Preview

- The waitpid() System Call
- The system() System Call
- Concept of Signals
  - Linux Signals
  - The signal System Call
  - Unreliable Signals
- Signal() System Call
- The kill() and raise() System Call
- The alarm() System Call
- The pause() System Call

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   1

Process Termination

- The exit status of a child will be used for parent process termination
- When a child exits, the parent process will receive a SIGCHLD signal to indicate that one of its children has finished executing; the parent process will typically call the wait() system call at this point.
- That call will provide the parent with the child's exit status, and will cause the child to be *reaped*, or removed from the process table.

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   2

wait and waitpid() System Call

- When a process terminate either normally or abnormally, the kernel sent a signal (SIGCHD) to a parent.
- A parent can ignore the signal or call a function (wait or waitpid) to take care the signal.

```
#include <sys/wait.h>
pid_t wait (int *status);
pid_t waitpid(pid_t pid, int *status, int option);

                        Returns ID if OK, -1 on error
```

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   3

wait and waitpid() System Call

- The execution of **wait()** could have two possible situations.
  - If there are at least one child processes, the caller will be blocked until one of its child processes exits.
  - If there is no child process running, then this **wait()** has no effect at all.
- **The status is the pointer** where terminated process's status is saved.

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   4

```
/*wait.c  demonstrate wait() system call */
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#define  MAX_COUNT  1000
#define  BUF_SIZE  100
void  main(void)
{

   pid_t  pid1, pid, p1;
   int    status;
   int    i;
   char   buf[BUF_SIZE];
   printf("*** Parent is about to fork  ***\n");
   if ((pid1 = fork()) < 0)
   {
                   printf("Failed to fork process 1\n");
                   exit(1);
   }
   if (pid1 == 0) /* child's part */
   {
          p1= getpid();
       for (i = 1; i <MAX_COUNT; i++)
       {
          sprintf(buf,"child %d is running\n", p1);
          write(1, buf, strlen(buf));
       }
   }
   else
   {
          sprintf(buf, "*** Parent enters waiting status .....\n");
          write(1, buf, strlen(buf));
          pid = wait(&status); /*wait for child finish it's job */
          sprintf(buf, "*** Parent detects process %d was done ***\n", pid);
          write(1, buf, strlen(buf));
          printf("*** Parent exits ***\n");
          exit(0);
   }
}
```

waitpid() System Call

- If a parent create more than one child, wait returns on termination of any of children process.
- **waitpid**() system call can be used to wait for specific process to terminate.

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int option);

                    Returns ID if OK, -1 on error
```

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   6

waitpid() System Call

- □ The differences between wait() and waitpid()
  - ▪ The wait can block the caller(parent) until a child process terminates.
  - ▪ The waitpid() has option that prevents it from blocking
  - ▪ The waitpid() function doesn't wait for the child that terminate first; it has options that control which process parent wait for.

waitpid() System Call

- □ The waitpid() system call provides three features that does not by the wait()
  - ▪ Wait for one particular process
  - ▪ Provide non blocking version of wait
  - ▪ Provides support for job control
- □ Options
  - ▪ 0: wait one by one until all terminated
  - ▪ WNOHANG: will not block if the child is not available immediately
  - ▪ WCONTINUED: if implementation support job control,
  - ▪ WUNTRACED : if implementation support job control,

```
/* waitpid.c: demonstrate waitpid system call */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;
    int i, j;

