Process Control (Chapter 8)

- Process Identifiers
- Process Life Cycle
  - **`fork & vfork`**
  - **`Wait & waitpid`**
- Race condition
- exec functions
- system functions

Process IDentifiers

- Every process has a unique process identifier (PID)
- PIDs are non-negative integers
- There are two special processes
  - 0 :scheduler, a system process
  - 1 : init, the parent of all other UNIX processes on a Linux/Unix system
- Relevant C Functions:

```
pid_t getpid();  // process ID
pid_t getppid(); // parent PID
uid_t getuid(); // real user ID of proc
uid_t geteuid() // effective uid of proc
gid_t getgid()   // group ID
gid_t getegid(); // effective group ID
```

Shell command: "ps", "top"

- Shell command "ps":
  - Report a snapshot of the current processes
  - To see every process on the system using standard syntax (UNIX System V)

    ps -e

    ps -ef

    ps -eF

    ps -ely
  - To see every process on the system using BSD syntax:

    ps ax or ps axu
    - a = show processes for all users
    - u = display the process's user/owner
    - x = show all running processes
- Shell command "top"

  check man page for detail

Process State Codes

Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") displayed to describe the state of a process.

D Uninterruptible sleep (usually IO)

R Running or runnable (on run queue)

S Interruptible sleep (waiting for an event to complete)

T Stopped, either by a job control signal or because it is being traced.

W paging (not valid since the 2.6.xx kernel)

X dead (should never be seen)

Z Defunct ("zombie") process, terminated but not reaped by its parent.

For BSD formats and when the stat keyword is used, additional characters may be displayed:

< high-priority (not nice to other users)

N low-priority (nice to other users)

L has pages locked into memory (for real-time and custom IO)

s is a session leader

l is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)

+ is in the foreground process group

Google “output fields of the ps command in UNIX”

