*CS 453: Operating Systems*

**Exercises**

# Contents

# 1 Introduction

1. Install Fedora Linux in a virtual machine on your system. Use VMWare Player or Virtual Box for setting up and running the virtual machine.

2. Download the latest stable Linux kernel source code from the Linux Kernel Archives. Explore the high-level code organization. See how it maps roughly to the major parts of the Operating Systems discussed earlier.

3. In particular, see if you can locate the system call branch table in the Linux kernel code. *Hint: It will be architecture specific. Look for kernel code specific to an architecture like x86. Use* `find` *and* `grep -r` *to help you!*

4. Read the man page for the `/proc` virtual filesystem that allows us to examine/modify the Linux kernel in real-time.

5. Does the Mac OS X have a `/proc` like virtual filesystem? If not, what facility does it have to obtain system information similar to the `/proc` virtual filesystem.

6. Download Sysinternals suite of tools for Microsoft Windows. These tools allow you to explore detailed technical information about the OS. For example, try the `coreinfo.exe` from the sysinternals suite from a powershell window.

## 2 Basic Systems Programming

1. **Passing data to your progeny?** Write a program that creates and fills some data structure (like an array). Then it forks a child process. Check if the child process inherits the initialized data structure. What about opening a file in the child process that was opened in the parent process before the fork?

2. **The Friendly Forked Family**. How many new processes are created in the following code?

```
/* process A */
/* ... */
    pid = fork();
    if (pid == 0) {
        pid = fork();
        printf("Hi!");
    } else {
        pid = fork();
        printf("Hi!");
        if (pid == 0) {
            pid = fork();
            printf("Hi!");
        } else {
            pid = fork();
            printf("Hi!");
        }
    }
/* ... */
```

3. Consider the following program:

```
void forkthem(int n) {
    if (n > 0) {
        fork();
        forkthem(n-1);
    }
}

int main(int argc, char **argv)
{
    forkthem(5);
}
```

   How many **new** processes are created if the above program is run?

4. **Auto Save**. Many editors have an "auto-save" feature which periodically (at a predetermined interval) saves the file for the current session. Explain how would you implement this feature? (You may give a Linux specific answer).

5. Setup Visual Studio and play with the examples from the class repository to familiarize yourself with the environment.

6. **Waiting for Godot!** Write a program that creates as many processes as the number of CPUs on your system (using the CreateProcess call). Have each created process print its process id and then sleep for five seconds before exiting. The number of CPUs can be determined via the following MS Windows API call: GetSystemInfo(...)

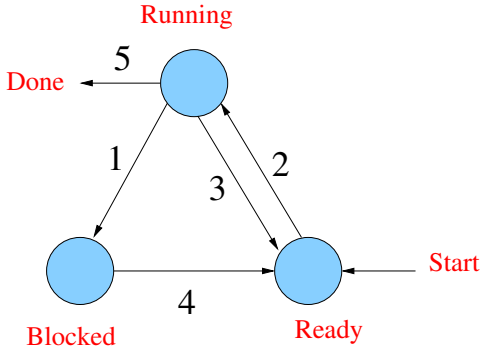
7. **Processes in Java**. Browse the Java class `java.lang.Runtime` to see how we can create and manage processes in Java. Is the model of managing processes in Java more similar to Linux or to MS Windows? Why?

## 3 Process Management

1. Write a bash script that determines the following pieces of information from the `/proc` virtual filesystem: processor type, kernel version, time since last boot (in hours), time spent in idle (as a percentage), number of context switches, number of processes created since booted, number of reads/writes on various disks.

2. (More difficult) Write a sample implementation of a binarization of a general tree with leftmost-child-right-sibling representation discussed in class. Also write a general tree using an array of child pointers. Write a test driver that creates various trees in both formats and calculates and compares the wasted space in each representation.

3. Consider the following process state transition diagram:




   For each of the transitions give an example of a specific event that can cause that transition.

