Languages for Informatics

10 – Multi-Tasking

Department of Computer Science
University of Pisa
Largo B. Pontecorvo 3
56127 Pisa

Topics

- Linux programming environment (2h)
- Introduction to C programming (12h)
- Basic system programming in Linux (10h)
    1. Signals and Error Handling
    2. Low-Level System Calls in C
    3. Multi-Tasking in C
    4. Multi-Threading in C
    5. Machine-To-Machine Communication in C

Overview

Creating New Processes?

Q1 Why we want that?

A1 to run **multiple tasks** (concurrently when cores are available) independently of one another.

- **Web server** receives request; creates additional instance of itself to handle the request; original instance continues listening for more requests. This will prevent client-side code on one page from bringing your whole browser down.
- **Daemon** runs in the background on a multi-tasking operating system. This means that it is detached from a terminal and runs continuously in a non-interactive mode such as `SMTP` daemon for sending mail, `inetd` daemon for network connection requests, `sshd`, ...

Q2 How to do that?

A2 A ”parent” process **forks** a ”child” process.

Create new Process

- **`fork()`** system call creates a new process, called **child** process
- Prototype

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

pid_t fork(void);
```

Return value

- On success, process ID (parent) or 0 (child)
- On error, -1 and sets **`errno`**.

`fork()` and Process State

- Immediately after `fork()`, **parent and child have identical but distinct process states**
  - Contents of registers on the stack
  - Contents of memory in the address space
  - File descriptor tables
  - pending signals
  - etc.
- Any process has a **unique non-negative ID**
  - Parent process and child processes have different process IDs
  - `pid_t getpid(void)` returns the process ID (PID) of the calling process.
  - `pid_t getppid(void)` returns the process ID (PID) of the parent of the calling process.

Example

```
#include<stdio.h>
#include<unistd.h>

int main (void) {
int pid;
printf("Start \n");
pid = fork();
if (pid == 0)
   printf ("child: I received %d; my pid is %d and that
    of my parent is %d\n", pid, getpid(), getppid());
else
   printf("parent: I received %d; my pid is %d and that
    of my parent is %d\n", pid, getpid(), getppid());

return 0;
}
```

Example

shell

```
Start
parent:  I received 11257; my pid is 11256 and
that of my parent is 11146
child:  I received 0; my pid is 11257 and that
of my parent is 11256
bash~$ echo $$
11146
```

Note

The current bash shell has PID 11146

Example - revisited

shell

```
bash~$ ./forkpid > out
bash~$ more out
Start
parent:  I received 11324; my pid is 11323 and
that of my parent is 11146
Start
child:  I received 0; my pid is 11324 and that
of my parent is 11323
```

Note

Why does `Start` appear twice ???

Example - revisited

Note

Why does `Start` appear twice ???

```
#include<stdio.h>
#include<unistd.h>

int main (void) {
int pid;
printf("Start \n");
/* stdout in the parent process contains "Start" */
pid = fork();
if (pid == 0)
   /* stdout in the parent process contains "Start" */
   /* stdout in the child process contains "Start" */
else
   /* stdout in the parent process contains "Start" */
return 0; /* With return statement all buffers are flushed and "
    Start" is saved twice */
}
```

Another Example

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

void char_at_a_time( const char * str ) {
  while( *str!= '\0' ) {
  putchar( *str++ ); // Write a char and increment the pointer
  fflush( stdout ); // Print now
  usleep(10000);   //100 ms
  }
}

int main() {
  if( fork() == 0 )   //child
    char_at_a_time( "*************" );
  else {              //parent
    char_at_a_time( "|||||||||||||" );
  }
return 0;
}
```

What is going wrong?
Example

Result

```
bash~$ gcc myfork.c -Wall -o myfork
./myfork
|*|*|*|**||**||**||**|*|*|
```

- Concurrency leads to unpredictable process execution order.
- **Synchronization** between the streams is needed.
- The parent process should **wait** for a child process to finish its computations at a particular execution point where it needs the output of the child process.
- Suppose, we wish to obtain

```
*************|||||||||||||
```

Synchronization among Processes (1)

- To **control execution order**, parent is **blocked** until its child that has **changed** state (suspended,stopped,continued)
- Prototype for the system call **`wait()`**:

```
#include <sys/types.h>
#include <sys/wait.h>
int wait(int* status_ptr)
```

- Suspends execution of the calling process until one of its children terminates.