    if ( (pid = fork()) < 0) /*create the first child */
    {
        printf("fork error");
        exit (1);
    }
    if (pid == 0) /* code for first child */
    {
        //create a second child by the first child
        if ( (pid = fork()) < 0)
        {
            printf("fork error");
            exit (2);
        }
        if (pid > 0) //code for the first child
        {
            for (i = 0; i<=10; i++)
            {
                printf ("This is the first child with id = %d, parent pid = %d\n", getpid(),getppid());
                sleep(1);
            }
            _exit(0);
        }
        else //code for the second child
        {
            for (j =0; j <= 20; j++)
            {
                printf("This is the second child with id = %d, parent pid = %d\n", getpid(),getppid());
                sleep(1);
            }
            _exit(0);
        }
    }
    printf("Now parent is waiting for first child's termination\n");
    if (waitpid(pid, NULL, 0) != pid)  /* wait for first child */
    {
        printf("waitpid ERROR \n");
        exit(3);
    }
    for (j =0; j <= 20; j++)
    {
        printf("This is parent with id = %d\n", getpid());
        sleep(1);
    }
    printf ("now parent %d terminated \n",pid);
    exit(0);
}
```

```
/* waitpid1.c: demonstrate waitpid system call */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;

    printf("My pid = %d \n", getpid());
    if ( (pid = fork()) < 0) /*create the first child */
    {
        printf("fork error");
        exit (1);
    }
    if (pid == 0) /* code for first child */
    {
        printf("A Child pid = %d \n", getpid());
        //create a second child by the first child
        if ( (pid = fork()) < 0)
        {
            printf("fork error");
            exit (2);
        }
        if (pid > 0)
        {
            printf("first child pid=%d is terminated\n", getpid());
            exit(0);  /* parent from second fork == first child */
        }
        else
        {
            printf("Gradchild's pid = %d, parent pid = %d\n",getpid(), getppid());
            sleep(10);
            printf("second child terminated parent pid = %d\n",getpid());
            exit(0);
        }
    }
    if (waitpid(pid, NULL, 0) != pid)        /* wait for first child*/
    {
        printf("waitpid ERROR \n");
        exit(3);
    }
    printf ("now parent %d terminated \n",getppid());

    exit(0);
}
```

```
/* waitpid2.c */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i, status;
    pid_t childID, endID;
    time_t when;
    if ((childID = fork()) == -1) /* Create a Child */
    {
        perror("fork error");
        exit(1);
    }
    else if (childID == 0) /* This is the child. */
    {
        time(&when);
        printf("Child process started at %s", ctime(&when));
        sleep(15);                 /* Sleep for 10 seconds.      */
        exit(0);
    }
    else   /* This is the parent. */
    {
        time(&when);
        printf("Parent process started at %s", ctime(&when));
        /* Wait 15 seconds for child process to terminate. */
        endID = waitpid(childID, &status, WNOHANG|WUNTRACED);
        for(i = 0; i < 15; i++)
        {
            if (endID == 0) /* child still running */
            {
                time(&when);
                printf("Parent waiting for child at %s", ctime(&when));
                sleep(1);
            }
        }
        printf("Now my child finish job at %s", ctime(&when));
        exit (0);
    }
}
```

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

int main(void)
{
    int i, status;
    pid_t childID, endID;
    time_t when;

    if ((childID = fork()) == -1) /* Create a Child */
    {
        perror("fork error");
        exit(1);
    }
    else if (childID == 0) /* This is the child. */
    {
        time(&when);
        printf("Child process started at %s", ctime(&when));
        if (gran_fork()) <0)
        {
            perror("fork error");
            exit(2);
        }
        if (pid >0)
        {
            for (i=0; i <5; i++)
            {
                printf("the first child is running at %s",ctime(&when));
                sleep(1);
            }
            printf("the first child finish job at %s", ctime(&when));
            exit(0);
        }
        else /* second child */
        {
            while (1)
            {
                printf("the second child is running at %s",ctime(&when));
                sleep(1);
            }
        }
    }
    else  /* This is the parent. */
    {
        time(&when);
        printf("Parent process started at %s", ctime(&when));
        /* Wait for the first child process to terminate. */
        while(1)
        {
            endID = waitpid(childID, &status, WNOHANG|WUNTRACED);
            if (endID == 0) /* child still running */
            {
                time(&when);
                printf("Parent waiting for child at %s", ctime(&when));
                sleep(1);
            }
            else
                break;
        }
        printf("Now my child finish job at %s", ctime(&when));
        exit (0);
    }
}
```

