Unix System Programming

Processes

Overview

Last Week:

- How to program with directories
- Brief introduction to the UNIX file system

This Week:

- How to program UNIX processes (Chapters 7-9)
  - » Follow the flow of Ch 8. Process control sprinkled with reflections from Ch 7 (e.g., exit, process/program memory layout).
- `fork()` and `exec()`

Outline

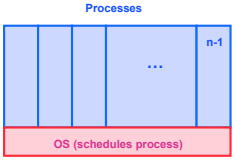
- What is a process?
- `fork()`
- `exec()`
- `wait()`
- Process Data
- Special Exit Cases
- Process Ids
- I/O Redirection
- User & Group ID real and effective (revisit)
- `getenv` & `putenv`
- `ulimit`

What is a Process?

- Review: A process is a program in execution (an active entity, i.e., it is a *running* program , this was also on the
  - » Basic unit of work on a computer
  - » Examples:
    - – compilation process,
    - – word processing process
    - – a.out process
    - – Shell process
    - – (we just need to make sure the program is running)

What is a Process?

- Each user can run many processes at once (e.g., by using &)
- A process:
  - » `cat file1 file2 &`
- Two processes started on the command line.
  - » `ls | wc -l`
- A time sharing system (such as UNIX) run several processes by *multiplexing* between them



What *is a* Process?

- It has both time, and space
  - » A container of instructions with some resources
- Process reads, and writes (or updates) machine resources
  - » e.g., CPU time (CPU carries out the instructions),
  - » memory,
  - » files,
  - » I/O devices (monitor, printer) to accomplish its task

Formal Process Definition

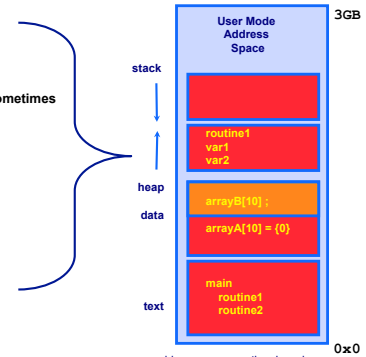
A process is a 'program in execution', a sequential execution characterized by trace. It has a context (the information or data) and this 'context' is maintained as the process progresses through the system.

What Makes up a Process?

`size`?

```
{nike:maria:27} size task2
   text    data     bss     dec     hex filename
   1040     484      16    1540     604 task2
```

- Program code (text)
  - » Compiled version of the text
- Data (cannot be shared)
  - » global variables
    - – Uninitialized (BSS segment) sometimes listed separately.
    - – Initialized
- Process stack (scopes)
  - » function parameters
  - » return addresses
  - » local variables and functions
- <<Shared Libraries >>
- Heap: Dynamic memory (alloc)
- OS Resources, environment
  - » open files, sockets
  - » Credential for security
- Registers
  - » program counter, stack pointer

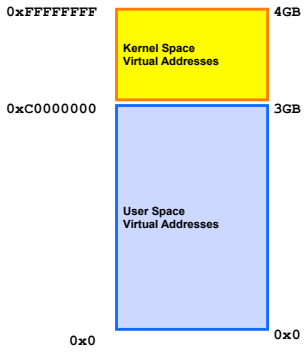


address space are the shared resources of a(ll) thread(s) in a program

Info about a process (running and foot print)

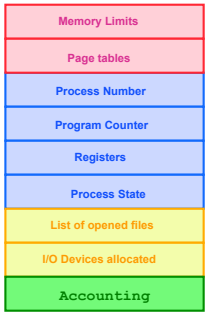
- `longsleepHelloW` (binary with long sleep)
- `size a.out.` (foot print)
- `ps`
  - » `61542 pts/5 00:00:00 longsleepHelloW`
- `cat /proc/61542/status`
- `cat /proc/61542/maps`
- ppp

- Example: Process with 4GB Virtual Address Space (32 bit architectures)
- User Space (focused on earlier, lower address space)
- Kernel Space



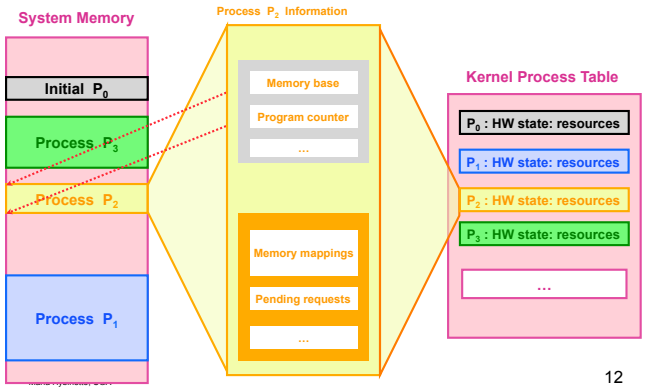
What is needed to keep track of a Process?

- Memory information:
  - » Pointer to memory segments needed to run a process, i.e., pointers to the address space -- text, data, stack segments.
- Process management information:
  - » Process state, ID
  - » Content of registers:
    - – Program counter, stack pointer, process state, priority, process ID, CPU time used
- File management & I/O information:
  - » Working directory, file descriptors open, I/O devices allocated
- Accounting: amount of CPU used.



