Operating system

Assignment 2.

---

1. 사전조사보고서

2. 실습보고서

컴퓨터과학과 2013147513
조영재

사 전 조 사 보 고 서

1. 쉘의 구조와 실습에서 요구하는 명령어의 구현 방식

쉘은 사용자로부터의 명령을 키보드로 입력받아 수행해 주는 프로그램이다. 쉘에서 명령어를 입력하게 되면, 쉘은 fork를 통해 프로세스 복제를 하고, exec를 통해 복제된 프로세스를 사용자가 원하는 명령을 수행할 수 있는 프로세스로 변환해 준다.

실습에서 요구하는 명령어 중 ①cd와 입출력 리다이렉션의 경우와, ②환경변수에 등록되어 있는 명령어 그리고 ③&입력 시 쉘이 백그라운드 프로세스를 기다리지 않게 하는 것 3가지로 나누어서 구현한다.

① cd와 입출력 리다이렉션은 그 명령을 가리키는 단어나 문자인 cd, <, >가 나타나는 순간 실행되게 한다.

② 환경변수에 등록되어 있는 프로그램을 실행하기 위해서 execvp함수를 사용하였다. exec family중 환경변수에 등록되어 있는 프로그램을 실행하기 위한 함수이므로, 이를 통하여 프로그램들을 실행시킨다.

③ &가 마지막에 붙을 경우, wait명령어를 건너뛰도록 설계한다.

2. Process와 Thread의 차이, 리눅스에서의 구조, 구현 등

Process는 CPU에서 실행되고 있는 프로그램을 말한다. Thread는 Process내에서 공유할 수 있는 자원을 공유하고 있지만, 서로 다른 작업을 수행중인 것들의 단위로 쪼갠 것을 말한다. 기존의 Solaris에서의 Thread의 구조는 Process내부에 Thread만을 위한 별도의 자료구조가 있고, 이를 통하여 Thread를 관리하였다. 하지만 리눅스에 Thread는 Process를 복제해서 생기며, 따라서 Process와 동일한 자료구조를 갖는다. 구체적으로는 pthread_create함수를 사용할 경우, 내부에서 clone()시스템콜이 사용되어 부모 Process와 동일한 프로세스를 생성하게 된다.

실 습 보 고 서

1. 파일 설명

```
yj@ubuntu:~/Desktop/project2$ ls
Makefile    header.h    lr_process   miniShell        output.txt   prototype.h
getcmd.c    init.c      lr_thread    multiprocess.c   parser.c     prototype.h.gch
gt_init.c   input.txt   main.c       multithread.c    prompt.c
```

