System Programming (ELEC462)

Processes and Programs

Dukyun Nam

HPC Lab@KNU

Contents

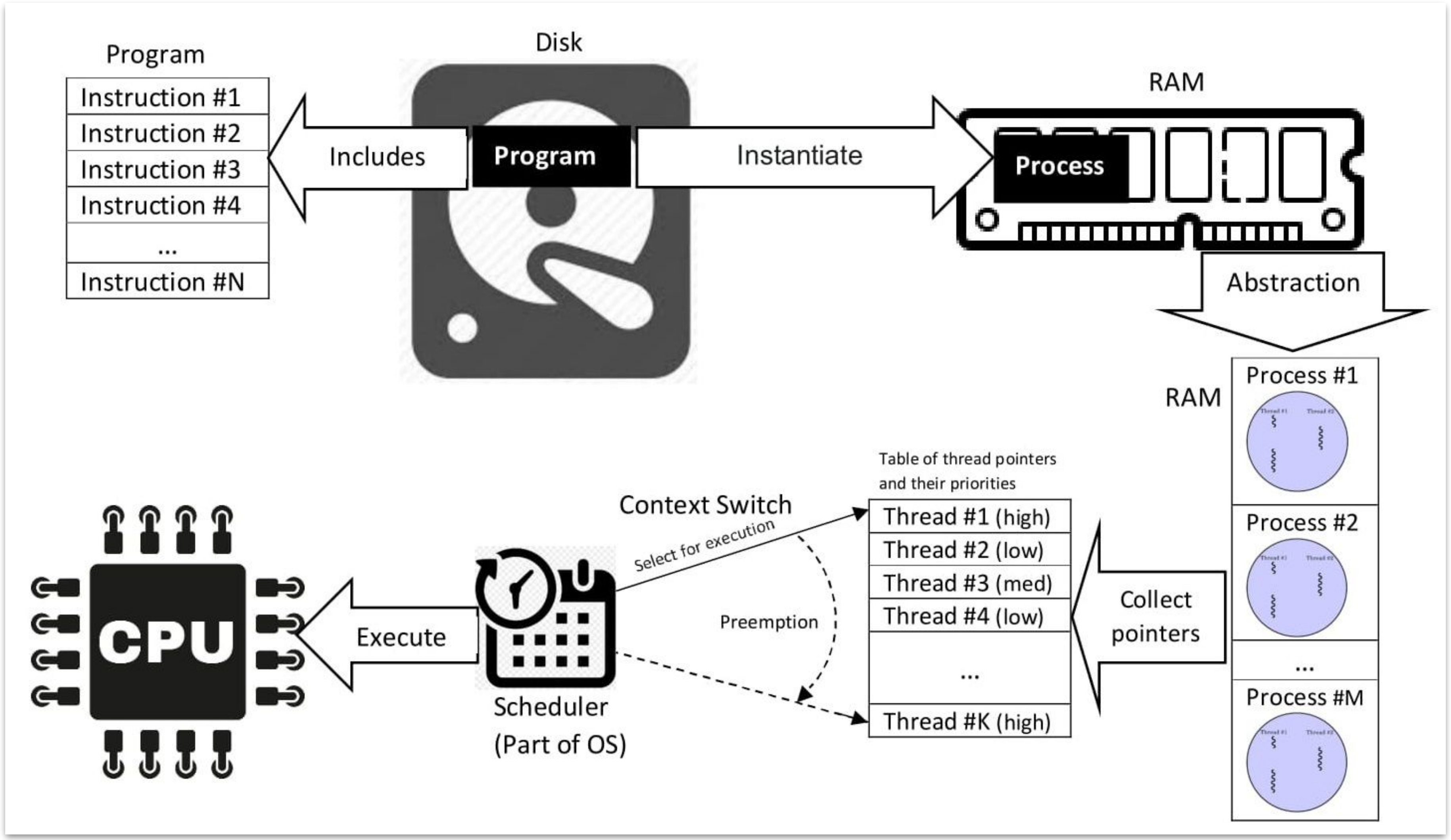
Introduction

- Ideas and Skills
  - What a Linux shell does
  - The Linux model of a process
  - How to run a program
  - How to create a process
  - How to communicate between parent and child processes
- System Calls and Functions
  - `fork`, `exec`, `wait`, and `exit`
- Commands
  - `sh`
  - `ps`

Processes = Programs in Action

- Program (or executable)?
  - A sequence of machine instructions stored in a file
    - Produced by compiling source code into binary code
- What’s the meaning of “running a program”?
  - Loading a list of machine instructions into memory
  - Having CPU execute each of the instructions in sequence (or parallel)
- Executable program vs. Process (in Unix/Linux)
  - Executable program: a list of machine language instructions and data
  - Process: the memory space and dynamic settings with which the program runs

Processes = Programs in Action (cont.)

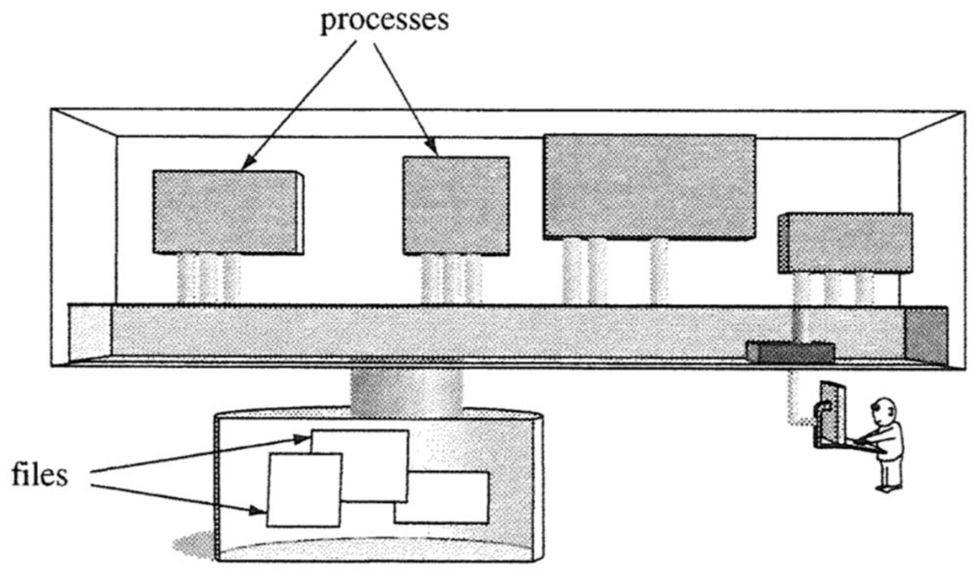


< Program vs. Process vs. Thread Scheduling, Preemption, Context Switching (from wikipedia) >

Programs in Action, or Processes

- Data + Processing
  - Data and programs are STORED

    in files on **<u>disk</u>**
  - Programs are RUN in **<u>processes</u>**
- A program is RUN (executed) by
  - 1. Kernel copies the code into memory
  - 2. CPU executes the instructions
- How can we view what processes are running?

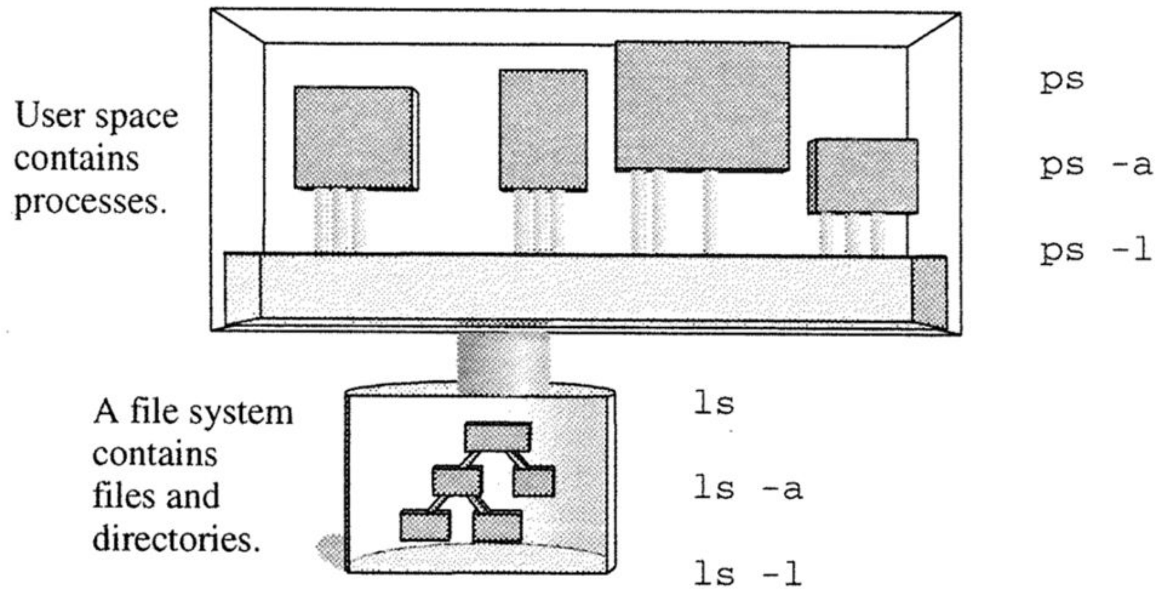


< Processes are programs in action >

Exploring Darkest, Deep User Space using `ps`

- `ls`
  - just as the `ls` command
    lists the objects and attributes
    in the file system