Process control Block (PCB)

Process Representation



System Control: Process Attributes

`ps` and `top` command can be used to look at current processes

- PID - process ID: each process has a unique ID
- PPID - parent process ID: The process that "forked" to start the (child) process
- nice value - priority (-20 highest to 19 lowest)
- TTY associated with terminal (TTY teletype terminal)

OS View: Process Control Block (PCB)

- How does an OS keep track of the state of a process?
  - » Keep track of 'some information' in a structure.
    - – Example: In Linux a process' information is kept in a structure called `struct task_struct` declared in `#include linux/sched.h`
    - – What is in the structure?

```
struct task_struct
    pid_t pid;                  /* process identifier */
    long state;                 /* state for the process */
    unsigned int time_slice     /* scheduling information */
    struct mm_struct *mm        /* address space of this process */
```

    - – Where is it defined:
      - not in /usr/include/linux – only user level code
      - usr/src/kernel/2.6.32-431.29.2.elf6.x86_64/include/linux

Back to user-level

- Finding PIDs
  - » At the shell prompt
    - – ps u, ps, ps aux,
      - ps no args # your process
      - ps –ef # every process
      - ps -p 77851 # particular process
    - – top interative
  - » In a C program: int p = getpid(); // more later

Other Process Attributes

- Real user ID
- Effective user ID
- Current directory
- File descriptor table
- Environment
- Pointer to program code, data stack and heap
- Execution priority
- Signal information

3 General Process Types in UNIX

Interactive
- – foreground (shell must wait until complete [takes user input], or
- – background (&) [no user input]
- – initiated an controlled terminal session
- – can accept input form user as it runs and output to the terminal

Daemons
- – server processes running in the background (e.g., listening to a port)
- – Not associated with the terminal
- – typically started by `init` process at boot time
- – Examples: `ftpd`, `httpd`, …, mail
- – If user wants to creates one, detach it from the terminal, kill its parent. (init adopts)

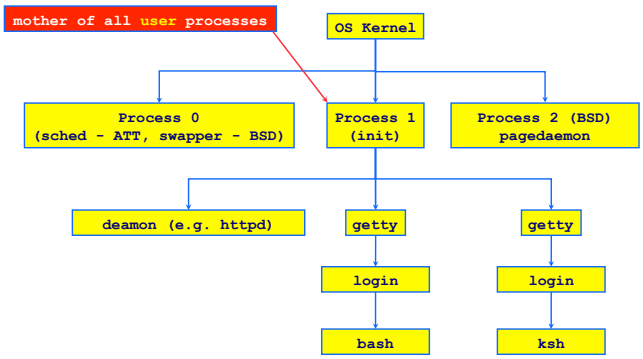
Batch (at, cron, batch)
- – Jobs that are queued and processed one after another
- – recurrent tasks scheduled to run from a queue
- – periodic, recurrent tasks run when system usage is low, cron-jobs (administered by the `daemon` crond).
- – Examples: backups, experimental runs.

» Zombies… don't count.

Process ID conventions, and the Process Life Cycle

- PID 0
  - » is usually the `scheduler` process (`swapper`), a system process (does not correspond to a program stored on disk, the grandmother of all processes).
- `init` - Mother of all user processes, init is started at boot time (at end of the boot strap procedure) and is responsible for starting other processes
  - » It is a user process with PID 1
  - » `init` uses file `inittab` and directory `/etc/rc?.d`
  - » brings the user to a certain specified state (e.g. multiuser)
- `getty` - login process that manages login sessions

Hierarchical Processes Tree on a (historical) UNIX System



Display Process Hierarchy

pstree (processes)

- **Syntax: pstree | more (all process)**
- **Syntax: pstree <PID>**
- **Syntax: pstree <username>**

tree (directory)

- **-d (directories), -a (hidden), -s (size), -p (permissions)**
- **tree –H .**

Daemon Processes

- **Print out status information of various processes in the system: `ps -axj` (BSD) , `ps -efjc` (SVR4) , switches / flags varies**
- **process status (ps)**
- **Daemons (d) run with root privileges, no controlling terminal, parent process is `init`**

```
{atlas:maria} ps -efjc | sort -k 2 -n | more  // solaris below
    UID   PID PPID  PGID SID  CLS PRI    STIME TTY  TIME CMD
   root     0    0     0   0  SYS  96   Mar 03 ?    0:01 sched
   root     1    0     0   0   TS  59   Mar 03 ?    1:13 /etc/init -r
   root     2    0     0   0  SYS  98   Mar 03 ?    0:00 pageout
   root     3    0     0   0  SYS  60   Mar 03 ? 4786:00 fsflush
   root    61    1    61  61   TS  59   Mar 03 ?    0:00 /usr/lib/sysevent/syseventd
   root    64    1    64  64   TS  59   Mar 03 ?    0:08 devfsadmd
   root    73    1    73  73   TS  59   Mar 03 ?   30:29 /usr/lib/picl/picld
   root   256    1   256 256   TS  59   Mar 03 ?    2:56 /usr/sbin/rpcbind
   root   259    1   259 259   TS  59   Mar 03 ?    2:05 /usr/sbin/keyserv
   root   284    1   284 284   TS  59   Mar 03 ?    0:38 /usr/sbin/inetd -s
 daemon   300    1   300 300   TS  59   Mar 03 ?    0:02 /usr/lib/nfs/statd
   root   302    1   302 302   TS  59   Mar 03 ?    0:05 /usr/lib/nfs/lockd
   root   308    1   308 308   TS  59   Mar 03 ?  377:42 /usr/lib/autofs/automountd
   root   319    1   319 319   TS  59   Mar 03 ?    6:33 /usr/sbin/syslogd
```