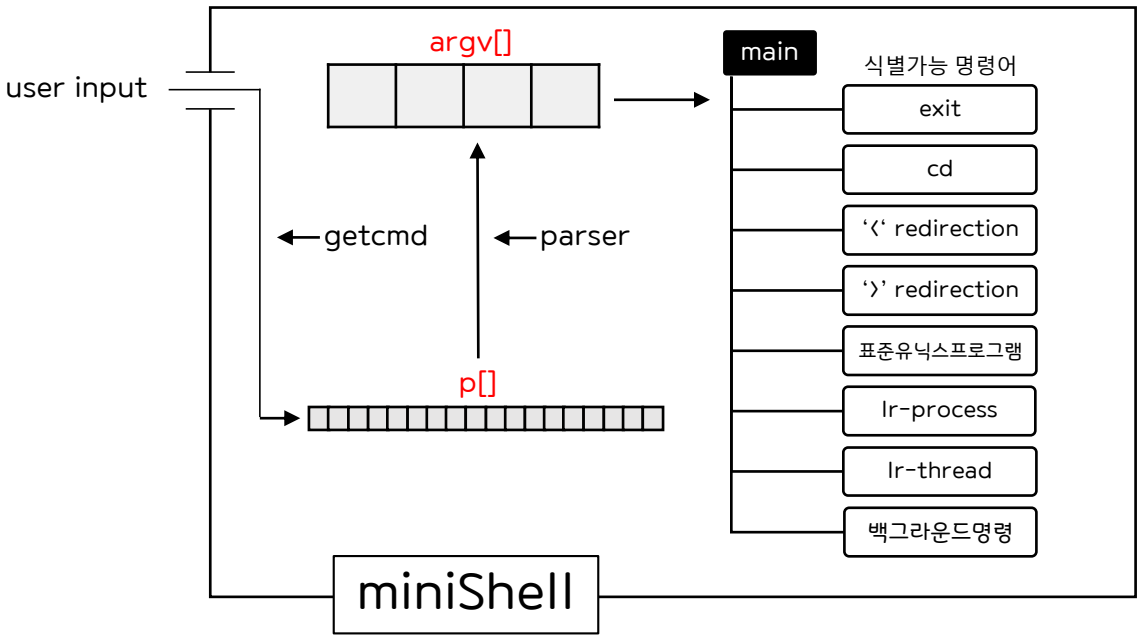
main.c: miniShell 실행파일을 생성하는 소스코드

- getcmd.c: 사용자로부터 커맨드를 입력받아 배열에 저장하는 함수코드
- init.c: 한 명령이 수행되고 나면 쉘의 기록들을 초기화 해주는 함수코드
- gt_init.c: '>'리다이렉션시에 쉘을 초기화 해주는 함수코드
- parser.c: 사용자의 입력을 분석하는 함수코드
- prompt.c: 프롬프트를 출력하는 함수코드
- prototype.h, header.h: 위의 코드에 필요한 헤더파일과 상수, 함수원형 선언 헤더

multiprocess.c, multithread.c: Makefile에 의해 각각 lr_process와 lr_thread 실행파일을 생성하는 소스코드. 멀티프로세스와 멀티스레드를 구현한 코드이다.

2. 수행과정 및 세부내용

전반적인 miniShell의 구조는 다음과 같다



2-1. getcmd()와 parser()의 작동원리, '<', '>', '&'처리방법

miniShell을 만들기 위해 사용자의 입력이 어떠한 명령어인지 구분할 필요가 있다. 이를 위해 다음과 같은 절차로 사용자 입력을 받았다.

1. 사용자의 입력을 p[] 배열에 저장한다.
2. p[]배열의 맨 끝에 '&'가 있는지 확인하여 명령어의 백그라운드 실행여부를 결정한다
3. 이후 p[]배열을 parsing함수로 넘겨주게 되는데, argument들이 저장될 argv[]배열과 그 배열의 개수 argc를 같이 넘겨주고, p[]배열의 공백을 기준으로 argument들을 임시로 저장할 임시배열 tempArg[]또한 인자로 넘겨준다. 이때, '<', '>'는 argv[]배열에서 들어가지 않게된다.
4. parsing()함수 후 사용자로부터 입력받은 명령어가 argv[]에 순차적으로 들어가 있다.
5. argv[0]의 내용을 확인하여 exit, cd, lr-thread, lr-process를 수행할지 결정한다.
6. p[]배열에 '<' 또는 '>'가 있는지 확인하여 있다면 해당 리다이렉션을 실행하도록 한다. '<' 리다이렉션의 경우 '<'를 제외한 모든 argument들을 argv[]로 묶어 execvp()에 인자로 넘겨주었다.

2-2. 프로세스 작동

miniShell에서 표준 리눅스 프로그램을 실행하거나, 새로운 프로세스를 생성할 때 다음과 같은 방법을 사용하였다.

```
switch(fork()){
    case -1:
        printf("ERROR1\n");
        exit(0);
    case 0:
        if(execvp(argv[0], argv) == -1){
            printf("fail\n");
            exit(0);
        }
        break;
    default:
        //background process
        if(!background) wait(&proc_status);
        break;
}
```

fork()를 통하여 프로세스를 복제한 다음, 그 값이 0일경우는 자식프로세스이므로, 그 안에서 execvp를 하여 원하는 프로세스로 바뀌도록 하였다. execvp는 환경변수에 등록된 프로그램을 실행시킬때 사용되며, 인자로 배열을 넘겨주도록 되어있으므로 위와같이 사용하였다. 또한 부모프로세스는 switch문 내에서 default부분으로 이동하게 되므로, 이곳에서 백그라운드 실행여부에 따라 자식프로세스의 종료를 기다리는 wait()함수를 사용하였다.

