MAJOR ASSIGNMENT-2

UNIX Systems Programming (CSE 3041)

Working with argument array, process creation and use of `fork-exec-wait` combination for the child to execute a new program

The assigment contains three parts. **In the first part**, students will be able to design their own argument array as like **`**argv`** parameter in `main` and be able to use the argument array in various to `exec`-family of functions. **In the second part** student will be able to create a chain of *n* processes by calling the **`fork()`** in a loop. **In the third part** students will be able to create a hybrid structure by combing the chain and fan of processes.

PART-1

This part aims to use **`execvp`** function to execute a different program that is different from that of the parent. The `exec` family of functions provides a facility for overlaying the process image of the calling process with a new image, and the **`execvp`** is one of them. The function prototype of **`execvp`** is given as;

SYNOPSIS:

```
intexecvp(constchar   *file,char  *constargv[]);
```

The arguments to the **`execvp`** will be set from command-line argumnets. The number of command line argument must be 2. Your code must supply an error checking to avoid more or less number(s) of command-line argument(s). An argumnet array named **`myargv`** will be constructed from the command-line string passed to the program. For example if you run the code **`$ ./a.out ''This is a test''`**, the argument array **`myargv`** will be displayed as follows;

```
myargv[0] ---- > This
myargv[1] ---- > is
myargv[2] ---- > a
myargv[3] ---  > test
```

Now, create a user-defined function named **`makeargv`** to create an argument array from a string passed on the command line. The following prototype shows a **`makeargv`** function that has a delimiter set parameter.

```
intmakeargv(constchar   *s,constchar   *delimiters,char  ***argvp)
   ;
```

The `const` qualifier means that the function does not modify the memory pointed to by the first two parameters.

Design your C code called **`ImageOverlay.c`** that creates a child process to execute a command string passed as the first command-line arguments. The program **`ImageOverlay.c`** will call the function **`makeargv`** to create the argumnet array and the created argument array must be used to set the parameter in the **`execvp`** call. The code must handle the error checking such as (1) if **`makeargv`** function returns -1, (2) fork() fails, (3) execvp fails etc. The code will be compiled as **`gcc ImageOverlay.c -o loadnew`** and run as **`./loadnew ''argument list''`**

Tesing

(a) **`./loadnew ``ls -l''`** To show the long listing of files in the current directory

(b) **`./loadnew ``ls -l *.c''`** To show all C files in the current directory

(c) **`./loadnew ``wc''`** to run **`wc`** command as like shell.

(d) **`./loadnew ``grep <patternname> <filename>''`** To search a pattern in the given file

(e) **`./loadnew ``cp <filename1> <filename2>''`** To copy a file

PART-2

Create a chain of *n* processes. It takes a single command-line argument that specifies the number of processes to create. Before exiting, each process outputs its i value, its process ID, its parent process ID and the process ID of its child. The parent does not execute wait. If the parent exits before the child, the child becomes an orphan. In this case, the child process is adopted by a special system process called `init`. As a result, some of the processes may indicate a different parent process ID. The Sample code for chain of *n* processes is given below;