   **(1)**

   **(2)**

   **(3)**

   **(4)**

4. - Give two specific examples of how a process could be involuntarily removed from the CPU.
   - Give two specific examples of how a process could voluntarily give up the CPU.
   - Give a specific example of how a process could move from a running state to the *ready-Suspended* state.

# 4 Scheduling

1. **Context Switch**. Explain what is a context switch. What steps are taken in a context switch by the operating system?

2. Devise a workload where FIFO is pessimal— it does the worst possible scheduling choices— for average response time.

3. Suppose you do your homework assignments in SJF-order. After all, you feel like you are making a lot of progress! What might go wrong?

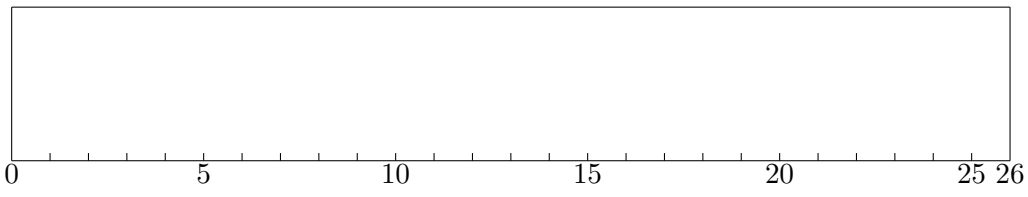
4. Many CPU scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Some of these algorithms are related to one another (for example, FCFS is RR with infinite time quantum). What (if any) relation holds among the following pairs of algorithms? Please provide a concise (but complete) answer.

    1. Priority and SJF
    2. Multilevel feedback queues and FCFS
    3. Priority and FCFS
    4. RR and SJF

5. Five processes arrived at the same time with the following burst times (that is the CPU time for which they will run before blocking for I/O).

| Process Name | Burst Time |
|---|---|
| P1 | 10 |
| P2 | 7 |
| P3 | 5 |
| P4 | 1 |
| P5 | 3 |

    Show how the processes are scheduled by filling in the following Gantt chart and calculate the average response and average turnaround time if the scheduling policy is round-robin with time quantum of 3 time units.



Average Response Time =

Average Turnaround Time =

6. Round Robin scheduling algorithm can be implemented without using any timer interrupts.
   **TRUE FALSE**

7. Give an argument in favor of and against a small quantum for a round robin scheduler?

# 5 Synchronization (Part 1)

1. **Evil in the Garden of Threads?** The following code shows the usage of two threads to sum up a large array of integers in parallel. The code seems to add a small set of numbers correctly but does not always show the correct sum for larger set of numbers. Explain why? Then show how to fix the problem.

```
/* appropriate header files */
void *partial_sum(void *ptr);
int *values;
int n;
int result[2]; /* partial sums arrays */

int  main( int argc, char **argv)
{
    int i;
    pthread_t thread1, thread2;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <n> \n", argv[0]); exit(1);
    }
    n = atoi(argv[1]);
    values = (int *) malloc(sizeof(int)*n);
    for (i=0; i<n; i++)
        values[i] = 1;

    pthread_create(&thread1, NULL, partial_sum, (void *) "1");
    pthread_create(&thread2, NULL, partial_sum, (void *) "2");

    do_some_other_computing_for_a_while();

    printf("Total sum = %d \n", result[0] + result[1]);
    exit(0);
}
void *partial_sum(void *ptr)
{
    char *message;
    int sum, i, start, end, index;

    message = (char *) ptr;
    sum = 0;
    if (strcmp(message,"1") == 0)  {
        index = 0; start = 0; end = n/2;
    } else {
        index = 1; start = n/2 + 1; end = n - 1;
    }

    for (i=start; i<=end; i++)
        sum += values[i];

    result[index] = sum;
```

```
    pthread_exit(NULL);
}
```

2. **Busy Bees**. Several concurrent processes are attempting to share an I/O device. In an attempt to achieve mutual exclusion, each process is given the following structure. (*busy* is a shared Boolean variable that is initialized to **false** before the processes start running.)

```
<code unrelated to the device use>
while (busy == true); // empty loop body
if (busy == false) busy = true;
<code to access shared device>
busy = false;
<code unrelated to the device use>
```

   Does the above solution guarantee mutual exclusion. If yes, then prove it. If not, then give a suitable interleaving which causes mutual exclusion to be violated.

