CSE 312 HOMEWORK 1 MAKEUP REPORT

Introduction

This homework is mostly influenced by the first of the three-example program. The only changes which are done is that, providing user-computer interactions such as context switching by keyboard interrupts (mouse click interrupts is not handled because it is a complex task and it is not shown in the videos), getting necessary input from the user to complete the task.

There are three processes except from the init process in the program. Contrary to previous homework, in this homework, all three processes must be added by the forking the init process. This is the provided with the following code. This code create 3 children processes and waits for the each of the children completes their task. Meanwhile there could be some time interrupts before the child process completes its task.

```
void initProcess()
{
    printf("Init: Started...\n");
    const uint32_t numOfProcess = 4;
    pid_t pids[numOfProcess];
    void (*process[])() = {binarySearch, linearSearch, collatz, infinite};

    printf("Init: Forking is started...\n");
    for (uint32_t i = 0 ; i < numOfProcess ; ++i)
        pids[i] = fork(process[i]);
    printf("Init: Forking is finished...\n");

    printf("Init: Waiting child to terminate...\n");
    for (uint32_t i = 0 ; i < numOfProcess ; ++i)
        waitpid(pids[i]);
    printf("Init: All child finished and collected. Init is exiting...\n");

    exit(exit_success);

}
```

After all children processes completes their task, init process terminates itself with the exit function.

```
void binarySearch()
{
  int * arr, arrSize = 0, key, low = 0, mid = 0, high = 0, index = -1;

  currentBuffer = 0;

  interractUser(&arr, &arrSize, &key, currentBuffer);

  high = arrSize - 1;

  while (low <= high){
    mid = (low + high) / 2;

    if (arr[mid] == key){
        index = mid;
        break;
    }
    else if (arr[mid] < key)
        low = mid + 1;
    else
        high = mid - 1;
  }

  int32_t printArr[] = {index};

  print("\nBinary search result: %d\n", 1, printArr);

  exit(exit_success);
}
```

```
void linearSearch()
{
    int * arr, arrSize = 0, key, low = 0, mid = 0, high = 0, index = -1;

    currentBuffer = 1;

    interractUser(&arr, &arrSize, &key, currentBuffer);

    high = arrSize - 1;

    for (uint32_t i = 0 ; i < arrSize ; ++i)
        if (arr[i] == key)
        {
            index = i;
            break;
        }

    int32_t printArr[] = {index};

    print("Linear search result: %d\n", 1, printArr);

    exit(exit_success);
}
```

```
void collatz()
{
    int32_t printArr[1];

    for (uint32_t i = 20 ; i < 25 ; ++i)
    {
        printArr[0] = i;
        print("Collatz %d: ", 1, printArr);
        uint32_t key = i;

        while (key != 1)
        {
            printArr[0] = key;
            print("%d, ", 1, printArr);

            if (key % 2 == 0)
                key /= 2;
            else
                key = key * 3 + 1;
        }

        printArr[0] = 1;
        print("%d\n", 1, printArr);
    }

    exit(exit_success);
}
```

The working principle of system calls is that, when there is a system calls in the program the required assembly codes runs with the necessary parameters such as entry point for fork system call, pid of the process for waitpid system call.

```
pid_t fork(void (*entry_point)(void))
{
    uint32_t ret = -1;

    __asm__ volatile("movl %0, %%eax; movl %1, %%ebx; int $0x80;" :
                : "r" (sys_fork), "r" (entry_point) : "eax", "ebx");
    __asm__ volatile("" : "=a"(ret));

    return ret;
}

void waitpid(uint32_t pid)
{
    Process::Status ret = Process::Running;

    while(ret != Process::Terminated)
    {
        __asm__ volatile("movl %0, %%eax; movl %1, %%ebx; int $0x80;" :
                    : "r" (sys_waitpid), "r" (pid) : "eax", "ebx");
        __asm__ volatile("" : "=a"(ret));
    }
}

void exit(Exit exit)
{
    __asm__ volatile("movl %0, %%eax; movl %1, %%ebx; int $0x80;" :
                : "r" (sys_exit), "r" (exit) : "eax", "ebx");

    while(true);
}
```

To handle system calls, there must be a system call handler. System call handler must take necessary action after the catch a system call. Necessary action depends on the system call, for example the process must be terminated in the exit system call, or the exact copy of the current process must be added process table after its entry point changed.

```
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp)
{
    CPUState* cpu = (CPUState*)esp;

    switch(cpu->eax)
    {
        case sys_fork:
            cpu->eax = interruptManager->tableManager->Fork(cpu);
            break;
        case sys_waitpid:
            cpu->eax = interruptManager->tableManager->GetStatus(cpu->ebx);
            break;
        case sys_exit:
            interruptManager->tableManager->TerminateProcess(interruptManager->tableManager->GetCurrent(), cpu->ebx);
            break;
        default:
            break;
    }

    return esp;
}
```

To provide user interactions for the system, timer interrupt is set. Timer interrupt is so fast that user cannot interact with the system. Period of the timer interrupt is increased with the following code.

```
if(interrupt == hardwareInterruptOffset && counter % 200 == 0)
{
    esp = (uint32_t)tableManager->Schedule((CPUState*)esp);
}
```

