

**ECU178 Computer Science:
207SE - Operating Systems, Security and Networks
Coursework**

Due on March 16th 2015

Robert Rigler : 4939377

Contents

Week 12: Multitasking vs Multiprogramming	3
Multiprogramming	3
Multitasking	3
Week 14: Process Manipulation & Nohup	4
Process Manipulation	4
Nohup	8
Week 17: Pipes and Sockets	10
RPN Calculator	10
Week 19: Security and Networks	15
Activity 1: Access Control	15
Domain Matrix	15
Access-List	15
Compatibility-Lists	16
Activity 2: Hash Function	17

Week 12: Multitasking vs Multiprogramming

In this task I am going to be comparing two different types of process scheduling: Multitasking, and Multiprogramming. I will look into what they are, their differences and their similarities.

Multiprogramming

Definition: A way of scheduling processes to maximise CPU usage by switching processes that are 'waiting' for I/O, it ensures that the CPU is never idle.

Much older systems, unlike modern computers were very expensive and slow and often, when a process needed to use a peripheral device It often meant that the CPU was sitting idle for a long period of time. The solution to this is 'batch processing'.

Multiprogramming allows a computer to do several tasks at the same time. When a group of processes are marked 'Ready' for execution they are placed in a queue in main memory. The first process from this queue is then loaded into the CPU and is executed. There may come a time when this process is interrupted because It needs I/O to continue. At this point the process changed to a 'waiting' state. The process is then swapped out of the CPU into the I/O queue, and the next process in the 'Ready Queue' is swapped into the CPU. When the I/O request of the first process is completed, it is then placed back into the 'Ready queue'. This cycle continues until there are no jobs to be processed.

Multitasking

Definition: A logical extension of Multiprogramming, it involves rapidly switching between processed in the 'Ready state' to give the impression that they are all running simultaneously.

In Multiprogramming, processes are executing one at a time, in the order that they are placed into the ready queue. This means that only one process can be actively used at a time. Similarly in multitasking, processes are executed individually, but ther is also a certain level of concurrency; Because once a process has used it allotted processing time, It is swapped back into main memory.

This is beneficial, because with multiprogramming, a process has complete control over the CPU until an interrupt is called. There may be a situation where a process does not call an interrupt and takes a long time to finish processing. This will cause shorter, more time efficient or more important processes to be delayed until the first process is finished.

Week 14: Process Manipulation & Nohup

Process Manipulation

For this task I will look into the different ways to manipulate a process, and show examples of how to use each command.

Command	Description
<i>command</i>	Type the name of the process to start it
<i>command &</i>	Start the process in the background (symbolised by the & symbol)
<i>ps -au</i>	Shows all the processes currently running on the machine
<i>ps -ux</i>	Shows all the processes currently running owned by the current user
<i>jobs</i>	Shows the processs that are currently suspended.
<i>CTRL - C</i>	Kills the process running in the foreground
<i>kill -9 x</i>	Kills the process with the PID <i>x</i>
<i>kill %1</i>	Kills the process with job number <i>1</i>
<i>CTRL - Z</i>	Susoends the process curently running in the foreground.
<i>kill -cont %1</i>	Continues the execution of suspended job %1
<i>bg %1</i>	Pushes job number 1 to to the background
<i>fg %1</i>	Pushes job number 1 to to the foreground

In the pages below, I will show two scenarios in which I use all of these commands. You will find a snippet of terminal code and an explanation of each step that was taken.

