UNIT - 5

PROCESS CONTROL

5.1 Process identifiers

- Every process has a unique process ID, a non negative integer
- Special processes : process ID 0 scheduler process also known as swapper process ID 1 init process init process never dies ,it's a normal user process run with super user privilege process ID 2 pagedaemon

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid (void);
pid_t getppid (void);
uid_t getuid (void);
uid_t geteuid (void);
gid_t getgid (void);
gid_t getegid (void);
```

Fork function

- The only way a new process is created by UNIX kernel is when an existing process calls the fork function

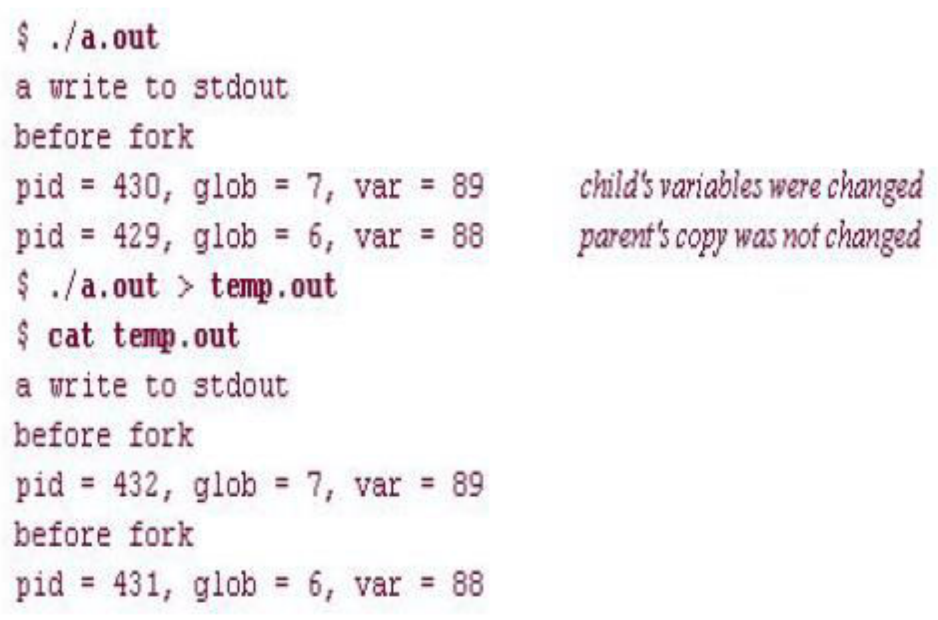
```
#include <sys/types.h>
#include <unistd.h>
 pid_t fork (void);
```

- The new process created by fork is called child process
- The function is called once but returns twice
- The return value in the child is 0
- The return value in parent is the process ID of the new child
- The child is a copy of parent
- Child gets a copy of parents text, data , heap and stack
- Instead of completely copying we can use COW copy on write technique

```
#include<sys/types.h>
#include "ourhdr.h"
int  glob = 6;
/* external variable in initialized data */
char          buf[ ] = "a write to stdout\n";
int main(void)
{
    int       var;
    /* automatic variable on the stack */
    pid_t   pid;
    var = 88;
```

```
        if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
                err_sys("write error");
              printf("before fork\n");
        if ( (pid = fork()) < 0)
                err_sys("fork error");
        else if (pid == 0)
        {                          /* child */
                glob++;            /* modify variables */
                var++;
        }
else
                sleep(2);
        /* parent */
        printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
        exit(0);
}
```

Output



```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89        child's variables were changed
pid = 429, glob = 6, var = 88        parent's copy was not changed
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

file sharing

- Fork creates a duplicate copy of the file descriptors opened by parent
- There are two ways of handling descriptors after fork

1. The parent waits for the child to complete
2. After fork the parent closes all descriptors that it doesn't need and the does the same thing

Besides open files the other properties inherited by child are

- Real user ID, group ID, effective user ID, effective group ID
- Supplementary group ID
- Process group ID
- Session ID
- Controlling terminal
- set-user-ID and set-group-ID
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Resource limits