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
intmain (intargc,char    *argv[]) {
   pid_t childpid = 0;
   inti, n;
   if(argc != 2){
     /* checkforvalidnumberofcommand-linearguments       */
     fprintf(stderr, "Usage: %s processes\n", argv[0]);
     return1;
   }
   n = atoi(argv[1]);
   for(i = 1; i < n; i++)
     if(childpid = fork())
       break;
   fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld
      \n",i, (long)getpid(), (long)getppid(), (long)childpid);
   return0;
}
```

(a) Run Program above sample code and observe the results for different numbers of processes.

(b) Fill in the actual process IDs of the processes in the chain diagram constructed from the sample code for a run with command-line argument value of 4.

(c) Experiment with different values for the command-line argument to find out the largest number of processes that the program can generate. Observe the fraction that are adopted by `init`.

(d) Place **`sleep(10);`** directly before the final **`fprintf`** statement in the sample code. What is the maximum number of processes generated in this case?

(e) Put a loop around the final fprintf in the given sample code. Have the loop execute **k** times. Put `sleep(10);` inside this loop after the **`fprintf`**. Pass **k** and **m** on the command line. Run the program for several values of **n**, **k** and **m**. Observe the results.

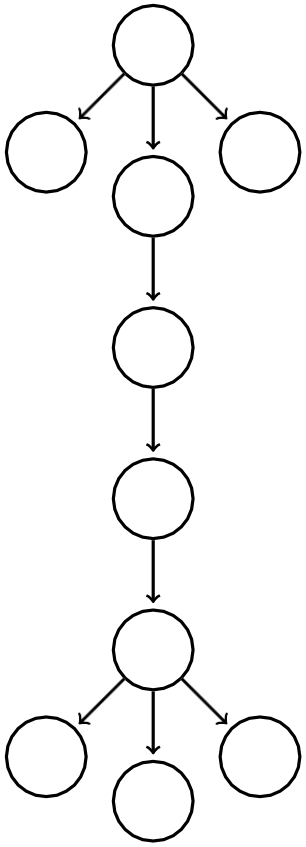
(g) Modify the sample code by putting a **`wait`** function call before the final **`fprintf`** statement. How does this affect the output of the program?

(h) Modify the sample code by replacing the final **`fprintf`** statement with **four `fprintf`** statements, one each for the four integers displayed. Only the last one should output a newline. What happens when you run this program? Can you tell which process generated each part of the output? Run the program several times and see if there is a difference in the output.

(i) Modify the sample code by replacing the final **`fprintf`** statement with a loop that reads **nchars** characters from standard input, one character at a time, and puts them in an array called **mybuf**. The values of **n** and **nchars** should be passed as command-line arguments. After the loop, put a **null** character in entry **nchars** of the array so that it contains a string. Output to standard error in a single **`fprintf`** the process ID followed by a colon followed by the string in **mybuf**. Run the program for several values of **n** and **nchars**. Observe the results. Press the Return key often and continue typing at the keyboard until all of the processes have exited.

PART-3

Create a hybrid of chain and fan of *n* processes. It takes a single command-line argument that specifies the number of processes to create. Design the C code to construct the below process tree structure. Use the wait/waitpid system call so that parent-child relationship will hold and no process will be adopted by the system special process.



Solution:

Part-1:

makeargv.c

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<errno.h>
#include<string.h>
int main(int argc,char *argv[])
{
     int i=0;
     char *myargv[10];
     if(argc!=2)
     {
     }
     else
     {
          fprintf(stderr, "Usage: %s string\n", argv[0]);
          return 1;
          execvp("a.out",&argv[1]);
          myargv[0]=strtok(argv[1]," ");
          while(myargv[i]!=NULL)
          {
               printf("myargv[%d] ----> %s\n",i,myargv[i]);
               i++;
               myargv[i]=strtok(NULL," ");
          }
     }
}
```

Output:

```
$ ./a.out "This is a test" myargv[0] ---- > This
myargv[1] ---- > is
myargv[2] ---- > a
myargv[3] ---- > test
```

ImageOverlay.c

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<string.h>
#include<unistd.h>
```

```
#include<sys/wait.h>

int makeargv(const char *s, const char *delimiters, char ***argvp);
int main(int argc, char *argv[])
{
char delim[] = " \t"; char **myargv;
int i,numtokens,retval=-1; pid_tpid=fork();
if(pid==0)
{
if (argc != 2)
{
fprintf(stderr, "Usage: %s string\n", argv[0]); return 1;
}
if ((numtokens = makeargv(argv[1], delim, &myargv)) == -1)
{
fprintf(stderr, "Failed to construct an argument array for %s\n",
argv[1]); return 1;
}
if(!strcmp(myargv[0],"ls") &&myargv[2]==NULL )
retval=execvp("ls",myargv);
if(!strcmp(myargv[0],"ls"))
{
if(!strcmp(myargv[1],"-l") && !strcmp(myargv[2],"*.c"))
retval=system("ls -l *.c");
else
}
retval=execvp("ls",myargv);
if(!strcmp(myargv[0],"wc")) retval=execvp("wc",myargv);
if(!strcmp(myargv[0],"grep")) retval=execvp("grep",myargv);
if(!strcmp(myargv[0],"cp")) retval=execvp("cp",myargv);
if(retval==-1)
printf("execvp error\n");
return 0;
}
else if(pid==-1)
{
}
else
{
}
return -1;
wait(NULL); return 2;
return 0;
}
int makeargv(const char *s, const char *delimiters, char ***argvp)
{
int error,i,numtokens; const char *snew; char *t;
if ((s == NULL) || (delimiters == NULL) || (argvp == NULL))
{
errno = EINVAL; return -1;
}
```

