1 SYSTEMS PROGRAMMING LABORATORY IV - Processes

Examples&Exercises:

- Complete the following codes if necessary, then compile and run the code.

- Analyze the code and output.

1. Using **fork** and **exec** Together,fork-exec.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
/* Spawn a child process running a new program.  PROGRAM is the name
   of the program to run; the path will be searched for this program.
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list.  Returns the process id of
   the spawned process.  */
int spawn (char* program, char** arg_list)
{
  pid_t child_pid;
  /* Duplicate this process.  */
  child_pid = fork ();
  if (child_pid != 0)
    /* This is the parent process.  */
    return child_pid;
  else {
    /* Now execute PROGRAM, searching for it in the path.  */
    execvp (program, arg_list);
    /* The execvp function returns only if an error occurs.  */
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
  }
}
int main ()
{
  /* The argument list to pass to the "ls" command.  */
  char* arg_list[] = {
    "ls",     /* argv[0], the name of the program.  */
    "-l",
    "/",
    NULL      /* The argument list must end with a NULL.  */
  };
  /* Spawn a child process running the "ls" command.  Ignore the
     returned child process id.  */
  spawn ("ls", arg_list);
  printf ("done with main program\n");
  return 0;
}
```

2. Using a Signal Handler; complete the following program sigusr1.c

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
{
  ++sigusr1_count;
}

int main ()
{
  struct sigaction sa;
  memset (&sa, 0, sizeof (sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);

  /* Do some lengthy stuff here.  */
  /* ...  */

  printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

3. The **wait** System Calls; complete the following program wait1.c

```
int main ()
{
  int child_status;
  /* The argument list to pass to the "ls" command. */
  char* arg_list[] = {
    "ls", /* argv[0], the name of the program. */
    "-l",
    "/",
    NULL /* The argument list must end with a NULL. */
  };
  /* Spawn a child process running the "ls" command. Ignore the
     returned child process ID. */
  spawn ("ls", arg_list);
  /* Wait for the child process to complete. */
  wait (&child_status);
  if (WIFEXITED (child_status))
    printf ("the child process exited normally, with exit code %d\n",
            WEXITSTATUS (child_status));
  else
    printf ("the child process exited abnormally\n");
  return 0;
}
```

4. Making a Zombie Process, zombie.c

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
  pid_t child_pid;

  /* Create a child process.  */
  child_pid = fork ();
  if (child_pid > 0) {
    /* This is the parent process.  Sleep for a minute.  */
    sleep (60);
  }
  else {
    /* This is the child process.  Exit immediately.  */
    exit (0);
  }
  return 0;
}
```

- Run it, and while it's still running, list the processes on the system by invoking the following command in another window:

  ```
  $ ps -e -o pid,ppid,stat,cmd
  ```

5. Cleaning Up Children Asynchronously; complete the following program cleaning.c

   ```
   #include <signal.h>
   #include <string.h>
   #include <sys/types.h>
   #include <sys/wait.h>

   sig_atomic_t child_exit_status;

   void clean_up_child_process (int signal_number)
   {
     /* Clean up the child process.  */
     int status;
     wait (&status);
     /* Store its exit status in a global variable.  */
     child_exit_status = status;
   }

   int main ()
   {
     /* Handle SIGCHLD by calling clean_up_child_process.  */
     struct sigaction sigchld_action;
     memset (&sigchld_action, 0, sizeof (sigchld_action));
     sigchld_action.sa_handler = &clean_up_child_process;
     sigaction (SIGCHLD, &sigchld_action, NULL);

     /* Now do things, including forking a child process.  */
     /* ...  */

     return 0;
   }
   ```

6. The **child_demo1.c** Program, child_demo1.c

   - Use the Makefile to compile and run the code.
   - The program **child_demo1.c** demonstrates the child library by invoking four child processes that do little other than announce their existence, sleep a random amount of time, and then die.

- Processes that die are automatically restarted.
- For no especially good reason, it installs signal handlers for several common signals and responds to those signals by doing a **longjmp()** and then killing the children.

child.c, child.h

- The library (**child.c**) contains functions to spawn a set number of child processes, to replace these processes when they die, and to send signals to these processes.
- It also includes a function that implements a safer and more flexible replacement for the **system()** and **popen()** standard library functions.
- The type **child_fp_t** defines a pointer to a function that will be executed in the child process. The two arguments are a pointer to the **child_info_t** structure that describes the child and an arbitrary (user defined) void pointer.
- The data structure **child_info_t** has information about a particular child process, including its process id (pid), its parent process id (ppid), its process number (zero through the number of child processes in a given group), and a pointer to the function to be executed.
- The data structure **child_group_info_t** contains information about a group of child processes. The member *nchildren* defines how many processes are listed in the child array.
- The data structure **child_groups_t** defines multiple groups; each group may be running a different function. Member *ngroups* indicates how many groups are defined in the array group of type **child_group_info_t**. This allows functions that wait for or manipulate dissimilar child processes.
- The function **child_create()** creates an individual child process. The third argument, **private_p**, is a user defined void pointer that is passed to the created child function.
- The function **child_group_create()** creates between ”min” and ”max” copies of a child process (currently the number created will equal ”min”).
- The function **child_groups_keepalive()** replaces children from one or more groups of children when they terminate for any reason.

- The function **child_group_signal()** sends a signal to all children in a single group.
- The function **child_groups_kill()** counts the number of children by sending them signal 0, sends each of them SIGTERM, and waits until they all die or a couple minutes have elapsed, at which time it aborts them using SIGKILL.
- The function **child_pipeve()** is a replacement for **system()** and **popen()**.

7. Rewrite the *Makefile* so that all the C-codes for today's lab can be compiled.