Synchronization among Processes (2)

- Prototype for the system call **`wait()`**:

```
#include <sys/types.h>
#include <sys/wait.h>
int wait(int* status_ptr)
```

- If **status** is not `NULL`, **`wait()`** stores status information in the `int` to which it points.
  - `WIFEXITED(status)` becomes true if the child terminated normally by calling **`exit()`** or **`_exit()`**.
  - `WEXITSTATUS(status)` becomes the exit status of the child.
  - `WIFSIGNALED(status)` becomes true if the child process was terminated by a signal.
  - `WTERMSIG(status)` returns the number of the signal that caused the child process to terminate.

Synchronization among Processes (3)

- Prototype for the system call **`wait()`**:

```
#include <sys/types.h>
#include <sys/wait.h>
int wait(int* status_ptr)
```

Return value

- On success, returns pid of the terminated child process
- On failure, -1

Synchronization among Processes (4)

- To wait for a **particular** child
- Prototype for the system call **`waitpid()`**:

```
int waitpid(
  pid_t pid, /* pid or proces group id */
  int* status_ptr, /* status pointer or NULL */
  int options,
)
```

- **Options**: one ore more flags combinable by OR (—):
  - `WSTOPPED` - Wait for children that have been stopped by delivery of a signal.
  - `WCONTINUED` - Wait for (previously stopped) children that have been resumed by delivery of SIGCONT.
  - `WNOHANG` -when the status is not available, the fct. returns 0 rather than blocking.

Synchronization among Processes (5)

- Prototype for the system call **`waitpid()`**:

```
int waitpid(
  pid_t pid, /* pid or proces group id */
  int* status_ptr, /* status pointer or NULL */
  int options,
)
```

- **pid**. The value can be

< -1	wait for any child process whose process group ID is equal to the absolute value of `pid`.
-1	wait for any child process.
0	wait for any child process in the **process group** ID.
> 0	wait for the child with process ID equal `pid`

Note

`waitpid(-1, &status, 0) = wait(&status);.`

Synchronization among Processes (6)

- Prototype for the system call **`waitpid()`**:

```
int waitpid(
  pid_t pid, /* pid or proces group id */
  int* status_ptr, /* status pointer or NULL */
  int options,
)
```

Return value

- On success, returns pid of the terminated child process
- On failure, -1

What is going wrong?
Example (cont'd)

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

void char_at_a_time( const char * str ) {
  while( *str!= '\0' ) {
  putchar( *str++ ); // Write a char and increment the pointer
  fflush( stdout ); // Print now
  usleep(10000);  //100 ms
  }
}

int main() {
  if( fork() == 0 )  //child
    char_at_a_time( "*************" );
  else {  wait(NULL); //synchronize with child
    char_at_a_time( "|||||||||||||" );
  }
return 0;
}
```

Example
Status flags

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

int main() {
  pid_t pid;
  time_t t;
  int status;

  if ((pid = fork()) < 0) {
    perror("fork() error"); exit(EXIT_FAILURE); }
  else if (pid == 0) {      /* child */
    sleep(5);  //sleeps 5sec and exits
    exit(1);
  }
```

Example (cont'd)

Status flags

```
else do {                    /* parent */
    if ((pid = waitpid(pid, &status, WNOHANG)) == -1)
      perror("wait() error");  /*checls child without caller
    being suspended */
     else if (pid == 0) {   /* child prints to stdout 1/sec */
      time(&t);
      printf("child is still running at %s", ctime(&t));
      sleep(1);
    }
    else {
      else {        /* meanwhile, parent observes status */
        printf("child exited with status of %d\n", WEXITSTATUS(
    status));
      else puts("child did not exit successfully");
    }
  } while (pid == 0);  /* as long as child exist */

  return 0;
}
```