3. **Here we go: 1, 2, 3,** ... Consider the following program that attempts to assign a unique index to each thread in the range $0 \ldots numThreads-1$. Does this code have a race condition? If yes, then identify the race condition and show how to fix it. Otherwise argue for its correctness.

```
/* appropriate include statements */

void *run(void *);
pthread_t *tids;
int numThreads = 10;
int counter;

int main(int argc, char *argv[])
{
    int i;
    if (argc == 2)
        numThreads = atoi(argv[1]);
    tids = (pthread_t *) malloc (sizeof(pthread_t)*numThreads);
    counter = 0;
    for (i=0; i<numThreads; i++)
        pthread_create(&tids[i], NULL, run, (void*) NULL );

    for (i=0; i<numThreads; i++)
        pthread_join(tids[i], NULL);
    exit(0);
}

void *run(void *arg)
{
    int index;
    index = counter;
    counter++;
    printf("Thread id = %ld has index = %d\n", pthread_self(), index);
    pthread_exit(NULL);
}
```

4. **Implementing barriers using semaphores** (More difficult). Write pseudo-code for implementing a barrier using semaphores. Recall that a barrier implies that all participating threads reach the barrier before any thread can get past the barrier. Look at the example `synchronization/threads-barrier.c` to see how to use a barrier.

5. **Implementing mailboxes using semaphores** A shared mailbox area in shared memory will be used to hold the mailboxes. Each mailbox data structure contains a fixed-size array of message slots. Each mailbox has a variable keeping track of the number of full slots and also has two queues that keep track of waiting processes/threads.

   - Each process has an associated semaphore (initially zero) on which it will block when a send/receive must block.
   - A global mutex is used to ensure mutual exclusion in accessing the shared mailbox area in memory.

   Based on the above, now sketch out the pseudo-code for synchronous send/receive.

   How can we improve the performance of the above implementation?

6. **Dining Philosophers? Dining Semaphores?** We have 5 philosophers that sit around a table. There are 5 bowls of rather entangled spaghetti that they can eat if they get hungry. There are five forks on the table as well. However, each philosopher needs two forks to eat the tangled spaghetti. No two philosophers can grab the same fork at the same time. We want the philosophers to be able to eat amicably. Consider the following solution to this problem.

```
/* dining_philosophers */
sem_t fork[5]; // array of binary semaphores
sem_t table; // general semaphore

void philosopher(void *arg)
{
    i = *(int *) arg;
    for (;;) {
        think();
        sem_wait(&table);
        sem_wait(&fork[i]);
        sem_wait(&fork[(i+1) % 5];
        eat();
        sem_post(&fork[i]);
        sem_post(&fork[(i+1) % 5];
        sem_post(&table);
    }
}
int main()
{
    int i;

    for (i=0; i<5; i++) {
        sem_init(&fork[i], 0, 1); //initialize to 1
    }
    sem_init(&table, 0, 4); //initialize to 4
    for (i=0; i<5; i++) {
        pthread_create(&tid[i], NULL, philosopher, (void *)&i);
    }
    for (i=0; i<5; i++) {
        pthread_join(tid[i], NULL);
    }
    exit(0);
}
```

(a) Argue why it is not possible for more than one philosopher to grab the same fork?

(b) Can the philosophers ever deadlock. Explain.

(c) Can a philosopher starve?