```
*argvp = NULL;
snew = s + strspn(s, delimiters); /* snew is real start of string
*/
if ((t = malloc(strlen(snew) + 1)) == NULL) return -1;
strcpy(t, snew); numtokens = 0;
if (strtok(t, delimiters) != NULL)/* count the number of tokens in
s */
for (numtokens = 1; strtok(NULL, delimiters) != NULL; numtokens++)
;/* create argument array for ptrs to the tokens */
if ((*argvp = malloc((numtokens + 1)*sizeof(char *))) == NULL)
{
error = errno; free(t);
errno = error; return -1;
}/* insert pointers to tokens into the argument array */
if (numtokens == 0)free(t); else
{
strcpy(t, snew);
**argvp = strtok(t, delimiters); for (i = 1; i<numtokens; i++)
*((*argvp) + i) = strtok(NULL, delimiters);
}
*((*argvp) + numtokens) = NULL;/* put in final NULL pointer */
return numtokens;
}
```

Output:

```
$ ./loadnew "ls -l" total 48
-rwxr-xr-x 1 student student 12912 Apr 2 00:05 a.out
-rw-rw-r-- 1 student student 1677 Feb 23 08:15 argarrv3.c
-rw-r--r-- 1 student student 137 Apr 2 00:06 argarrv3-output.txt
-rw-r--r-- 1 student student 2431 Feb 27 18:41 ImageOverlay.c
-rwxr-xr-x 1 student student 13088 Feb 23 09:39 loadnew
-rw-r--r-- 1 student student 517 Feb 23 08:13 MA3.c
$ ./loadnew "ls -l *.c"
-rw-r--r-- 1 student student 2431 Feb 27 18:41 ImageOverlay.c
-rw-r--r-- 1 student student 449 Apr 2 00:09 MA3.c
$ ./loadnew "wc" v
e
[ ctrl + c ]
$ ./loadnew "grep include MA3.c" #include<stdio.h>
#include<stdlib.h> #include<unistd.h> #include<errno.h>
#include<string.h>
$ ./loadnew "cp file1.txt file2.txt"
$ ls
a.out file1.txt ImageOverlay.cloadnew argarrv3-output.txt file2.txt
ImageOverlay-output.txt MA3.c
```

Part-2:

(a)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
pid_tchildpid = 0; int i, n;
if (argc != 2)
{
fprintf(stderr, "Usage: %s processes\n", argv[0]); return 1;
}
n = atoi(argv[1]); for (i = 1; i< n; i++)
if (childpid = fork())
break;
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld
\n",i, (long)getpid(), (long)getppid(), (long)childpid);
return 0;
}
```

Output of (a):

```
$ ./a.out 4
i:1 process ID:5742 parent ID:2017 child ID:5743 i:2 process
ID:5743 parent ID:5742 child ID:5744 i:3 process ID:5744 parent
ID:5743 child ID:5745 i:4 process ID:5745 parent ID:5744 child ID:0
$ ./a.out 5
i:1 process ID:6241 parent ID:2017 child ID:6242 i:2 process
ID:6242 parent ID:6241 child ID:6243 i:3 process ID:6243 parent
ID:6242 child ID:6244 i:4 process ID:6244 parent ID:6243 child
ID:6245 i:5 process ID:6245 parent ID:6244 child ID:0
$ ./a.out 1
i:1 process ID:6716 parent ID:2017 child ID:0
$ ./a.out 2
i:1 process ID:6713 parent ID:2017 child ID:6714 i:2 process
ID:6714 parent ID:6713 child ID:0
```

(b)

```
$ ./a.out 4
i:1 process ID:5742 parent ID:2017 child ID:5743 i:2 process
ID:5743 parent ID:5742 child ID:5744 i:3 process ID:5744 parent
ID:5743 child ID:5745 i:4 process ID:5745 parent ID:5744 child ID:0
```

(c)

```
i:31166 process ID:27717 parent ID:27716 child ID:0
Maximum number of processes generated are 31166 and is always
changing.
```

(d)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
pid_tchildpid = 0; int i, n, m, k;
if (argc != 4)
{
fprintf(stderr, "Usage: %s processes\n", argv[0]); return 1;
}
n = atoi(argv[1]);
k = atoi(argv[2]);
m = atoi(argv[3]); for (i = 1; i< n; i++)
if (childpid = fork())
break; for(int j=0; j< k ;j++)
{
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld
\n",i, (long)getpid(), (long)getppid(), (long)childpid);
sleep(m);
}
return 0;
}
```

Output of (d):