Example (cont'd)
Status flags

Shell

```
child is still running at Thu Nov 19 12:34:03 2020
child is still running at Thu Nov 19 12:34:04 2020
child is still running at Thu Nov 19 12:34:05 2020
child is still running at Thu Nov 19 12:34:06 2020
child is still running at Thu Nov 19 12:34:07 2020
child exited with status of 1
```

Zombie Processes

- Suppose a process that forks a child process.
- The **Process Table** in the kernel memory records accounting and scheduling information of the child.
- The child process completes its work and exits.
- Note: Info on the child keeps in the Process Table until the parent process reads its `exit` code (”reaping” the child).
- In the **period between exit of child and reaping**, the child process is called a **Zombie**

Example
Zombie

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main () {
  pid_t pid;
  pid = fork ();
  if (pid > 0) {
  printf("parent %d: Gonna sleep\n",getpid());
  sleep (60);     /* Parent goes to sleep */
  }
  else {   /* Child exits in the meantime */
  printf("child %d: Exiting\n",getpid());
  exit (0);
  }
return 0;
}
```

Example (cont'd)
Zombie

Shell 1

```
bash~$ gcc zombie.c -Wall -o zombie
bash~$ ./zombie parent 17407: Gonna sleep
child 17408: Exiting
```

Shell 2

```
bash~$ ps axo stat,ppid,pid,comm | grep -w
defunct
Z+    17407    17408    zombie    <defunct>
```

Reaping Zombies (1)

Note

Zombies take no memory or CPU. However, the process table is a finite resource, and excessive zombies can fill it so that no more processes can launch.

Possible Solutions

1. Use **`wait(NULL)`** system call in the parent process.
2. Ignore `SIGCHLD` signal by the child process.
3. Implement a **signal handler**.

Reaping Zombies (2)

1. Use **`wait(NULL)`** system call in the parent process.
   - Parent wait for the child to complete and it will reap the exit status of the child.
   - The execution order has been serialized.
2. Ignore `SIGCHLD` signal by the kernel due to child process.
   - When a child is terminated, a corresponding `SIGCHLD` signal is delivered to the parent.
   - By ignoring the `SIGCHLD` signal, the child process entry is deleted from the process table.
   - The parent process continues working in parallel.
3. Use a **signal handler**.
   - The signal handler calls **`wait(NULL)`** system call within it.
   - On receipt of `SIGCHLD`, the corresponding handler is activated,notifying the parent almost immediately, and the child entry in the process table is cleared.

Example (Demo)
Prevent Zombie by signal handler

```
#include <sys/wait.h>
#include <unistd.h>     //sleep
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>

void handle_sigchld(int sig) {
  int saved_errno = errno;      //save errno
  pid_t   pid;
  int     stat;
  pid = wait(&stat);           //wait for child and
    cleanup process table
  printf("child %d: terminated \n", pid);
  errno = saved_errno;         //restore errno
}
```

Example (Demo)

Prevent Zombie by signal handler

```
int main () {
  struct sigaction sAction;  //register the handler
  sAction.sa_handler = &handle_sigchld;
  sigemptyset(&sAction.sa_mask);
  sAction.sa_flags = SA_RESTART | SA_NOCLDSTOP;
  if (sigaction(SIGCHLD, &sAction, 0) == -1) {
    perror(0); exit(1);
  }
  int iRet; pid_t pid;  pid = fork ();
  if (pid > 0) {
    iRet = sigaction(SIGCHLD, &sAction, NULL);    //catch SIGCHLD
      */
    if (iRet != 0) {    /* Something went wrong */
      exit(EXIT_FAILURE);
    }  /* parent does sthg. */
  }
  else { exit (0);  // child leaves */
  }
return 0;
}
```

Orphans

- A process becomes **orphan** when its parent process does no more exist
    - either finished or terminated without waiting for its child process to terminate.
- Orphaned children are immediately ”adopted” by the first process `init`.
    - Hence, no zombies.
- `Init` reaps the orphan by `wait()`ing on the child when it receives `SIGCHLD`.

Example

