Chapter 10

Process Environment

10.1 Introduction

Program
A program is a sequence of instructions. After compilation, a program is transformed into an executable code. Neither of these actually do anything,as they are only passive entities.

Process
A process is a dynamic active entity of a program along with the resources required to run. These include user and system stacks, memory, file handles, etc. Each process generally acts in a separate address space.

Concurrent processes
A process is usually sequential and consists of a sequence of actions that take place one at a time. When a set of processes are run on a single CPU, using timeslicing to multitask them, this is referred to as concurrent sequential processing or pseudo-parallel processing. If there is more than one CPU available, then processes may be run in parallel.

Process Context

In UNIX, a process executes both in the user and the kernel space. It has three contexts: a user-level context, register context, and system-level context.

1. User-level context: text, data and stack, represented by the translation table.
2. Register context: program counter (PC), processor status word (PSW), stack pointer, and general registers.
3. System-level context: has two parts, a static and a dynamic part. Static parts include the process control block (PCB), U-area and the translation tables. The dynamic parts include the kernel stack and the ”context” layers (basically exception frames).

Process state
A UNIX process can be in one of following states:

- Running in user mode
- Running in kernel mode
- Preempted
- Blocked, but in memory
- Blocked, not in memory

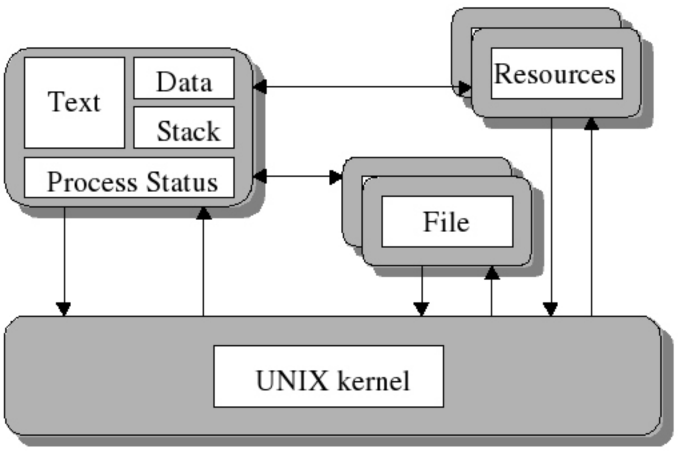

Figure 10.1: User-level context of a process

- Ready to run, not in memory
- Ready to run, but in memory
- Created
- Zombie

Memory layout of a C/C++ program

- Text segment
- Initialized data segment
- Uninitialized data segment (bss - block started by symbol)
- Stack (for function calls)
- Heap (for dynamic memory allocations)
- Command-line arguments (argc/argv[ ]//envp[ ])
- Environment variables (extern char **environ)

Command-line arguments

They are always represented as an array of strings. This array is called "argv" (for "argument values") and there is also an integer called "argc" (for "argument count") which is the number of command-line arguments. That's why the main function looks like

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

Here, *argc* is declared to be an *int*, whereas *argv* is an array of char *

System Environment Variables

There are a number of well-known environment variables you can expect to find defined on startup of a program from the Unix shell.

USER : Login name of the account under which this session is logged in (BSD convention)

LOGNAME : Login name of the account under which this session is logged in (System V convention)

HOME : Home directory of the user running this session

```
#include <stdio.h>
int
main(int argc, char* argv[])
{
  int i;
  printf("This program is named %s.\n", argv[0]);
  for (i = 1; i < argc; ++i)
   printf("the argument #%d is %s\n", i, argv[i]);
}
```

SHELL : The name of the user's command shell (often used by shellout commands)
PATH : The list of directories that the shell searches when looking for executable commands to match a name
TERM : Name of the terminal type of the session console or terminal emulator window

```
#include <stdio.h>
#include <unistd.h>
extern char **environ;
int main(int argc, char *argv[])
{
  char **p = environ;
  while (*p != NULL)
  {
    printf("%s (%p)\n", *p, *p);
    *p++;
  }
  return 0;
}
```