7. Construct an interleaving for the following program in which the final value of $n$ is 2. Note that $n$ is a global variables and $p$ and $q$ are two threads with local variables $i$ and *temp*.

| int n = 0; | |
|---|---|
| int temp; | int temp; |
| p1: for (i=0; i<10; i++) | q1: for (i=0; i<10; i++) |
| p2: temp = n; | q2: temp = n; |
| p3: n = temp + 1; | q3: n = temp + 1; |

# 6 Synchronization (Part 2)

1. **Monitors in PThreads**. Consider the following implementation of a circular queue in C. Show how to make the implementation be thread-safe using PThreads. The solution needs to be valid C code.

```
#ifndef __ARRAYQUEUE
#define __ARRAYQUEUE
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#define FALSE 0

typedef struct ArrayQueue ArrayQueue;
struct ArrayQueue {
    void **A;
    int length;
    int head;
    int tail;
    int count;
};

ArrayQueue *InitArrayQueue(ArrayQueue *Q, int size)
{
    Q = (ArrayQueue *) malloc(sizeof(ArrayQueue));
    Q->A = (void **) malloc(sizeof(void *)*size);
    Q->length = size; Q->head = 0; Q->tail = 0; Q->count = 0;
    return Q;
}

int enqueue(ArrayQueue *Q, void *x)
{
    if (Q->count == Q->length){ return FALSE; //overflow }
    Q->A[Q->tail] = x;
    Q->count++;
    Q->tail=(Q->tail + 1) % Q->length;
    return TRUE;
}

void *dequeue(ArrayQueue *Q)
{
    void *x;
    if(Q->count <= 0) { return NULL; //underflow }
    x = Q->A[Q->head];
    Q->count--;
    Q->head = (Q->head + 1) % Q->length;
    return x;
}
#endif /* __ARRAYQUEUE */
```

2. **Monitors in Windows API**. Repeat the last problem using MS Windows API.

3. **Monitors in Java**. Consider the following implementation of a circular queue in Java. Show how to make the implementation be thread-safe. The solution needs to be valid Java code.

```
public class ArrayQueue{
    private int head;
    private int tail;
    private int count;
    private Object[] A;

    public ArrayQueue(int size) {
        A = new Object[size];
        this.head=0; this.tail=0; this.count=0;
    }

    public void enqueue(Object x)
    throws QueueException
    {
        //check for queue overflow
        if(count==A.length){
             throw new QueueException("overflow - " + A.length);
        }
        A[tail] = x;
        count++;
        tail=(tail+1)%A.length;
    }

    public Object dequeue()
    throws QueueException
    {
        Object x;
        //check for queue overflow
        if(count<=0){ throw new QueueException("underflow"); }
        x = A[head];
        count--;
        head=(head+1)%A.length;
        return x;
    }
}
```

4. **File Access Monitor**. A file is to be shared among different processes. The file can be accessed simultaneously by several processes, subject to the constraint that the total number of processes accessing the file should be less than $n$. Write a monitor to enforce the given constraint. Use the following template.

```
monitor  FileAccess {
 ...
public:
 void StartAccess() {
    ...
 }

 void EndAccess() {
    ...
 }
}
```

5. **Another File Access Monitor**. A file is to be shared among different processes, each of which has a unique process-id. The file can be accessed simultaneously by several processes, subject to the constraint that the sum of all unique process-ids associated with all the processes currently accessing the file must be less than a parameter $n$. Write a monitor to coordinate access to the file using the following template.

```
monitor  FileAccess {
    ...

 public:

 void StartAccess(int id) {
 }

 void EndAccess(int id) {
 }
```

