scenario is when the actual physical memory is larger than 256 frames, but the number of frames allocated to a process is 256. The page table is stored outside of the allocated 256 frames.

Note that the frames and pages are the same size, and the number of frames and pages are the same. So you do not need to be concerned about page replacements during a page fault.

A C program "vm.c" is supposed to simulate the virtual memory manager program. It reads a file named "address.txt", which is a list of virtual addresses. The program should run as follows:

./a.out

The program reads the virtual addresses from "address.txt". For each virtual address, it outputs on a line

- the virtual address
- the corresponding physical address
- whether it resulted in a page fault
    - in particular, it outputs "PAGE-MISS" or "PAGE-HIT" for a fault or no-fault, respectively

- whether it resulted in a TLB hit
    - in particular, it outputs “TLB-HIT” or “TLB-MISS”

After all this is done, the program will output on separate lines

- the number of virtual addresses in the file “address.txt”
- the number of page faults
- the number of TLB hits

In the directory “Final” and subdirectory “Vm”, there is a partially working “vm.c”. Instead of a TLB with 16 entries, it has a TLB with one entry. Also, it doesn’t keep track or display the number of virtual address in the file address.txt, number of page faults, or the number of TLB hits.

Modify “vm.c” so that it works correctly. Make sure vm.c works on wiliki or Ubuntu. If you can’t get it to work then explain what went wrong or why it doesn’t work. Hint: to implement the LRU policy for the TLB, you need to keep track of each entry’s age.

Other comments: Note that vm.c just simulates the address translations and keeps track of when page faults and TLB misses occur.

Example: Suppose address.txt has the contents

1
256
32768
32769
128
65534
33153

The following are the results

Virtual addr	Page #	Frame #	Page Fault	TLB	Page offset	Phys addr
1	**0**	**0**	**YES**	**MISS**	**1**	**1**
256	**1**	**1**	**YES**	**MISS**	**0**	**256**
32768	**128**	**2**	**YES**	**MISS**	**0**	**512**
32769	**128**	**2**	**NO**	**HIT**	**1**	**513**
128	**0**	**0**	**NO**	**HIT**	**128**	**128**
65534	**255**	**3**	**YES**	**MISS**	**254**	**1022**
33153	**129**	**4**	**YES**	**MISS**	**129**	**1153**

So running

./a.out address.txt

will output

```
1      1      PAGE-MISS    TLB-MISS
256    256    PAGE-MISS    TLB-MISS
32768  512    PAGE-MISS    TLB-MISS
32769  513    PAGE-HIT     TLB-HIT
128    128    PAGE-HIT     TLB-HIT
65534  1022   PAGE-MISS    TLB-MISS
33153  1153   PAGE-MISS    TLB-MIS
```

Number of virtual addresses = 7
Number of page faults = 5
Number of TLB misses = 5

```
/*
 * EE 468 Final Exam Problem
 *
 * This program simulates a virtual memory manager with
 * a TLB with 16 entries.  Total number of memory bytes is 2^16 bytes.
 * Page and frame sizes are 256 bytes.  Number of pages and
 * frames is 256.  It uses the least recently used replacement
 * policy.
 *
 * For the simulation it gets virtual addresses from a file
 * named "address.txt"
 *
 * However this version doesn't work.  It simulates a
 * memory system with a TLB of size one.  Fix it so
 * that it works according to Final Exam Problem 6.
 */

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

struct tlb_entry {
   char valid;  /* Valid bit implemented as a char */
   int page;    /* Page number */
   int frame;   /* Frame number */
};

struct ptable_entry {
   char valid;  /* Valid bit implemented as a char */
   int frame;   /* Frame number */
};

/* The next two data structures is the TLB (tlb) and page table (ptable).
 * TLB is usually an array but in this (bad) implementation there is only
 * one entry.
 */

struct tlb_entry tlb[16];              /* Global variable for TLB. */
struct ptable_entry ptable[256];   /* Global array for page table */
```

```
int main() {
  FILE *fp;
  int vaddr;  /* virtual address */
  int offset; /* page offset */
  int page;   /* page number */
  int frame;  /* frame number of current access */
  int freeframe;  /* pointer to the lowest free frame */
  int tlbhit;     /* indicates a TLB hit */
  int pfault;     /* indicates page fault */
  int physaddr;   /* physical address */
  int i;
  int num_tlbmiss = 0;
  int num_pfault = 0;
  int num_virtual = 0;

  /* Initialize TLB and page table*/
  for (i=0; i<16; i++) tlb[i].valid = 'n';
  for (i=0; i<256; i++) ptable[i].valid = 'n';

  /* Open the file which has virtual addresses */
  fp = fopen("address.txt", "r");
  if (fp ==NULL) {
    printf("Cannot open file\n");
    exit(1);
  }

  /* Initialize free frame pointer */
  freeframe = 0;

  /* Read virtual addresses */

  while(fscanf(fp, "%d", &vaddr)== 1) { /* Read a virtual address */

    page = vaddr >> 8; /* Page # of virtual address */
    tlbhit = 0; /* indicatess if there is a TLB hit */
    pfault = 0; /* indicates page fault */
    // printf("page is %d\n",page);
    num_virtual++;

    tlbhit = 0;
    /* Access TLB */
    for (i=0; i<16; i++) {
      if (tlb[i].valid == 'y' && tlb[i].page == page) { /* TLB hit */
        tlbhit = 1;
        frame = tlb[i].frame;
        break;
      }
    }
    if (tlbhit == 0) {  /* TLB miss */
      num_tlbmiss++;
      /* Access the page table */
      if (ptable[page].valid == 'y') { /* No page fault */
        frame = ptable[page].frame;
      }
```

```
      else { /* Page fault */
        pfault = 1;
        num_pfault++;
        frame = freeframe;
        ptable[page].frame = freeframe++;
        ptable[page].valid = 'y';
      }

      /* Update TLB */
      for (i=0; i<16; i++) {
        if (tlb[i].valid == 'n') {
          tlb[i].valid = 'y';
          tlb[i].page = page;
          tlb[i].frame = frame;
          break;
        }
      }
    }


    physaddr = (frame<<8)+ (0xff & vaddr);  /* Physical address */
    printf("VirtAddr=%d PhysAddr=%d ",vaddr,physaddr);
    if (tlbhit == 1) printf(" TLB-HIT ");
    else printf(" TLB-MISS ");

    if (pfault == 1) printf(" PAGE-MISS ");
    else printf(" PAGE-HIT ");

    printf("\n");
  }
  printf("\n");
  printf("Number of virtual addresses = %d\n",num_virtual);
  printf("Number of page faults = %d\n",num_pfault);
  printf("Number of TLB misses = %d\n",num_tlbmiss);
  printf("\n");

  /*
  for (i=0; i<16; i++) {
    printf("TLB Table: valid=%c, page=%d, frame=%d\n",tlb[i].valid, tlb[i].page,
tlb[i].frame);
  }
  for (i=0; i<256; i++) {
    if (ptable[i].frame!=0) printf("Page Table: valid=%c,
frame=%d\n",ptable[i].valid, ptable[i].frame);
  }
  */

  fclose(fp);
  return 0;
}
```