PID and Parentage

- **A process ID or PID is a positive integer that uniquely identifies a running process and is stored in a variable of type `pid_t`**
- **Example: print the process PID and parent's PID**

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(void)
   {
   pid_t pid, ppid;
   printf( "My PID is" %d\n", (pid = getpid()) );
   printf( "My PPID is" %d\n\n", (pid = getppid()) );
   }
```

```
{saffron} print-pid
My PID is 3891
MY PPID is 3794
```

```
PID COMMAND      %CPU   TIME
3891 print-ids    0.0%  0:00.00
3874 top         13.6%  0:19.71
3794 ksh          0.0%  0:00.04
```


- **pstree**
- **pidstat**
- **top, htop**
- **mpstat**
- **jobs**
  - » **^Z, ^C**
- **kill %1**

- **Linux processes**
- **ps -efjc | sort -k 2 -n | more**

```
 {nike:maria:125} ps -efjc | sort -k 2 -n | more  # linux Oct 2014 below
UID          PID  PPID  PGID   SID CLS PRI STIME TTY          TIME CMD
root           1     0     1     1 TS   19 Oct01 ?        00:02:11 /sbin/init
root           2     0     0     0 TS   19 Oct01 ?        00:00:04 [kthreadd]
root           3     2     0     0 FF  139 Oct01 ?        01:23:55 [migration/0]
root           4     2     0     0 TS   19 Oct01 ?        00:01:10 [ksoftirqd/0]
root           5     2     0     0 FF  139 Oct01 ?        00:00:00 [migration/0]
root           6     2     0     0 FF  139 Oct01 ?        00:07:19 [watchdog/0]
root           7     2     0     0 FF  139 Oct01 ?        01:14:54 [migration/1]
root           8     2     0     0 FF  139 Oct01 ?        00:00:00 [migration/1]
root           9     2     0     0 TS   19 Oct01 ?        00:00:32 [ksoftirqd/1]
root          10     2     0     0 FF  139 Oct01 ?        00:07:59 [watchdog/1]
root          11     2     0     0 FF  139 Oct01 ?        01:17:56 [migration/2]
root          12     2     0     0 FF  139 Oct01 ?        00:00:00 [migration/2]
root          13     2     0     0 TS   19 Oct01 ?        00:00:16 [ksoftirqd/2]
root          14     2     0     0 FF  139 Oct01 ?        00:07:17 [watchdog/2]
```

Linux Processes

- [] in ps (kernel processes)
  - » Example: [kthreadd]

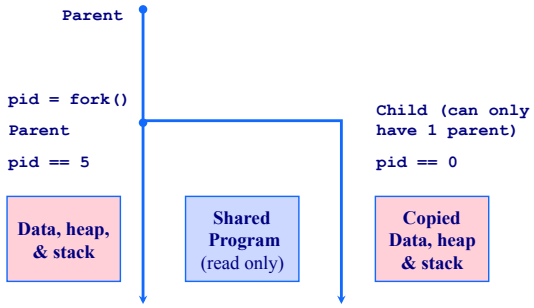
```
root         3  0.0  0.0      0     0 ?        S    Nov02   4:39 [ksoftirqd/0]
root         6  0.0  0.0      0     0 ?        S    Nov02   0:00 [migration/0]
root         7  0.0  0.0      0     0 ?        S    Nov02   0:01 [watchdog/0]
root         8  0.0  0.0      0     0 ?        S    Nov02   0:00 [migration/1]
```

- ksoftirqd – scheduling process kernel process (per CPU, soft interrupt handling.
- migration – migrates processes between CPUs
- Watchdog – checks that the system is running OK.

Process Life Cycle

- `Create, Run, Die`
- (Creation and Running) In the beginning:
  - » `init` and it's descendants creates all subsequent processes by a `fork()-exec()` mechanism
  - » `fork()` creates an exact copy of itself called a child process
  - » `exec()` system call places the image of a new program over the newly copied program of the parent
- (Die, Exit)
  - » When a process demises (completion of killed) it sends a signal to it's parent.

`fork()` a child



`fork()`

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

pid_t fork( void );
```

- Creates a child process by making a copy of the parent process
- Both the child *and* the parent continue running
- The return of `fork()`
  - » depends whether you are the child or the parent process:
    - `pid == 0` in the child process
    - `pid == <process ID of child>` in the parent process
- `pid` enables the programmer to define different actions for the parent and the child

