**System Programming**

**P-1: Displaying data**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

char globBuf[65536]; /\* Uninitialized data segment \*/

int primes[] = {2, 3, 5, 7}; /\* Initialized data segment \*/

static int square(int x) {

int result;

result = x \* x;

return result;

}

static void doCalc(int val) {

printf("The square of %d is %d\n", val, square(val));

if (val < 1000) {

int t;

t = val \* val \* val;

printf("The cube of %d is %d\n", val, t);

}

}

int main(int argc, char \*argv[]) {

static int key = 9973; /\* Initialized data segment \*/

static char mbuf[10240000]; /\* Uninitialized data segment \*/

char \*p; /\* Allocated in frame for main() \*/

p = malloc(1024); /\* Points to memory in heap segment \*/

printf("\nMemory addresses of variables and segments:\n");

printf("---------------------------------------------------\n");

printf("Address of function main() (Text): %lu (%p)\n", (unsigned long)main, (void \*)main);

printf("Address of function doCalc() (Text): %lu (%p)\n", (unsigned long)doCalc, (void \*)doCalc);

printf("Address of globBuf (BSS): %lu (%p)\n", (unsigned long)globBuf, (void \*)globBuf);

printf("Address of primes (Initialized Data): %lu (%p)\n", (unsigned long)primes, (void \*)primes);

printf("Address of key (Initialized Data): %lu (%p)\n", (unsigned long)&key, (void \*)&key);

printf("Address of mbuf (BSS): %lu (%p)\n", (unsigned long)mbuf, (void \*)mbuf);

printf("Address of malloc'd memory (Heap): %lu (%p)\n", (unsigned long)p, (void \*)p);

printf("Address of local variable p (Stack): %lu (%p)\n", (unsigned long)&p, (void \*)&p);

doCalc(key);

free(p);

return 0;

}

**P-2: putenv,setenv,unsetenv**

#define \_GNU\_SOURCE /\* To get various declarations from <stdlib.h> \*/

#include <stdio.h>

#include <stdlib.h>

//run program by using following commands: gcc new2.c -o new2 && ./new2 SHELL=/bin/sh BYE=byebye

extern char \*\*environ;

int main(int argc, char \*argv[]) {

int j;

char \*\*ep;

clearenv();

for (j = 1; j < argc; j++) {

printf("Setting environment variable: %s\n", argv[j]); // Print argument before setting

if (putenv(argv[j]) != 0) {

printf("Error setting: %s\n", argv[j]); // Minimal error message

}

}

if (setenv("GREET", "Hello world", 0) == -1) {

printf("Error setting GREET\n");

}

unsetenv("BYE");

for (ep = environ; \*ep != NULL; ep++) {

puts(\*ep);

}

return 0;

}

**P-3: Dynamic memory allocation**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main() {

void \*ptr1, \*ptr2;

printf("Initial program break: %ld (%p)\n", (long)sbrk(0), (void \*)sbrk(0));

// Allocate 1024 bytes using sbrk

ptr2 = sbrk(1024);

if (ptr2 == (void \*)-1) {

printf("Error: sbrk failed\n");

return 1;

}

printf("After sbrk(1024): %ld (%p)\n", (long)sbrk(0), (void \*)sbrk(0));

// Free manually allocated memory (not needed with sbrk, but included for symmetry)

sbrk(-1024);

printf("After sbrk(-1024): %ld (%p)\n", (long)sbrk(0), (void \*)sbrk(0));

// Allocate 1024 bytes using malloc

ptr1 = malloc(1024);

if (ptr1 == NULL) {

printf("Error: malloc failed\n");

return 1;

}

printf("After malloc(1024): %ld (%p)\n", (long)sbrk(0), (void \*)sbrk(0));

// Free allocated memory

free(ptr1);

printf("After free(ptr1): %ld (%p)\n", (long)sbrk(0), (void \*)sbrk(0));

return 0;

}

**P-4: Fork**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <errno.h>

#include <string.h>

static int idata = 111; /\* Allocated in data segment \*/

int main(int argc, char \*argv[]) {

int istack = 222; /\* Allocated in stack segment \*/

pid\_t childPid;

switch (childPid = fork()) {

case -1:

printf("Error: fork failed: %s\n", strerror(errno));

exit(EXIT\_FAILURE);

case 0: /\* Child process \*/

idata \*= 3;

istack \*= 3;

break;

default: /\* Parent process \*/

sleep(3); /\* Give child a chance to execute \*/

break;

}

/\* Both parent and child come here \*/

printf("%s PID=%ld PARENT PID=%ld idata=%d istack=%d\n",

(childPid == 0) ? "(child) " : "(parent)", (long) getpid(), (long) getppid(), idata, istack);

return 0;

}

**P-5: Execv()**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

int main() {

pid\_t pid;

// Create a new child process

pid = fork();

if (pid < 0) {

// Fork failed

printf("Error: fork failed\n");

exit(EXIT\_FAILURE);

} else if (pid == 0) {

// Child process

printf("Child process: Replacing program with execv...\n");

// Path to the new program

char \*program = "./hello\_world"; // Path to the "hello\_world" program which is compiled already

char \*args[] = {program, NULL}; // Arguments array

// Replace the child process with the new program

if (execv(program, args) == -1) {

printf("Error: execv failed\n");

exit(EXIT\_FAILURE);

}

} else {

// Parent process

printf("Parent process: Waiting for child to complete...\n");

wait(NULL); // Wait for the child process to finish

printf("Parent process: Child finished execution.\n");