10.2 Summary

Exercises

Chapter 11

Process Control

11.1 Introduction

Process Control operations
The set of operations on processes includes

- create a process (fork(), vfork())
- destroy a process (exit(), _exit(), abort(), kill())
- run a process (exec family calls)
- suspend/synchronize processes (wait(), waitpid())
- get process information (getpid(), getppid(), getgid(), getegid(), getuid(), geteuid())
- set process information (setpid(), setppid(), setgid(), setegid(), setuid(), seteuid())

Process creation
A new process will be created using an existing one. There are several possible organisations of this creation such as Synchronous and Asynchronous In Sychronous, if a process is created from another, then the new must complete execution before the old one can resume. In asynchronous, if the new process is created from another, then the two processes may be run in pseudo-parallel.

Parent and child process
When a new process is created, it may use the old one denoted as parent or new process denoted as child. The possible relationship between these process are as follows: The child gets a copy of the parent, it inherits the parent's data, heap and stack. The parent and the child share the text segment, if it is read-only. COW (copy-on-write) is available in most current implementations, only the page that gets modified is copied, typically in a virtual memory system.

The *fork()* system call is used to create a child process after calling this system call each process can execute a different sections of the code at the same time or One process can execute a different program.

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

pid_t fork(void);

                         Returns: 0 in child process, child process ID in parent, -1 on error
```

fork() may get failed if it exceeds user limit or total system limit and it corresponding error values are *EAGAIN* and *ENOMEN* respectively. The first execution of process is implementation-dependent. But, all file descriptors that are open in the parent are duplicated in the child. They also share the same file offset (Files opened after fork are not shared).

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

pid_t vfork(void);

                    Returns: 0 in child process, child process ID in parent, -1 on error
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int
main(int argc, char* argv[])
{
  pid_t child_pid;
  printf("Current Process ID = %d \n", getpid());
  printf("Parent Process ID = %d \n", getppid());
  switch (child_pid = fork())
  {
   case (pid_t) -1 :
                perror("fork fails");
                break;
   case (pid_t) 0 :
                printf("Child created : PID: %d", getpid());
                printf("Parent PID: %d", getppid());
                exit(0);
   default:
                printf("Parent after fork : PID: %d", getpid());
                printf("Child : PID: %d", child_pid);
 }
  return 0;
}
The sample outputs of this program, when executed may be
Current Process ID = 345
Parent Process ID = 111
Child created : PID: 500, Parent PID: 345
Parent after fork : PID: 345, Child PID : 500
```

Similar to *fork()*; *vfork()* is used to create a process, but it guarantees that the child runs first until it calls *exec()/exit()* system call. The deadlock is possible if the child needs information from the parent.

Process termination
Normal termination

- Return from main()
- Calling exit()
- Calling _exit()

Abnormal termination

- Calling abort()

- Terminated by a signal

```
#include <stdlib.h>

void exit(int status);

#include <unidstd.h>

void _exit(int status);
```

The *exit()* system call performs a standard I/O cleanup, executes all registered exit handlers, flushes all C/C++ output buffers, closes all open streams and terminates the calling process.

The *exit()* function first calls all functions registered by *atexit()*, in the reverse order of their registration. Each function is called as many times as it was registered.

The *atexit()* a register function to be called at normal program termination. The functions so registered are called in the reverse order of their registration; no arguments are passed.

```
#include <stdlib.h>

int atexit(void (*function)(void));

Returns: 0 if Ok, otherwise it returns a nonzero value
```

The _exit() system call terminates the calling process without performing some cleanup.

exit() Vs _exit()

There are a few differences between *exit()* and *_exit()* that become significant when fork(), and especially *vfork()*, is used. The basic difference between exit() and _exit() is that the former performs clean-up related to user-mode constructs in the library, and calls user-supplied cleanup functions, whereas the latter performs only the kernel cleanup for the process.

In the child branch of a *fork()*, it is normally incorrect to use *exit()*, because that can lead to stdio buffers being flushed twice, and temporary files being unexpectedly removed.

