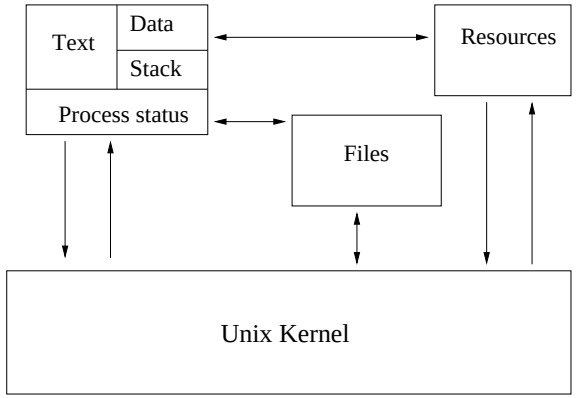
Objectives

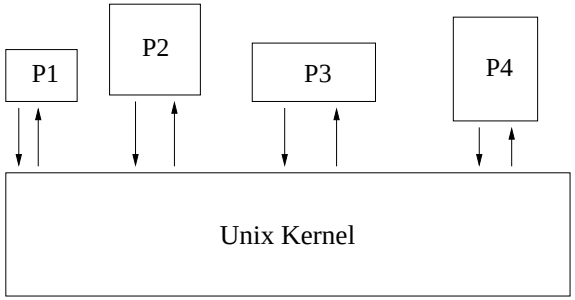
- Ideas and Skills
  - What a Unix shell does
  - The Unix model of a process
  - How to run a program
  - How to create a process
  - How parent and child process communicate
- System calls and functions
  - Fork
  - exec
  - wait
  - exit
- Commands
  - sh
  - ps

What is a process

- A process is the memory space and settings with which a program runs.



(a Unix process)



(Processes in User space)

- Processes live in user space, which holds data that can be swapped out of the memory at any time.

Some Attributes of a process

- UIDs, GIDs, Process ID
- scheduling parameters
  - SCHED_FIFO: A First-In, First-Out real-time process.
  - SCHED_RR: A Round Robin real-time process.
  - SCHED_OTHER: A conventional, time-shared process.
- limits - per-process resource limits
- supplemental groups - a list of groups (GIDs) in which this user has membership.
- timers, resources, pipes
- signals, threads, semaphores

List the current processes: ps

To list longer, more informative lines about processes, use **-l** option:

```
bash-2.02$ ps -la
 F S   UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN TTY      TIME CMD
 8 O  1807 24492 24454  0  40 20 7873fc70    124          pts/7    0:00 ps
 8 S   159 19812 19800  0  40 20 78737598   1023 7a70c93a pts/6    0:00 pine-4.3
 8 S  1605 23353 16960  0  40 20 73eed7d8   1239 79a64e72 pts/4    0:02 pine-4.3
 8 S   289 19515 19483  0  40 20 72b28020   1742 74287582 pts/1    0:08 pine4.64
 8 S   289 21530 21498  0  40 20 76cdd800   2283 7a6ae6f2 pts/5    0:40 pine4.64
 8 S  1807 24454 24431  0  40 20 73ef49f8    284 73ef4a64 pts/7    0:00 bash-2.0
bash-2.02$
```

Process and computer memory

Kernel manages processes in a similar way as it manages files.

- The memory is divided into two areas: Kernel space and user space. Processes are in the user space
- The kernel allocates memory for a process in pages. The memory occupied by a process does not have to be continuous.