Listing 1: Scenario 1

```
1 Script started on Thu 12 Mar 2015 14:58:19 GMT
2 rob@rob-HP-ProBook-6470b:$ xclock
3 ^Z
4 [1]+  Stopped                  xclock
5 rob@rob-HP-ProBook-6470b:$ jobs
6 [1]+  Stopped                  xclock
7 rob@rob-HP-ProBook-6470b:$ fg %1
8 xclock
9 ^C
10 rob@rob-HP-ProBook-6470b:$ exit
11 exit
12
13 Script done on Thu 12 Mar 2015 14:59:23 GMT
```

This typescript recording shows how I:

1. Starting the process *xclock* in the foreground,
2. Suspending *xclock* via CTRL-z,
3. Bringing *xclock* back to the foreground using *fg %1*
4. Finally Killing the process with CTRL-C

Listing 2: Scenario 2

```
1  Skip to content
2  This repository
3
4      Explore
5      Gist
6      Blog
7      Help
8
9      Rob Rigler Riglerr
10
11      1
12
13  0
14
15      0
16
17  Riglerr/University-Work
18
19  University-Work/207SE-Networks-&-Security/Portfolio2/Week14/com2.txt
20  Rob Rigler Riglerr 14 hours ago
21  12-3-15
22
23  1 contributor
24  39 lines (29 sloc) 1.256 kb
25  rob@rob-HP-ProBook-6470b:$ xclock &
26  [1] 21811
27  rob@rob-HP-ProBook-6470b:$ xclock
28  ^Z
29  [2]+ Stopped xclock
30  rob@rob-HP-ProBook-6470b:$ jobs
31  [1]- Running xclock &
32  [2]+ Stopped xclock
33  rob@rob-HP-ProBook-6470b:$ kill %1
34  rob@rob-HP-ProBook-6470b:$ jobs
35  [1]- Terminated xclock
36  [2]+ Stopped xclock
37  rob@rob-HP-ProBook-6470b:$ kill -cont %2
38  rob@rob-HP-ProBook-6470b:$ jobs
39  [2]+ Running xclock &
40  rob@rob-HP-ProBook-6470b:$ ps au |grep rob
41  USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
42  rob 16721 0.0 0.0 27336 4448 pts/5 Ss 14:26 0:00 /bin/bash
43  rob 21794 0.0 0.0 21892 960 pts/5 S+ 15:47 0:00 script -a com2.txt
44  rob 21795 0.0 0.0 21896 396 pts/5 S+ 15:47 0:00 script -a com2.txt
45  rob 21796 0.0 0.0 27224 4180 pts/15 Ss 15:47 0:00 bash -i
46  rob 21813 0.0 0.0 70556 4840 pts/15 S 15:47 0:00 xclock
47  rob 21872 0.0 0.0 22648 1320 pts/15 R+ 15:48 0:00 ps au
48  rob@rob-HP-ProBook-6470b:$ kill -9 21813
49  rob@rob-HP-ProBook-6470b:$ jobs
50  [2]+ Killed xclock
51  rob@rob-HP-ProBook-6470b:$ exit
52  exit
```

```
53 | Script done on Thu 12 Mar 2015 15:48:38 GMT
54 |
55 |     Status
56 |     API
57 |     Training
58 |     Shop
59 |     Blog
60 |     About
61 |
62 |     2015 GitHub, Inc.
63 |     Terms
64 |     Privacy
65 |     Security
66 |     Contact
```

1. Starting the *xclock* process in the background,
2. Starting another *xclock* process in the foreground,
3. Suspend the *xclock* foreground process using CTRL-Z,
4. Use the Jobs Keyword to show the two *xclock* processes,
5. Kill the first *xclock* job using kill %1
6. Continue the second *xclock* process in the foreground using kill -cont %2
7. Show a list of my running processes using ps -au — grep rob
8. Finally Kill the remaining *xclock* process by using kill -9 21813