In the child branch of a *vfork()*, the use of *exit()* is even more dangerous, since it will affect the state of the parent process.

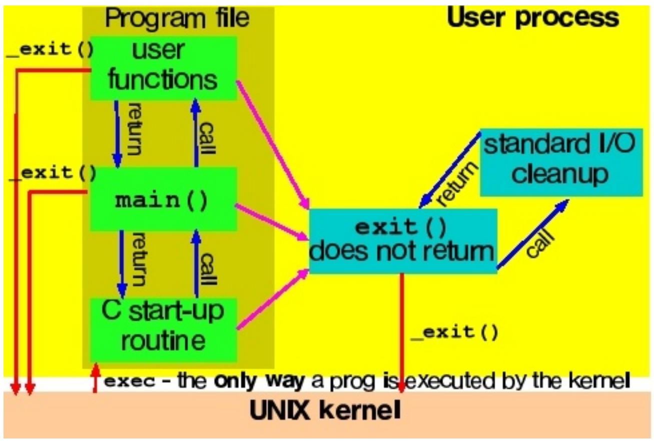

Figure 11.1: Creation and termination of process

Process execution

The *exec()* family of functions replaces the current process image with a new process image. The text, data and stack segment of the process are replaced and only the u (user) area of the process remains the same. It is important to realize that control is not passed back to the calling process unless an error occurred with the *exec()* call.

The versions of exec() are *execl (), execv(), execle(), execve(), execlp(), execvp()*. The naming convention of *exec()* is as follows

’l’ indicates a list arrangement (a series of null terminated arguments)

’v’ indicate the array or vector arrangement (like the argv structure).

’e’ indicates the programmer will construct (in the array/vector format) and pass their own environment variable list

’p’ indicates the current PATH string should be used when the system searches for executable files.

```
#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg, ...);

int execlp(const char *file, const char *arg, ...);

int execle(const char *path, const char *arg , ..., char * const envp[]);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);
```

Process Suspension

The *wait()* function suspends execution of its calling process until status information is available for a terminated child process, or a signal is received.

```
#include <sys/types.>

#include <sys/wait.h>

pid_t wait(int *stat_loc);

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

The *waitpid()* function shall be equivalent to wait() if the pid argument is (pid_t)-1 and the options argument is 0. Otherwise, its behavior shall be modified by the values of the pid and options arguments. The *wait()* and *waitpid()* functions shall obtain status information pertaining to one of the caller’s child processes. Various options permit status information to be obtained for child processes that have terminated or stopped. If status information is available for two or more child processes, the order in which their status is reported is unspecified.

The pid argument specifies a set of child processes for which status is requested. The *waitpid()* function shall only return the status of a child process from this set:

If pid is greater than 0, it specifies the process ID of a single child process for which status is requested

If pid is 0, status is requested for any child process whose process group ID is equal to that of the calling process

If pid is less than (pid_t)-1, status is requested for any child process whose process group ID is equal to the absolute value of pid.

The options argument is constructed from the bitwise-inclusive OR of zero or more of the following flags, defined in the <sys/wait.h> header:

WCONTINUED

The *waitpid()* function shall report the status of any continued child process specified by pid whose status has not been reported since it continued from a job control stop.

WNOHANG

The *waitpid()* function shall not suspend execution of the calling thread if status is not immediately available for one of the child processes specified by pid.

WUNTRACED

The status of any child processes specified by pid that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the requesting process.

The following macros may be used to test the manner of exit of the process. One of the first three macros will evaluate to a non-zero (true) value:

WIFEXITED(status) - True if the process terminated normally by a call to _exit(2) or exit(3).

WIFSIGNALED(status) - True if the process terminated due to receipt of a signal.

WIFSTOPPED(status) - True if the process has not terminated, but has stopped and can be restarted. This macro can be true only if the wait call specified the WUNTRACED option or if the child process is being traced. Depending on the values of those macros, the following macros produce the remaining status information about the child process

WEXITSTATUS(status) - If WIFEXITED(status) is true, evaluates to the low-order 8 bits of the argument passed to _exit(2) or exit(3) by the child.

WTERMSIG(status) - If WIFSIGNALED(status) is true, evaluates to the number of the signal that caused the termination of the process.

WCOREDUMP(status) - If WIFSIGNALED(status) is true, evaluates as true if the termination of the process was accompanied by the creation of a core file containing an image of the process when the signal was received.

WSTOPSIG(status) - If WIFSTOPPED(status) is true, evaluates to the number of the signal that caused the process to stop.

[h]

Process ID

The process ID is a (long) positive integer uniquely identifying the process to the operating system. It is possible to determine the process ID of a parent and its child through the following system calls.