6. **Print Monitor**. Consider a system consisting of $n$ processes $P_1, P_2, \ldots, P_n$. Write a monitor (in pseudo-code) that allocates three printers printer[1..3] among these processes. Processes get printers in a first-come-first-served fashion (one per process). If there is more than one printer available, then each process wants printer[1] since it is the fastest. If it can't get printer[1], then a process wants printer[2] as it is the second fastest and printer[3] is the least favorite choice if the first two are not available. For your solution use Hoare semantics (where a process must immediately leave the monitor after signaling).

```
monitor PrintMonitor{
    const int NUM=3;
    condition  WaitForPrinter;
    int  NumberOfPrintersLeft;
    boolean  PrinterFree[1..NUM]; /* true implies free, false implies busy */

    int GetPrinter()
    {
    ...
    }
    void ReleasePrinter(int p)
    {
    ...
    }
    // constructor to initialize the monitor
    PrintMonitor(void) {
        NumOfPrintersLeft = NUM;
        for (int i=1; i<= NUM; i++)
            PrinterFree[i] = true;
    }
}

// Code for each process
void P(int pid);
{
    int Printer;

    Printer = PrintMonitor.GetPrinter();
    PrintFile(Printer);
    PrintMonitor.ReleasePrinter(Printer);
    goDoSomethingElse();
}

void main(void) {
    // start the following n processes in parallel
    P(1); P(2); ... P(n);
}
```

7. **Print Monitor in Java**. Rewrite the previous problem in Java.

```
public class PrintMonitor
{
    public final int NUM=3;
    private int  NumberOfPrintersLeft;
    private boolean  PrinterFree[1..NUM]; /* true implies free, false implies busy */

    public PrintMonitor(void) {
        NumOfPrintersLeft = NUM;
        for (int i=1; i<= NUM; i++)
            PrinterFree[i] = true;
    }
    public int GetPrinter() {
    /* add code here */
    }
    public void ReleasePrinter(int p) {
    /* add code here */
    }
}
public class P implements Runnable {
    int pid;
    PrintMonitor monitor;
    public P(int pid, PrintMonitor monitor) {
        this.pid = pid;
        this.monitor = monitor;
    }
    public void run() {
        int printer;
        while (true) {
            printer = monitor.GetPrinter();
            printFile(printer);
            monitor.ReleasePrinter(printer);
            goDoSomethingElse();
        }
    }
    /* other methods */
}

pubic static void main(String [] args) {
    PrintMonitor monitor = new PrintMonitor();
    // start the n processes in parallel
    for (int i=0; i<n; i++) {
        new Thread(new P(i, monitor)).start();
    }
}
```

8. **Implementing Message Passing using Monitors**. Each mailbox is managed by a monitor. Each mailbox contains an array of message slots. Each mailbox has a variable keeping track of the number of full slots and also has two queues that keep track of waiting processes/threads.

   Each process has an condition variable on which it will block when a send/receive must block. Based on the above, now sketch out the pesudo-code for synchronous send/receive.

9. **Implementing Asynchronous Message Passing using Monitors**. Modify the solution the previous problem so that it implements asynchronous (i.e. non-blocking) send/receive.

10. **List Monitor Mayhem**. Explain why protecting the `removeNode` function with a mutex in your List class isn't sufficient to prevent race conditions (Even though this was sufficient to protect the `removeFront` and `removeRear` functions).

11. **FIFO Monitor**. Modify the alarm alock monitor example in Java so that it implements a FIFO `wait()`.

12. **Priority-wait Monitor**. Modify the alarm alock monitor example in Java so that it implements a prisority-based `wait(int priority)`, where we will assume that the smaller number implies higher priority with 1 being the highest.

13. **Thread-pool in Java**. Java implements a pool of thread using the *Executor* interface. Study the *interface* and write a minimal Java code example that creates threads using the Executor interface.

14. **Thread-pool in C** (More difficult). Discuss how you would implement a pool of threads. The idea is to have a queue of jobs that need to be executed and a pool of threads is created to execute those jobs. The user can limit the number of threads when they create the thread-pool. Write a header file for a "class" to implement a thread-pool in C using PThreads.

15. **Walkie-Talkie**. Write a program that uses two pipes to allow two-way communication between a parent and child process. May be they can finally have a real conversation. . ..

16. **Chit Chat**. Write a server program that creates two named pipes (FIFOs) and then waits for a client program to write a request to one of the named pipes. Then it uses the other named pipe to reply to the client. Since we are using named pipes, you will be able to run these two programs in two separate windows and watch.