```
int main() {
  int pid = fork();
  if (pid > 0)
    {
    printf("Parent: %d \n",getpid());
    sleep(1);
    printf("Parent is leaving\n");
    exit(0);
    }
  else if (pid == 0)
    {
    printf("\nChild: %d \n", getpid());
    printf("Parent: %d\n\n",getppid());

    sleep(10);  //parent is surely gone

    printf("\nChild: %d \n",getpid());
    printf("Parent: %d\n",getppid());
    }
  return 0;
}
```

Example (cont'd)

Result

```
bash~$ gcc orphan.c -Wall -o orphan
bash~$ ./orphan
Parent:  21740

Child:  21741
Parent:  21740

Parent is leaving
bash~$
Child:  21741
Parent:  1
```

Run a Separate Program (1)

- We have seen that `fork`ed process simply do different work concurrently.
- Approach can be used to launch **completely separate program** while
  - **maintaining control** over the program
  - being able to send data to the program through `stdin`
  - capturing the output of the program through its `stdout`.
- The **`exec()`** family of functions replace the current process image with a new one coming from loading a new program
  - all code (text) and data in the current process is replaced with the executable of the new program
  - all open file descriptors remains open.

Run a Separate Program (2)

- Prototypes

```
#include <unistd.h>
int execl(const char *path, const char *arg, (char*) NULL );
int execlp(const char *file, const char *arg, (char*) NULL);
int execle(const char *path, const char *arg, (char *) NULL, char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

- All the functions take the executable path as first argument

l functions accept variable amount of null-terminated char *

v functions accept the executable path and an array of null-terminated char *

  - Both forward arguments to the executable (arg0 must be set to executable name)

p functions access PATH environment variable to find exec.

e functions accept also an array of null-terminated char * storing environment variables

Example - Demo

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>
int spawn(const char * program, char ** arg_list) {
  pid_t child_pid = fork();
  if (child_pid != 0)
     return child_pid; /* This is the parent process. */
  else {
     execvp (program, arg_list); /* exec. prog. in child proc.*/
     perror ("spawn"); exit(1); }
}
int main() {  //open xterm, print sthg., sleep, exit
  char * arg_list[] = { "/usr/bin/xterm", "-fn", "10x20","-e", "
    cowsay Big brother is watching you;sleep 3;exit", NULL };
  spawn("/usr/bin/xterm", arg_list);
  wait(NULL);    // wait until child has finished
  printf ("Thanks for telling me. Bye........\n");
  return 0;      // parent exits
}
```

The **`system`** Function

- Common combination of operations
  - **`fork()`** to create a new child process
  - **`execvp()`** to execute new program in child process
  - **`wait()`** in the parent process for the child to complete
- Single call that combines all three **`int system(const char *cmd);`**
- Example - revisited

```
#include <stdio.h>
#include <stdlib.h> //system
int main() {
  system("/usr/bin/xterm -fn 10x20 -e 'cowsay Big
    brother is watching you;sleep 3;exit'");
  printf ("Thanks for telling me. Bye........\n");
  return 0;
}
```

Daemons

- A **daemon**[1] is a program that continuously runs as a background process rather than being under the direct control of an interactive user.
- Does not belong to a `TTY`.
- For handling periodic service requests that a computer system expects to receive.
- Traditionally, the process names of a daemon **end with the letter `d`** (crond,inetd,sshd,...).

[1] from *demon*, a spiritual being that constantly works in the background.

Prolog: Creating a new UNIX session (1)

- **`setsid`** creates a session and sets the process group ID
- Prototype