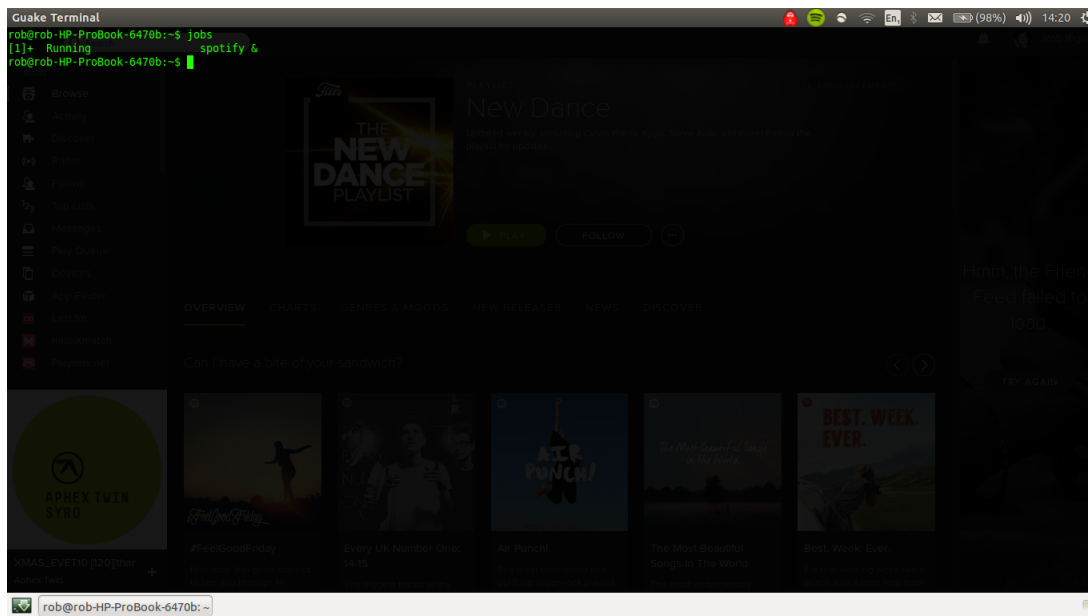
Nohup

Definition : A command which allows a process to continue executing after the parent process has been stopped.

'Nohup' means 'No Hang Up'. Commands that are executed with 'nohup', ignore hang up signals, so that the user can log out of the terminal and the process will still be running in the background. When a process is run in the foreground (no &), it effectively blocks the use of the shell whilst that process is being executed. When a process is run in the background (with &), it is placed into the list of background jobs that the shell is managing, but it is still connected to that shell, so if the shell closes, the process is terminated. NOHUP effectively separates the command from the shell, allowing it to close and the process to continue.

In this example, I am going to use the 'spotify' process as an example.

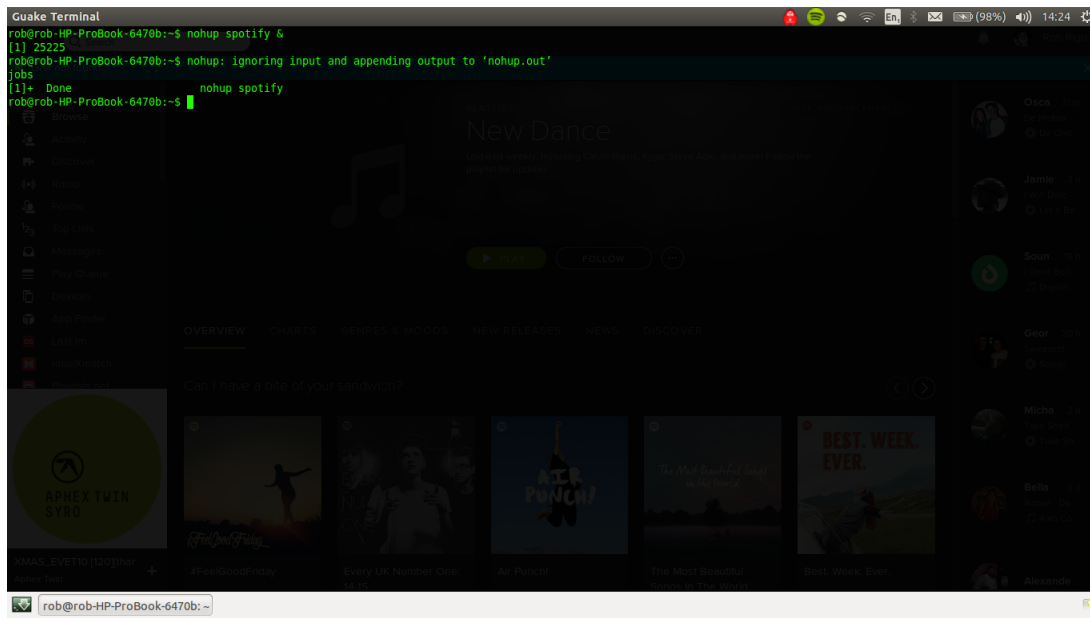
If I type 'spotify &', the spotify application is started and is run in the background, allowing to continue using the shell. When I use the 'jobs' command, we can see that the spotify process is running in the background.



