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

Due on March 16th 2015

Robert Rigler : 4939377

Contents

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

```
Script started on Thu 12 Mar 2015 14:58:19 GMT
rob@rob-HP-ProBook-6470b:$ xclock
^Z
[1]+  Stopped                 xclock
rob@rob-HP-ProBook-6470b:$ jobs
[1]+  Stopped                 xclock
rob@rob-HP-ProBook-6470b:$ fg %1
xclock
^C
rob@rob-HP-ProBook-6470b:$ exit
exit

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

Week 14: Process Manipulation & Nohup continued on next page...

Listing 2: Scenario 2

```
 Skip to content
This repository

    Explore
    Gist
    Blog
    Help

    Rob Rigler Riglerr

    1

0

    0

Riglerr/University-Work

University-Work/207SE-Networks-&-Security/Portfolio2/Week14/com2.txt
Rob Rigler Riglerr 14 hours ago
12-3-15

1 contributor
39 lines (29 sloc) 1.256 kb
rob@rob-HP-ProBook-6470b:$ xclock &
[1] 21811
rob@rob-HP-ProBook-6470b:$ xclock
^Z
[2]+ Stopped xclock
rob@rob-HP-ProBook-6470b:$ jobs
[1]- Running xclock &
[2]+ Stopped xclock
rob@rob-HP-ProBook-6470b:$ kill %1
rob@rob-HP-ProBook-6470b:$ jobs
[1]- Terminated xclock
[2]+ Stopped xclock
rob@rob-HP-ProBook-6470b:$ kill -cont %2
rob@rob-HP-ProBook-6470b:$ jobs
[2]+ Running xclock &
rob@rob-HP-ProBook-6470b:$ ps au |grep rob
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
rob 16721 0.0 0.0 27336 4448 pts/5 Ss 14:26 0:00 /bin/bash
rob 21794 0.0 0.0 21892 960 pts/5 S+ 15:47 0:00 script -a com2.txt
rob 21795 0.0 0.0 21896 396 pts/5 S+ 15:47 0:00 script -a com2.txt
rob 21796 0.0 0.0 27224 4180 pts/15 Ss 15:47 0:00 bash -i
rob 21813 0.0 0.0 70556 4840 pts/15 S 15:47 0:00 xclock
rob 21872 0.0 0.0 22648 1320 pts/15 R+ 15:48 0:00 ps au
rob@rob-HP-ProBook-6470b:$ kill -9 21813
rob@rob-HP-ProBook-6470b:$ jobs
[2]+ Killed xclock
rob@rob-HP-ProBook-6470b:$ exit
exit
```

```
Script done on Thu 12 Mar 2015 15:48:38 GMT

    Status
    API
    Training
    Shop
    Blog
    About

      2015 GitHub, Inc.
    Terms
    Privacy
    Security
    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

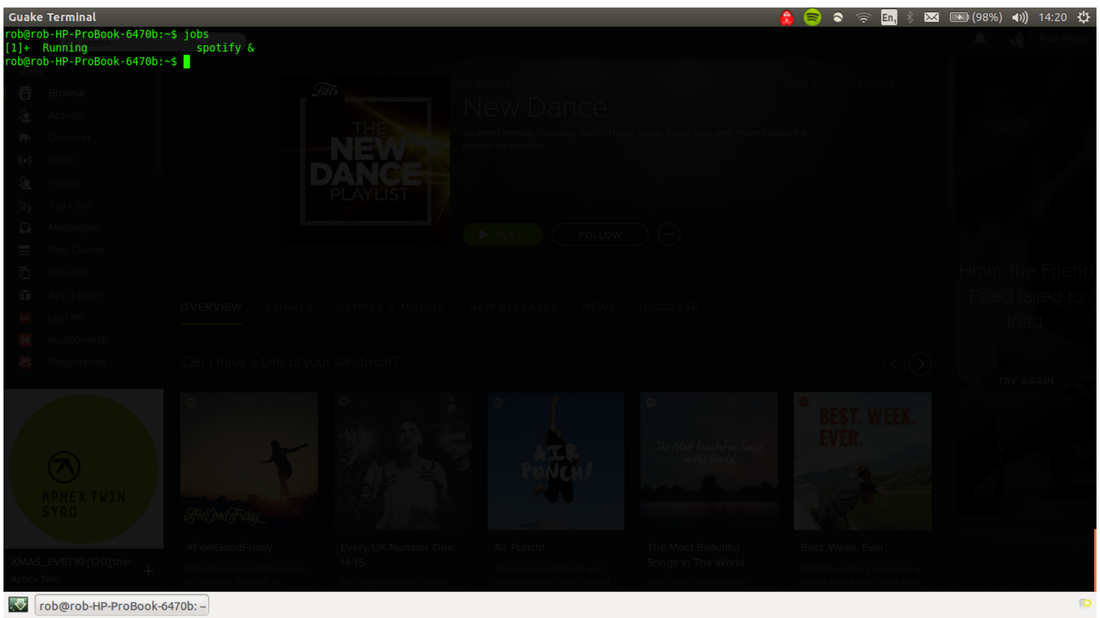
6. Continue the second *xclock* process in the foreground using kill -cont %2