}

return 0;

}

**P-6: Creating N child**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

/\* Global variable for number of children \*/

const int numChildren = 104; // Set the desired number of child processes

int main() {

int j;

pid\_t childPid;

setbuf(stdout, NULL); // Make stdout unbuffered

for (j = 0; j < numChildren; j++) {

switch (childPid = fork()) {

case -1:

printf("Error: fork failed\n");

return EXIT\_FAILURE;

case 0:

printf("%d child\n", j);

exit(EXIT\_SUCCESS);

default:

printf("%d parent\n", j);

wait(NULL); // Wait for child to terminate

break;

}

}

return EXIT\_SUCCESS;

}

**P-7: Creating N child with predefined sleep time**

#include <sys/wait.h>

#include <time.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <errno.h>

#define NUM\_CHILDREN 3

const int sleepTimes[NUM\_CHILDREN] = {2, 4, 6}; // Sleep times for children

int main() {

int numDead = 0; /\* Number of children so far waited for \*/

pid\_t childPid;

int j;

setbuf(stdout, NULL); /\* Disable buffering of stdout \*/

for (j = 0; j < NUM\_CHILDREN; j++) { /\* Create children with predefined sleep times \*/

switch (fork()) {

case -1:

printf("Error: fork failed\n");

exit(EXIT\_FAILURE);

case 0: /\* Child sleeps for a while then exits \*/

printf("Child %d started with PID %ld, sleeping %d seconds\n", j + 1, (long) getpid(), sleepTimes[j]);

sleep(sleepTimes[j]);

exit(EXIT\_SUCCESS);

default: /\* Parent just continues around loop \*/

break;

}

}

for (;;) { /\* Parent waits for each child to exit \*/

childPid = wait(NULL);

if (childPid == -1) {

if (errno == ECHILD) {

printf("No more children - bye!\n");

exit(EXIT\_SUCCESS);

} else { /\* Some other (unexpected) error \*/

printf("Error: wait failed\n");

exit(EXIT\_FAILURE);

}

}

numDead++;

printf("wait() returned child PID %ld (numDead=%d)\n", (long) childPid, numDead);

}

}

**P-8 Pid**

#include <unistd.h>

#include <stdio.h>

int main(){

pid\_t pid = getpid(); // Get the process ID of the current process

pid\_t ppid = getppid(); // Get the parent process ID of the current process

printf("Parent Process ID: %d\n", ppid); // Print the parent process ID

printf("Process ID: %d\n", pid); // Print the process ID

}

**P-9 Command Line Arguments (argv)**

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{

int j;

for (j = 0; j < argc; j++)

printf("argv[%d] = %s\n", j, argv[j]);

exit(EXIT\_SUCCESS);

}

// gcc commandLineArg.c -o a.out && ./a.out hello world Sajid

OR

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*argv[])

{

char \*\*p;

for (p = argv; \*p != NULL; p++)

puts(\*p);

exit(EXIT\_SUCCESS);

}

**P-10 getenv()**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, char \*argv[])

{

if (getenv("SHELL") != NULL)

puts(getenv("SHELL"));

return EXIT\_SUCCESS;

}

**P-11 putenv()**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, char \*argv[])

{

if (!putenv("TEXT=hello"))

puts(getenv("TEXT"));

return EXIT\_SUCCESS;

}

‬ **P-12 unsetenv()**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

int main(int argc, char \*argv[])

{

if (!putenv("TEXT=hello"))

puts(getenv("TEXT"));

if(!unsetenv("TEXT"))

puts(getenv("TEXT"));

}

**Shell Script**

**P – Input Output**

#!/bin/bash

echo "What's your name?"

read entered\_name

echo -e "\nWelcome to bash tutorial $entered\_name"

**P – Reading File**

#!/bin/bash

while read line

do

echo $line

done < file.txt

**P – Variables**

#!/bin/bash

country=Bangladesh

same=$country

echo -e "$country\n"

echo -e "$same\n"

**P – If else**

#!/bin/bash

echo "Please enter a number: "

read num

if [ $num -gt 0 ]; then

echo "$num is positive"

elif [ $num -lt 0 ]; then

echo "$num is negative"

else

echo "$num is zero"

fi

**P – Display Folders**

#!/bin/bash

echo "Today is " `date`

echo -e "\nenter the path to directory"

read the\_path

echo -e "\n you path has the following files and folders: "

ls $the\_path

**P – For loop**

#!/bin/bash

for i in {1..5}

do

echo $i

done

**P – While Loop**

#!/bin/bash

i=1

while [ $i -le 10 ] ; do

echo $i

((i += 1 ))

done

**P – Command Line Arguments**

#!/bin/bash

echo "Hello, $1!"

# ./commandLineArgument.sh Sajid (Run this)

**P – Case Statement**

#!/bin/bash

echo "Enter the name of a fruit:"

read fruit

case $fruit in

"apple")

echo "This is a red fruit."

;;

"banana")

echo "This is a yellow fruit."

;;

"orange")

echo "This is an orange fruit."

;;

\*)

echo "Unknown fruit."

;;

esac

**P- Creating Folders**

echo "Please enter a number: "

read num

if [ "$num" -gt 0 ]; then

echo "$num is positive"

echo "Creating $num folders..."

for ((i = 1; i <= num; i++)); do

mkdir "Folder\_$i"

echo "Folder\_$i created"

done

elif [ "$num" -lt 0 ]; then

echo "$num is negative"

else

echo "$num is zero"

fi