```
#include <unistd.h>
pid_t setsid(void);
```

Return Value

- On success, the session ID of the calling process.
- On error, -1 is returned, and `errno` is set.

Prolog: Creating a new UNIX session (2)

- Prototype

```
#include <unistd.h>
pid_t setsid(void);
```

- Description:
  - The calling process is the leader of the new session, the process group leader of the new process group, and has no controlling terminal.
  - The process group ID and session ID of the calling process are set to the PID of the calling process.
  - The calling process will be the only process in this new process group and in this new session.

Note

The calling process is now detached from its TTY. It will not be killed by closing the terminal without the comand `exit`.

Spawning Daemons

1. `fork` the parent process and let it terminate. The child process now runs in the background.
2. `setsid` - Create a new session.
3. **catch** `signals` - Ignore and/or handle signals.
4. `fork` again and let the parent process terminate. The child process is an orphan and the OS cleans up after termination of the grandchild (as the parent process is already dead), to prevent resource consumption.
5. `chdir` - Change the working directory of the daemon.
6. `umask` - Change the file mode mask according to the needs of the daemon.
7. `close` - Close all open file descriptors that may be inherited from the parent process.

Daemon skeleton

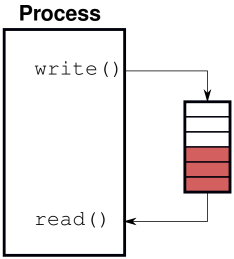
```
static void mydaemon() {
    pid_t pid;
    pid = fork(); //fork parent
    if (pid < 0)
        exit(EXIT_FAILURE);
    elseif (pid > 0)
        exit(EXIT_SUCCESS);   //parent terminates
    umask(0);     //no privileges
    if (setsid() < 0)    /* child becomes session leader */
        exit(EXIT_FAILURE);
    /* Catch, ignore or handle signals */
    signal(SIGCHLD, SIG_IGN);
    signal(SIGHUP, SIG_IGN);
    pid = fork();  //fork again, to prevent orphans
    if (pid < 0)
        exit(EXIT_FAILURE);
    else if (pid > 0)  exit(EXIT_SUCCESS); /* parent exits */
    chdir("/");
    for (int x = sysconf(_SC_OPEN_MAX); x>=0; x--) { close (x);}
}
```

Daemon skeleton
DEMO

```
int main()
{
    skeleton_daemon();

    /* Open the log file */
    openlog ("firstdaemon", LOG_PID, LOG_DAEMON);

    char command[] = "/usr/bin/xterm -fn 10x20 -e 'echo hi! I
    am your daemon; sleep 3; exit'";
    while (1) {
        syslog (LOG_NOTICE, "My first daemon.");
        system(command);
        sleep(600);       //sleep 10 minutes
    }

    syslog (LOG_NOTICE, "First daemon terminated.");
    closelog();
    return EXIT_SUCCESS;
}
```

The `pipe()` System Call (1)

- **Unnamed pipes**
- A producer writes and a consumer reads in a FIFO fashion



- In Linux, the OS guarantees that only one process at a time can access the pipe.
- Data written by the producer (**`write()`**) are stored into a buffer by the OS (Ubuntu 64-bit: 16 pages, each 4096 Bytes) until a consumer (**`read()`**) reads it.

The `pipe()` System Call (2)

- **Prototype**

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

- **Parameters**

pipefd[0]	: the FD for the read end of pipe.
pipefd[1]	: the FD for the write end of pipe.

- **Return value**

0	: on success.
-1	: on error; `errno` is set appropriately, `pipefd` is left unchanged.

pipe() Example

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define MSGSIZE 16
char* msg = "hello, world!";

int main() {
    char inbuf[MSGSIZE];
    int p[2];
    if (pipe(p) < 0)
        perror("pipe");

    write(p[1], msg, MSGSIZE);  /* write pipe */
    printf("Message sent: %s\n", msg);

    read(p[0], inbuf, MSGSIZE);  /* read pipe */
    printf("Message received: %s\n", inbuf);
    return 0;
}
```