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

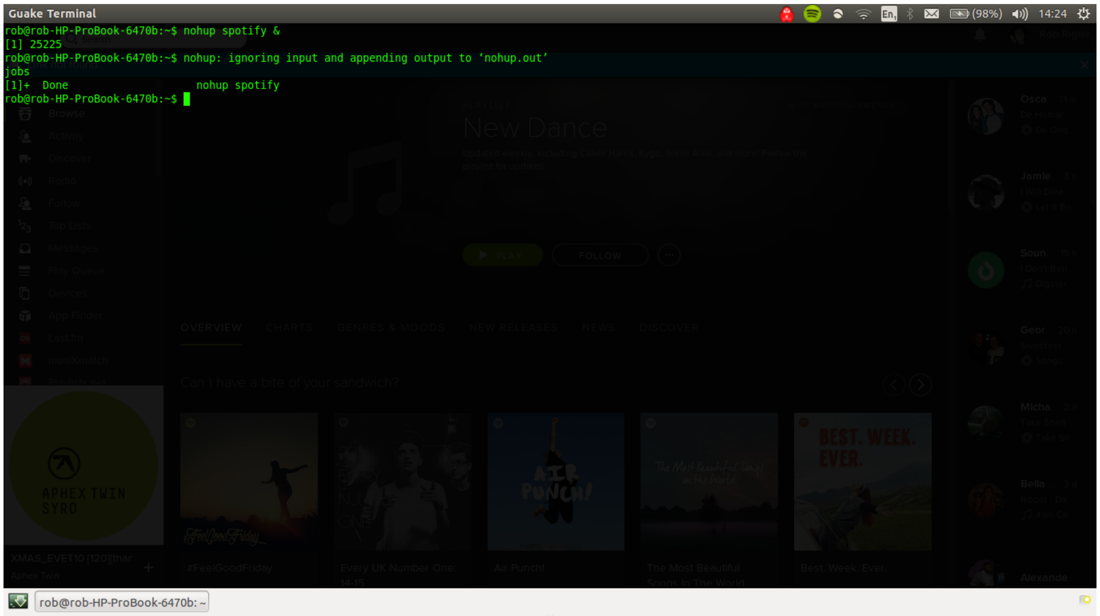
Nohup

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 effictively 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 tasked 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

```
#include <arpa/inet.h>

#include <netdb.h>
#include <netinet/in.h>
#include <unistd.h>
#include <iostream>
#include <cstring>
#include <stdlib.h>
#include <stdio.h>

#define MAX_MSG 100
#define LINE_ARRAY_SIZE (MAX_MSG+1)

using namespace std;

//Create stack, variable and
//top fo stack variable.
int stack[MAX_MSG];
string str_temp;
int top;

//function to push a value.
void push(int x){ stack[top++]=x; }

//function to pop a value.
int pop(){

    int t = stack[--top];
    stack[top]=0;
  return t;

}
```

```
int main()
{
  int listenSocket, connectSocket, i;
  unsigned short int listenPort;
  socklen_t clientAddressLength;
  struct sockaddr_in clientAddress, serverAddress;
  char line[LINE_ARRAY_SIZE];

  cout << "Enter port number to listen on (between 1500 and 65000): ";
  cin >> listenPort;

  // Create socket for listening for client connection
  // requests.
  listenSocket = socket(AF_INET, SOCK_STREAM, 0);
  if (listenSocket < 0) {
    cerr << "cannot create listen socket";
    exit(1);
  }

  // Bind listen socket to listen port. First set various
  // fields in the serverAddress structure, then call
  // bind().

  // htonl() and htons() convert long integers and short
  // integers (respectively) from host byte order (on x86
  // this is Least Significant Byte first) to network byte
  // order (Most Significant Byte first).
  serverAddress.sin_family = AF_INET;
  serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
  serverAddress.sin_port = htons(listenPort);

  if (bind(listenSocket,
           (struct sockaddr *) &serverAddress,
           sizeof(serverAddress)) < 0) {
    cerr << "cannot bind socket";
    exit(1);
  }

  // Wait for connections from clients. This is a
  // non-blocking call; i.e., it registers this program with
  // the system as expecting connections on this socket, and
  // then this thread of execution continues on.
  listen(listenSocket, 5);
```

```
  while (1) {
    cout << "Waiting for TCP connection on port " << listenPort << " ...\n";

    // Accept a connection with a client that is requesting
    // one. The accept() call is a blocking call; i.e., this
    // thread of execution stops until a connection comes
    // in. connectSocket is a new socket that the system
    // provides, separate from listenSocket. We *could*
    // accept more connections on listenSocket, before
    // connectSocket is closed, but this program doesn't do