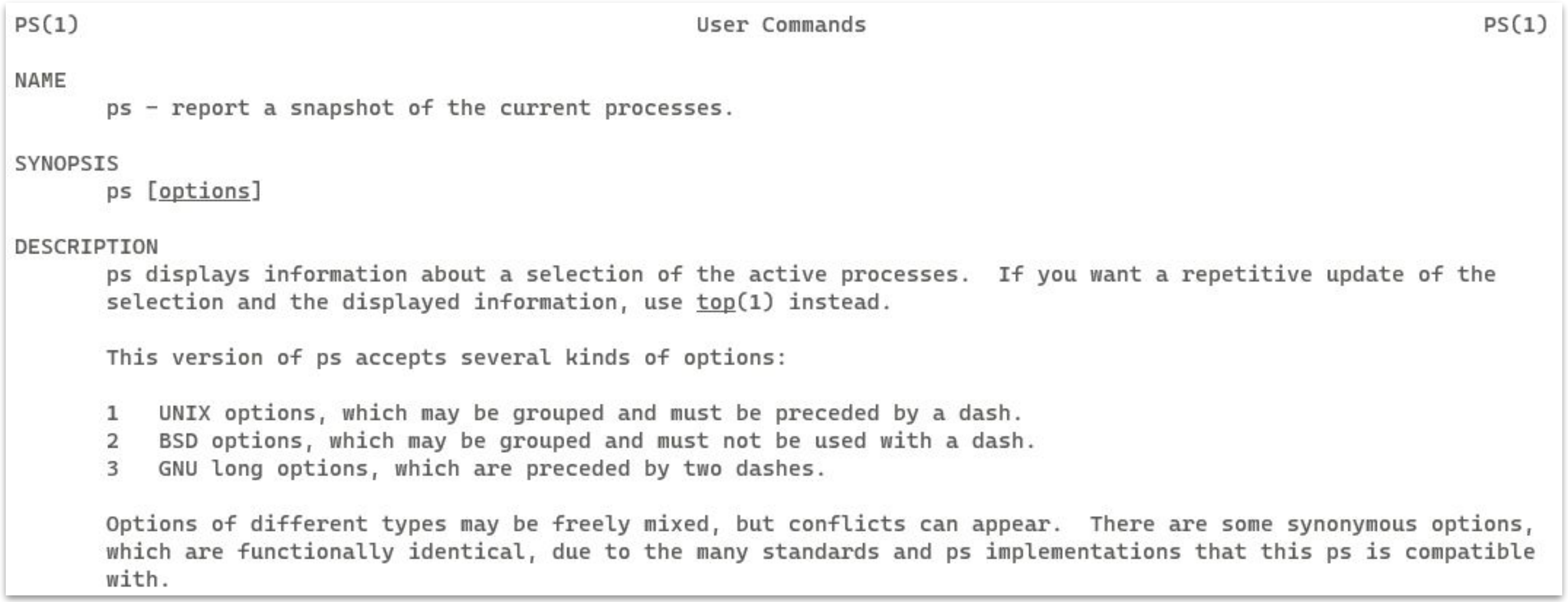
- `ps`
  - the `ps` command
    lists the objects and their properties in the process table
  - Attributes of a process
    - user, tty, addr, status, etc.


User space contains processes.
ps
ps -a
ps -l
A file system contains files and directories.
ls
ls -a
ls -l

< The `ps` command lists current processes >

Learning about Processes with `ps`

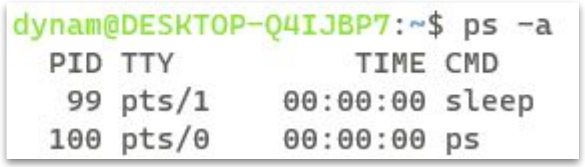
- `ps` (process status): a Linux user command
  - Can allow us to explore the contents of ***user space***
    - User space: portion of computer memory, holding running programs & data
  - Shows “current” processes running in user space



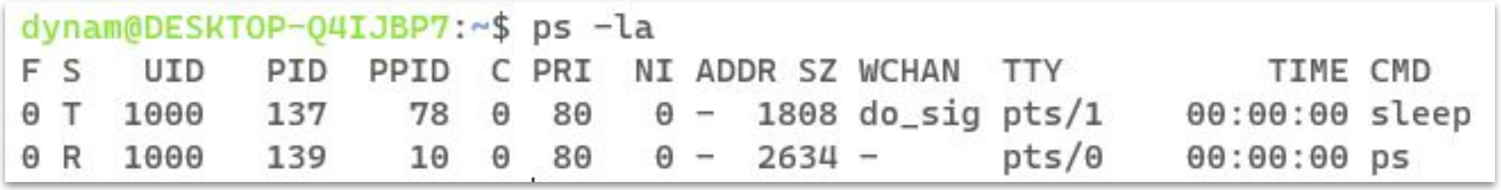
```
PS(1)                                  User Commands                                  PS(1)

NAME
       ps - report a snapshot of the current processes.

SYNOPSIS
       ps [options]

DESCRIPTION
       ps displays information about a selection of the active processes.  If you want a repetitive update of the
       selection and the displayed information, use top(1) instead.

       This version of ps accepts several kinds of options:

       1   UNIX options, which may be grouped and must be preceded by a dash.
       2   BSD options, which may be grouped and must not be used with a dash.
       3   GNU long options, which are preceded by two dashes.

       Options of different types may be freely mixed, but conflicts can appear.  There are some synonymous options,
       which are functionally identical, due to the many standards and ps implementations that this ps is compatible
       with.
```

Practice with `ps`

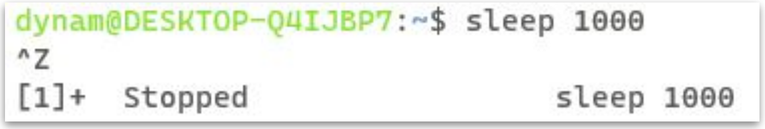
- ‘`-a`’ option
  - Lists more processes, including ones being run by other users and at other terminals
  - Exclude the shell (bash)
- ‘`-l`’ option: ‘`-a`’ + long format (displaying more information)
  - `S` – the status of each process
    - `S`: sleeping, `R`: running, `T`: stopped, and more …
  - `UID` – user id
  - `PPID` – parent process ID
  - `C` – CPU usage by a process
  - `PRI` – (immediate) priority
    - Higher numbers mean lower priority
  - `NI` – niceness level (assigned priority, or preemption level?!)
    - 19 (nicest) ~ -20 (not nice to other)
    - A process with a higher number will yield CPU time to other processes



```
dynam@DESKTOP-Q4IJBP7:~$ ps -a
  PID TTY          TIME CMD
   99 pts/1    00:00:00 sleep
  100 pts/0    00:00:00 ps
```



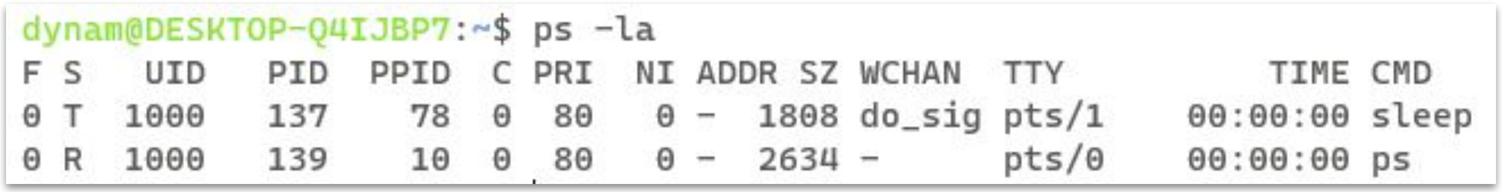
```
dynam@DESKTOP-Q4IJBP7:~$ ps -la
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 T  1000   137    78  0  80   0 -  1808 do_sig pts/1    00:00:00 sleep
0 R  1000   139    10  0  80   0 -  2634 -      pts/0    00:00:00 ps
```



```
dynam@DESKTOP-Q4IJBP7:~$ sleep 1000
^Z
[1]+  Stopped                 sleep 1000
```

Practice with `ps` (cont.)

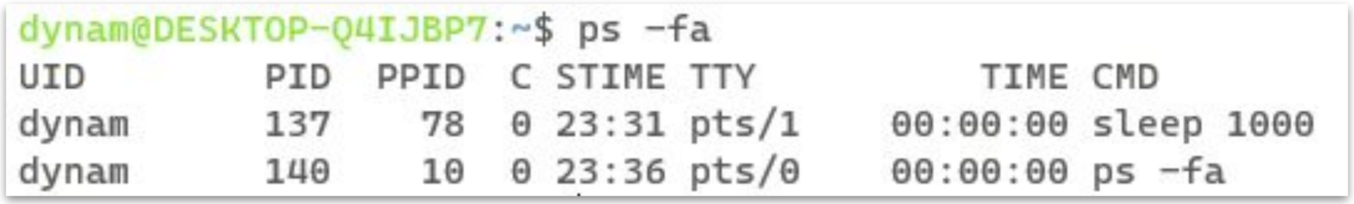
- '`-l`' option: '`-a`' + long format (displaying more information) (Cont'd)
  - `SZ`: size of a process (in KB)
    - The amount of memory in use for this process
  - `WCHAN`: show why a process is sleeping
    - Memory address of the event that the process is waiting for
    - Shows the name of the kernel function in which the process is sleeping
      - Ex. 'inet_c', 'poll_s' shown below.
      - '-': denoting a running process
  - `ADDR` (process memory address) and `F` (flag): not used any longer
    - For compatibility with programs to expect to see them



```
dynam@DESKTOP-Q4IJBP7:~$ ps -la
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 T  1000   137    78  0  80   0 -   1808 do_sig pts/1    00:00:00 sleep
0 R  1000   139    10  0  80   0 -   2634 -      pts/0    00:00:00 ps
```

Practice with `ps` (cont.)

