CS 342 – OPERATING SYSTEMS

Project 1

Elif Gülşah Kaşdoğan

21601183

Section 2

Bilshell is a command line interpreter which can work in batch and interactive modes, for further explanation please refer to README file. After implementing bilshell command line interpreter, we are required to do some experiments with the built in commands producer and consumer. I performed my experiments as follows:

N restricts the number of bytes flowing through the pipes, as N increases number of system calls (which is read()) decrease, this has an effect on elapsed time during the execution of the program. We will be particularly interested in amount of time spent in kernel mode since the read and write calls are kernel system calls. I made two tables for experiment results. In each execution of the compound command, we provide number M to producer and consumer which is the number of characters. By using different M and N values I made 2 tables to examine effect of N and M on execution time, and amount of kernel time spent by program. “sys” value obtained from the time command represents the amount of time spent in CPU executing kernel calls, in other words it gives us the time amount spent in “kernel-mode”.

First table shows the correlation between kernel time and value M (input size)

M	N	read-call-count	character-count	user	sys	Total CPU time
10	10	1	10	0,002	0,006	0,008
40	10	4	40	0,001	0,007	0,008
50	10	5	50	0,002	0,006	0,008
80	10	8	80	0,001	0,008	0,009
100	10	10	100	0,001	0,007	0,008
400	10	40	400	0,001	0,008	0,009
800	10	80	800	0,001	0,008	0,009
1000	10	100	1000	0,002	0,008	0,010
4000	10	400	4000	0,003	0,008	0,011
8000	10	800	8000	0,006	0,007	0,013
10000	10	1000	10000	0,007	0,008	0,015
15000	10	1500	15000	0,016	0,008	0,024
20000	10	2000	20000	0,007	0,019	0,026
30000	10	3000	30000	0,008	0,027	0,035
50000	10	5000	50000	0,007	0,035	0,042
60000	10	6000	60000	0,012	0,039	0,051

As we can see from the table, amount of time spent in kernel mode increases as the input size increase when we use a fixed size of N. For different input sizes we have N = 10 meaning we will call a read system call for each 10 input, as input size increases number of system calls made by our program increases. This results in increased kernel time.

The second table demonstrates the effect of N over a fixed input size ie the relationship between kernel time and N(amount of bytes read at a time).

M	N	read-call-count	character-count	user	sys	Total CPU time
60000	1	60000	60000	0,075	0,305	0,380
60000	2	30000	60000	0,056	0,146	0,202
60000	3	20000	60000	0,020	0,115	0,135
60000	4	15000	60000	0,025	0,083	0,108
60000	5	12000	60000	0,015	0,073	0,088
60000	6	10000	60000	0,024	0,050	0,074
60000	7	8572	60000	0,020	0,040	0,064
60000	8	7500	60000	0,013	0,048	0,061
60000	9	6667	60000	0,017	0,038	0,055
60000	10	6000	60000	0,009	0,039	0,048
60000	20	3000	60000	0,000	0,028	0,028
60000	30	2000	60000	0,003	0,021	0,024
60000	40	1500	60000	0,004	0,012	0,016
60000	50	1200	60000	0,012	0,007	0,019
60000	100	600	60000	0,012	0,002	0,014
60000	400	150	60000	0,007	0,005	0,012
60000	800	75	60000	0,007	0,004	0,011
60000	1000	60	60000	0,001	0,010	0,011
60000	1900	32	60800	0,006	0,004	0,010
60000	2000	30	60000	0,004	0,006	0,010
60000	8000	8	64000	0,008	0,001	0,009
60000	10000	6	60000	0,010	0,001	0,011

60000	15000	4	60000	0,004	0,006	0,010
60000	60000	1	60000	0,004	0,006	0,010

When we have N = i this means we will read i elements at a time, changing the N value over a fixed size of M causes change in number of read system call. By looking at the first and the second table we can see the general relation:

$$M = N * read\_call\_count$$

By looking at the second table we can see that kernel mode-time increases as the number of system calls increase. This part of the experiment gives better results with large input size M, for small M difference might be considered negligible.

As a conclusion, characters read/written at a time during the execution of a piped process affect the number of read/write system calls. Since the read/write calls are provided by kernel, amount of time spent in kernel mode is affected by the number of read/write calls. By looking at the provided tables, we can see that the experimental results are also consistent with this fact.

Some methods:

Random character generator

```
//Generates M random variables and fills string.
void generateRandom(char *string, int M){
   static const char alphanumeric[] =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
   for (int i = 0; i < M; ++i) {
      string[i] = alphanumeric[rand() % (sizeof(alphanumeric) - 1)];
   }
}
```

Built-in producer command, generates M random alphanumerics and displays them N by N

```
void producer(int M, int N){
   char string[M];
   generateRandom(string, M);
   int remaining = M;
   int count = 0;
   while(remaining > 0){
      write(1, string, (size_t) N); //ins
      remaining -= N;
      count++;
   }
}
```

Built-in consumer command, reads M alphanumerics from standart input

```
void consumer(int M, int N){
   int remaining = M;
   char buffer[N];
   int count = 0;
   while(remaining > 0){
      read(0, buffer, sizeof(buffer));
      remaining -= N;
      count++;
   }
}
```