The difference between the parent and child

- The return value of fork
- The process ID
- Parent process ID
- The values of tms_utime , tms_stime , tms_cutime , tms_ustime is 0 for child
- file locks set by parent are not inherited by child
- Pending alrams are cleared for the child
- The set of pending signals for the child is set to empty set

■ The functions of fork

1. A process can duplicate itself so that parent and child can each execute different sections of code
2. A process can execute a different program

vfork

- It is same as fork
- It is intended to create a new process when the purpose of new process is to exec a new program
- The child runs in the same address space as parent until it calls either exec or exit
- vfork guarantees that the child runs first , until the child calls exec or exit

```
int glob = 6;
    /* external variable in initialized data */
    int main(void)
    {
            int         var;
      /* automatic variable on the stack */
              pid_t    pid;
              var = 88;
              printf("before vfork\n");
       if ( (pid = vfork()) < 0)
              err_sys("vfork error");
      else if (pid == 0) {               /* child */
              glob++;
      /* modify parent's variables */
              var++;
              _exit(0);                  /* child terminates */
      }
      /* parent */
      printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
      exit(0);
}
```

5.2 exit functions

- Normal termination

1. Return from main
2. Calling exit – includes calling exit handlers
3. Calling _exit – it is called by exit function

- Abnormal termination

1. Calling abort – SIGABRT
2. When process receives certain signals

- Exit status is used to notify parent how a child terminated
- When a parent terminates before the child, the child is inherited by init process
- If the child terminates before the parent then the information about the is obtained by parent when it executes wait or waitpid

- The information consists of the process ID, the termination status and amount of CPU time taken by process
- A process that has terminated , but whose parents has not yet waited for it, is called a zombie
- When a process inherited by init terminates it doesn't become a zombie
- Init executes one of the wait functions to fetch the termination status

5.3 Wait and waitpid functions

- When a child id terminated the parent is notified by the kernel by sending a SIGCHLD signal
- The termination of a child is an asynchronous event
- The parent can ignore or can provide a function that is called when the signal occurs
- The process that calls wait or waitpid can
  1. Block
  2. Return immediately with termination status of the child
  3. Return immediately with an error

  ```
  #include <sys/wait.h>
  #include <sys/types.h>
  pid_t wait (int *statloc);
  pid_t waitpid (pid_t pid,int *statloc , int options );
  ```
- Statloc is a pointer to integer
- If statloc is not a null pointer ,the termination status of the terminated process is stored in the location pointed to by the argument
- The integer status returned by the two functions give information about exit status, signal number and about generation of core file
- Macros which provide information about how a process terminated

Program to demonstrate the use of the exit status

```
#include "apue.h"
#include <sys/wait.h>
```

```
Void pr_exit(int status)
{
   if (WIFEXITED(status))
      printf("normal termination, exit status = %d\n",WEXITSTATUS(status));
   else if (WIFSIGNALED(status))
      printf("abnormal termination, signal number = %d%s\n",WTERMSIG(status),
   #ifdef  WCOREDUMP
            WCOREDUMP(status) ? " (core file generated)" : "");
   #else
            "");
   #endif

          else if (WIFSTOPPED(status))
            printf("child stopped, signal number = %d\n",WSTOPSIG(status));
     }
```

WIFEXITED	TRUE – if child terminated normally WEXITSTATUS – is used to fetch the lower 8 bits of argument child passed to exit or _exit
WIFSIGNALED	TRUE – if child terminated abnormally WTERMSIG – is used to fetch the signal number that caused termination WCOREDUMP – is true is core file was generated
WIFSTOPPED	TRUE – for a child that is currently stopped WSTOPSIG -- is used to fetch the signal number that caused child to stop

5.4 Waitpid

- The interpretation of pid in waitpid depends on its value

  1. Pid == -1 – waits for any child
  2. Pid > 0 – waits for child whose process ID equals pid
  3. Pid == 0 – waits for child whose process group ID equals that of calling process
  4. Pid < -1 – waits for child whose process group ID equals to absolute value of pid

- Waitpid helps us wait for a particular process
- It is nonblocking version of wait
- It supports job control