When I exit the shell, the 'spotify' application also closes.

Now, if I type 'nohup spotify &' and look at the terminal jobs. It shows 'nohup spotify &', but now if I

type exit, the application will stay open regardless of the terminal.



Week 17: Pipes and Sockets

RPN Calculator

This task asked me to modify the 'tcp-server' code to evaluate an expression in Reverse Polish Notation (RPN) and return the value to the client. Below I have included the modified code of 'tcp-server', and a screen-shot of the program working. The program is able to evaluate complex expressions such as:

' 100 3 5 6 + * - '

which in prefix notation is :

'100 - 3 * (5+6) '

both of these expressions evaluate to 67.

Listing 3: 'tcp-server.cc' code

```
1 #include <arpa/inet.h>
2
3 #include <netdb.h>
4 #include <netinet/in.h>
5 #include <unistd.h>
6 #include <iostream>
7 #include <cstring>
8 #include <stdlib.h>
9 #include <stdio.h>
10
11 #define MAX_MSG 100
12 #define LINE_ARRAY_SIZE (MAX_MSG+1)
13
14 using namespace std;
15
16 //Create stack, variable and
17 //top fo stack variable.
18 int stack[MAX_MSG];
19 string str_temp;
20 int top;
21
22 //function to push a value.
23 void push(int x){ stack[top++]=x; }
24
25 //function to pop a value.
26 int pop() {
27
28     int t = stack[--top];
29     stack[top]=0;
30     return t;
31 }
32
33
34
35
36
```

```
37 int main()
38 {
39     int listenSocket, connectSocket, i;
40     unsigned short int listenPort;
41     socklen_t clientAddressLength;
42     struct sockaddr_in clientAddress, serverAddress;
43     char line[LINE_ARRAY_SIZE];
44
45     cout << "Enter port number to listen on (between 1500 and 65000): ";
46     cin >> listenPort;
47
48     // Create socket for listening for client connection
49     // requests.
50     listenSocket = socket(AF_INET, SOCK_STREAM, 0);
51     if (listenSocket < 0) {
52         cerr << "cannot create listen socket";
53         exit(1);
54     }
55
56     // Bind listen socket to listen port. First set various
57     // fields in the serverAddress structure, then call
58     // bind().
59
60     // htonl() and htons() convert long integers and short
61     // integers (respectively) from host byte order (on x86
62     // this is Least Significant Byte first) to network byte
63     // order (Most Significant Byte first).
64     serverAddress.sin_family = AF_INET;
65     serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
66     serverAddress.sin_port = htons(listenPort);
67
68     if (bind(listenSocket,
69             (struct sockaddr *) &serverAddress,
70             sizeof(serverAddress)) < 0) {
71         cerr << "cannot bind socket";
72         exit(1);
73     }
74
75     // Wait for connections from clients. This is a
76     // non-blocking call; i.e., it registers this program with
77     // the system as expecting connections on this socket, and
78     // then this thread of execution continues on.
79     listen(listenSocket, 5);
80
81
82
83
84
85
86
87
88
89
```

```
90 while (1) {
91     cout << "Waiting for TCP connection on port " << listenPort << " ...\n";
92
93     // Accept a connection with a client that is requesting
94     // one. The accept() call is a blocking call; i.e., this
95     // thread of execution stops until a connection comes
96     // in. connectSocket is a new socket that the system
97     // provides, separate from listenSocket. We *could*
98     // accept more connections on listenSocket, before
99     // connectSocket is closed, but this program doesn't do
100    // that.