Unix Shell: Tool for process and program control

All popular shells provide three main functions

- Shells run programs
- Shells manage input and output, pipes (|), I/O redirection ($\langle$, $\rangle$)
- Shells can be programmed. Shell is a high level programming language. Several links to Shell programming tutorial:

  `http://steve-parker.org/sh/sh.shtml`
  `http://www.freeos.com/guides/lsst/`

Unix Shell: A sample shell program

The following script converts all png files under a directory to tga files.

```
FILENAME="buffer"
START_INDEX=$1
NUM_SLICES=‘ls *.png |wc -l‘
COUNT=0
TEN=10
while [ $COUNT -lt $NUM_SLICES ]
do
let CURRENT_INDEX=$START_INDEX+$COUNT
if [ $CURRENT_INDEX -lt $TEN ]; then
        convert "$FILENAME"0"$CURRENT_INDEX".png "$FILENAME"0"$CURRENT_INDEX".tga
else
        convert "$FILENAME""$CURRENT_INDEX".png "$FILENAME""$CURRENT_INDEX".tga
fi
let COUNT=$COUNT+1;
done
```

How the Shell run programs

A shell follows four steps in executing a program

- Get program name and arguments from the user
- Creates a new process to run the program
- Load the program from the disk into the process
- Wait for the program to finish.

To write a shell, we should find out how to run a program, create a process, and wait for the exit of the program.

How does a program run a program: execvp

`execvp(progname, arglist)`

- copies the named program into the calling process
- passes the specified list of strings to the program as argv[]
- run the program

Sample code exec1.c

```
/* exec1.c - shows how easy it is for a program to run a program
 */
main()
{
        char    *arglist[3];

        arglist[0] = "ls";
        arglist[1] = "-l";
        arglist[2] = 0 ;
        printf("* * * About to exec ls -l\n");
        execvp( "ls" , arglist );
        printf("* * * ls is done. bye\n");
}
```

manpage of execvp

```
SYNOPSIS
       #include <unistd.h>

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

DESCRIPTION
       The  exec()  family  of  functions replaces the current
       process image with a new process image.
RETURN VALUE
       If any of the exec() functions returns, an error will have
       occurred.  The return value is -1, and  the  global
       variable errno will be set to indicate the error.
```

Run another program within a program

Open and run exec1.c

```
 /* exec1.c - shows how easy it is for a program to run a program
 */

main()
{
        char    *arglist[3];

        arglist[0] = "ls";
        arglist[1] = "-l";
        arglist[2] = 0 ;
        printf("* * * About to exec ls -l\n");
        execvp( "ls" , arglist );
        printf("* * * ls is done. bye\n");
}
```

A prompting shell

Open and run psh1.c

how to create a new process–fork()

```
NAME
       fork - create a child process

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

       pid_t fork(void);

DESCRIPTION
       fork  creates a child process that differs from the parent process only
       in its PID and PPID, and in the fact that resource utilizations are set
       to 0.  File locks and pending signals are not inherited.

RETURN VALUE
       On success, the PID of the child process is returned  in  the  parents
       thread  of execution, and a 0 is returned in the childs thread of exe-
       cution.  On failure, a -1 will be returned in the parents context,  no
       child process will be created, and errno will be set appropriately.
```

What the kernel do in fork system call

- Allocates a new chunk of memory and kernel data structures
- Copies the original process into the new process
- Adds the new process to the set of running processes
- Returns control back to both process

Different return values in the parent and child processes

```
/*  forkdemo1.c
 *      shows how fork creates two processes, distinguishable
 *      by the different return values from fork()
 */

#include        <stdio.h>

main()
{
        int     ret_from_fork, mypid;

        mypid = getpid();                           /* who am i?         */
        printf("Before: my pid is %d\n", mypid);    /* tell the world    */

        ret_from_fork = fork();

        sleep(1);
        printf("After: my pid is %d, fork() said %d\n",
                        getpid(), ret_from_fork);
}
```

Where child process starts running

The sample program is in forkdemo2.c

```
/* forkdemo2.c - shows how child processes pick up at the return
 *               from fork() and can execute any code they like,
 *               even fork().  Predict number of lines of output.
 */

main()
{
        printf("my pid is %d\n", getpid() );
        fork();
        fork();
        fork();
        printf("my pid is %d\n", getpid() );
}
```

