Lab 3: Task

Yu-Lun Huang

2011-07-11

1 Objective

- Be familiar with system calls: `fork()`, `wait()`, `waitpid()`, `nice()`, `exec()`, etc.
- Be familiar with POSIX programming: `pthread_create()`, `pthread_exit()`, etc.

2 Prerequisite

- Read man pages of the above system calls.

3 Process Control

- `fork()` creates a child process that differs from the parent process only in its PID and PPID, and in fact that resource utilizations are set to 0. The memory of child process is copied from the parent and a new process structure is assigned by the kernel. The environment, resource limits, umask, controlling terminal, current working directory, root directory, signal masks and other process resources are also duplicated from the parent in the forked child process, while file locks and pending signals are not inherited. The return value of the `fork()` function discriminates the two processes of execution. A zero is returned by the fork function in the child's process, while the parent process gets the PID (a non-zero integer) of its child.

```
1  /*
2   * process.c
3   */
4
5  #include <errno.h>      /* Errors */
6  #include <stdio.h>      /* Input/Output */
7  #include <stdlib.h>     /* General Utilities */
8  #include <sys/types.h>  /* Primitive System Data Types */
9  #include <sys/wait.h>   /* Wait for Process Termination */
10 #include <unistd.h>     /* Symbolic Constants */
11
12 pid_t childpid; /* variable to store the child's pid */
13
14 void childfunc(void)
15 {
16    int retval;     /* user-provided return code */
17
18    printf("CHILD: I am the child process!\n");
19    printf("CHILD: My PID: %d\n", getpid());
```

```
20      printf("CHILD: My parent's PID is: %d\n", getppid());
21      printf("CHILD: Sleeping for 1 second...\n");
22      sleep(1); /* sleep for 1 second */
23
24      printf("CHILD: Enter an exit value (0 to 255): ");
25      scanf(" %d", &retval);
26      printf("CHILD: Goodbye!\n");
27
28      exit(retval); /* child exits with user-provided return code */
29  }
30
31  void parentfunc(void)
32  {
33      int status;     /* child's exit status */
34
35      printf("PARENT: I am the parent process!\n");
36      printf("PARENT: My PID: %d\n", getpid());
37      printf("PARENT: My child's PID is %d\n", childpid);
38      printf("PARENT: I will now wait for my child to exit.\n");
39
40      /* wait for child to exit, and store its status */
41      wait(&status);
42      printf("PARENT: Child's exit code is: %d\n", WEXITSTATUS(status));
43      printf("PARENT: Goodbye!\n");
44
45      exit(0);  /* parent exits */
46  }
47
48  int main(int argc, char *argv[])
49  {
50      /* now create new process */
51      childpid = fork();
52
53      if (childpid >= 0) { /* fork succeeded */
54          if (childpid == 0) { /* fork() returns 0 to the child process */
55              childfunc();
56          } else { /* fork() returns new pid to the parent process */
57              parentfunc();
58          }
59      } else { /* fork returns -1 on failure */
60          perror("fork"); /* display error message */
61          exit(0);
62      }
63
64      return 0;
65  }
```

You can check the identifiers of the parent and child process by executing `ps` command in shell. Please refer to its man page for more details.