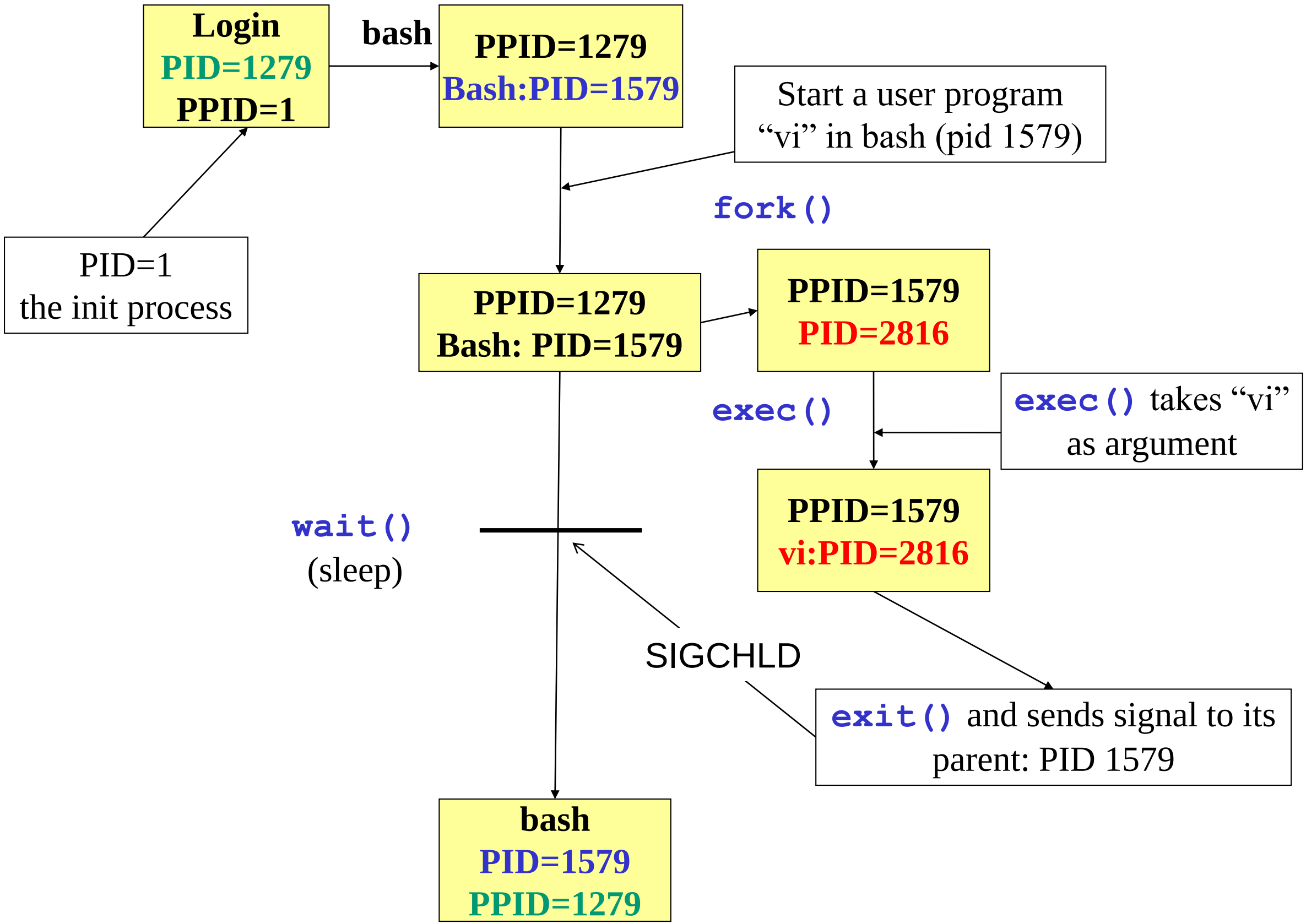
Life Cycle of a UNIX/Linux Process

- On UNIX, process creation, execution and termination are done by a set of four system calls
  - **fork, wait, exec**, & **exit**
- **fork** system call

  A process is created in UNIX with the fork() system call
  - It creates a duplicate process of the calling process, the duplicated process is the child process of the calling one
  - The calling process is called the parent
  - A parent process can have many child processes, but a child process can have only one parent process
  - The child process starts right after the call of fork. It is responsible to do the work, such as the command called through command line
- **wait** system call
  - Suspend the parent process after the **fork** system call

Life Cycle of a UNIX Process (Cont'd)

- The **`exec`** system call
  - Real execution is done by the child process through the **`exec`** system call
  - The child process calls `exec` with the name of the command as its argument
  - **The kernel loads this new program into memory in place of the shell that called it (the child process) and becomes the child process (hold its PID) and starts executing.**
- The **`exit`** system call
  - Called when the child process terminates. It sends a signal (SIGCHLD) to its parent,
  - The parent wakes up when it receives the SIGCHILD signal, and you get your prompt back


Login
PID=1279
PPID=1
bash
PPID=1279
Bash:PID=1579
Start a user program
“vi” in bash (pid 1579)
fork()
PID=1
the init process
PPID=1279
Bash: PID=1579
PPID=1579
PID=2816
exec()
exec() takes “vi”
as argument
PPID=1579
vi:PID=2816
wait()
(sleep)
SIGCHLD
exit() and sends signal to its
parent: PID 1579
bash
PID=1579
PPID=1279

Process Termination

- Normal termination
  - Return from “main”
  - Calling exit (glibc function)
  - Calling _exit or _Exit (system call)
- Control-Key(s)
  - `Ctrl-C`: send SIGINT (interrupt) signal
  - `Ctrl-\`: send SIGQUIT signal
    - stronger than ctrl-C, used when ctrl-C does not work
- cmd `kill` with the following arguments and signals
  - `kill 12345` # PID=12345 how to get the pid?), send default signal
  - `killall foo  #` Process name foo, might have multiple “foo” running
  - `send signal to the running process: such as kill -9 12345`
    - `-TERM`: termination, by default (15)
    - `-QUIT`: quit (3); -KILL (9)
    - `-KILL`: `kill -9 12345`
    - `-s signal pid`
- `kill -l`: List all the available signals on the system