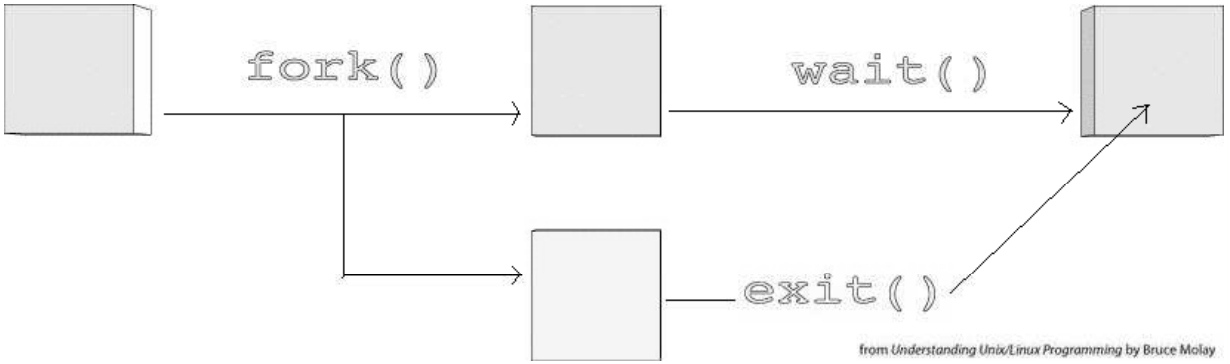
Distinguishing parent from child

The sample program is in forkdemo3.c

```
/*  forkdemo3.c - shows how the return value from fork()
 *                allows a process to determine whether
 *                it is a child or process
 */
#include        <stdio.h>
main()
{
        int     fork_rv;

        printf("Before: my pid is %d\n", getpid());

        fork_rv = fork();               /* create new process   */

        if ( fork_rv == -1 )            /* check for error      */
                perror("fork");

        else if ( fork_rv == 0 )
                printf("I am the child.  my pid=%d\n", getpid());
        else
                printf("I am the parent. my child is %d\n", fork_rv);
}
```

How does the parent wait for the child to exit: wait()

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

       pid_t wait(int *status);
DESCRIPTION
       The wait function suspends execution of the current process until a
       child has exited, or until a signal is delivered whose action is to
       terminate the current process or to call a signal handling function.
       If a child has already exited by the time of the call (a so-called
       "zombie" process), the function returns immediately. Any system
       resources used by the child are freed.
```



A demo of notification to wait

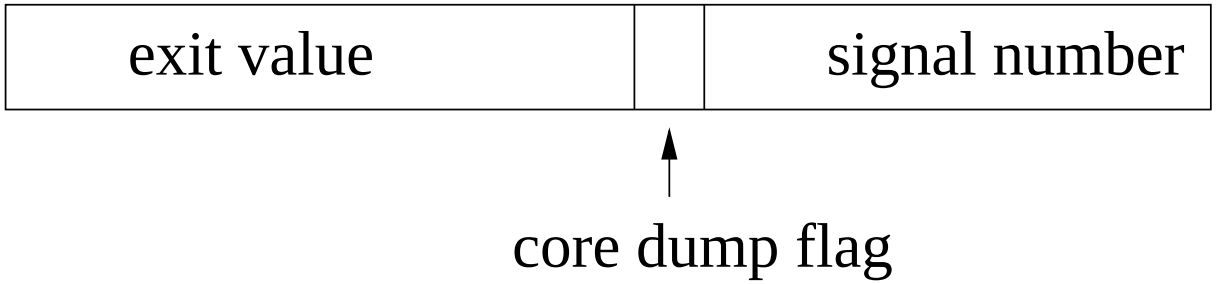
```
/* waitdemo1.c - shows how parent pauses until child finishes
 */

#include        <stdio.h>

#define DELAY   2

main()
{
        int  newpid;
        void child_code(), parent_code();

        printf("before: mypid is %d\n", getpid());

        if ( (newpid = fork()) == -1 )
                perror("fork");
        else if ( newpid == 0 )
                child_code(DELAY);
        else
                parent_code(newpid);
}
/*
 * new process takes a nap and then exits
```