Example: `parent-child.c`

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

int main()
  {
   int i;
   pid_t pid;
   pid = fork();
   if( pid > 0 )
        {               /* parent */
        for( i = 0; i < 1000; i++ )
                printf( "\tPARENT %d\n", i );
    }
   else
    {   /* child */
        for( i = 0; i < 1000; i++ )
        printf( "\t\tCHILD %d\n", i );
    }
  }
```

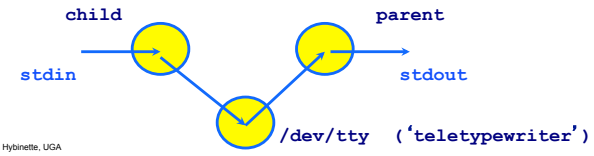
```
{saffron} parent-child
        PARENT 0
        PARENT 1
        PARENT 2
                CHILD 0
                CHILD 1
        PARENT 3
        PARENT 4
                CHILD 2
        .
        .
```


Things to Note

- `i` is copied between parent and child
- The switching between parent and child depends on many factors:
  - » Machine load, system process scheduling, …
- I/O buffering effects the output shown
  - » Output interleaving is *non-deterministic*
    - – Cannot determine output by looking at code

Example: `talk-to.c`

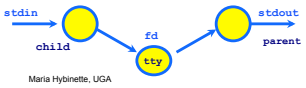
- A simple communications program :
  - » A "terminal"
  - » copies `chars` from `stdin` to a specified port and from that port to `stdout`
    - – `Read from stdin then write to port (copy)`
    - – `Read from port then write to stdout`
- Use port at `/dev/ttya` (terminal connected to standard input – a serial communication driver)



Example: `talk-to.c`

```
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFSIZE 10

int main(void )
  {
  int fd, count;
  char buffer[BUFSIZE];

  if( fd = open( "/dev/tty", O_RDWR ) < 0 )
   {
   fprintf( stderr, "Cannot open port\n" );
   exit(1);
   }
```

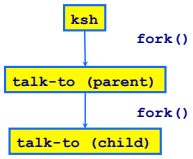
```
if( fork() > 0 )
  {    /* parent */
  while( 1 )
     {
     count = read( fd, buffer, BUFSIZ );
     write( 1, buffer, count ); /* stdout */
     }
  }
else   /* child */
  {
  while( 1 )
     {
     count = read( 0, buffer, BUFSIZ );
     write( fd, buffer, count );
     }
  }      /* else */
return 0;
} /* main */
```




```
{saffron} talk-to
hello this is maria
hello this is maria
^C
```


`ps` Output

```
{saffron} ps -l
UID      PID      PPID     COMMAND
501      3945      371     -ksh
501      3984     3945     talk-to
501      3985     3984     talk-to
.
.
.
```



Process Summary

- Process: a program in execution
  - » Time and Space entity
  - » System View : A set of data structures that changes over time.
    - – Entity that needs system resources (e.g., CPU & Memory, Files).
  - » Address Space : User / System
    - – Stack / Heap / Data (initialized, uninitialized) / Text
    - – Program pointer, Stack pointer
- Creation/Fork: Identical 'copy' of parent initially starting at next instruction after fork
  - » logical (separate) copy of parents address space
  - » separate stack and heap
  - » Caveats: Multi-threaded Processes, Lightweight Processes
    - – Shares 'more' (e.g., address space).

Replace Program: w/ `exec()`

```
Command line arguments: note argv0 is often = file
#include <unistd.h>
int execlp( char *file, char *argv0, char *argv1, … (char *) 0 );

execlp( "sort", "sort", "-n", "foobar", (char *) 0 );
```

same as "`sort -n foobar`"

- Family of functions for replacing a process's running program (text, data, heap and stack segment) with the one specified in the `exec()` call
- Process ID does not change across exec calls
  - » new process is not created, just it's context is replaced.
- The old program is obliterated by the new
  - » → no return back to the exec caller - unless there is an ERROR

Example: `tiny-menu.c`

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

int main()
  {
  char *cmd[] = { "who", "ls", "date" };
  int i;
  printf( "0 = who : 1 = ls : 2 = date" );
  scanf( "%d", &i );

  execlp( cmd[i], cmd[i], (char *) 0 );
  printf( "execlp failed\n" );
  }
```

```
{saffron:ingrid:40} tiny-menu
0 = who : 1 = ls : 2 = date
0
ingrid   console  Apr  4 10:58
{saffron:ingrid:41} tiny-menu
0 = who : 1 = ls : 2 = date
2
Fri Apr  8 16:56:47 EDT 2005
{saffron:ingrid:42}
```

`printf()` not executed unless there is a problem with `execlp()`

exec(…) family: execute a file (program)

- There are 6 versions of the exec function and they all basically do the same thing; they replace the current program with the text of the new program.
- Main difference is how the parameters are passed:
  - » Permutations:
    - – pathname/file (p) :
      - Program name searched for in current execution path (no p, must give full path name
    - – vector/list (v, l) :
      - Null terminated array of pointers to strings
      - L varargs mechanism
    - – environment (e)
      - Also accept Environmental variables.