“fork” Function-Create a New Process

Every process except system processes like “swapper”, a scheduler process, and the init process are started via a function call named **fork**

```
pid_t fork(void);
```

- Creates a “copy” of the current process
- It returns **twice**
  - **0: to the child**
  - **Positive Int (PID of the child): to the parent**
  - **-1 on error**
- Reasons for fork to fail
  - Too many processes on the system
  - Too many processes for the RUID (there is a limit CHILD_MAX)
- After fork, both the child and parent continue executing concurrently (racing condition can happen)

```c
#include “../cs590.h“ //fork.c
int     glob = 6;
char    buf[] = “a write to stdout\n";

int main(void){
    int      var = 88;
    pid_t    pid;
     if (write(1, buf, sizeof(buf)-1) != sizeof(buf)-1)
         perror("write error");
    printf("before fork, var=%d, glob=%d\n", var, glob);
    //fflush(stdout);
    if ((pid = fork()) < 0) {
        perror("fork error");
    } else if (pid == 0) {       /* child */
        glob++;                  /* modify variables */
        var++;
        printf(“child pid=%d\n”, getpid());
    } else {
        sleep(2);                /* parent */
    }
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

```
./fork
a write to stdout
before fork, var=88, glob=6
pid = 30825, glob = 7, var = 89
pid = 30824, glob = 6, var = 88

./fork >output, output has:
a write to stdout
before fork, var=88, glob=6
pid = 30979, glob = 7, var = 89
before fork, var=88, glob=6
pid = 30978, glob = 6, var = 88
```

- Both the parent and the child continue executing the instructions that follows the call to **`fork`**. The child is a copy of the parent
  - Child gets a copy of the parent's data space, heap, and stack.
  - Child and parent do not share these memory. Hence the **`printf`** call display the values of their own
  - They share the text segment
- "**`write`**" is unbuffered , we only got one line of output
- **`printf`** is buffered output, when print to terminal, it's line buffered, so by the time creating child process, the buffer has been flushed out.
  - When redirecting the output to a file, **`printf`** will be fully buffered output, so when the child is forked, the un-empty buffer is also copied to the child. So the child process also get the copy of the `printf` line
  - To avoid the buffer I/O copied to the child, call **`fflush`** before fork call
- The parent sleep(2) sleep for 2 seconds to make sure the child finishes first, but this is not guaranteed.
- Possible uses for **`fork`**
  - Concurrency – a process may want to duplicate itself so that the parent and child can each execute different sections of code at the same time
  - Program execution – a process may want to execute a different program with exec

File Sharing

- Parent and child share a file table entry for every open descriptor
- Normally, either (1) the parent waits for the child to complete, and the file descriptors are not an issue or (2) the parent and the child go separate ways, the parent closes the no need descriptors, so does the child
- Properties copied from parent to child
  - RUID RGID EUID EGID Supplementary group IDs
  - Process group ID
  - Session ID
  - Controlling terminal
  - SUID/SGID flags
  - Current working directory
  - Root directory
  - File mode creation mask
  - Signal mask and dispositions
  - Close-on-exec flag for any open file descriptors
  - Environment
  - Attached shared memory segments
  - Resource limits
- Differences b/w parent & child
  - **Return value from fork**
  - PID & Parent PID
  - Child has tms_utime tms_stime, tms_cutime, and tms_cstime set to 0
  - File locks set by parent are not inherited
  - Pending alarms are cleared for the child
  - Pending signals for the child is set to the empty set

“vfork” Function

From man page: vfork – Create a child process and block parent

- Intends to create a new process when the purpose of the new process is to exec a new program
- **`vfork`** creates the new process like fork, but until it calls either **`exec`** or **`exit`**, the child runs in the address space of the parent
- **`vfork`** guarantees the child runs first, until the child calls exec or exit. The parent is put to sleep by the kernel until the child calls either exec or exit.