```
 */
void child_code(int delay)
{
        printf("child %d here. will sleep for %d seconds\n", getpid(), delay);
        sleep(delay);
        printf("child done. about to exit\n");
        exit(17);
}
/*
 * parent waits for child then prints a message
 */
void parent_code(int childpid)
{
        int wait_rv;            /* return value from wait() */
        wait_rv = wait(NULL);
        printf("done waiting for %d. Wait returned: %d\n", childpid, wait_rv);
}
```

Communication via wait

The integer argument in the wait system call records the information set by the kernel.

- The first eight bits of the integer saves the exit number of the child process
- The seventh bit indicates a core dump
- The last seven bits saves the signal number

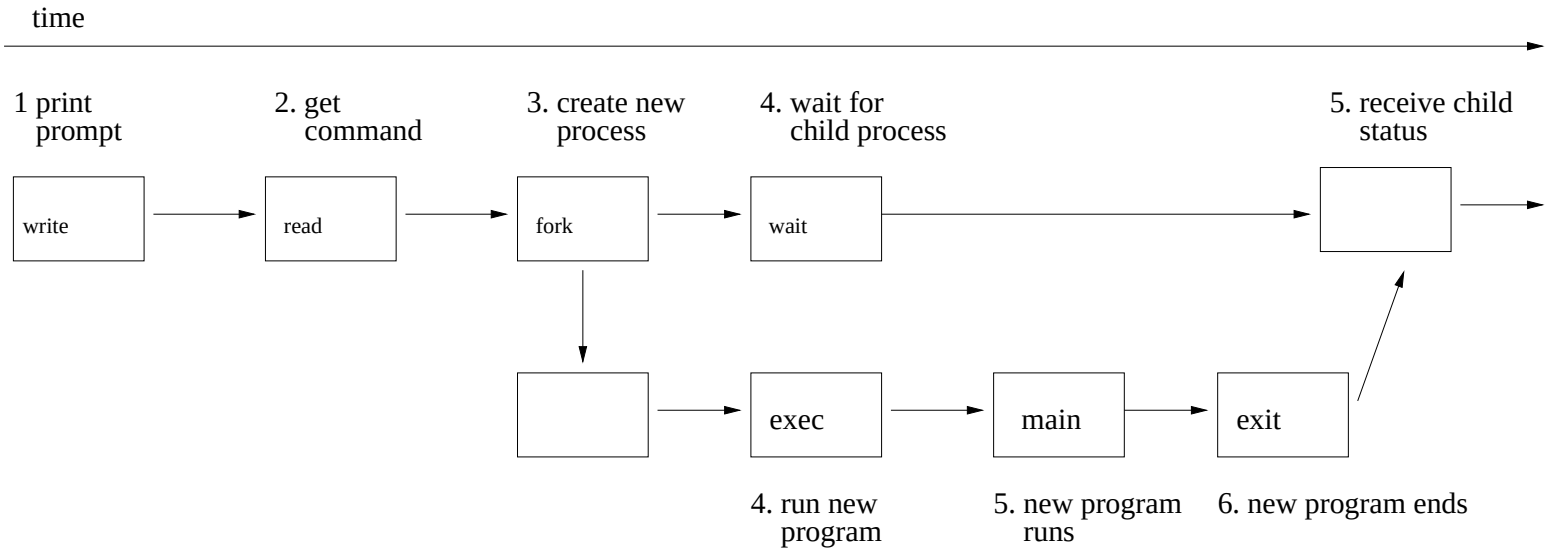


Communication via wait–demo

```
/* waitdemo2.c - shows how parent gets child status
 */
#include        <stdio.h>

#define DELAY   5

main()
{
        int  newpid;
        void child_code(), parent_code();

        printf("before: mypid is %d\n", getpid());

        if ( (newpid = fork()) == -1 )
                perror("fork");
        else if ( newpid == 0 )
                child_code(DELAY);
        else
                parent_code(newpid);
}
/*
 * new process takes a nap and then exits
 */
```