```
$ ./a.out 10
#(after 10 seconds)
i:1 process ID:17025 parent ID:2017 child ID:17026 i:2 process
ID:17026 parent ID:17025 child ID:17027 i:3 process ID:17027 parent
ID:17026 child ID:17028 i:4 process ID:17028 parent ID:17027 child
ID:17029 i:5 process ID:17029 parent ID:17028 child ID:17030 i:6
process ID:17030 parent ID:17029 child ID:17031 i:7 process
ID:17031 parent ID:17030 child ID:17032 i:8 process ID:17032 parent
ID:17031 child ID:17033 i:9 process ID:17033 parent ID:17032 child
ID:17034 i:10 process ID:17034 parent ID:17033 child ID:0
```

(e)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
pid_tchildpid = 0; int i, n;
if (argc != 2)
{
fprintf(stderr, "Usage: %s processes\n", argv[0]); return 1;
}
n = atoi(argv[1]); for (i = 1; i< n; i++)
if (childpid = fork())
break;
sleep(10);
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld
\n",i, (long)getpid(), (long)getppid(), (long)childpid);
return 0;
}
```

Output of (e):

```
$ ./a.out 4 4 5
i:1 process ID:2099 parent ID:1952 child ID:2100 i:2 process
ID:2100 parent ID:2099 child ID:2101 i:3 process ID:2101 parent
ID:2100 child ID:2102 i:4 process ID:2102 parent ID:2101 child ID:0
.....(after 5 seconds)
i:2 process ID:2100 parent ID:2099 child ID:2101 i:1 process
ID:2099 parent ID:1952 child ID:2100 i:3 process ID:2101 parent
ID:2100 child ID:2102 i:4 process ID:2102 parent ID:2101 child ID:0
.....(after 5 seconds)
i:1 process ID:2099 parent ID:1952 child ID:2100 i:4 process
ID:2102 parent ID:2101 child ID:0
i:2 process ID:2100 parent ID:2099 child ID:2101 i:3 process
ID:2101 parent ID:2100 child ID:2102
.....(after 5 seconds)
i:2 process ID:2100 parent ID:2099 child ID:2101 i:1 process
ID:2099 parent ID:1952 child ID:2100 i:4 process ID:2102 parent
ID:2101 child ID:0
i:3 process ID:2101 parent ID:2100 child ID:2102
.....(after 5 seconds)
```

(g)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main (int argc, char *argv[])
{
pid_tchildpid = 0; int i, n;
if (argc != 2)
{
fprintf(stderr, "Usage: %s processes\n", argv[0]); return 1;
}
n = atoi(argv[1]); for (i = 1; i< n; i++)
if (childpid = fork())
break;
wait(NULL);
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld \n",i,
(long)getpid(), (long)getppid(), (long)childpid);
return 0;
}
```

Output of (g):

```
$ ./a.out 5
i:5 process ID:2208 parent ID:2207 child ID:0
i:4 process ID:2207 parent ID:2206 child ID:2208 i:3 process
ID:2206 parent ID:2205 child ID:2207 i:2 process ID:2205 parent
ID:2204 child ID:2206 i:1 process ID:2204 parent ID:1952 child
ID:2205
Parent waits for child execution first.
```

(h)

```
#include <stdio.h> #include <unistd.h> #include <sys/wait.h>
int main (int argc, char *argv[])
{
pid_tchildpid = 0; int i, n;
if (argc != 2)
{
fprintf(stderr, "Usage: %s processes\n", argv[0]); return 1;
}
n = atoi(argv[1]); for (i = 1; i< n; i++)
{
if (childpid = fork())
break;
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child
ID:%ld\n",i, (long)getpid(), (long)getppid(), (long)childpid);
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child
ID:%ld\n",i, (long)getpid(), (long)getppid(), (long)childpid);
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child
```

```
ID:%ld\n",i, (long)getpid(), (long)getppid(), (long)childpid);
fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child
ID:%ld\n",i, (long)getpid(), (long)getppid(), (long)childpid);
}
return 0;
}
```

Output of (h):