101    clientAddressLength = sizeof(clientAddress);
102    connectSocket = accept(listenSocket,
103                          (struct sockaddr *) &clientAddress,
104                          &clientAddressLength);
105    if (connectSocket < 0) {
106        cerr << "cannot accept connection ";
107        exit(1);
108    }
109
110    // Show the IP address of the client.
111    // inet_ntoa() converts an IP address from binary form to the
112    // standard "numbers and dots" notation.
113    cout << "    connected to " << inet_ntoa(clientAddress.sin_addr);
114
115    // Show the client's port number.
116    // ntohs() converts a short int from network byte order (which is
117    // Most Significant Byte first) to host byte order (which on x86,
118    // for example, is Least Significant Byte first).
119    cout << ":" << ntohs(clientAddress.sin_port) << "\n";
120
121    // Read lines from socket, using recv(), storing them in the line
122    // array. If no messages are currently available, recv() blocks
123    // until one arrives.
124    // First set line to all zeroes, so we'll know where the end of
125    // the string is.
126    memset(line, 0x0, LINE_ARRAY_SIZE);
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
```

```
143 while (recv(connectSocket, line, MAX_MSG, 0) > 0) {
144
145     // RPN evaluation loop.
146     for (i =0; i<LINE_ARRAY_SIZE;i++)
147     {
148         //If number then add it to str_temp
149         if(line[i] >='0' && line[i] <= '9'){
150             str_temp+=line[i];
151         }
152         // if separator char (SPACE)
153         else if(line[i] == ' ')
154         {
155             // if str_temp has a value in it,
156             // then push and clear the string.
157             if(!str_temp.empty()){
158                 push(atoi(str_temp.c_str()));
159                 str_temp.clear();}
160             // else go to next char in input string.
161             else
162                 continue;
163         }
164         // if operator
165         else if(line[i] == '+' || line[i] == '-'
166             ||line[i] == '*' ||line[i] == '/') {
167
168             // pop two value from stack.
169             int a = pop();
170             int b = pop();
171
172             // Do operation depending on operator
173             // push result to stack.
174             switch (line[i]){
175
176             case '+':
177                 push(a+b);
178                 break;
179
180             case '-':
181                 push(b-a);
182                 break;
183
184             case '*':
185                 push(a*b);
186                 break;
187
188             case '/':
189                 push(b/a);
190                 break;
191             }
192         }
193     }
194     // Only value left in stack is the answer.
195     sprintf(line,"%d",pop());
```

```
196
197     // Send converted line back to client.
198     if (send(connectSocket, line, strlen(line) + 1, 0) < 0)
199         cerr << "Error: cannot send modified data";
200
201     memset(line, 0x0, LINE_ARRAY_SIZE); // set line to all zeroes
202 }
203 }
204 }
```

Firstly to be able to evaluate an RPN expression, I needed to implement stack functionality. Lines 16 to 32 show me creating a stack.

I created an integer array to hold the values, and an integer to 'point' to the front on the stack. I then created the two functions which would allow me to push and pop values to and from the stack respectively.

On lines 143 to 195 is where I evaluated the RPN Expression.