```
void child_code(int delay)
{
        printf("child %d here. will sleep for %d seconds\n", getpid(), delay);
        sleep(delay);
        printf("child done. about to exit\n");
        exit(17);
}
/*
 * parent waits for child then prints a message
 */
void parent_code(int childpid)
{
        int wait_rv;            /* return value from wait() */
        int child_status;
        int high_8, low_7, bit_7;

        wait_rv = wait(&child_status);
        printf("done waiting for %d. Wait returned: %d\n", childpid, wait_rv);

        high_8 = child_status >> 8;     /* 1111 1111 0000 0000 */
        low_7  = child_status & 0x7F;   /* 0000 0000 0111 1111 */
        bit_7  = child_status & 0x80;   /* 0000 0000 1000 0000 */
        printf("status: exit=%d, sig=%d, core=%d\n", high_8, low_7, bit_7);
}
```

How the shell runs programs



Shell loop with fork(), exec(), wait()

Writing a shell: psh2.c

```
/**  prompting shell version 2
 **
 **             Solves the 'one-shot' problem of version 1
 **                     Uses execvp(), but fork()s first so that the
 **                     shell waits around to perform another command
 **             New problem: shell catches signals.  Run vi, press ^c.
 **/
#include        <stdio.h>
#include        <signal.h>
#define MAXARGS         20                              /* cmdline args */
#define ARGLEN          100                             /* token length */
main()
{
        char    *arglist[MAXARGS+1];            /* an array of ptrs     */
        int     numargs;                        /* index into array     */
        char    argbuf[ARGLEN];                 /* read stuff here      */
        char    *makestring();                  /* malloc etc           */

        numargs = 0;
        while ( numargs < MAXARGS ) {
                printf("Arg[%d]? ", numargs);
                if ( fgets(argbuf, ARGLEN, stdin) && *argbuf != '\n' )
                        arglist[numargs++] = makestring(argbuf);
```

```
                else {
                        if ( numargs > 0 ){             /* any args?    */
                                arglist[numargs]=NULL;  /* close list   */
                                execute( arglist );     /* do it        */
                                numargs = 0;            /* and reset    */
                        }
                }
        }
        return 0;
}

execute( char *arglist[] )
/* use fork and execvp and wait to do it  */
{
        int     pid,exitstatus;                         /* of child     */

        pid = fork();                                   /* make new process */
        switch( pid ){
                case -1:
                        perror("fork failed");
                        exit(1);
                case 0:
                        execvp(arglist[0], arglist);            /* do it */
                        perror("execvp failed");
```

```
                        exit(1);
                default:
                        while( wait(&exitstatus) != pid )
                                ;
                        printf("child exited with status %d,%d\n",
                                        exitstatus>>8, exitstatus&0377);
        }
}
char *makestring( char *buf )
/* trim off newline and create storage for the string */
{
        char    *cp, *malloc();

        buf[strlen(buf)-1] = '\0';              /* trim newline */
        cp = malloc( strlen(buf)+1 );           /* get memory   */
        if ( cp == NULL ){                      /* or die       */
                fprintf(stderr,"no memory\n");
                exit(1);
        }
        strcpy(cp, buf);                /* copy chars   */
        return cp;                      /* return ptr   */
}
```

Death of a process: exit and _exit

- **exit**
  - flushes all streams
  - calls functions that have been registered with atexit and on_exit, and perform any other functions associated with exit for the current system
  - calls the system call _exit.
- **_exit**
  - terminates the calling process "immediately".
  - Closes any open file descriptors belonging to the process
  - Change the parent PID of all its children to the PID of process 1, init
  - Sends the processs parent a SIGCHLD signal.

The exec family

The following is a list of the functions in this family

```
#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[]);
```

The execve system call

```
#include <unistd.h>

int execve(const char *filename, char *const argv [], char *const envp[]);
```