Method for executing a built-in command

```
int executeBuiltIn(char** command, int N){
   //command comes space parsed.
   //some commands are not executed by using cvp, these will be added to here
   int numberOfCommands = 5;
   int i;
   char* builtIn[numberOfCommands];
   builtIn[0] = "exit";
   builtIn[1] = "cd";
   builtIn[2] = "help";
   builtIn[3] = "producer";
   builtIn[4] = "consumer";
   char** rest_of_command = NULL;

   for(i = 0; i< numberOfCommands; i++){
      if(strcmp(*command, builtIn[i]) == 0){
         //they are same
         break;
      }
   }
   switch(i){
      case 0: printf(" Bye!\n"); exit(0); break;
      case 1: chdir(command[1]); break;
      case 2: printf("Welcome bilshell!\nType exit to terminate shell\n**Please checkout the README
file for details\n"); break;
      case 3: //producer
         if(command[1] != NULL ){
            int size = atoi(command[1]);
            if(size > 0)
               producer(size, N);
         }
         break;
```

```
        case 4: //consumer
            if(command[1] != NULL){
                int size = atoi(command[1]);
                if(size > 0)
                    consumer(size, N);
            }
            break;
        default: break;
    }
    if(i < numberOfCommands)
        return 1; //it is one of our functions
    else
        return 0; //it is not a built in
}
```

Method for executing simple command(not compound)

```
int executeSingular(char** parsed, int N){
    //execute arguments for singular command
    if(executeBuiltIn(parsed, N))
        return 2;
    pid_t pid = fork();
    if(pid < 0){
        //failed
        printf("ERROR: Creation of child process!\n");
        return 0;
    }
    else if(pid == 0){
        //child
        int executed = execvp(*parsed, parsed);
        if(executed < 0){
            printf("WARNING: Could not execute command\n");
        }
        exit(0);
    }else{
        //parent
        wait(NULL); //wait for child process
        return 1;
    }
}
```

Method for executing compound command with two pipes

```
void executePipedArg(char** piped1, char** piped2, int N){
    //piped1 --> child1 process
    //piped2 --> child2 process
    int pipe1_fd[2];
    int pipe2_fd[2]; //file descriptors
    ssize_t nbytes = 0;
    ssize_t read_byte = 0;
    ssize_t write_byte = 0;

    pid_t child1, child2;
    char buffer[N];

    if(pipe(pipe1_fd) < 0 || pipe(pipe2_fd) < 0){
       //either of them had a problem during initialization
       printf("ERROR: Pipe initialization!\n");
       return;
    }
    child1 = fork();
    if(child1 == 0){
       //child1
       //printf("Child1 executes \n");
       dup2(pipe1_fd[1], 1);
       //execvp(piped1[0], piped1);
       executeSingular(piped1, N);

       exit(0);
    }
    else if(child1 > 0){
       //parent
       close(pipe1_fd[1]); //write end of pipe1
       wait(NULL);

       while(nbytes = read(pipe1_fd[0], buffer, (size_t)N) > 0){
          write_byte += write(pipe2_fd[1], buffer, (size_t) N);
          read_byte += nbytes;
       }

       close(pipe1_fd[0]);
       close(pipe2_fd[1]);

       child2 = fork();
       if(child2 == 0){
          //child2
          //printf("Child2 executes\n");
          dup2(pipe2_fd[0], 0);
          //execvp(piped2[0], piped2);
          executeSingular(piped2, N);

          exit(0);
```

```
        }
        else if(child2 > 0){
            //parent
            close(pipe2_fd[0]);
            //printf("Child2 terminated\n");
            wait(NULL);
            printf("read-call-count: %ld\n", read_byte);
            printf("character-count: %ld\n", write_byte); //total number rof bytes written

                            //printf("%ld\n", read_byte);
            //printf("character-count: %ld\n", write_byte); //total number rof bytes written
        }
    }
}
```

Batch mode executer, reads commands from a file rather than stdin

```
void executeBatch(char** parsed, int N){
    //open the file with the given name
    char* filename = parsed[2];
    FILE* fp = NULL;
    char buffer[50]; //maximum number of letters you can read
    fp = fopen(filename, "r");
    char* spaceParsed[NUM_OF_COMMANDS];
    char* textCmds[NUM_OF_COMMANDS];
    int newN = 1;
    char* param;

    if(fp == NULL){
        printf("\nWARNING: File does not exist!\n");
        return;
    }
    else{
        char c;
        int count = 0;
        for (c = getc(fp); c != EOF; c = getc(fp)){
            if (c == '\n')
                count = count + 1;
        }
        //printf("Number of lines: %d ", count);
        fclose(fp);
        fp = fopen(filename, "r");

        for(int i = 0; i < count; i++){
            fgets(buffer, sizeof(buffer), fp);
            //printf(buffer);
```

```
         buffer[sizeof(buffer)-1] = '\0';

         param = strtok(buffer, "\n\r\a");
         newN = atoi(parsed[1]);
         if(newN <= 0) //not valid
            newN = N;
         if(param != NULL)
            commandProcessing(param, newN);
      }
      fclose(fp);
   }
}
```

Checks the type of command(singular, compound or batch), calls related executer.

```
void commandProcessing(char* str, int N){
   char* spaceParsed[NUM_OF_COMMANDS];
   char* parsed1[NUM_OF_COMMANDS];
   char* parsed2[NUM_OF_COMMANDS];

   char pipeInput[NUM_OF_LETTERS]; //input to pipe
   char timeInput[NUM_OF_LETTERS]; //input to time
   strcpy(pipeInput, str);
   strcpy(timeInput, str);

   parseSpace(str, spaceParsed); //now all elements are in spaceParsed array
   int batch = isBatch(spaceParsed);
   int isPipe = parsePiped(pipeInput, parsed1, parsed2);

   if(spaceParsed[0] == NULL)
      return;

   if(batch > 0){
      executeBatch(spaceParsed, N);
   } else if(isPipe > 0){
      executePipedArg(parsed1, parsed2, N);
   }
   else{
      executeSingular(spaceParsed, N);
   }

}
```