- '`-fa`' option: more human-readable format than the plain '`-a`' option
    - The username is displayed instead of the `UID` number
    - The complete command line is listed in the `CMD` column



```
dynam@DESKTOP-Q4IJBP7:~$ ps -fa
UID          PID   PPID  C STIME TTY          TIME CMD
dynam        137     78  0 23:31 pts/1    00:00:00 sleep 1000
dynam        140     10  0 23:36 pts/0    00:00:00 ps -fa
```

- Do '`man ps`' for further details…

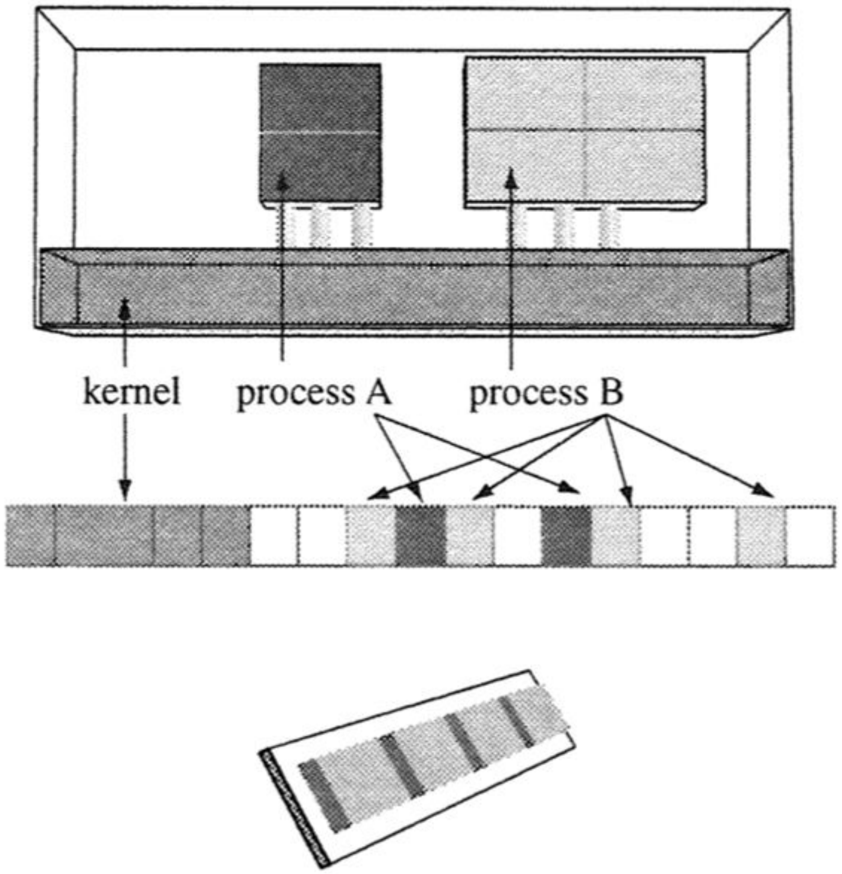
File Management and Process Management

- Files
  - Contains data
  - Have attributes
  - Created or destroyed by the kernel
  - Stored on disk
  - Has an allocation list of disk blocks

- Processes
  - Contains executable code
  - Have attributes
  - Created or killed by the kernel
  - Stored in memory by the kernel,
    - Allocating space
    - Keeping track of which processes use which blocks of memory
  - Has a structure to hold an allocation list of memory pages

Computer Memory and Programs

- Memory can be viewed as an expanse of space containing the kernel and processes
  - Divided into kernel space and user space
    - Again, processes live in user space
- Many systems view memory as an array of "pages" and split processes into pages
  - Memory: a sequence of bytes (just a big array)
  - Just as disk files split into disk blocks
- The array of pages may be stored physically in solid state chips on little circuit boards



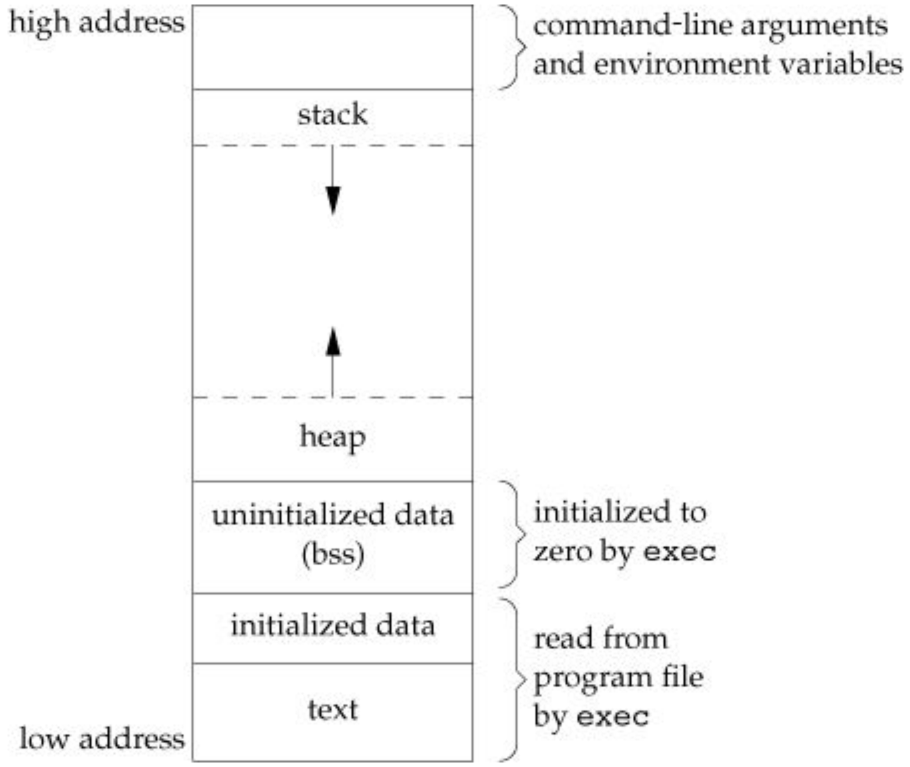
< Three models of computer memory >

Computer Memory and Programs (cont.)

- Kernel: creates a process
  - In a very similar way of creating a file
  - Finds some “free” pages of memory to hold the machine code and data bytes for the process (Or, the program in action)
  - Sets up some data structures to store memory allocation information and attributes of the processes
  - Converts a sequence of bits on chips to a society of processes: born, live, forked, job-running, and dying
    - C.f. disk: sectors on platters => directory structure
- For better understanding processes, let’s be familiar with a shell!

Memory Layout of a C Program

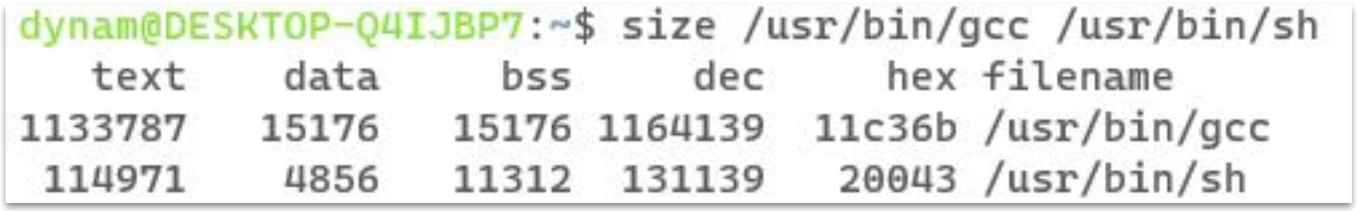
- Stack
  - Automatic variables are stored, along with information that is saved each time a function is called
- Heap
  - Dynamic memory allocation usually takes place
- Uninitialized data segment
  - Often called the "bss (block started by symbol)" segment
    - e.g., `long sum[1000];`
- Initialized data segment
    - e.g., `int maxcount = 99;`
- Text segment
  - Consisting of the machine instructions that the CPU executes


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

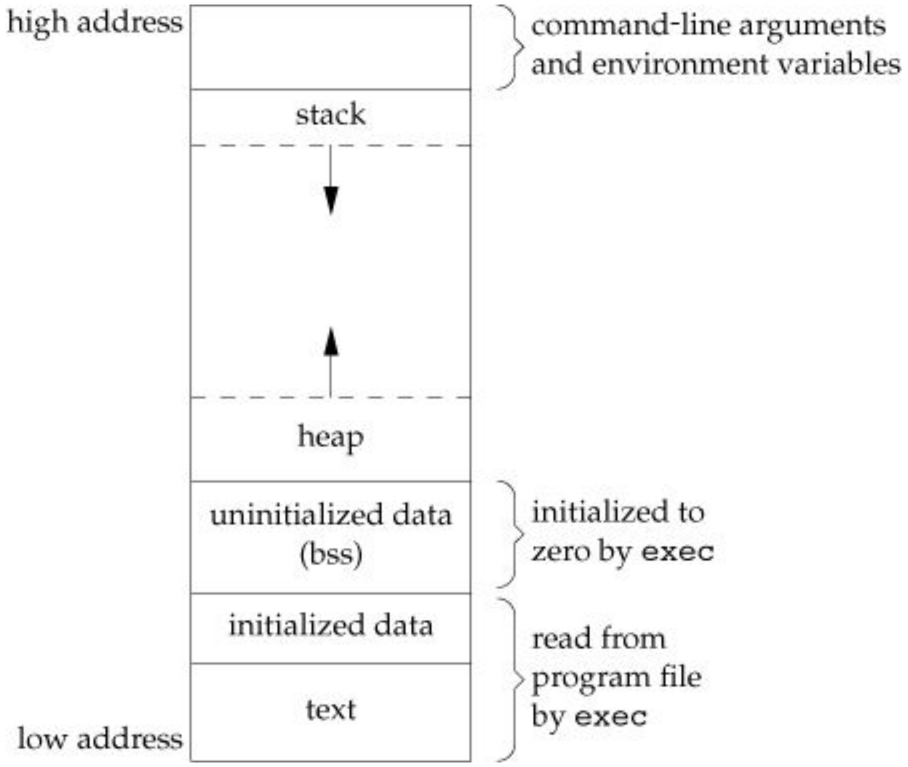
< Typical memory arrangement >

Memory Layout of a C Program (cont.)