2-3. lr-thread

다음과 같이 구현하였다.

1. input.txt의 처음부터 끝까지 쭉 읽어가면서 x와 y의 총 합계를 구하고, 총 몇줄인지 '₩n'의 개수를 센다.

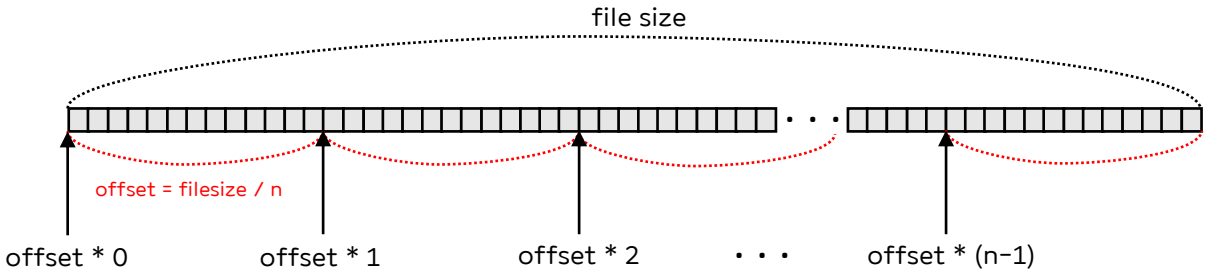
2. 스레드 배열, 인자로 넘겨줄 구조체 배열, 리턴값을 받을 구조체배열을 각각 스레드 개수만큼 선언

```
pthread_t threads[n];
//passing arguments;
struct t_arg *targ[n];

//receive thread's return value
struct fraction *frac[n];
```

3. input.txt를 할당된 스레드 개수만큼 등분한다. 등분된 조각들의 시작점의 offset을 스레드함수의 인자로 넘겨준다. 등분하는 방법은 다음과 같다.

a) input.txt를 하나의 선형적인 1차원 배열로 생각한다. 이후 fseek과 ftell을 통하여 총 문자열의 개수를 구한다(file size). 이를 스레드 개수 n으로 나눈다. 이 값을 unit offset으로 한 후, 스레드 개수 n만큼 for문을 돌면서 (unit offset * i) 해준다.



b) 만약 구한 offset이 줄의 시작점이나 끝부분이 아니라 숫자의 중간지점이라면, 다음 개행문자 '₩n'가 있는곳까지 offset을 조절해준다. 이렇게 구한 offset을 해당 스레드의 인자로 전달해준다.

4. 각각의 스레드마다 파일을 열고, 인자로 넘겨받은 offset만큼 파일포인터를 이동시켜 주어진 line수만큼 x, y값을 읽어 합을 구한다. 구한 값은 struct coord형태의 구조체에 저장하여 return값으로 한다.

5. n개의 스레드 수 만큼 for문을 돌면서 pthread_create를 하고, 스레드 종료까지 메인프로세스가 끝나면 안되므로 pthread_join을 사용해 준다.

```
for(int i=0; i<n; i++){
    pthread_create(&threads[i], NULL, linear_regression, (void*)targ[i]);
}
for(int i=0; i<n; i++){
    pthread_join(threads[i], (void *)&frac[i]);
    boonja += frac[i]->numerator;
    boonmo += frac[i]->denominator;
    free(frac[i]);
}
```