To give authority to user context switching by keyboard, a condition is added to keyboard handler. When the user pressed the ‘p’, counter is reset, so that timer interrupt occurs immediately.

```
void InterruptManager::ResetCounter(){
    counter = -1;
}
```

To get input from the user, interactUser function is implemented. This function gets input from the user until ‘s’ is pressed. Even if there is a timer interrupt in the process of getting input of the array elements from the user, the program will continue to get input from the user when the process runs again. This feature must be provided because there could be more than one timer interrupts during the process of getting input from the user in real life. It also has some codes to convert string to integer.

```
void interractUser(int ** arr, int * arrSize, int * key, int bufferNo){
  int length = 0;
  char toInt[80];

  buff_size[bufferNo] = 0;

  printf("\nEnter the array elements: ");

  while(buffer[bufferNo][buff_size[bufferNo] - 1] != 's'){

  }
  buffer[bufferNo][buff_size[bufferNo] - 1] = '\0';

  for(int i = 0; i < buff_size[bufferNo]; i++){
    if('0' < buffer[bufferNo][i] && buffer[bufferNo][i] < '9'){
      toInt[length] = buffer[bufferNo][i];
      length++;
    }
    else{
      toInt[length] = '\0';
      length = 0;
      array[bufferNo][*arrSize] = strToInt(toInt);
      (*arrSize)++;
    }
  }
  *arr = array[bufferNo];

  buff_size[bufferNo] = 0;

  printf("\nEnter the key: ");

  while(buffer[bufferNo][buff_size[bufferNo] - 1] != 's'){

  }
  buffer[bufferNo][buff_size[bufferNo] - 1] = '\0';

  *key = strToInt(buffer[bufferNo]);
}
```

Tests

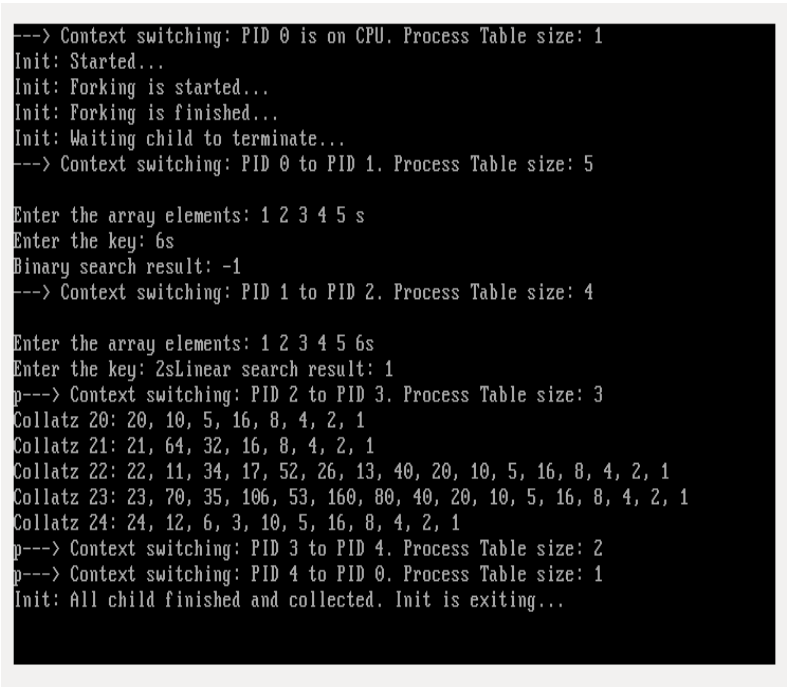
```
---> Context switching: PID 0 is on CPU. Process Table size: 1
Init: Started...
Init: Forking is started...
Init: Forking is finished...
Init: Waiting child to terminate...
ppppppppppppp---> Context switching: PID 0 to PID 1. Process Table size: 5

Enter the array elements: 1 2 3 4 5s
Enter the key: 2s
Binary search result: 1
---> Context switching: PID 1 to PID 2. Process Table size: 4

Enter the array elements: 1 2 3 4 5 s
Enter the key: 6sLinear search result: -1
---> Context switching: PID 2 to PID 3. Process Table size: 3
Collatz 20: 20, 10, 5, 16, 8, 4, 2, 1
Collatz 21: 21, 64, 32, 16, 8, 4, 2, 1
Collatz 22: 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1
Collatz 23: 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1
Collatz 24: 24, 12, 6, 3, 10, 5, 16, 8, 4, 2, 1
p---> Context switching: PID 3 to PID 4. Process Table size: 2
p---> Context switching: PID 4 to PID 0. Process Table size: 1
Init: All child finished and collected. Init is exiting...
```

In the first test, [1 2 3 4 5] array is sent to binary search with user input. 2 is sent as key. The program gives the correct result which is 1.

In the second test, same array is sent to linear search with user input. 6 is sent as key. The program gives the correct result which is -1.



In the first test, [1 2 3 4 5] array is sent to binary search with user input. 6 is sent as key. The program gives the correct result which is -1.

In the second test, same array is sent to linear search with user input. 2 is sent as key. The program gives the correct result which is 1.