17. **Multi-process Chat Server**. This generalizes the solution from the last problem. Write a multi-process server that can chat with multiple clients simultaneously by forking multiple copies of itself. Also develop a client program to test the server. This will use named pipes for the communication.

# 7 Device Management

1. Do a rough calculation of how much slower a typical 7200RPM disk drive is compared to the main memory (such as DRAM). (Use 60ns per 64 bits for DRAM versus 10 ms to seek a 2K block for a disk drive) Find out typical speed for a SSD drive and redo the comparison with main memory.

2. **Loading code on the fly in C**. Modify the plugin example `plugins/ex1/` such that the driver program sets an alarm and checks if the plugin library has been updated and if so, it replaces the old version with the new version.

3. **Loading code on the fly in Java**. Write a simple Java example that shows how a driver class can load another class on the fly using the class loader. Further modify the example using a timer such that the driver class periodically checks if the class has changed and reloads the new version of the class. *Hint:* Use the TimerTask for setting up a periodic check. Use the File class to check timestamp for the class file to see if it was updated.

4. **Monitors in device drivers**. Explain why making a device drive a monitor may not be a good idea? Under what circumstances is it a good idea to make a device driver into a monitor?

# 8 Memory Management

1. Consider the following C fragment that allocates some memory.

```
struct job {
    struct job *next;
    struct job *prev;
    char *name;
    char **parameters;
    struct status *job_status;
};

struct job_status {
    int state;
    struct event *pending_events;
    struct time_used time;
}

struct job *newjob;

newjob = (struct job *) malloc(sizeof(struct job));
newjob->next = NULL;
newjob->prev = NULL;
newjob->name = (char *) malloc(sizeof(char)*MAXLENGTH);
newjob->parameters = (char **) malloc(sizeof(char *)*MAX_PARAMETERS);
for (i=0; i<MAX_PARAMETERS; i++)
    newjob->parameters[i] = (char *) malloc(sizeof(char)*MAXLENGTH);
newjob->job_status = (struct job_status *) malloc(sizeof(struct status));

newjob->job_status->state = READY;
newjob->job_status->pending_events =
                    (struct event *) malloc(sizeof(struct event)*MAX_NUM_EVENTS);
```

   Write a C code fragment that frees the memory properly.

2. The following code shows how to dynamically allocate a two-dimensional $n \times n$ array in C.

```
int i;
int **array = (int **) malloc(n * sizeof(int *));
for (i=0; i<n; i++)
    array[i] = (int *) malloc(n * sizeof(int));
```

   The following code also allocates a two-dimensional $n \times n$ array in C. How is the memory layout different from the above version? Why would it be useful?

```
int *ptr = (int *) malloc(sizeof(int) * n * n);
int **array = (int **) malloc(sizeof(int *) * n);
for (i=0; i<n; i++)
    array[i] = ptr + i * n;
```

3. **Memory manager overhead**. Assume that `malloc` memory manager uses the following block header

```
typedef double Align; /* for alignment to double boundary */

union header {  /* block header */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned int size;/* size of this block */
    } s;
    Align x;
};
```

   Calculate the fraction of memory that will used as overhead by the following code snippet.

```
    for (i = 0; i < 1000000; i++)
        ptr = (int *) malloc(sizeof(int));
```

4. How would you change your buddy system implementation such that it can manage multiple pools of memory instead of a single pool. Assume that we know the number of such pools at the time of creation of the buddy system and that each pool is of a size that is a power of two. Consider solutions that modify or don't modify the buddy system interface.

5. (More difficult) Make your buddy memory manager implementation be thread-safe. Now test to see if you are able to preload your buddy memory manager with a larger variety of programs?