2-4. lr-process

lr-process에서 input파일을 분할하고, 값을 읽고 계산하는 방식은 lr-thread와 동일하다. 하지만 lr-thread에서는 pthread_join함수에서 마지막 인자로 스레드의 리턴값이 저장되는 곳을 지정할 수 있었지만, process간에는 shared memory를 사용하거나 pipe를 사용해서만 서로의 결과값을 공유할 수 있다. 하지만 내 쉘에서 pipe는 구현하지 않았고, shared memory에서는 synchronize문제가 생길것 같았다. 그래서 마음편히 임의의 파일을 생성하고, 그 파일을 마치 shared memory인 것처럼 그곳에 자식 프로세스는 결과값을 쓰고, 부모프로세스는 그 결과값을 읽어서 계산을 수행하는 방식으로 구현하였다. 핵심적인 코드는 다음과 같다.

```
for(int i=0; i<n; i++){
    pid_t pid = fork();          <-- n개의 프로세스 생성
    double x, y;
    switch(pid){
        case -1:
            printf("fork error\n");
            exit(0);
        case 0:
            fp = fopen(argv[3], "r");
            fseek(fp, line*i, SEEK_SET);
            while((nl = fgetc(fp)) != '\n'){};
            for(int j=0; j<line; j++){
                fscanf(fp, "%lf %lf\n", &x, &y);
                p[i].boonja += (x - avgx)*(y - avgy);
                p[i].boonmo += (x - avgx)*(x - avgx);
            }
            fclose(fp);                                   <-- output파일에 이어쓰기
            destfp = fopen("output.txt", "a");
            fprintf(destfp, "%f %f\n", p[i].boonja, p[i].boonmo);
            fclose(destfp);
            return 0;            <-- 자식프로세스 종료
            break;
        default:
            break;
    }
}
wait((int*) 0);                  <-- 자식프로세스 종료 대기
```

wait()이후 다음과 같이 부모프로세스에서 output.txt의 결과값을 읽어서 값을 계산한다.

```
fp = fopen("output.txt", "r");
for(int i=0; i<n; i++){
    fscanf(fp, "%lf %lf\n", &x, &y);
    final_boonja += x;
    final_boonmo += y;
}
```

3. 결과분석 및 토론

스레드와 프로세스간의 실행시간을 비교하고, 각각의 수에 따른 실행시간을 비교하기 위하여

	스레드 대 프로세스 비교	
개수 비교	4스레드	4프로세스
	40스레드	40프로세스

4가지 경우로 나누어 실행시간을 비교하였고, 그 결과는 다음과 같다.

멀티스레드

```
[17:46:29]YJ/home/yj/Desktop/project2$lr-thread -n 4 < input.txt
y = -0.098744 + 0.849012x
```

멀티프로세스

```
[17:47:11]YJ/home/yj/Desktop/project2$lr-process -n 4 < input.txt
y = 0.073236 + 0.848978x
```

4스레드

```
yj@ubuntu:~/Desktop/project2$ ps -eaT | grep lr_thread
 52514   52514 pts/4    00:00:19 lr_thread
 52514   52527 pts/4    00:00:05 lr_thread
 52514   52528 pts/4    00:00:05 lr_thread
 52514   52529 pts/4    00:00:05 lr_thread
 52514   52530 pts/4    00:00:05 lr_thread
```

4프로세스

```
yj@ubuntu:~/Desktop/project2$ ps -eaT | grep lr_process
 52550   52550 pts/4    00:00:19 lr_process
 52559   52559 pts/4    00:00:04 lr_process
 52560   52560 pts/4    00:00:04 lr_process
 52561   52561 pts/4    00:00:04 lr_process
 52562   52562 pts/4    00:00:04 lr_process
```