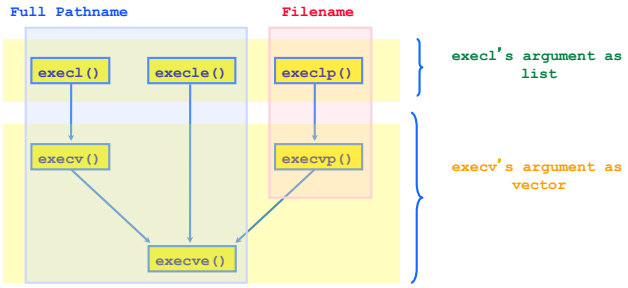
exec(…) family: execute a file (program)

- There are 6 versions of the exec function and they all basically do the same thing; they replace the current program with the text of the new program.
- Main difference is how the parameters are passed:

```
#include <unistd.h>

int execl( const char *path, const char *arg, ... argn,(char *)0 );
int execlp( const char *file, const char *arg, ... argn,(char *)0 );
int execle( const char *path, const char *arg, ... , argn,(char *)0
      char *const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *file, char *const argv [],
      char *const envp[] ); /* actual system call */
```

- Permutations: pathname/file : vector/list : environment

exec(…) Family Tree -



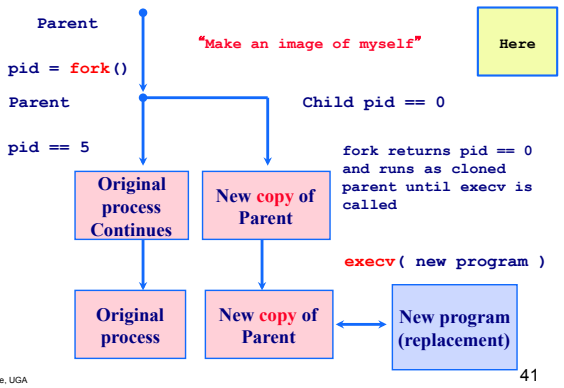
- Permutations: pathname/file : vector/list : environment
- System call: execve() -> all paths leads to this one
  - execve(const char *path, char *const argv[], char *const envp[]);

Summary

1. We created a process the unix way –
   - » Forking
2. We communicated
3. And we ran a file/program from a process
   - » Exec'd.

Combine these 3 things….

Want: fork() & execv()



Terminating processes

- Problem: Our original menu program only allowed a user to execute
  - » Only one command
  - » But now we are forking, couldn't we do more?
- Want:
  - » Would like child program to finish before continuing.
  - » (other instances) perhaps we would like to get result from child before continuing

WAIT

Process control: wait() & waipid()

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

pid_t wait( int *stat );
pid_t waitpid( pid_t pid, int *status, int options );
```

- Suspends calling process until child has finished.
- Returns the process ID of the terminated child if ok, -1 on error (check `errno` for error code)
- `status` can be `(int *)0` or a variable which will be bound to status information about the child when `wait` returns (e.g., exit-status of child passed through exit).
- `waitpid(-1, &status, 0); /* = wait() */`
- `options` : bitwise OR of any of the following options … (see man page)

wait() or waitpid() Actions

- Parent Suspend (block) if all of its children are still running, or
- Return immediately with the termination status of a child, or
- Return immediately with an error if there are no child processes
- Example …

wait() or waitpid() Example

- Example program: `menu-shell.c` illustrates `wait()` and includes:

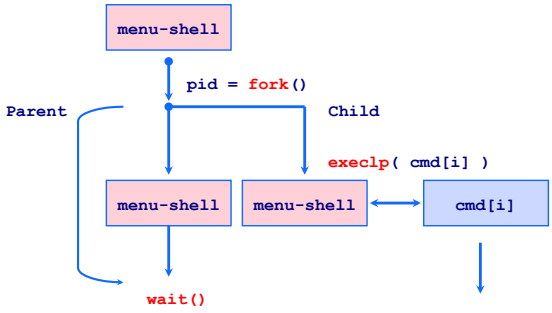
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```


Example: menu-shell.c

```
int main()
   {
   char *cmd[] = { "who", "ls", "date" };
   int i;
   pid_t pid;
   while( 1 )
         {
         printf( "0 = who : 1 = ls : 2 = date );
         scanf( "%d", &i );
         if( (pid = fork()) == 0 )
            {     /* child */
            execlp( cmd[i], cmd[i], (char *) 0 );
            perror( "execlp failed\n" );
            }
         else
            {     /* parent */
            printf( "waiting for child %d...", pid );
            wait( (int *) 0 );
            printf( "child %d finished\n", pid );
            }
   }
```