- The `size` command
  - Reports the sizes (in bytes) of the text, data, and bss segments
  - The fourth and fifth columns are the total of the three sizes, displayed in decimal and hexadecimal, respectively



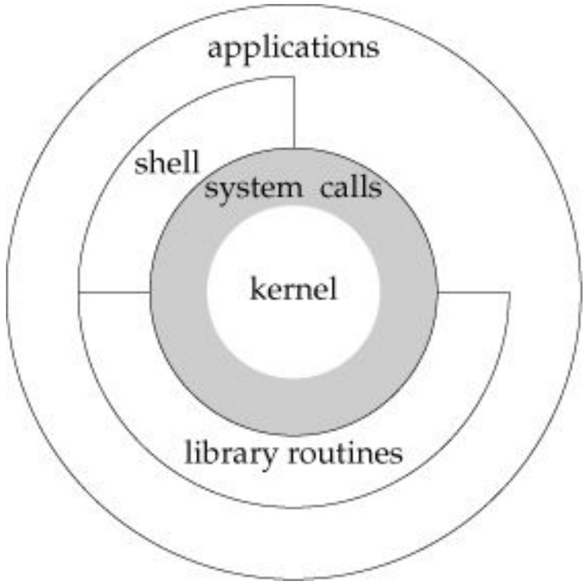
```
dynam@DESKTOP-Q4IJBP7:~$ size /usr/bin/gcc /usr/bin/sh
   text    data     bss     dec     hex filename
1133787   15176   15176 1164139  11c36b /usr/bin/gcc
 114971    4856   11312  131139   20043 /usr/bin/sh
```


high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

< Typical memory arrangement >

Shell: A Tool for Process and Program Control

- A user interface for access to an OS's services
  - Typically uses a command-line interface (CLI)
- A program that manages processes and runs (other) programs
  - A lot of shells available for Linux/Unix: `sh`, `bash`, `tcsh`, `csh`, etc.
- Why is it "named so"?
  - As it's the "outmost" layer around the kernel
  - As it encloses the kernel to talk to it
- Has three main functions
  - Program running: execute programs
  - Input/output managing: receives input args and controls output
  - Directly programmable UI: possible to do the coding on the fly

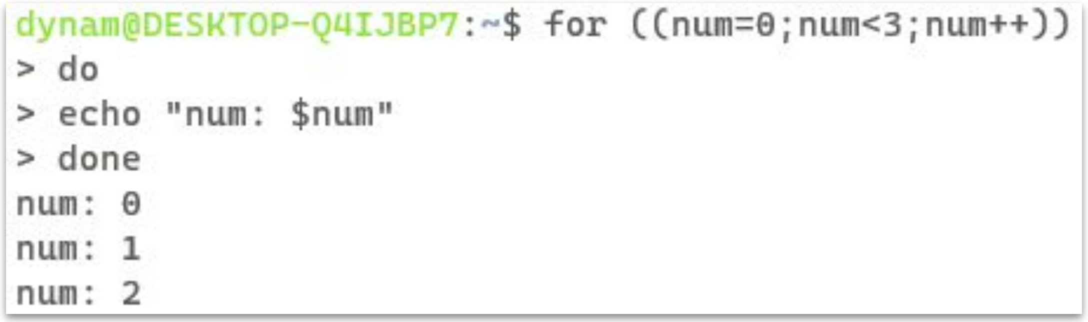


Major Functionalities of Shells

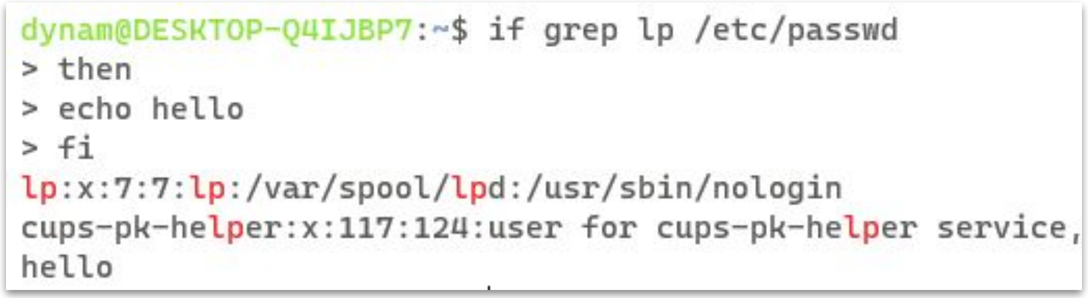
- 1) Program running
    - Shells interpret user commands, load relevant programs into memory and run the programs.
        - Regarded as “program launcher”
- 2) Managing I/O
    - >, <, | are symbols for input/output redirection
        - Redirection: directing input and output to files and devices other than the default I/O devices.
    - With the symbols, a user tells the shell to attach the input/output of processes
        - To disk files or
        - To other processes

Major Functionalities of Shells (cont.)

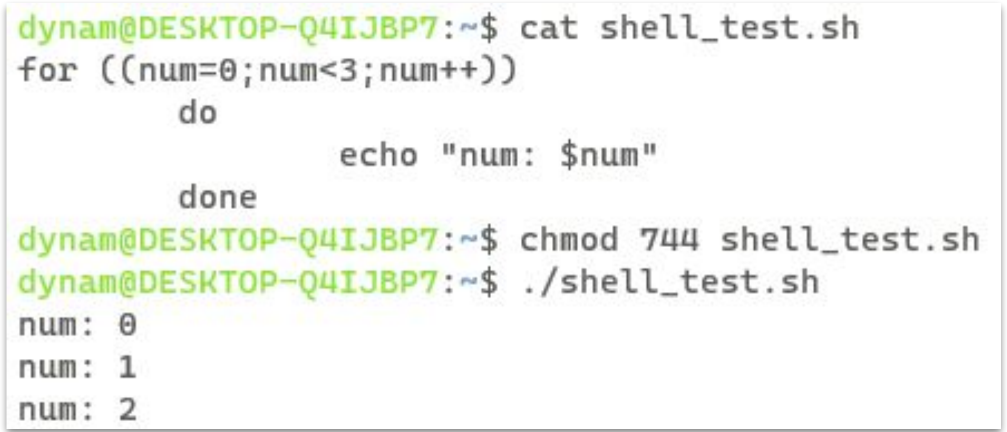
- 3) Programming
  - "Programming language" with variables and flow control (if, while, for, etc.)
    - We can put the following code into a (shell) script, from which the commands can be run in an automated way.



```
dynam@DESKTOP-Q4IJBP7:~$ for ((num=0;num<3;num++))
> do
> echo "num: $num"
> done
num: 0
num: 1
num: 2
```



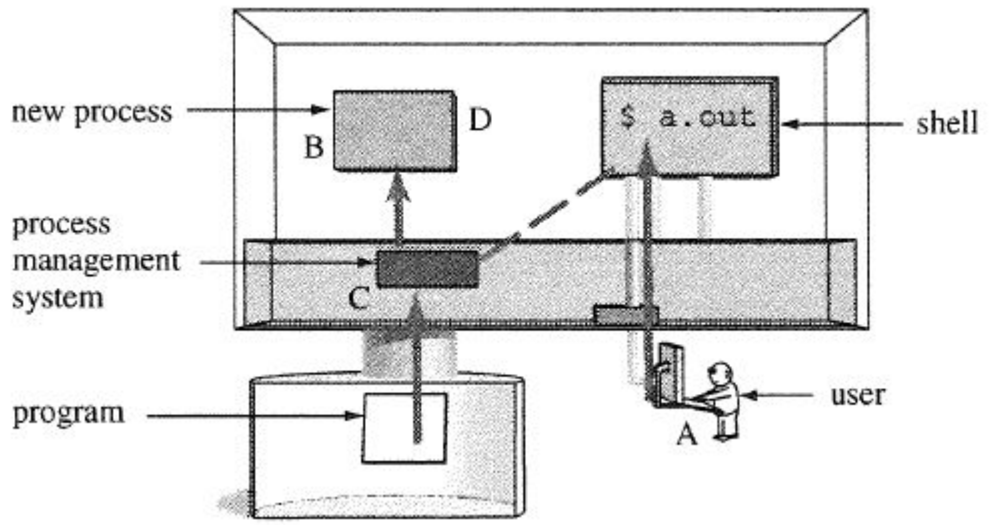
```
dynam@DESKTOP-Q4IJBP7:~$ if grep lp /etc/passwd
> then
> echo hello
> fi
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
cups-pk-helper:x:117:124:user for cups-pk-helper service,
hello
```



```
dynam@DESKTOP-Q4IJBP7:~$ cat shell_test.sh
for ((num=0;num<3;num++))
        do
                echo "num: $num"
        done
dynam@DESKTOP-Q4IJBP7:~$ chmod 744 shell_test.sh
dynam@DESKTOP-Q4IJBP7:~$ ./shell_test.sh
num: 0
num: 1
num: 2
```

How the Shell Runs Programs

- The shell prints a prompt, you type a command, and the shell runs the command. Then, the shell prints a prompt over and over again. What's happening behind the scene?
  - A. The user types in `a.out`
  - B. The shell creates a new process to run the program
  - C. The shell loads the program from the disk into the process
  - D. The program runs in its process until it is done

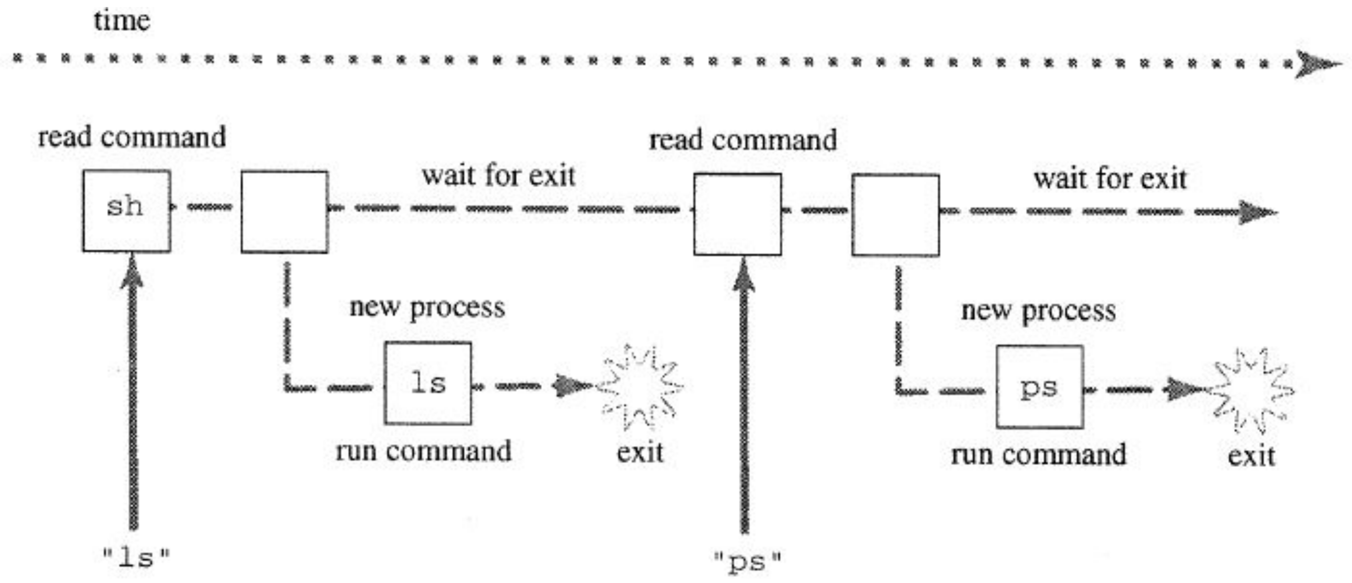


< A user asks a shell to run a program >

The Main Loop of a Shell

- The shell consists of the following loop:

```
while ( ! end_of_input )
        get command
        execute command
        wait for command to finish
```