- `wait()` suspends execution of the current process until one of its children terminates.
- `waitpid()` suspends execution of the current process until a child specified by pid argument has changed state. By default, `waitpid()` waits only for terminated children, but this behaviour is modifiable via the options argument. Please refer to its man page.

```
1  /*
2   * waitpid.c -- shows how to get child's exit status
3   */
4
5  #include <errno.h>      /* Errors */
6  #include <stdio.h>      /* Input/Output */
7  #include <stdlib.h>     /* General Utilities */
8  #include <sys/types.h>  /* Primitive System Data Types */
9  #include <sys/wait.h>   /* Wait for Process Termination */
10 #include <time.h> /* Time functions */
11 #include <unistd.h>     /* Symbolic Constants */
12
13 pid_t childpid; /* variable to store the child's pid */
14
15 void childfunc(void)
16 {
17    int randtime;     /* random sleep time */
18    int exitstatus;       /* random exit status */
19
20    printf("CHILD: I am the child process!\n");
21    printf("CHILD: My PID: %d\n", getpid());
22
23    /* sleep */
24    srand(time(NULL));
25    randtime = rand() % 5;
26    printf("CHILD: Sleeping for %d second...\n", randtime);
27    sleep(randtime);
28
29    /* rand exit status */
30    exitstatus = rand() % 2;
31    printf("CHILD: Exit status is %d\n", exitstatus);
32
33    printf("CHILD: Goodbye!\n");
34    exit(exitstatus); /* child exits with user-provided return code */
35 }
36
37 void parentfunc(void)
38 {
39    int status;     /* child's exit status */
40    pid_t pid;
41
42    printf("PARENT: I am the parent process!\n");
43    printf("PARENT: My PID: %d\n", getpid());
44
45
46    printf("PARENT: I will now wait for my child to exit.\n");
```

```
47
48      /* wait for child to exit, and store its status */
49      do
50      {
51          pid = waitpid(childpid, &status, WNOHANG);
52          printf("PARENT: Waiting child exit ...\n");
53          sleep(1);
54      }while (pid != childpid);
55
56      if (WIFEXITED(status)) {
57          // child process exited normally.
58          printf("PARENT: Child's exit code is: %d\n",
59                  WEXITSTATUS(status));
60      }else{
61          // Child process exited thus exec failed.
62          // LOG failure of exec in child process.
63          printf("PARENT: Child process executed but exited failed.\n");
64      }
65
66      printf("PARENT: Goodbye!\n");
67
68      exit(0);  /* parent exits */
69  }
70
71  int main(int argc, char *argv[])
72  {
73      /* now create new process */
74      childpid = fork();
75
76      if (childpid >= 0) { /* fork succeeded */
77          if (childpid == 0) { /* fork() returns 0 to the child process */
78              childfunc();
79          } else { /* fork() returns new pid to the parent process */
80              parentfunc();
81          }
82      } else { /* fork returns -1 on failure */
83          perror("fork"); /* display error message */
84          exit(0);
85      }
86
87      return 0;
88  }
```

- `nice()` adds *inc* to the nice value for the calling process. (A higher nice value means a low priority.) Only the superuser may specify a negative increment, or priority increase. The range for nice values is described in getpriority(2).

```
#include <unistd.h>
...
int incr = -20;
int ret;

ret = nice(incr);
```

- The `exec()` family of functions initiates a new process image within a program. The initial argument for these functions is the pathname of a file which is to be executed.

```
/*
 * exec.c
 */

#include <unistd.h>
int main(int argc, char *argv[])
{
    execl("/bin/ls", "/bin/ls", "-r", "-t", "-l", (char *) 0);

    return 0;
}
```

4 Thread

A standardized programming interface was required to take full advantages provided by threads. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.

- `pthread_create()` creates a new thread, with attributes specified by *attr*, within a process. Upon successful completion, `pthread_create()` will store the ID of the created thread in the location specified by thread. For more information, please see its man page.
- `pthread_join()` suspends execution of the calling thread until the target thread terminates.
- `pthread_detach()` tells the underlying system that resources allocated to a particular thread can be reclaimed once it terminates. This function should be used when an exit status is not required by other threads.
- `pthread_exit()` terminates the calling thread.

The above APIs are included in the header file, 'pthread.h'.

```
/*
 * pthread.c -- shows how to create a thread
 */

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
```