Below is a screen-shot of the output on 'tcp-client'. It shows three separate expressions being evaluated :

1. 100 3 5 6 + * - (100 - 3 * (5+6))
2. 1000 345 - (1000 - 345)
3. 280 2 / 4 * (4 * (280/2))

```
rob@rob-HP-ProBook-6470b: ~/Git/University-Work/207SE-Networks-Security/Portfolio2/Week17
rob@rob-HP-ProBook-6470b:~$ cd Git/University-Work/207SE-Networks-Security/Portfolio2/Week17
rob@rob-HP-ProBook-6470b:~/Git/University-Work/207SE-Networks-Security/Portfolio2/Week17$ ./tcp-client0
Enter server host name or IP address: 192.168.0.16
Enter server port number: 8080

Enter an RPN Expression:
Input: 100 3 5 6 + * -
Answer: 67
Input: 1000 345 -
Answer: 655
Input: 280 2 / 4 *
Answer: 560
Input: █
```

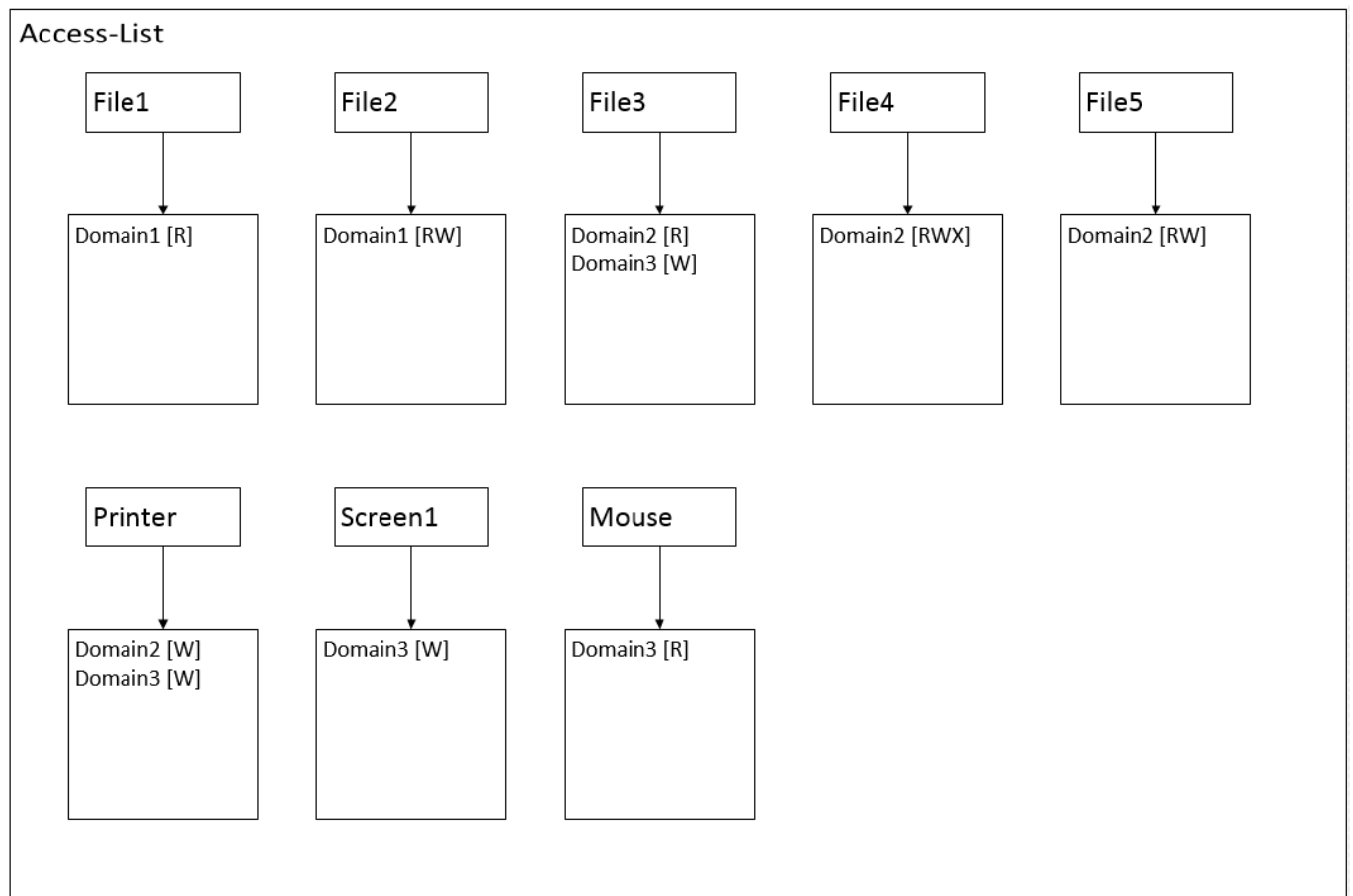
Week 19: Security and Networks

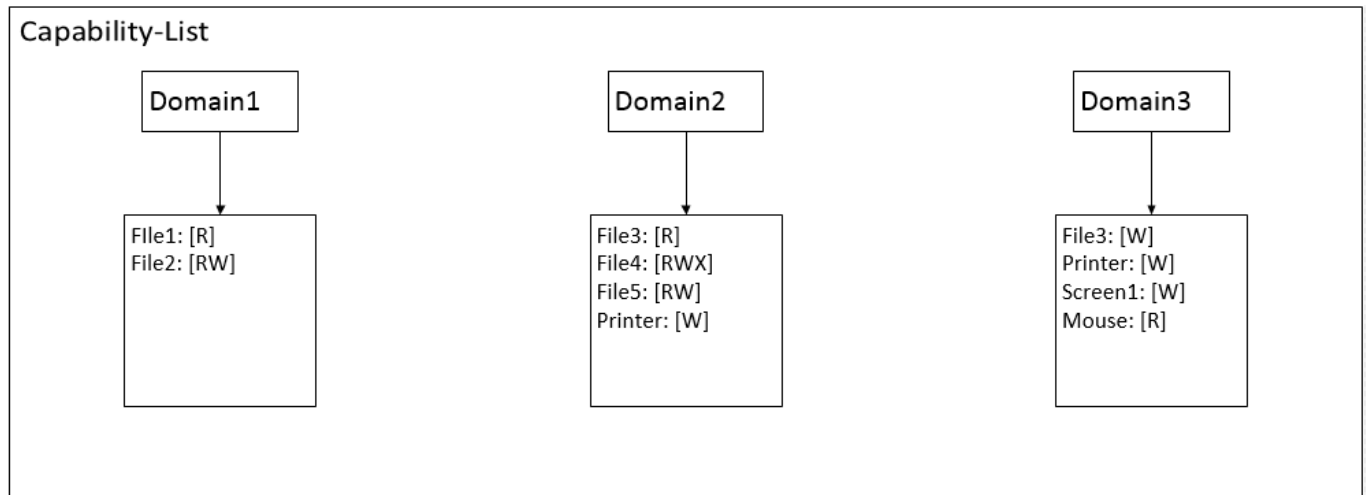
Activity 1: Access Control

Domain Matrix

	File1	File2	File3	File4	File5	Printer	Screen1	Mouse
D1	R	RW						
D2			R	RWX	RW	W		
D3			W			W	W	R

Access-List




Compatibility-Lists

Activity 2: Hash Function

Listing 4: A Simple Hash function in C#

```
1 using System.Text;
2 using System.Threading.Tasks;
3
4 namespace HashFunction
5 {
6     class Program
7     {
8         static void Main(string[] args)
9         {
10             String myName = "Robert";
11             double sum = 0;
12
13
14             for (int i = 0; i < myName.Length; i++) {
15
16                 sum += (myName[i]) * Math.Pow(i+5,i+1);
17
18             }
19             Console.WriteLine("Hash Function of " + myName + ": " + sum);
20             Console.ReadLine();
21         }
22     }
23 }
```

A screenshot of a Windows console window. The title bar shows the file path: file:///C:/Users/Rob/Documents/University-Work/207SE-Networks-Security/Port... The console output displays the text "Hash Function of Robert: 123183302". The console background is black, and the text is white. There is a small white cursor at the end of the first line of output. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.