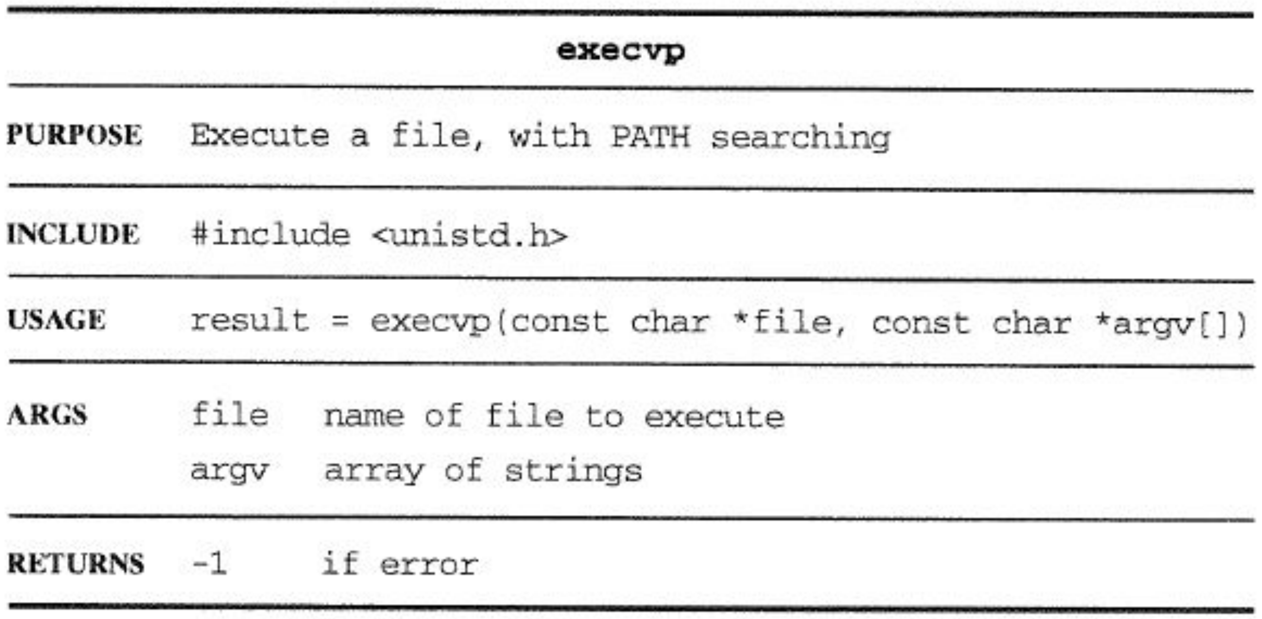
< A time line of the main loop of the shell >

The Main Loop of a Shell (cont.)

- To write a shell, we need to learn how to
  - 1) Run a program
  - 2) Create a process
  - 3) Wait for `exit()`
- Once we get familiar with these, then we'll be able to write our own shell!

Question 1: How Does a Program Run a Program?

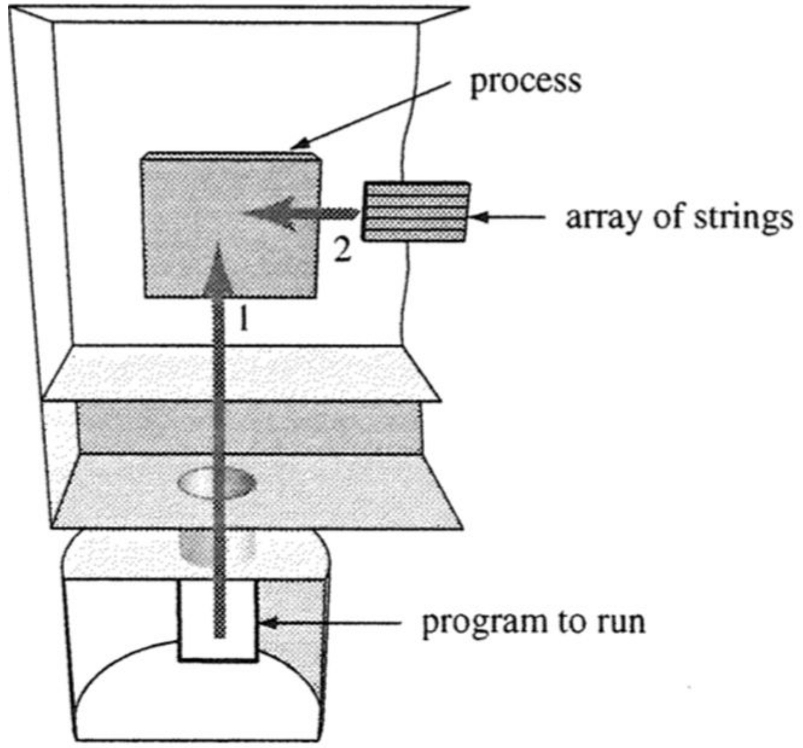
- Answer: the program invokes `execvp()`
  - The `execvp()` function replaces the current process image with a new process image specified by *file*
    - The new image is constructed from a regular, executable file called the new process image file



execvp		
PURPOSE	Execute a file, with PATH searching	
INCLUDE	#include <unistd.h>	
USAGE	result = execvp(const char *file, const char *argv[])	
ARGS	file	name of file to execute
	argv	array of strings
RETURNS	-1	if error

Question 1: How Does a Program Run a Program? (cont.)

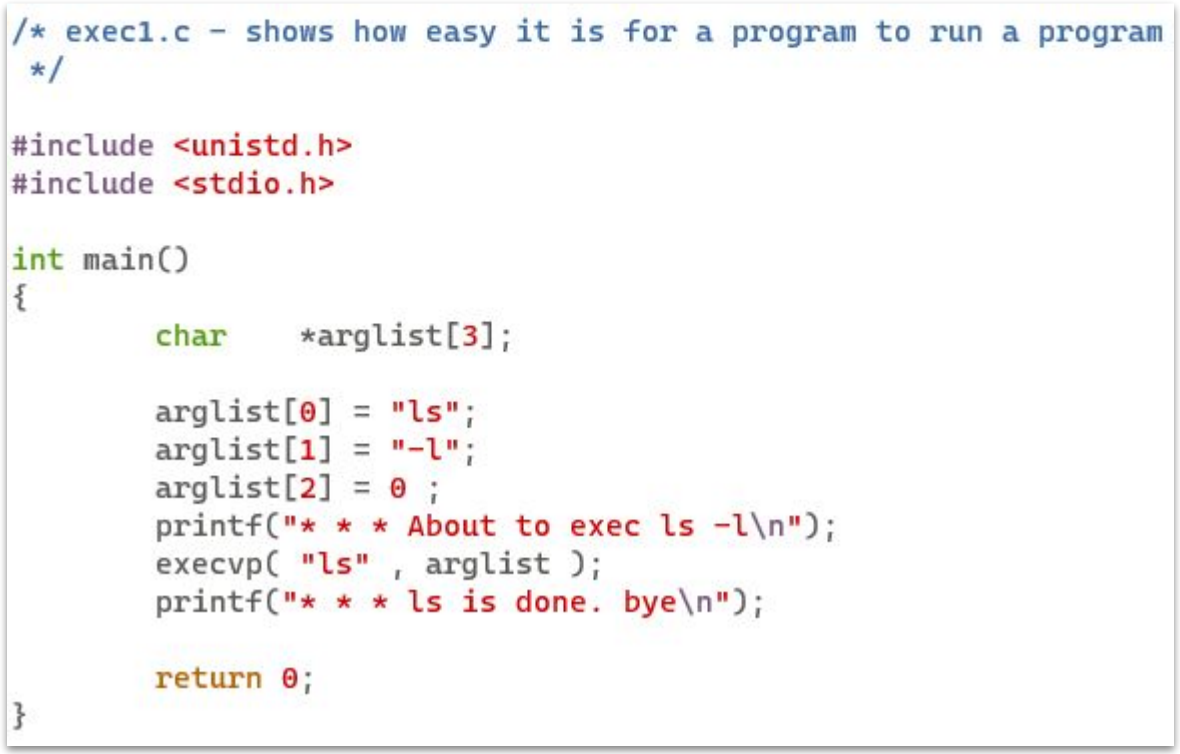
- How Unix/Linux runs programs:
  - `execvp`(progname, arglist)
    - 1) Copies the named program into the calling process of `execvp`
    - 2) Passes the specified list of strings to the program as `argv[]`
    - 3) Runs the program
  - In short,
    - 1. Program in action (hereafter, “process”) calls `execvp`
    - 2. Kernel loads program from disk into the process
    - 3. Kernel copies `arglist` into the process
    - 4. Kernel calls `main`(`argc`, `argv`)



< `execvp` copies into memory and runs a program >

Let’s Write a Program that Runs Another Program (`ls`) with an Option (`-l`)

- `execvp` takes two arguments
  - The name of the program to run
  - An array of command-line arguments for that program
- Note
  - Set the first string to the name of the program
  - The array must have a null pointer as the last element