```
#include "../cs590.h"
int     glob = 6;
int main(void){
    int      var;
    pid_t    pid;
    var = 88;
    printf("before vfork\n");

    if ((pid = vfork()) < 0) {
        perror("fork error");
        exit (1);
    } else if (pid == 0) {  /* child */
        glob++;  var+;      /* modify variables */
        printf("child: pid =%d,glob=%d,var=%d\n",  getpid(), glob, var);
        _exit(0);    // child terminates here
   }
    printf("pid=%d, glob=%d,var =%d\n",getpid(), glob, var);
    exit(0);
}
```

Output of the program

```
before vfork
child:pid = 3642, glob = 7, var = 89
pid = 3641, glob = 7, var = 89
```

- The incrementing of the variables done by the child changes the values in the parent;
- the child finished first

“exit” Functions

- Normal process termination:
  - Calling return from main
  - Calling exit(int status)
    - Will close all the standard I/O stream
  - Calling _exit(int status) or _Exit(int status)
    - Terminate the process without running exit handlers and signal handlers
- Abnormal termination
  - Calling abort
  - Receiving certain signals
- Argument to the exit functions is the exit status which can be retrieved by the parent

Special Cases

- Parent terminates before child – Orphaned process
  - Parent process dies/exits before child process ends
  - **`Init/systemd`** process is the parent of all orphaned processes

    When a process terminates, the kernel goes through all the active processes to see if the terminating process is the parent of any process that still exists. If so, the parent process ID of the active child process is change to be 1
- Zombie or defunct process
  - A process that has terminated, but whose parent has not yet waited for it, the process becomes a zombie
  - Zombie processes can be seen in "ps" listings occasionally as “Z” or <defunct>
- whenever a child terminates, `init` calls one of the wait functions to fetch the termination status. So orphaned process will not become zombie

wait & waitpid

```
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int opts);
```

- Returns PID of a terminated child process, -1 on error
- **statloc**: a pointer to integer stores the termination status of the child process. Use NULL if don't care about the status

  The following macros can be used to decipher the status:

  - WIFEXITED(status)
    - true for a child terminated normally
    - Can call WEXITSTATUS(status) to return the exit status
  - WIFSIGNALED(status)
    - true for a child terminated abnormally due to a signal
    - Call WTERMSIG(status) to give the signal number
  - WIFSTOPPED(status)
    - true if a child is currently stopped
    - Call WSTOPSIG(status) to return signal number

wait & waitpid

```
pid_t wait(int *statloc);
pid_t waitpid(pid_t pid, int *statloc, int opts);
```

- **opts** constants (can be zero)
  - WNOHANG – do not block
  - WUNTRACED – status of stopped job
  - WCONTINUED – return if a stopped child has resumed

- **pid arg:**
  - `pid =-1` – wait for any child process (like `wait`)
  - `pid >0` – wait for child whose PID=pid
  - `Pid =0` wait for any child whose process group ID = caller's
  - `pid <-1`: waits for any child whose process group ID equals to |pid|

wait vs. waitpid

wait(&status)⇔ waitpid(-1,&status,0)

- Differences
  - The **wait** function can block the caller until a child process terminates, whereas `waitpid` has an option that prevents it from blocking.
  - The **waitpid** function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.
- Advantage of `waitpid` over `wait`
  - **waitpid** allows parent process to wait for one particular process
  - **waitpid** has a nonblocking option
  - **waitpid** supports job control with the options: WUNTRACED and WCONTINUED