6. **Bitmaps** are an alternate way of keeping track of free blocks. Design a bitmap data structure that stores one bit per block and provides the following operations:

   - *initBitMap* initializes the bitmap to appropriate size
   - *clearBitMap* sets the whole bitmap to zeroes
   - *freeBitMap* frees all memory associated with the bitmap
   - *set* the $i$th bit (to 1)
   - *get* get the $i$th bit
   - *clear* the $i$th bit (to 0)
   - *findFirstFree* finds the first free bit in the map
   - any other operations that you think may be useful

   The simplest way to store a bit map would be an array of `unsigned int` but then we would waste 31 bits out of each 32 bits (assuming unsigned int is 32 bits). The most space efficient way would be to pack `sizeof(unsigned int)*8` bits per `unsigned int` (as sizeof(int) returns number of bytes and not bits). Use this approach and come up with the implementation of the above operations.

# 9 Virtual Memory

1. Suppose we have a computer system with a 52-bit virtual address, 36-bit physical address and a page size of 64K.
   (a) What is the size of a page frame?
   (b) How many page frames are there in the physical address?
   (c) How many pages are there in the virtual address?
   (d) Show how the 52-bit virtual address gets mapped into the 36-bit physical address by drawing out a virtual address and specifying which bits are used to index the page table and which bits are used to determine the offset into a page?

2. A computer with a 32-bit virtual address uses a two-level page table. Virtual addresses are split into a 9-bit top-level page table field, an 11-bit second-level page table field, and an offset. How large are the pages and how many are there in the address space?

3. Suppose that a machine has 38-bit virtual addresses and 32-bit physical addresses.
   (a) What is the main advantage of a multi-level page table over a single-level page table?
   (b) With a two-level page table, 16-KB pages, and 4-byte page table entries, how many bits should be allocated for the top-level page table field and how many for the next level page table field? Explain.

4. Suppose an application is assigned four page frames that are initially empty (demand paging). It then references pages as shown in the following reference stream

   `a c b d b a e f b f a g e f a`

   (a) Draw a table to show how the system would fault pages into the four page frames using the FIFO replacement algorithm.
   (b) Draw a table to show how the system would fault pages into the four page frames using the Belady's optimal algorithm.
   (c) Draw a table to show how the system would fault pages into the four page frames using the LRU replacement algorithm.

5. Consider a server running a general mix of programs. It has two disks: one for paging and swapping and another for the file system. Here is the measured utilization on the system:

| | |
|---|---|
| CPU utilization | 20.0% |
| Paging disk | 99.7% |
| File system disk | 10.0% |
| Network | 5.0% |

   For each of the following changes, what would be the effect on the processor utilization: significantly increase, marginally increase, significantly decrease, marginally decrease, or have no effect.
   (a) Get a faster CPU
   (b) Get a faster paging disk
   (c) Increase the number of threads in the running programs

6. Suppose that a virtual page reference stream contains repetitions of long sequences of page references, followed occasionally by a random page reference. For example, the sequence $0, 1, \ldots, 511, 431, 0, 1, \ldots, 511, 332, 0, 1, \ldots$ consists of repetitions of the sequence $0, 1, \ldots, 511$ followed by a random reference to pages 431 and 332.

   (a) Why won't the standard replacement algorithms (LRU, FIFO, Clock) be effective in handling this workload for a page allocation that is less than the sequence length?

   (b) If this program were to be allocated 500 page frames, describe a page replacement approach that would perform much better than the LRU, FIFO or Clock algorithms.

# 10 File Management

1. Consider the following output from running the `du` command twice under Linux for a large folder containing tens of thousands of files and folders. There was no change in the filesystem in between and the output is the same in both cases. Explain why it ran over 50 times faster on the second attempt?

```
[amit@onyx ~]$ time du -hs somelargefolder
....
real    0m25.779s
user    0m0.259s
sys     0m1.607s

[amit@onyx ~]$ time du -hs somelargefolder
...
real    0m0.471s
user    0m0.137s
sys     0m0.299s
[amit@onyx ~]$
```

2. **Disk usage calculation**. Modify the example `file-management/filetype-survey.c` so that it instead calculates the total disk usage (in KB) for a given directory recursively. This would make it similar to the `du` command in Linux.