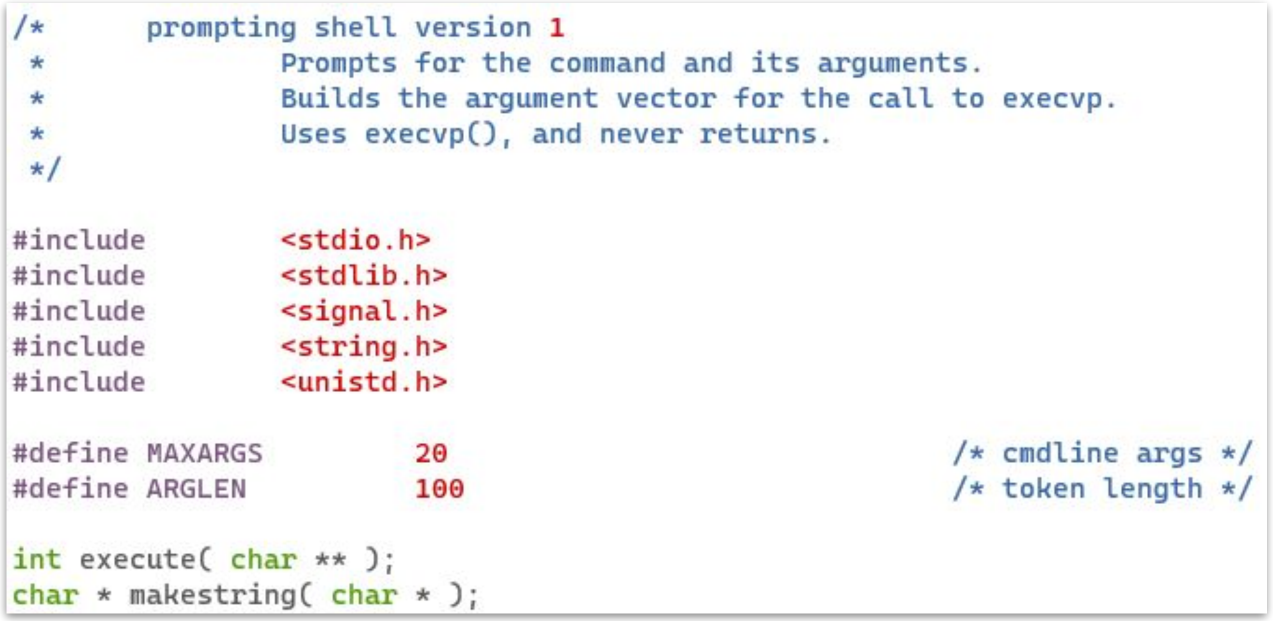
```c
/* exec1.c - shows how easy it is for a program to run a program
 */

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

int 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");

        return 0;
}
```

Here's What Happened

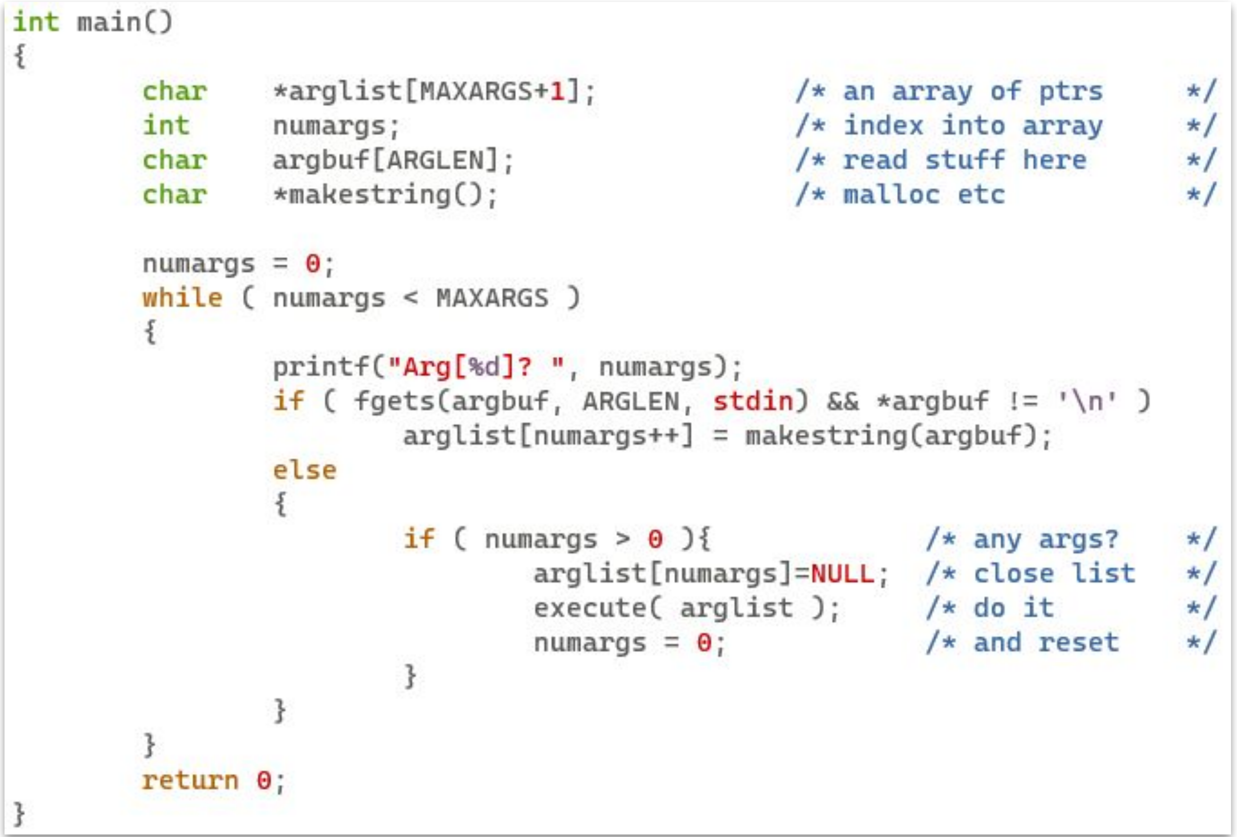
- The kernel loads the new program into the current process, replacing the code and data of that current process
    - The current program is removed from the process, and then that new program executes in the current process.
- The `exec` system call:
    - Clears out the machine code of the calling process (or, current program)
    - Puts the code of the program given in the `exec` call
    - Runs that new program in the calling process
- The memory location of the calling process is changed to fit the space requirements of the callee program
    - The process is the same, but the contents are NEW!

Writing the First Shell: `psh1.c`

- A Prompting Shell
  - Prompts a user for a program name and arguments
  - Runs the program

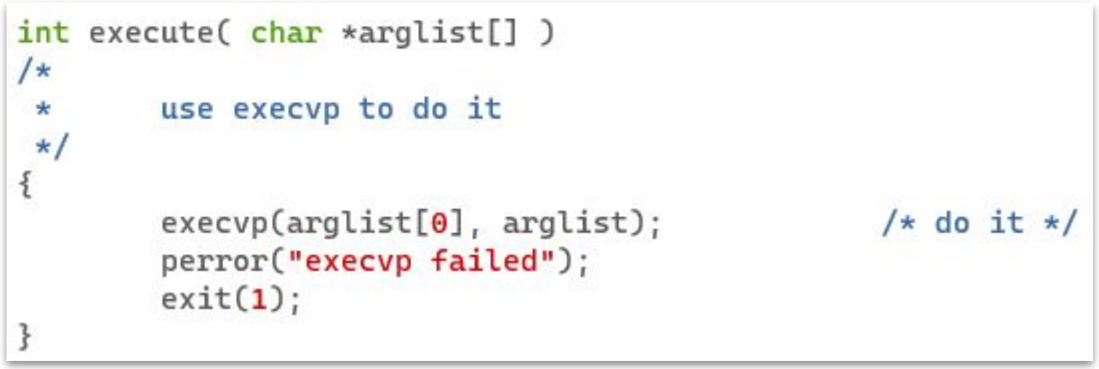


```c
/*      prompting shell version 1
 *              Prompts for the command and its arguments.
 *              Builds the argument vector for the call to execvp.
 *              Uses execvp(), and never returns.
 */

#include        <stdio.h>
#include        <stdlib.h>
#include        <signal.h>
#include        <string.h>
#include        <unistd.h>

#define MAXARGS         20                              /* cmdline args */
#define ARGLEN          100                             /* token length */

int execute( char ** );
char * makestring( char * );
```

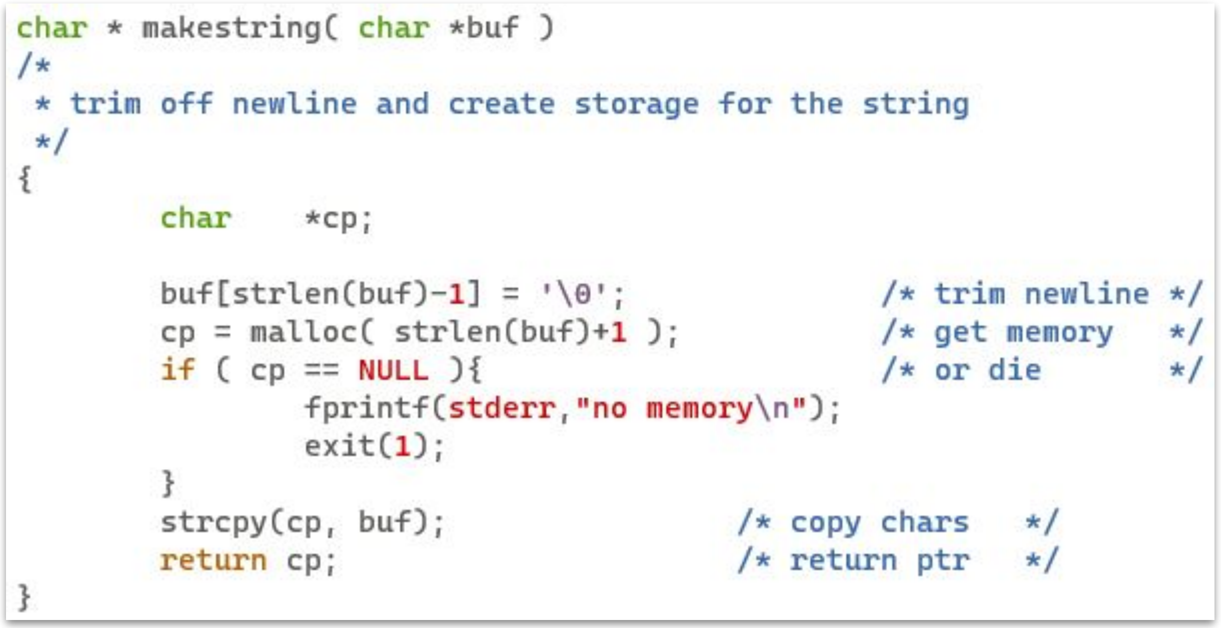


```c
int main()
{
        char    *arglist[MAXARGS+1];            /* an array of ptrs    */
        int     numargs;                        /* index into array    */
        char    argbuf[ARGLEN];                 /* read stuff here     */
        char    *makestring();                  /* malloc etc          */

        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;
}
```