```
{saffron:ingrid:40} menu-shell
0
ingrid   console  Apr  4 10:58
waiting for child 4953...child 4953
finished
0 = who : 1 = ls : 2 = date
0
ingrid   console  Apr  4 10:58
waiting for child 4954...child 4954
finished
0 = who : 1 = ls : 2 = date
2
Fri Apr  8 19:05:39 EDT 2005
waiting for child 4955...child 4955
finished
0 = who : 1 = ls : 2 = date
```


menu-shell Execution



Macros for wait (1) samples the status

Enables checking on status of child after wait returns:

- `WIFEXITED(status)`
  - » Returns true if the child exited normally
  - » Checks 8 low order bits, i.e., the most significant eight bits.
  - » If macro is zero then child been stopped by another process via a signal.
- `WEXITSTATUS(status)`
  - » Details on exit status
  - » Evaluates to *the least significant eight bits (high order bits)* of the return code of the child which terminated, which may have been set as the argument to a call to `exit( )` or as the argument for a return.
  - » This macro can only be evaluated if `WIFEXITED` returned non-zero.

Macros for `wait` (2)

- `WIFSIGNALED(status)`
  - » Returns true if the child process exited *because of a signal* which was not caught.
- `WTERMSIG(status)`
  - » Returns *the signal number* that caused the child process to terminate.
  - » This macro can only be evaluated if `WIFSIGNALED` returned non-zero.

`waitpid()`: Particular Child

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

pid_t waitpid( pid_t pid, int *status, int opts )
```

- `waitpid()` waits for a particular child and does not necessarily need to block until a child terminates
- `pid > 0`
  - » Waits for the child whose ID is equal to pid
- `pid < -1`
  - » Waits for any child process whose process group ID is equal to the absolute value of *pid.*
- `pid == -1`
  - » Wait for any child process (same behavior as `wait()`)
- `pid == 0`
  - » Wait for any child process whose process group ID is equal to that of the calling process.

`waitpid()`

- `opts` : options when `pid > 0`
  - » Zero or more of the following constants can be `OR'ed:`
    - – `WNOHANG`
      - Return immediately if no child has exited.
    - – `WUNTRACED`
      - Also return for children which are stopped, and whose status has not been reported (because of a signal).
- Returns process ID of child which exits, -1 on error, 0 if `WNOHANG` was used and no child was available.

Macros for `waitpid()`

- `WIFSTOPPED(status)`
  - » Returns true if the child process which caused the return is currently stopped.
  - » This is only possible if the call was done using `WUNTRACED`.
- `WSTOPSIG(status)`
  - » Returns the signal number which caused the child to stop.
  - » This macro can only be evaluated if `WIFSTOPPED` returned non-zero.

Example: `waiting.c`

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

int main(void)
   {
   pid_t pid;
   int status;

   if( (pid = fork() ) == 0 )
         {        /* child */
         printf("I am a child with pid = %d\n", getpid() );
         sleep( 60 );
         printf( "child terminates\n" );
         exit(0);
         }
```


returned if child is stopped and not reported (signal)

```
else
   { /* parent */
   while (1)
         {
         waitpid( pid, &status, WUNTRACED );
         if( WIFSTOPPED(status) )
            {
            printf( "child stopped,
                  signal(%d)\n",
                  WSTOPSIG(status) );
            continue;
            }
         else if( WIFEXITED(status) )
            printf( "normal termination with
                  status(%d)\n",
                  WEXITSTATUS(status));
         else if (WIFSIGNALED(status))
            printf( "abnormal termination,
                  signal(%d)\n",
                  WTERMSIG(status));
         exit(0);
         } /* while */
   } /* parent */
} /* main */
```

```
{saffron:ingrid:54} waiting
waiting for child 985
child terminates
normal termination with status(0)
```

```
{saffron:ingrid:54} waiting
waiting for child 5022
child stopped, signal(17)
waiting for child 5022
child terminates
normal termination with status(0)
```

```
{saffron:ingrid:40} kill -l
.
.
{saffron:ingrid:48} kill -STOP 5022
{saffron:ingrid:49} kill -CONT 5022
```

```
{saffron:ingrid:55} waiting
waiting for child 5024
abnormal termination,   signal(15)
{saffron:ingrid:56}
```

```
{saffron:ingrid:56} kill -TERM 5024
```

Special Exit Cases

- A child exits when its parent is not currently executing `wait()`
  - » the child becomes a *zombie*
  - » `status` data about the child is stored until the parent does a `wait()`
  - » Zombie: Terminated process that has not YET been cleaned up. Parents are responsible to clean up after their children. Possible parent has not YET called wait.
- A parent exits when 1 or more children are still running
  - » children are adopted by the system's init process (`/etc/init`)
    - – it can then monitor/kill them
    - – when the adopted child terminates however it does not become a zombie, because `init` automatically calls wait when the child finally terminates

Zombies

- Terminated child process, but still around, waiting for its parent : to wait() and do the cleanup.
- Still take up system resources, memory, and it will never be schedule since it is 'terminated'
- Problem: when there are lots of zombies, one by itself not bad, but a crowd can be a problem

http://en.wikipedia.org/wiki/Zombie_(fictional)

make-zombie.c

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

int main ()
  {
  pid_t child_pid;

  /* Create a child process. */
  child_pid = fork ();
  if( child_pid > 0 )
  {
        /* This is the parent process. Sleep for a minute. */
        sleep (60);
  }
  else
  {
        /* This is the child process. Exit immediately. */
        exit (0);
  }

  return 0;
}
```

```
{nike:maria:41}  ps -e -o pid,ppid,stat,cmd  | grep zom
76745 76624 S+   make-zombie
76746 76745 Z+   [make-zombie] <defunct>
```

- `ps -e -o pid,ppid,stat,cmd | grep zom`
- Child is marked as defunct
  - » Terminated child that has not yet been clean up!
- Parents exits without calling wait,
  - » Zombie child is adopted by init, and now init will clean up after the unclean parent!