40스레드

```
yj@ubuntu:~/Desktop/project2$ ps -eaT | grep lr_thread
 52576  52576 pts/4    00:00:19 lr_thread
 52576  52590 pts/4    00:00:00 lr_thread
 52576  52591 pts/4    00:00:00 lr_thread
 52576  52592 pts/4    00:00:00 lr_thread
 52576  52593 pts/4    00:00:00 lr_thread
 52576  52594 pts/4    00:00:00 lr_thread
 52576  52595 pts/4    00:00:00 lr_thread
 52576  52596 pts/4    00:00:00 lr_thread
 52576  52597 pts/4    00:00:00 lr_thread
 52576  52598 pts/4    00:00:00 lr_thread
 52576  52599 pts/4    00:00:00 lr_thread
 52576  52600 pts/4    00:00:00 lr_thread
 52576  52601 pts/4    00:00:00 lr_thread
 52576  52602 pts/4    00:00:00 lr_thread
 52576  52603 pts/4    00:00:00 lr_thread
 52576  52604 pts/4    00:00:00 lr_thread
 52576  52605 pts/4    00:00:00 lr_thread
 52576  52606 pts/4    00:00:00 lr_thread
 52576  52607 pts/4    00:00:00 lr_thread
 52576  52608 pts/4    00:00:00 lr_thread
 52576  52609 pts/4    00:00:00 lr_thread
 52576  52610 pts/4    00:00:00 lr_thread
 52576  52611 pts/4    00:00:00 lr_thread
 52576  52612 pts/4    00:00:00 lr_thread
 52576  52613 pts/4    00:00:00 lr_thread
 52576  52614 pts/4    00:00:00 lr_thread
 52576  52615 pts/4    00:00:00 lr_thread
 52576  52616 pts/4    00:00:00 lr_thread
 52576  52617 pts/4    00:00:00 lr_thread
 52576  52618 pts/4    00:00:00 lr_thread
 52576  52619 pts/4    00:00:00 lr_thread
 52576  52620 pts/4    00:00:00 lr_thread
 52576  52621 pts/4    00:00:00 lr_thread
 52576  52622 pts/4    00:00:00 lr_thread
 52576  52623 pts/4    00:00:00 lr_thread
 52576  52624 pts/4    00:00:00 lr_thread
 52576  52625 pts/4    00:00:00 lr_thread
 52576  52626 pts/4    00:00:00 lr_thread
 52576  52627 pts/4    00:00:00 lr_thread
 52576  52628 pts/4    00:00:00 lr_thread
 52576  52629 pts/4    00:00:00 lr_thread
```

40프로세스

```
yj@ubuntu:~/Desktop/project2$ ps -eaT | grep lr-process
 52735  52735 pts/4    00:00:19 lr-process
 52751  52751 pts/4    00:00:00 lr-process
 52752  52752 pts/4    00:00:00 lr-process
 52753  52753 pts/4    00:00:00 lr-process
 52754  52754 pts/4    00:00:00 lr-process
 52755  52755 pts/4    00:00:00 lr-process
 52756  52756 pts/4    00:00:00 lr-process
 52757  52757 pts/4    00:00:00 lr-process
 52758  52758 pts/4    00:00:00 lr-process
 52759  52759 pts/4    00:00:00 lr-process
 52760  52760 pts/4    00:00:00 lr-process
 52761  52761 pts/4    00:00:00 lr-process
 52762  52762 pts/4    00:00:00 lr-process
 52763  52763 pts/4    00:00:00 lr-process
 52764  52764 pts/4    00:00:00 lr-process
 52765  52765 pts/4    00:00:00 lr-process
 52766  52766 pts/4    00:00:00 lr-process
 52767  52767 pts/4    00:00:00 lr-process
 52768  52768 pts/4    00:00:00 lr-process
 52769  52769 pts/4    00:00:00 lr-process
 52770  52770 pts/4    00:00:00 lr-process
 52771  52771 pts/4    00:00:00 lr-process
 52772  52772 pts/4    00:00:00 lr-process
 52773  52773 pts/4    00:00:00 lr-process
 52774  52774 pts/4    00:00:00 lr-process
 52775  52775 pts/4    00:00:00 lr-process
 52776  52776 pts/4    00:00:00 lr-process
 52777  52777 pts/4    00:00:00 lr-process
 52778  52778 pts/4    00:00:00 lr-process
 52779  52779 pts/4    00:00:00 lr-process
 52780  52780 pts/4    00:00:00 lr-process
 52781  52781 pts/4    00:00:00 lr-process
 52782  52782 pts/4    00:00:00 lr-process
 52783  52783 pts/4    00:00:00 lr-process
 52784  52784 pts/4    00:00:00 lr-process
 52785  52785 pts/4    00:00:00 lr-process
 52786  52786 pts/4    00:00:00 lr-process
 52787  52787 pts/4    00:00:00 lr-process
 52788  52788 pts/4    00:00:00 lr-process
 52789  52789 pts/4    00:00:00 lr-process
 52790  52790 pts/4    00:00:00 lr-process
```