```
// figure 8.8 on page 225
#include "cs590.h"
#include <sys/wait.h>
int main(void){
   pid_t    pid;

   printf("my pid is %d\n", getpid());
   if ((pid = fork()) < 0) {
        perror("fork error");
    }
    else if (pid == 0)  /* first child */
    {
        printf("child 1, pid=%d, ppid=%d\n", getpid(), getppid());
        if ((pid = fork()) < 0)
             perror("fork error");
        else if (pid > 0){
          printf("parent of 2nd child, the 1st child: pid=%d exit\n", getpid());
           exit(0);
        }
        sleep(2);
        printf("child 2 pid=%d parent pid = %d\n", getpid(), getppid());
        exit(0);
    }
   // wait for the first child to finish
    if (waitpid(pid, NULL, 0) != pid)
       perror("waitpid error");

    printf("pid=%d is done\n", getpid());
    exit(0);
}
```

```
my pid is 12041
child 1, pid=12042, ppid=12041
parent of 2nd child, the 1st child: pid=12042 exit
pid=12041 is done
hlin@linux:~/cs590/chapter_8> child 2 pid=12043
parent pid = 1
```

Race conditions

- A race condition occurs when multiple processes are trying to do something with a shared resource, the final outcome depends on the order in which the processes run
- These conditions are resolved generally with signaling and synchronization mechanisms to be discussed in later chapters

```
#include "cs590.h"
static void pr_char(char *str){
    char    *ptr; int        c;
    setbuf(stdout, NULL) /* set unbuffered *
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);}
int main(void){
    pid_t    pid;
    if ((pid = fork()) < 0) {
        perror("fork error");
    } else if (pid == 0) {
        pr_char("output from child\n");
    } else {
        pr_char("output from parent\n");
    }
    exit(0);
}
```

Possible outputs:

output from parent
output from child

output foruotmp upta rfernotm
child

output from pareontu
tput from child

The result is undetermined, see the result on page 247

“exec” Functions

- There are 6 variation of exec functions
- exec functions allow a process to begin running a new program
- When a process calls one of the exec functions, the process is completely replaced by the new program, and **the PID is not changed**, **no new process created**

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[]);

Among these 6 functions

- l – list arguments individually, char* arg0, char* arg1, …, (char*) 0
- v – arguments are in a array, argv[]
- e – allow you to specify an environment pointer
- p – will search the path for the executable file based on PATH if filename contains no “/”

- All return -1 on error and no return for success

See page 249 (or man page) for properties inherited after exec

```
#include "cs590.h"  // exec.c
char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int main(void){
    pid_t    pid;
    if ((pid = fork()) < 0) {  perror("fork error");}
    else if (pid == 0) {    /* specify pathname, specify environment */
        if (execle("/home/csuser/hlin/cs590/echoall", "echoall",
              "myarg1","MY ARG2", (char *)0, env_init) < 0)
            perror("execle error");
    }
    if (waitpid(pid, NULL, 0) < 0)
        perror("wait error");

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

See the results on page 255

“system” Function

int system (const char* cmdstring);

System function provides an easy way to execute a command string from within a program

ex: `system(“date >file”);`

- System is implemented by calling **`fork`**, **`exec`**, and **`waitpid`**
- Return value:
  - -1 if fork fails or waitpid returns error other than EINTR
  - 127 if exec fails (i.e. shell can’t be executed)
  - Exit status: if all functions succeed, the value is the termination status of shell

//Figure 8.22 on page 266

```
int system(const char *cmdstring) {
    pid_t    pid;
    int      status;

    if (cmdstring == NULL)
        return(1);

    if ((pid = fork()) < 0) {
        status = -1;    /* probably out of processes */
    } else if (pid == 0) {                  /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
        _exit(127);         /* execl error */
    } else {                                /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) {
                status = -1; /* error other than EINTR from waitpid() */
                break;
            }
        }
    }
    return(status);
}
```

Identifying the race conditions (problems)

```
int main(void){
    int   id,  status;

    if ((pid=fork()) < 0) {
         perror("for error");
         exit (1);
    }
    if (pid == 0) {
         if ((pid=fork()) < 0){
             perror("for error");
             exit (1);
         }
         else if (pid == 0){
            printf("I am a child, my parent is %d\n", getppid());
         }
         else {
            return 0; // did not wait for its child
         }
    }
    else {
         printf("I am the parent, my pid is %d\n",    getpid());
         wait(&status);
    }
    return 0;
}
```