Process Data

- Recall a process is a copy of the parent, it has a copy of the parent's data.
- A change to a variable in the child will *not* change that variable in the parent.

Example: global-example.c

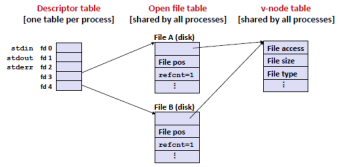
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int globvar = 6;
char buf[] = "stdout write\n";

int main(void)
{
int w = 88;
pid_t pid;

write( 1, buf, sizeof(buf)-1 );
printf( "Before fork()\n" );
if( (pid = fork()) == 0 )
   {  /* child */
   globvar++;
   w++;
   }
else if( pid > 0 )  /* parent */
   sleep(2);
else
   perror( "fork error" );

printf( "pid = %d, globvar = %d, w = %d\n",
   getpid(), globvar, w );
return 0;
} /* end main */
```

```
{saffron:ingrid:62} global-example
stdout write
Before fork()
pid = 5039, globvar = 7, w = 89
pid = 5038, globvar = 6, w = 88
{saffron:ingrid:63}
```

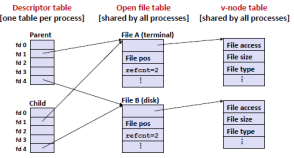
Caveat: Process File Descriptors

- While child and parent have (separate) copies of the file descriptors they share system file table entries.
  - » Effect is that the R-W pointer is shared
- This means that a `read()` or `write()` in one process will affect the other process since the R-W pointer is changed.

Before and after fork()



- Un-related processes



- Related processes

Example: `file-ptr.c`

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

void printpos( char *msg, int fd ) /* Print position in file */
  {
  long int pos;
  if( (pos = lseek( fd, 0L, SEEK_CUR) ) < 0L )
       perror("lseek");
  printf( "%s: %ld\n", msg, pos );
  }
```



```
int main( void )
  {
  int fd;               /* file descriptor */
  pid_t pid;
  char buf[10];         /* for file data */

  if( (fd = open( "file-ptr.txt", O_RDONLY )) < 0 )
        perror("open");
  read( fd, buf, 10 );  /* move R-W ptr */
  printpos( "Before fork", fd );
  if( (pid = fork()) == 0 )
        {       /* child */
        printpos( "Child before read", fd );
        read( fd, buf, 10 );
        printpos( "Child after read", fd );
        }
  else if ( pid > 0 )
        {       /* parent */
        wait( (int *) 0 );
        printpos( "Parent after wait", fd );
        }
  else  {
        perror( "fork" );
        }
  }
```

```
{saffron} cat
fileptr.txt
hello
this is
the data file
```

```
{saffron} shared-file
Before fork: 10
Child before read: 10
Child after read: 14
Parent after wait: 14
```

what's happened?

I/O redirection: `ls > lis.txt`

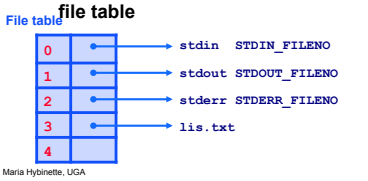
(1) Open create file – write mode –

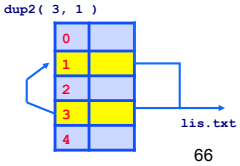
(2) How do we get stdout of ls to go to the file?

- The trick: you can change where the standard I/O streams are going/coming from after the `fork` but before the `exec`

I/O redirection

- Example implementation shell:
  - » `{saffron} ls > lis.txt`
  - » open a new file `lis.txt`
  - » Redirect standard output to `lis.txt` using `dup2`
    - – Everything that is sent to standard output is also sent to `lis.txt`
  - » Execute `ls` in the process
- `dup2( int fin, int fout )` - copies `fin` to `fout` in file table





Example: `ls > lis.txt`

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

int main( void )
  {
  int fileId;
  int int_stdout;

  fileId = creat( "lis.txt", 0640 );

  if( fileId < 0 )
        {
        fprintf( stderr, "error creating lis.txt\n" );
        exit (1);
        }

  dup2( fileId, STDOUT_FILENO ); /* copy fileID to stdout */
  close( fileId );
  execl( "/bin/ls", "ls", 0 );
  }
```