3. **Multithreaded disk usage calculation**. Modify your solution from the last problem so that is is multi-threaded. Use sempahores or monitors for synchronization.

4. **Ext3 File system**. The following fragment of code is from /usr/include/linux/ext3_fs.h in the Linux kernel source code.

```
#define EXT3_MIN_BLOCK_SIZE     1024
#define EXT3_MAX_BLOCK_SIZE     4096

/*
 * Constants relative to the data blocks
 */
#define EXT3_NDIR_BLOCKS        12
#define EXT3_IND_BLOCK          EXT3_NDIR_BLOCKS
#define EXT3_DIND_BLOCK         (EXT3_IND_BLOCK + 1)
#define EXT3_TIND_BLOCK         (EXT3_DIND_BLOCK + 1)
#define EXT3_N_BLOCKS           (EXT3_TIND_BLOCK + 1)

/*
 * Structure of an inode on the disk
 */
struct ext3_inode {
```

```
    ...
    __u32   i_block[EXT3_N_BLOCKS];/* Pointers to blocks */
    ...
}
```

Answer the following questions for the ext3 file system assuming a block size of 4096 bytes.

- What is the maximum size of a file using only direct pointers?
- What is the maximum size of a file using direct and single indirect pointers?
- What is the maximum size of a file using direct, single, and double indirect pointers?
- What is the maximum size of a file using direct, single, double, and triple indirect pointers?

5. How many disk operations are needed to fetch an i-node for the file:
   /home/faculty/amit/cs453/lab/README
   Assume that the i-node for the root directory is in memory, but nothing else along the path is in memory. Also assume that all directories fit within one disk block.

6. (More difficult) Write a program that scans all directories in a Linux file system and finds and locates all i-nodes with a hard link count of two or more. For each such file, it lists together all file names that point to the file.

# 11 Security

1. **Code Insecurity Therapy**. Find all potential security problems in the following piece of code. Examples would be buffer overflows, writing before the start of an array etc.

```
int main(int argc, char **argv)
{
    char buffer[128];
    char *input1;
    char input2[128];


    scanf("%s", input1);
    strcpy(buffer, input1);
    gets(input2);
    if strcmp(buffer, input2) {
        /* give access to all secrets */
    }

}
```

2. **Security Walkthrough**. Inspect your dash project code and identify and fix potential buffer overflow and other security issues.

3. **File protection bits, Access Control Lists, Capability Lists, and Encryption** are different methods for controlling access to files. Suppose we have a file that a given user has access to currently. For each of the four methods above explain how (if possible at all) to accomplish the following:

   - Remove access for a process (owned by the user) that has the file open while the process is running.
   - Forbid future access for the file to a specific user out of a group of users.
   - Forbid future access to even the system administrator (a.k.a. superuser).

4. (More difficult) **Trust in Me. Just in me....** Suppose you have two processes, say A and B, running on a "secure system." Suppose process A accesses some strategic information (let's say the password to the electronic cash account for The Billionaire.) The system doesn't allow process A to create pipes, named pipes, sockets, send messages, shared memory, open/access any files (except authorized files which the process B cannot open or detect that A has opened/accessed), create any semaphores, fork off a child process, create a new thread, or send/receive any signals. That is, no direct communication between A and B is allowed. *Can the process A somehow transmit the password to process B? Explain your answer.*

   Assume, for the sake of this problem, that the processes are running under a UNIX like system, although the scenario is valid for any operating system.

## 12 Networking

1. What would be advantage of having a single-threaded server?
2. Give pros and cons for a multi-threaded server versus a multi-process server.

## References


- *Operating Systems: Principles and Practice* by Thomas Anderson and Michael Dahlin.
- *Modern Operating Systems* (3rd ed) by Andrew Tanenbaum.
- *Operating Systems, 3rd ed.* by Gary Nutt.