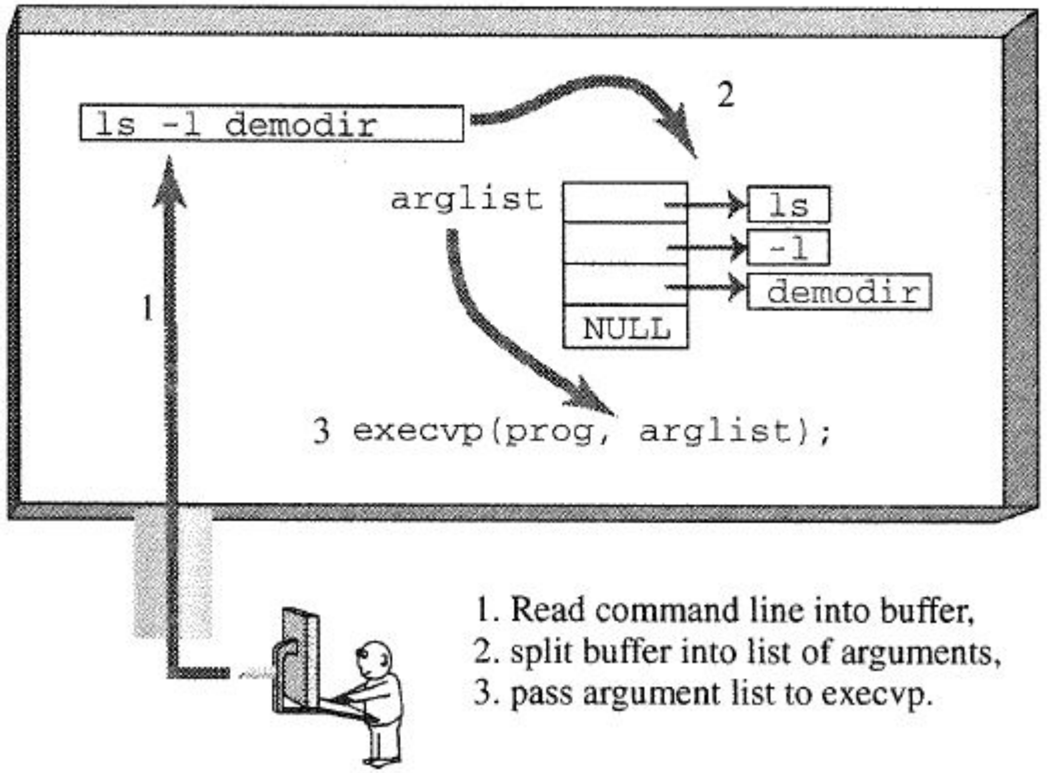
Writing the First Shell: `psh1.c` (cont.)



```
int execute( char *arglist[] )
/*
 *      use execvp to do it
 */
{
        execvp(arglist[0], arglist);            /* do it */
        perror("execvp failed");
        exit(1);
}
```



```
char * makestring( char *buf )
/*
 * trim off newline and create storage for the string
 */
{
        char    *cp;

        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   */
}
```



1. Read command line into buffer,
2. split buffer into list of arguments,
3. pass argument list to execvp.

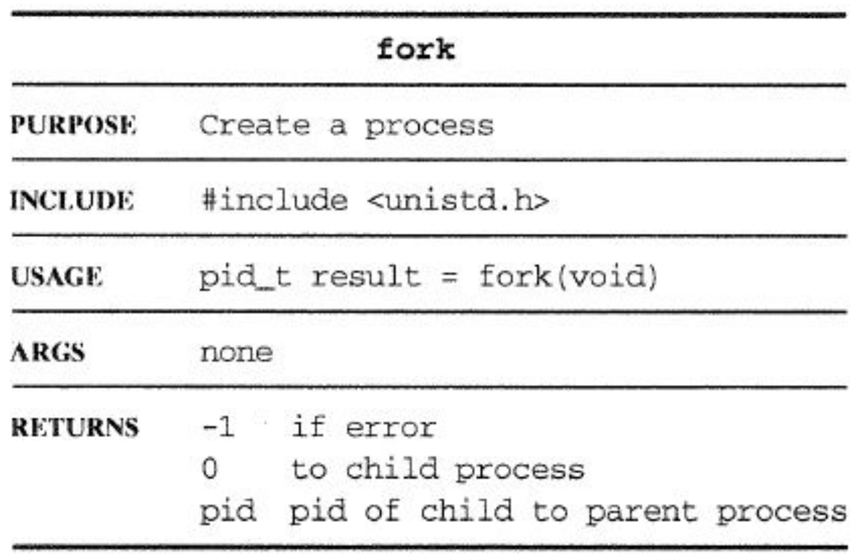
< Building an `arglist` from one string >

Enough?

- `psh1` just exits after the command is executed
- What if a user wants to run the shell again to run another command?
  - What if a shell runs a command and “loops back” to accept another command?
- A simple solution is …
  - Creating a new process, and
  - Having the new process execute the program
- No matter how many programs to run, we can just create as many processes responsible for each program as needed

Question 2: How Do We Get a New Process?

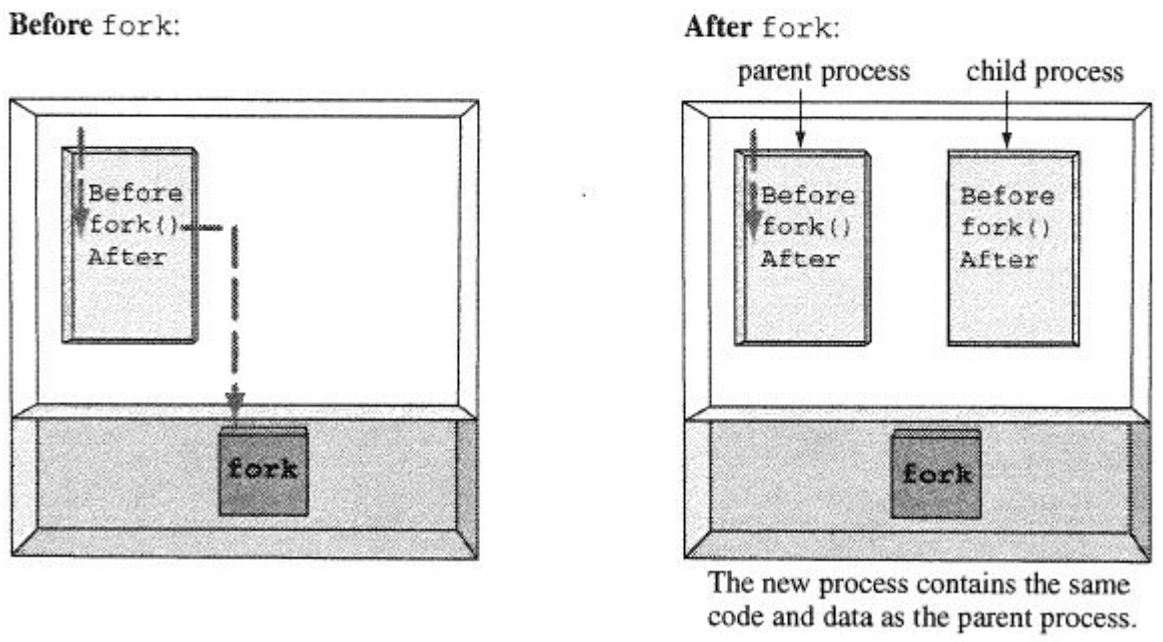
- The answer
  - A process calls **fork** to replicate itself.
  - System call: `fork(); // takes no argument`
    - The new process is referred to as the <u>child</u> process
    - The calling process is referred to as the <u>parent</u> process
    - The child process and the parent process run in separate memory spaces
    - At the time of `fork()` both memory spaces have the same content



fork	
PURPOSE	Create a process
INCLUDE	`#include <unistd.h>`
USAGE	`pid_t result = fork(void)`
ARGS	none
RETURNS	-1 if error 0 to child process pid pid of child to parent process

Explanation of `fork`: Making a Copy of a Process

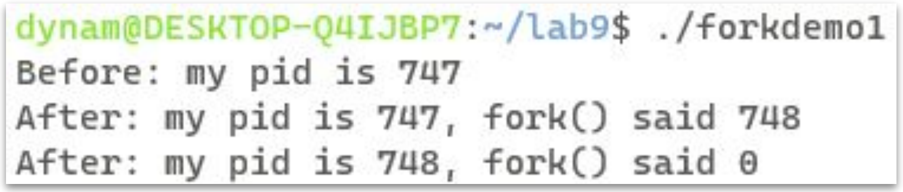
- When `fork` is invoked, what the kernel does is
  - Allocate a new chunk of memory and kernel data structures
  - Copy the original process (code and data) into the new process
  - Add the new process to the set of running processes
  - Return control back to *both* processes