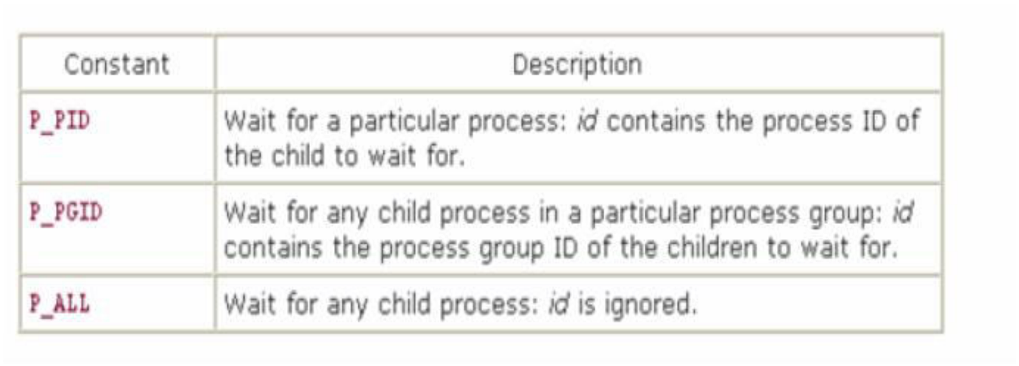
WNOHANG	Waitpid will not block if the child specified is not available
WUNTRACED	supports job control

```
#include    <sys/types.h>
#include    <sys/wait.h>
#include    "ourhdr.h"
Int main(void)
{
    pid_t   pid;
    int             status;
    if ( (pid = fork()) < 0)
            err_sys("fork error");
    else if (pid == 0)              /* child */
            exit(7);
if (wait(&status) != pid)
                        /* wait for child */
            err_sys("wait error");
      pr_exit(status);
                    /* and print its status */
  if ( (pid = fork()) < 0)
           err_sys("fork error");
  else if (pid == 0)              /* child */
           abort();
             /* generates SIGABRT */
if (wait(&status) != pid)
                        /* wait for child */
           err_sys("wait error");
  pr_exit(status);
```

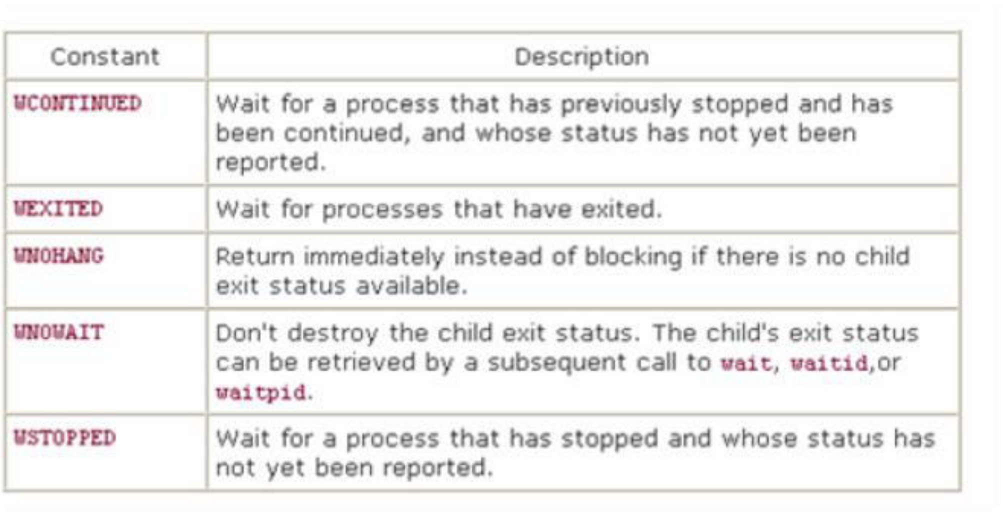
```
                              /* and print its status */
          if ( (pid = fork()) < 0)
                    err_sys("fork error");
          else if (pid == 0)                  /* child */
                    status /= 0;
               /* divide by 0 generates SIGFPE */
          if (wait(&status) != pid)
                                        /* wait for child */
                    err_sys("wait error");
               pr_exit(status);
                              /* and print its status */


          exit(0);
     }
```

5.5 Waitid

```
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t   *infop, int options);
Returns: 0 if OK, 1 on error
```



Constant	Description
P_PID	Wait for a particular process: *id* contains the process ID of the child to wait for.
P_PGID	Wait for any child process in a particular process group: *id* contains the process group ID of the children to wait for.
P_ALL	Wait for any child process: *id* is ignored.