결과를 분석해 보면, 우선 4가지 경우 모두 첫번째 프로세스는 19초가 걸린것을 알 수 있다. 이는 프로세스, 스레드, 개수에 상관없이 모두 우선적으로 input.txt를 처음부터 끝까지 한번 읽어 x와 y의 합을 구하는 과정이 있기 때문이다. 이 과정이 끝난 후에 본격적으로 스레드와 프로세스가 생성된다. 예상했던 바는 스레드가 프로세스보다 빠르고, 개수가 많을수록 빨라지는 것이었다. 개수가 많을수록 계산해야 하는 양이 적어지므로 당연히 개당 스레드나 프로세스의 실행시간은 짧아질 것이고, 실제로도 짧아짐을 확인 할 수 있었다. 하지만, 스레드와 프로세스의 속도차이는 그렇게 많이 차이나지는 않았다. 또한 개수를 늘릴수록 그만큼 전체적인 실행속도는 빨라지지 않았다. 예를들어 스레드나 프로세스의 양을 10배 늘리면 실행시간은 0.1배로 줄어들어야 하지만 그만큼 줄어들지 않는 것이었다. 멀티스레드나 멀티프로세스의 경우 데이터동기화 문제로 인해 blocking이 발생하면 원래의 실행속도보다 느려지는 경우가 생길 수도 있다. 하지만 나는 이러한 데이터공유문제가 없도록 스레드는 각자의 데이터를 자기가 가지고 있도록 분리해 놓았고, 프로세스는 상대의 데이터를 침범하지 못하게 구현해 놓았다. 그럼에도 불구하고 속도가 많이 빨라지지 않는 이유는 모든 스레드나 프로세스가 동시에 병렬처리되지는 않기 때문이라고 추측해 볼 수 있다. 그리고 그렇게 되지 않는 이유는 CPU코어의 물리적 개수의 제한이 있고, 이로인해 스레드나 프로세스가 아무리 많이 병렬로 처리한다 하더라도 실제로 실행되는 스레드나 프로세스의 개수는 정해져 있을것이기 때문이라고 생각한다.

4. 문제점 및 애로사항

이번 과제를 수행하면서 가장 애를 먹었던 부분은 Segmentation fault가 발생하는 부분이었다. 일반적인 문법 오류는 컴파일러가 잘못된 부분을 짚어주지만, Segmentation fault는 딱 저 한문장만 출력되고 프로그램이 꺼져버리기 때문에, 잘못된부분을 찾기가 힘들었다. 저 오류를 정말 많이 접하고 나서 Segementation fault는 주로 포인터를 잘못 사용할 때 발생된다는 것을 알게되었다. 이 오류는 자잘하게는 strcpy함수에서도 발생하였고, 특히 멀티스레딩 작업을 하는 과정에서 pthread_join함수의 마지막 인자인 스레드 리턴값의 주소를 지정하는 곳에서 정말 많이 겪었던 오류였다.

멀티프로세싱에서는 프로세스간의 통신을 shared memor나 pipe없이 하기 위해 동적할당을 써보려 하였다. 즉, 부모프로세스에 자식프로세스들이 접근할 수 있도록 메모리 영역을 할당받아 놓고, 자식프로세스가 그 메모리영역의 포인터를 가지고 수정하게 하도록 할 계획이었다. 하지만, 이 역시 자식프로세스가 종료되면 부모프로세스의 값은 변화가 없었다. 이와같은식으로 하면, fork()함수로 프로세스를 복제하면, 부모프로세스와 자식프로세스는 서로 다른 프로세스이므로 동적할당으로 선언한 포인터가 가리키는 곳은 주소는 같겠지만 서로 다른 heap영역이기 때문에, 독립적이라는것을 알게 되었다.

5. 개발환경

```
yj@ubuntu:~/Desktop/project2$ uname -a
Linux ubuntu 4.9.13-2013147513 #1 SMP Mon Mar 6 20:34:02 PST 2017 x86_64 x86_64
 x86_64 GNU/Linux
```