The system() System Call

- We can execute bash commands inside c program by using system() system call.
- The system() is implemented by calling fork(), exec, and waitpid, there are three types of return value
  - - 1: If either the fork() or waitpid() failed.
  - 127: if the exec failed
  - Other: the termination status of waitpid()

```
/* systemtest.c: demonstrate system() system call */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int status;

    if ( (status = system("date")) < 0)
    {
        printf("system() ERROR\n");
        exit(1);
    }
    printf ("Status = %d\n", status);
    /*non existed command */
    if ( (status = system("nosuchcommand")) < 0)
    {
        printf("system() ERROR\n");
        exit(2);
    }
    printf ("Status = %d\n", status);

    if ( (status = system("who; exit 44")) < 0)
    {
        printf("system() ERROR\n");
        exit(3);
    }
    printf ("Status = %d\n", status);

    exit(0);
}
```

Concepts of Signal

- Signals are **software interrupts**.
- Signals **provide a way of handling asynchronous event**.
- Signals are a fundamental **method for interprocess communication.**
- Each signal names begin with SIG.
- Linux support 31 standard signals (signal.h) and additional application defined signals.

Concepts of Signal

- SIGHUP 1 Hangup (POSIX)
- SIGINT 2 Terminal interrupt (ANSI)
- SIGQUIT 3 Terminal quit (POSIX)
- SIGILL 4 Illegal instruction (ANSI)
- SIGTRAP 5 Trace trap (POSIX)
- SIGIOT 6 IOT Trap (4.2 BSD)
- SIGBUS 7 BUS error (4.2 BSD)
- SIGFPE 8 Floating point exception (ANSI)
- SIGKILL 9 Kill(can't be caught or ignored) (POSIX)
- SIGUSR1 10 User defined signal 1 (POSIX)
- SIGSEGV 11 Invalid memory segment access (ANSI)
- SIGUSR2 12 User defined signal 2 (POSIX)
- SIGPIPE 13 Write on a pipe with no reader, Broken pipe (POSIX)
- SIGALRM 14 Alarm clock (POSIX)
- SIGTERM 15 Termination (ANSI)

Concepts of Signal

- SIGSTKFLT 16 Stack fault
- SIGCHLD 17 Child process has stopped or exited, changed (POSIX)
- SIGCONT 18 Continue executing, if stopped (POSIX)
- SIGSTOP 19 Stop executing(can't be caught or ignored) (POSIX)
- SIGTSTP 20 Terminal stop signal (POSIX)
- SIGTTIN 21 Background process trying to read, from TTY (POSIX)
- SIGTTOU 22 Background process trying to write, to TTY (POSIX)
- SIGURG 23 Urgent condition on socket (4.2 BSD)
- SIGXCPU 24 CPU limit exceeded (4.2 BSD)
- SIGXFSZ 25 File size limit exceeded (4.2 BSD)
- SIGVTALRM 26 Virtual alarm clock (4.2 BSD)
- SIGPROF 27 Profiling alarm clock (4.2 BSD)
- SIGWINCH 28 Window size change (4.3 BSD, Sun)
- SIGIO 29 I/O now possible (4.2 BSD)
- SIGPWR 30 Power failure restart (System V)

Concepts of Signal

- Conditions for generating signal
  - **The terminal-generated signals** occur when users press certain terminal key (ctr-c (SIGSTOP), ctr-z(SIGINT), ctr-d (EOF)…).
  - **Hardware exception generate signals** – invalid memory reference.
  - **kill system call allows a process to send a signal** to a process or group of process
  - **kill command (bash) allows us to send signals** to other process.
  - **Software condition can generate signal** when something happed – out of band data arrived over network

Concepts of Signal

- Kernel do one of three things for a signal
  - **Ignore the signal** – This works for most of signal except SIGKILL and SIGSTOP.
  - **Catch the signal** – ask the kernel to call a function of ours (signal handler) whenever the signal occurs. When a child process terminate, the SIGCHILD can catch by signal() or sigaction() system call and this signal can be used to initiate a user defined function.
  - **Let default action apply** – every signal has a default action, such as terminate or ignore.