```
$ ./a.out 4
i:1 process ID:2301 parent ID:2300 child ID:0 i:1 process ID:2301
parent ID:1237 child ID:0 i:1 process ID:2301 parent ID:1237 child
ID:0 i:1 process ID:2301 parent ID:1237 child ID:0
i:2 process ID:2302 parent ID:2301 child ID:0 i:2 process ID:2302
parent ID:2301 child ID:0 i:2 process ID:2302 parent ID:2301 child
ID:0 i:2 process ID:2302 parent ID:2301 child ID:0
i:3 process ID:2303 parent ID:2302 child ID:0 i:3 process ID:2303
parent ID:2302 child ID:0 i:3 process ID:2303 parent ID:2302 child
ID:0 i:3 process ID:2303 parent ID:2302 child ID:0
Real parent process will not be shown. (Parent ID:2300)
```

(i)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
pid_tchildpid = 0; int i, n, nchars; char mybuf[20];
if (argc != 3)
{
fprintf(stderr, "Usage: %s processes\n", argv[0]); return 1;
}
n = atoi(argv[1]); nchars = atoi(argv[2]);
for (i = 1; i< n; i++)
if (childpid = fork())
break;
for(i =0;i <nchars; i++)
read(0,mybuf,i); mybuf[nchars]='\0';
fprintf(stderr,"Process ID : %ld String in mybuf :%s
\n",(long)getpid(),mybuf);
return 0;
}
```

Output of (i):

```
Output-1
--------
$ ./a.out 4 4 hello iter
Process ID : 2570 String in mybuf :lo / Process ID : 2571 String in
mybuf :r m/ Process ID : 2572 String in mybuf :ame/ Process ID :
2573 String in mybuf : Su/
student@desktop:~/6th Semester/USP/Major Assignment-3/PART-2$ raj
Command 'raj' not found, did you mean:
command 'ra6' from deb ipv6toolkit command 'raw' from deb util-
linux command 'arj' from deb arj command 'rdaj' from deb horae
command 'rar' from deb rar command 'aj' from deb aspectj command
'ra' from deb argus-client
Try: sudo apt install <deb name> Output 2
```

Part-3:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
int i,n;
pid_tchildpid=0; if (argc != 2)
{
fprintf(stderr, "Usage: %s processes\n", argv[0]); return 1;
}
n = atoi(argv[1]); for(i=1;i<n;i++)
{
if( (childpid=fork())<= 0) break;
}
if(i!=(n/2))
{
printf("i:%d [process] PID : %ld from [parent] PID : %ld [child]
PID : %ld
\n",i,(long)getpid(),(long)getppid(),(long)childpid); wait(NULL);
}
if(i==(n/2))
{
for(i=n+1;i<2*n;i++)
{
if(childpid=fork())
break;
}
if(i!=2*n)
{
printf("i:%d [process] PID : %ld from [parent] PID : %ld [child]
PID :
%ld \n",i,(long)getpid(),(long)getppid(),(long)childpid);
wait(NULL);
}
if(i==2*n)
{
for(i=2*n;i<3*n-1;i++)
{
if( (childpid=fork())<= 0) break;
}
printf("i:%d [process] PID : %ld from [parent] PID : %ld [child]
PID :
%ld \n",i,(long)getpid(),(long)getppid(),(long)childpid);
wait(NULL);
}
}
```

```
return 0;
}
```

Output:

```
$ ./a.out 5
i:1 [process] PID : 29990 from [parent] PID : 29989 [child] PID : 0
i:5 [process] PID : 29989 from [parent] PID : 29599 [child] PID : 29994
i:3 [process] PID : 29992 from [parent] PID : 29989 [child] PID : 0
i:6 [process] PID : 29991 from [parent] PID : 29989 [child] PID : 29993
i:4 [process] PID : 29994 from [parent] PID : 29989 [child] PID : 0
i:7 [process] PID : 29993 from [parent] PID : 29991 [child] PID : 29995
i:8 [process] PID : 29995 from [parent] PID : 29993 [child] PID : 29996
i:9 [process] PID : 29996 from [parent] PID : 29995 [child] PID : 29997
i:10 [process] PID : 29998 from [parent] PID : 29997 [child] PID : 0
i:14 [process] PID : 29997 from [parent] PID : 29996 [child] PID : 30001
i:13 [process] PID : 30001 from [parent] PID : 29997 [child] PID : 0
i:11 [process] PID : 29999 from [parent] PID : 29997 [child] PID : 0 i:12
[process] PID : 30000 from [parent] PID : 29997 [child] PID : 0
```