    // that.
    clientAddressLength = sizeof(clientAddress);
    connectSocket = accept(listenSocket,
                           (struct sockaddr *) &clientAddress,
                           &clientAddressLength);
    if (connectSocket < 0) {
      cerr << "cannot accept connection ";
      exit(1);
    }

    // Show the IP address of the client.
    // inet_ntoa() converts an IP address from binary form to the
    // standard "numbers and dots" notation.
    cout << "  connected to " << inet_ntoa(clientAddress.sin_addr);

    // Show the client's port number.
    // ntohs() converts a short int from network byte order (which is
    // Most Significant Byte first) to host byte order (which on x86,
    // for example, is Least Significant Byte first).
    cout << ":" << ntohs(clientAddress.sin_port) << "\n";

    // Read lines from socket, using recv(), storing them in the line
    // array.  If no messages are currently available, recv() blocks
    // until one arrives.
    // First set line to all zeroes, so we'll know where the end of
    // the string is.
    memset(line, 0x0, LINE_ARRAY_SIZE);
```

```
    while (recv(connectSocket, line, MAX_MSG, 0) > 0) {

      // RPN evaluation loop.
    for (i =0; i<LINE_ARRAY_SIZE;i++)
    {
        //If number then add it to str_temp
        if(line[i] >='0' && line[i] <= '9'){
        str_temp+=line[i];
        }
        // if separator char (SPACE)
        else if(line[i] == ' ')
        {
          // if str_temp has a value in it,
          //  then push and clear the string.
            if(!str_temp.empty()){
                push(atoi(str_temp.c_str()));
                str_temp.clear();}
                // else go to next char in input string.
            else
                continue;
        }
        // if operator
        else if(line[i] == '+' || line[i] == '-'
                ||line[i] == '*' ||line[i] == '/' ){

          // pop two value from stack.
        int a = pop();
        int b = pop();

          // Do operation depending on operator
          // push result to stack.
        switch (line[i]){

        case '+':
            push(a+b);
            break;

        case '-' :
            push(b-a);
            break;

        case '*':
            push(a*b);
            break;

        case '/':
            push(b/a);
            break;
        }
    }
  }
// Only value left in stack is the answer.
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 funtions 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 out 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/Port
folio2/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: 
```