Constant	Description
WCONTINUED	Wait for a process that has previously stopped and has been continued, and whose status has not yet been reported.
WEXITED	Wait for processes that have exited.
WNOHANG	Return immediately instead of blocking if there is no child exit status available.
WNOWAIT	Don't destroy the child exit status. The child's exit status can be retrieved by a subsequent call to wait, waitid, or waitpid.
WSTOPPED	Wait for a process that has stopped and whose status has not yet been reported.

5.6 wait3 and wait4 functions

- These functions are same as waitpid but provide additional information about the resources used by the terminated process

  ```
  #include <sys/wait.h>
  #include <sys/types.h>
  #include <sys/times.h>
  #include <sys/resource.h>
  pid_t wait3 (int *statloc ,int options, struct  rusage *rusage );
  pid_t wait4 (pid_t pid ,int *statloc ,int  options, struct rusage *rusage );
  ```

5.7 Race condition

- Race condition occurs when multiple process are trying to do something with shared data and final out come depends on the order in which the processes run

Program with race condition

```
#include    <sys/types.h>
#include    "ourhdr.h"
static void charatatime(char *);
int main(void)
{
     pid_t   pid;
     if ( (pid = fork()) < 0)
               err_sys("fork error");
else if (pid == 0)
   {
               charatatime("output from child\n");
    }
 else
  {
```

```
                charatatime("output from parent\n");
        }
        exit(0);
    }
    static void
    charatatime(char *str)
    {
        char     *ptr;
        int               c;
        setbuf(stdout, NULL);
                              /* set unbuffered */
        for (ptr = str; c = *ptr++; )
                putc(c, stdout);
}
/*altered program*/
#include    <sys/types.h>
#include    "ourhdr.h"
static void charatatime(char *);
Int main(void)
{
     pid_t    pid;
     TELL_WAIT();
   if ( (pid = fork()) < 0)
                err_sys("fork error");
else if (pid == 0)
```

```
 {
   WAIT_PARENT();      /* parent goes first */
            charatatime("output from child\n");
 }
 else {
            charatatime("output from parent\n");
            TELL_CHILD(pid);
   }
   exit(0);
}
 static void charatatime(char *str)
 {
    char    *ptr;
    int                 c;
    setbuf(stdout, NULL);
                              /* set unbuffered */
    for (ptr = str; c = *ptr++; )
             putc(c, stdout);
```

5.8 exec functions

- Exec replaces the calling process by a new program
- The new program has same process ID as the calling process
- No new program is created , exec just replaces the current process by a new program

```
#include <unistd.h>
int execl ( const char *pathname,  const char *arg0 ,... /*(char *) 0*/);
int execv (const char *pathname, char * const argv[ ]);
int execle (const char *pathname, const   char *arg0 ,... /*(char *) 0,
                        char  *const envp[ ] */);
int execve ( const char *pathname,  char *const argv[ ] , char *const  envp [ ]);
int execlp (const char *filename, const char *arg0 ,... /*(char *) 0*/);
int execvp (const char *filename ,char  *const argv[ ] );
```

```
#include        <sys/types.h>
#include        <sys/wait.h>
#include        "ourhdr.h"
char    *env_init[ ] =
{ "USER=unknown", "PATH=/tmp", NULL };
int main(void)
{
        pid_t   pid;
        if ( (pid = fork()) < 0)
                err_sys("fork error");
else if (pid == 0) {
/* specify pathname, specify environment */
if ( execle ("/home/stevens/bin/echoall",
   "echoall", "myarg1", "MY ARG2",
  (char *)  0,   env_init) < 0)
                                  err_sys("execle error");
             }
             if (waitpid(pid, NULL, 0) < 0)
                         err_sys("wait error");

             if ( (pid = fork()) < 0)
                         err_sys("fork error");

 else if (pid == 0) {
/* specify filename, inherit environment */
                        if (execlp("echoall",
                                  "echoall", "only 1 arg",
                                               (char *) 0) < 0)
                                    err_sys("execlp error");
            }
            exit(0);
}
```