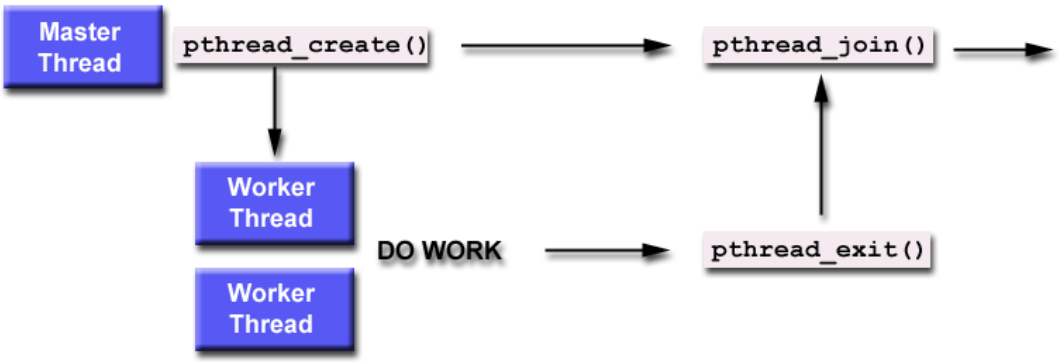

Figure 1: pthread_join()

```
8
9  #define NUM_THREADS  5
10
11 void *show(void *threadid)
12 {
13    long tid;
14
15    tid = (long)threadid;
16    printf("Hello! I am thread #%ld!\n", tid);
17
18    pthread_exit(NULL);
19 }
20
21 int main(int argc, char *argv[])
22 {
23    pthread_t threads[5];
24    int rc;
25    int t;
26
27    for(t=0;t<NUM_THREADS;t++){
28       printf("In main(): creating thread %d\n", t);
29       rc = pthread_create(&threads[t], NULL, show, (void *)t);
30       if (rc){
31          printf("ERROR; return code from pthread_create() is %d\n", rc);
32          exit(-1);
33       }
34    }
35
36    printf("Main: program completed. Exiting.\n");
37    pthread_exit(NULL);
38 }
```

```
1  /*
2   * join.c -- shows how to "wait" for thread completions
3   */
4
```

```
5  #include <math.h>
6  #include <pthread.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 #define NUM_THREADS  4
11
12 void *BusyWork(void *t)
13 {
14    int i;
15    long tid;
16    double result=0.0;
17
18    tid = (long)t;
19    printf("Thread %ld starting...\n",tid);
20
21    for (i = 0; i < 1000000; i++)
22    {
23       result += sin(i) * tan(i);
24    }
25
26    printf("Thread %ld done. Result = %e\n", tid, result);
27    pthread_exit((void*) t);
28 }
29
30 int main (int argc, char *argv[])
31 {
32    pthread_t thread[NUM_THREADS];
33    pthread_attr_t attr;
34    int rc;
35    long t;
36    void *status;
37
38    /* Initialize and set thread detached attribute */
39    pthread_attr_init(&attr);
40    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
41
42    for (t = 0; t < NUM_THREADS; t++) {
43       printf("Main: creating thread %ld\n", t);
44       rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
45       if (rc) {
46          printf("ERROR; return code from pthread_create() is %d\n", rc);
47          exit(-1);
48       }
49    }
50
51    /* Free attribute and wait for the other threads */
52    pthread_attr_destroy(&attr);
53    for (t = 0; t < NUM_THREADS; t++) {
54       rc = pthread_join(thread[t], &status);
55       if (rc) {
```

```
56              printf("ERROR; return code from pthread_join() is %d\n", rc);
57              exit(-1);
58          }
59          printf("Main: join with thread %ld (status: %ld)\n",t,(long)status);
60      }
61
62      printf("Main: program completed. Exiting.\n");
63      pthread_exit(NULL);
64  }
```

```
1   /*
2    * deteach.c -- shows how to detach a thread
3    */
4
5   #include <errno.h>
6   #include <pthread.h>
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <unistd.h>
10
11  void *threadfunc(void *parm)
12  {
13      printf("Inside secondary thread\n");
14      sleep(3);
15      printf("Exit secondary thread\n");
16      pthread_exit(NULL);
17  }
18
19  int main(int argc, char **argv)
20  {
21      pthread_t thread;
22      int rc = 0;
23
24      printf("Create a thread using attributes that allow detach\n");
25      rc = pthread_create(&thread, NULL, threadfunc, NULL);
26      if (rc) {
27          printf("ERROR; return code from pthread_create() is %d\n", rc);
28          exit(-1);
29      }
30
31      printf("Detach the thread after it terminates\n");
32      rc = pthread_detach(thread);
33      if (rc) {
34          printf("ERROR; return code from pthread_detach() is %d\n", rc);
35          exit(-1);
36      }
37
38      printf("Detach the thread again\n");
39      rc = pthread_detach(thread);
40      /* EINVAL: No thread could be found corresponding to that
41       * specified by the given thread ID.
```

```
42      */
43     if (rc != EINVAL) {
44         printf("Got an unexpected result! rc=%d\n", rc);
45         exit(1);
46     }
47     printf("Second detach fails as expected.\n");
48
49     /* sleep() is not a very robust way to wait for the thread */
50     sleep(6);
51     printf("Main() completed.\n");
52     return 0;
53 }
```

The link to `libpthread.a` library should be specified to the gcc compiler when compiling a program with pthread calls.

```
SHELL> arm-unknown-linux-gnu-gcc -o pthread pthread.c
-L /opt/arm-unknown-linux-gnu/arm-unknown-linux-gnu/lib/
-I /opt/arm-unknown-linux-gnu/arm-unknown-linux-gnu/include/
-lpthread
```

Lab 3 Assignments

- Figure out the differences between thread and process. Design your own program to demonstrate the differences to your TAs.
- Read the reference about controlling the LED. Write two programs using pthread and fork(), respectively. In each program:
  1. One thread/process reads the console input. The other thread/process controls the LEDs. The input is a number from 0 to 15. If the number is within 1 to 15, the corresponding LED's blinks at a 3-second interval. For example, if the number is 5, LED D9 and LED D11 will be blinking. If the number is 0, the program exits.
  2. The reading thread/process should keep reading inputs from console while the LED thread/process controls the LEDs. Do not block the inputs.
  3. The program has an argument which indicates the initial blinking LED's. The argument should not be 0.
  4. Compare the behaviours of the two implementations (thread- and process- implementations).