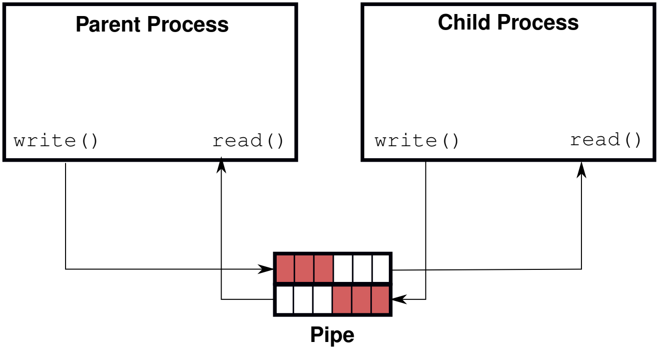
`pipe()` Example (cont'd)

Result

```
Message sent:  hello, world!
Message received:  hello, world!
```

Pipe based Message Passing

- When we use **`fork`** in any process, the **FDs remain open** across child process and also parent process.
- When we call **`fork`** **after** creating a pipe, the **parent and child can communicate via the pipe**.



Two-way Communication using Pipes

Algorithm

1. **Create pipe1** for the parent process to write and the child process to read.
2. **Create pipe2** for the child process to write and the parent process to read.
3. **Close the unwanted ends** of the pipe from the parent and child side.
4. **Parent process to write** a message and **child process to read** and display on the screen.
5. **Child process to write** a message and **parent process to read** and display on the screen.

Two-way Communication using Pipes
Example

```
#include<stdio.h>
#include<unistd.h>

int main() {
   int pipefds1[2], pipefds2[2];
   int stat1, stat2;
   int pid;
   char pipe1msg[] = "Hi baby";
   char pipe2msg[] = "Hi dad";
   char readmessage[20];
   stat1 = pipe(pipefds1);

   if (stat1 == -1) {
      perror("Pipe 1");
      }
   stat2 = pipe(pipefds2);

   if (stat2 == -1) {
      perror("Pipe 2");
   }
```

Two-way Communication using Pipes
Example (cont'd)

```
  if (pid != 0) // Parent process
     {
       close(pipefds1[0]); // Close the unwanted pipe1 read side
       close(pipefds2[1]); // Close the unwanted pipe2 write side
       printf("In Parent: Writing to pipe 1 – %s\n", pipe1msg);
       write(pipefds1[1], pipe1msg, sizeof(pipe1msg));
       read(pipefds2[0], readmessage, sizeof(readmessage));
       printf("In Parent: Reading from pipe 2 – %s\n",
     readmessage);
    } else { //child process
       close(pipefds1[1]); // Close the unwanted pipe1 write side
       close(pipefds2[0]); // Close the unwanted pipe2 read side
       read(pipefds1[0], readmessage, sizeof(readmessage));
       printf("In Child: Reading from pipe 1 – %s\n", readmessage
     );
       printf("In Child: Writing to pipe 2 – %s\n", pipe2msg);
       write(pipefds2[1], pipe2msg, sizeof(pipe2msg));
    }
    return 0;
}
```

Two-way Communication using Pipes
Example (cont'd)

Result

```
bash~$ gcc pipeandfork.c -Wall -o pipeandfork
bash~$ ./pipeandfork
In Parent:  Writing to pipe 1 - Hi baby
In Child:  Reading from pipe 1 - Hi baby
In Child:  Writing to pipe 2 - Hi dad
In Parent:  Reading from pipe 2 - Hi dad
```

Quiz

```
#include <stdio.h>
#include <unistd.h>
int main(int argc,char *argv[]){
    printf("A\n");
    fork();
    printf("B\n");
    fork();
    printf("C\n");
    return 0;
}
```

How many A, B and C will be printed?

1. A: 1 time, B: 2 times, C: 2 times
2. A: 1 time, B: 2 times, C: 4 times
3. A: 1 time, B: 1 times, C: 2 times
4. A: 1 time, B: 2 times, C: 3 times