```
{saffron:6} ls
lis*    lis.c
{saffron:7} lis
{saffron:8} ls
lis*  lis.c  lis.txt
{saffron:9} cat lis.txt
lis
lis.c
lis.txt
```


User and Group ID (revisit)

- Group ID: Real and effective
- User ID
  - » Real user ID
    - – Identifies the user who is responsible for the running process
  - » Effective user ID
    - – Used to assign ownership of newly created files, to check file access permissions and to check permission to send signals to processes
    - – To change `euid`: execute `setuid-program` that has the `set-uid` bit set or invodes the `setuid()` system call
    - – The `setuid( uid )` system call, if `euid` is not superuser, `uid` must be the real uid or saved uid (the kernel also resets `euid` to `uid`)
  - » Real and effective uid: inherit (`fork`), maintain (`exec`)

Read IDs

- `pid_t getuid( void );`
  - » Returns the real user ID of the current process
- `pid_t geteuid( void );`
  - » Returns the effective user ID of the current process
- `gid_t getgid( void );`
  - » Returns the real group ID of the current process
- `gid_t getegid( void );`
  - » Returns the effective group ID of the current process

Change UID and GID (1)

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

int setuid( uid_t uid )
int setgid( gid_t gid )
```

- Sets the effective user ID of the current process.
- Superuser process resets the real effective user IDs to `uid`.
- Non-superuser process can set effective user ID to `uid`, only when `uid` equals real user ID or the saved set-user ID (set by executing a `setuid-program` in `exec`).
- In any other cases, `setuid` returns error.

Change UID and GID (2)

ID	exec		setuid(uid)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real-uid	unchanged	unchanged	set to uid	unchanged
effective user ID	unchanged	set from user ID of program file	set to uid	set to uid
saved set-uid	copied from euid	copied from euid	uid	unchanged

- Different ways to change the three user IDs (pg 214)

Change UID and GID (3)

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

int setreuid( uid_t ruid, uid_t euid )
```

- Sets real and effective user ID's of the current process
- Un-privileged users may change the real user ID to the effective user ID and vice-versa.
- It is also possible to set the effective user ID from the saved user ID.
- Supplying a value of -1 for either the real or effective user ID forces the system to leave that ID unchanged.
- If the real user ID is changed or the effective user ID is set to a value not equal to the previous real user ID, the saved user ID will be set to the new effective user ID.

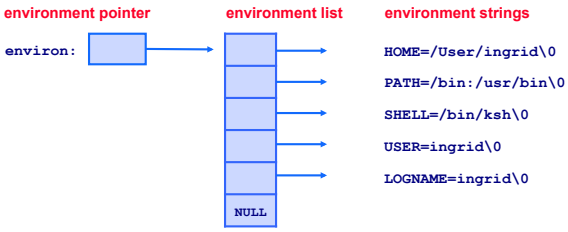
Change UID and GID (4)

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

int seteuid( uid_t uid );

int setregid( gid_t rgid, gid_t egid )
int setegid( gid_t gid );
```

- Functionally equivalent to setreuid(-1, euid)
- Setuid-root program wishing to temporarily drop root privileges, assume the identity of a non-root user, and then regain root privileges afterwards cannot use setuid, because setuid issued by the superuser changes all three IDs. One can accomplish this with seteuid.

Environment

```
extern char **environ;
int main( int argc, char *argv[], char *envp[] )
```



Example: `environ.c`

```
#include <stdio.h>

int main( int argc,
    char *argv[],
    char *envp[] )
  {
  int i;
  extern char **environ;

  printf( "**----> from argument envp\n" );
  for( i = 0; envp[i]; i++ )
        puts( envp[i] );

  printf( "\n**----> from global environ\n" );
  for( i = 0; environ[i]; i++ )
        puts( environ[i] );
  }
```

```
{saffron} environ
**----> from argument envp
_=environ
PAGER=/usr/bin/more
PATH=/usr/local/bin:/lib:/
sw:/sw/bin:/sbin:/usr/sbin:/
usr/games::/usr/games:/usr/
local/jdk.lat
est/bin:/Users/ingrid/bin:/
Users/ingrid/usr/bin
SHELL=ksh
TERM_PROGRAM_VERSION=100.1.4
HOSTNAME=saffron
USER=ingrid
.
.
**----> from global environ
_=environ
PAGER=/usr/bin/more
.
.
```


`getenv`

```
#include <stdlib.h>
char *getenv( const char *name )
```

- Searches the environement list for a string that matches the string pointed by `name`
- Returns a ointer to the value in the environment, or `NULL` if there is no match

`putenv`

```
#include <stdlib.h>
int putenv( const char *string )
```

- Adds or changes the values of environment variables
- The argument `string` is of the form "`name = value`"
- If the name does not already exist in the environment then `string` is added to the environment
- If `name` does exist then the value of `name` in the environment is changed to `value`
- Returns 0 on successs and -1 if an error occurs

Example: `getputenv.c`

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

int main( void )
  {
  printf( "Home directory is %s\n",
        getenv( "HOME" ) );
  putenv( "HOME=/" );
  printf( "New home directory is %s\n",
        getenv( "HOME" ) );
  }
```

```
{saffron:ingrid:95} getputenv
Home directory is /Users/ingrid
New home directory is /
```