```
#include <sys/types.>

#include <unistd.h>

pid_t getpid (void);

pid_t getppid(void);
```

There is also an effective user ID which identies the priviledge of that process in accessing resources such as shared memory, and semaphores. The user ID and effective user ID?s can be accessed through the following system calls.

```
#include <sys/types.>

#include <unistd.h>

pid_t getuid (void);

pid_t geteuid(void);
```

When a normal program is executed, the effective and real group ID of the process are set to the group ID of the user executing the file. When a set ID program is executed the real group ID is set to the group of the calling user and the effective user ID corresponds to the set group ID bit on the file being executed.

```
#include <sys/types.>

#include <unistd.h>

pid_t getgid (void);

pid_t getegid(void);
```

getgid() returns the real group ID of the current process. *getegid()* returns the effective group ID of the current process.

11.2 Summary

Exercises

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int
main(int argc, char *argv[])
{
  pid_t child_pid, wpid;
  int status;

  child_pid = fork();
  if (child_pid == -1)
  { /* fork() failed */
    perror("fork");
    exit(EXIT_FAILURE);
  }

  if (child_pid == 0)
  {  /* This is the child */
     /* Child does some work and then terminates */
  }
  else
  {
    /* This is the parent */
    do {
         wpid = waitpid(child_pid, &status, WUNTRACED);
         if (wpid == -1)
         {
            perror("waitpid");
            exit(EXIT_FAILURE);
         }
         if (WIFEXITED(status))
            printf("child exited, status=%d\n", WEXITSTATUS(status));
         else if (WIFSIGNALED(status))
            printf("child killed (signal %d)\n", WTERMSIG(status));
         else if (WIFSTOPPED(status))
            printf("child stopped (signal %d)\n", WSTOPSIG(status));
         else if (WIFCONTINUED(status))
            printf("child continued\n");
         else
         {    /* Non-standard case -- may never happen */
            printf("Unexpected status (0x%x)\n", status);
         }
      } while (!WIFEXITED(status) && !WIFSIGNALED(status));
   }
}
```

Chapter 12

Interprocess Communication

12.1 Introduction

Networking involves communication between processes. Not all interprocess communication involves networking. Typically, IPC services are provided by the Operating System.

Several Different Methods of IPC in UNIX are as follows:

- Pipe: one-way data stream
- FIFO: special kind of pipe
- Message Queue: message passing (not a data stream)
- Semaphores: Synchronization primitive
- Shared Memory: data does not go through the kernel (Operating System).

12.2 Pipes

Pipes are the oldest form of UNIX System IPC. It provides a one-way flow of data. For example, the command # who | sort | lpr gives the output of who is input to sort output of sort is input to lpr. A pipe is a pipe is a data structure in the kernel and it is created by using the pipe system call. But it have two limitations.

They have been half duplex (i.e., data flows in only one direction) and it can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child. A pipe is created by using the pipe system call

```
#include <unistd.h>


int pipe(int fd[2]);


                                        Returns: 0 on success, 1 on error
```

Two file descriptors are returned through the fd argument: fd[0] is open for reading, and fd[1] is open for writing.

12.3 FIFO

The acronym for FIFO is first in first out. They are also called as Named Pipes. Pipes can be used only between related processes when a common ancestor has created the pipe. They are similar to pipes except that these can be used to communicate between unrelated processes.