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   19

The kill() and raise() System Calls

- The **kill()** system call send a signal to a specific process or a group of processes.
- The **raise()** system call allows a process to send a signal to itself.

```
#include <signal.h>
int kill (pid_t pid, int signo);
int raise (int signo);

Return 0 if ok else return -1
```

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   20

The kill() and raise() System Calls

- There are four different condition for the pid argument to kill().
  - **pid >0**: The signal send to the process with ID = pid.
  - **pid ==0**: The signal send to all process whose process group ID equal to sender's group ID with sender has permission to send.
  - **pid <0**: The signal send to all processes whose process group ID equal to the absolute value of pid with sender has permission to send.
  - **pid == -1**: The signal send to all processes on the system with sender has permission to send.

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   21

The kill() and raise() System Calls

- Permission to send
  - The super-user can send a signal to any processes
  - If real or effective ID of a sender's ID is same as receiver process, the sender send signal to them.

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   22

The alarm() System Call

- The alarm() system call allows to set a timer that will expire at a specified time in the future.
- When the timer expires, the SIGALAM signal is generated.

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);

                Return 0 or number of seconds
```

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   23

The alarm() System Call

- There is only one of alarm clocks per a process.
- When we call alarm() system call and if a previously registered clock for the process has not yet expired, the remaining time will be return as a value of this function.

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park   24

The pause() System Call

- The pause() system call suspends the calling process until a signal is caught

```
#include <unistd.h>
int pause (void);

                    Return -1 with error
```

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park
25

The signal() System Call

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t);

                              Return -1 with error
```

signo: name of signal SIG…
func: the const SIG_IGN - ignore
      the const SIG_DEF - default
      a function address to be called when the signal occurs

- Function **signal** accept two arguments and return a pointer to a function that returns nothing.
- Second argument is pointer to a function that take a single integer argument and return nothing.

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park
26

The pause() System Call

```
/* alarm1.c: demonstrate a signal system call*/
#include <time.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void ding (int sig)
{
   time_t t =time((time_t *)0);
   printf("alarm fired at %s by signal = %d\n", asctime(localtime(&t)),sig);
}

int main()
{
   time_t t = time((time_t *)0);
   printf("set alram 5 second for a process at time %s\n", asctime(localtime(&t)));
   alarm(5);
   alarm(10);
   signal (SIGALRM, ding);
   pause();
   exit(0);

}
```

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park
27

The signal() System Call

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signo, sighandler_t);

                              Return -1 with error
```

signo: name of signal SIG…
func: the const SIG_IGN - ignore
      the const SIG_DEF - default
      a function address to be called when the signal occurs

- Function **signal** accept two arguments and return a pointer to a function that returns nothing.
- Second argument is pointer to a function that take a single integer argument and return nothing.

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park
28

The signal() System Call

- Most of the Linux users use the key combination ctrl+c to terminate processes in Linux.
- Whenever ctrl+c is pressed, a signal SIGINT is sent to the process.
- The default action of this signal is to terminate the process.
- But this signal can also be handled. The following code demonstrates this case

COSC350 System Software, Fall 2020
Dr. Sang-Eon Park
29

The signal() System Call

```
// catchsignal.c example for catching signal SIGINT
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
#include <time.h>

void sig_handler(int signo)
{
    if (signo == SIGINT)
    {
        sleep (1);
        time_t t =time((time_t *)0);
         printf("received SIGINT at %s\n", asctime(localtime(&t)));
    }
}
int main(void)
{
    if (signal(SIGINT, sig_handler) == SIG_ERR)
      printf("\ncan't catch SIGINT\n");
      // A long long wait for a signal. this process suspend for a signal
      while(1)
      {
                pause(1);
      }
      return 0;
}
```

Dr. Sang-Eon Park