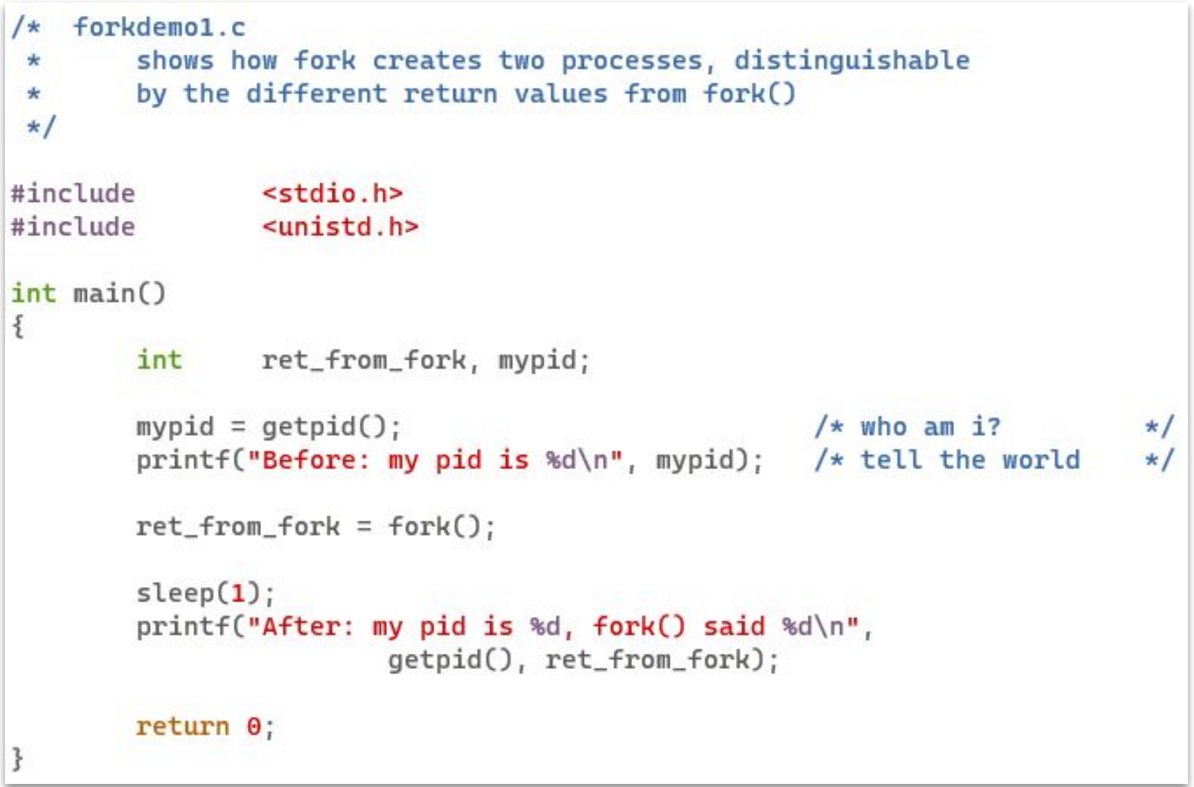
< `fork()` makes a copy of a process >

Creating a New Process: `forkdemo1.c`

- The kernel creates process 748
  - by replicating process 747, copying the code and the *current line* in the code into the new process

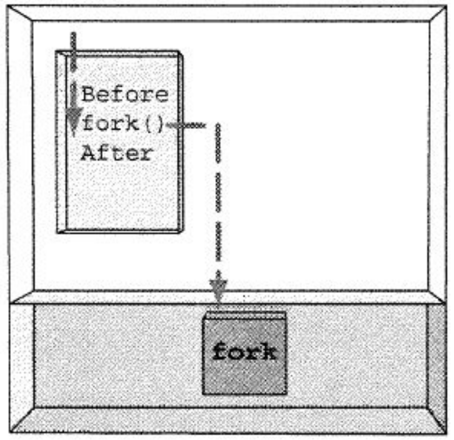


```
dynam@DESKTOP-Q4IJBP7:~/lab9$ ./forkdemo1
Before: my pid is 747
After: my pid is 747, fork() said 748
After: my pid is 748, fork() said 0
```



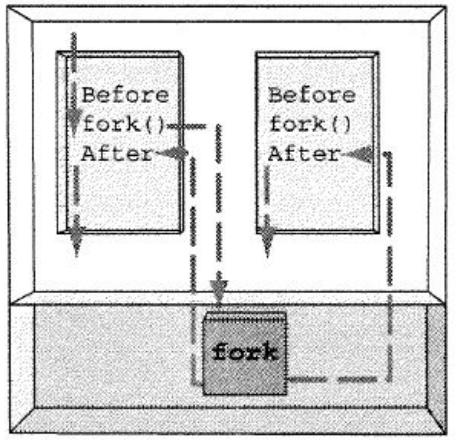
```c
/*  forkdemo1.c
 *      shows how fork creates two processes, distinguishable
 *      by the different return values from fork()
 */

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

int 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);

        return 0;
}
```

Before fork:



One flow of control enters the fork kernel code.

After fork:

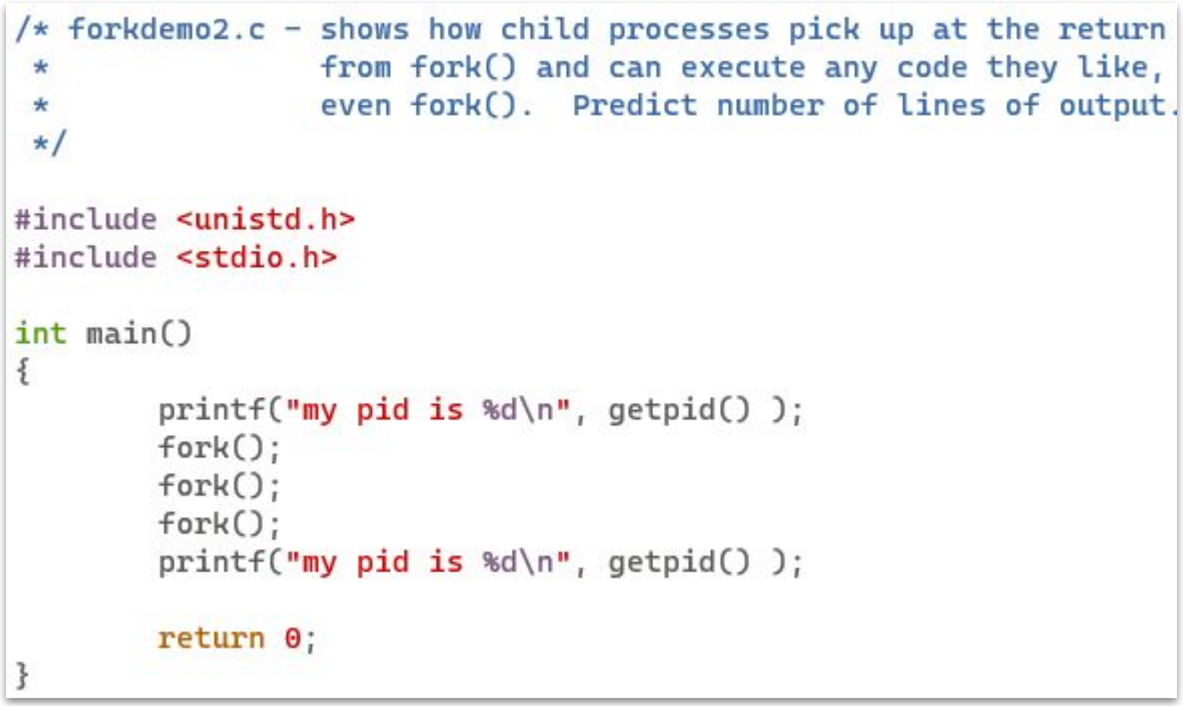


Two flows of control return from fork kernel code.

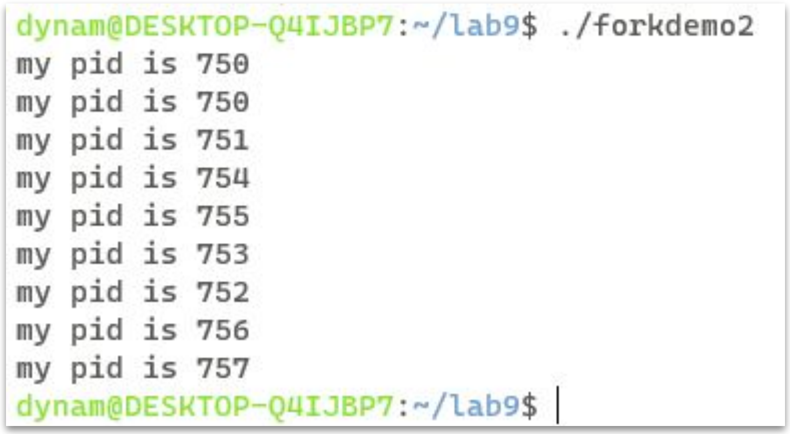
< The child executes the code after `fork()` >

Children Creating Processes: `forkdemo2.c`

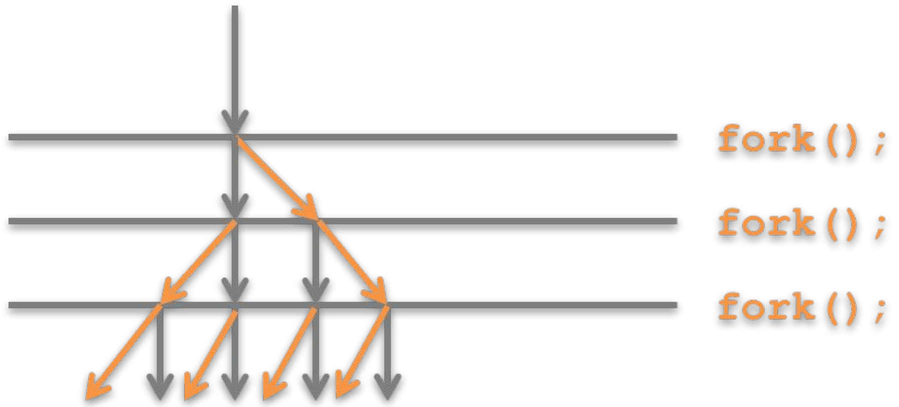
- The child process begins its life, not at the start of `main` but at the return from `fork`



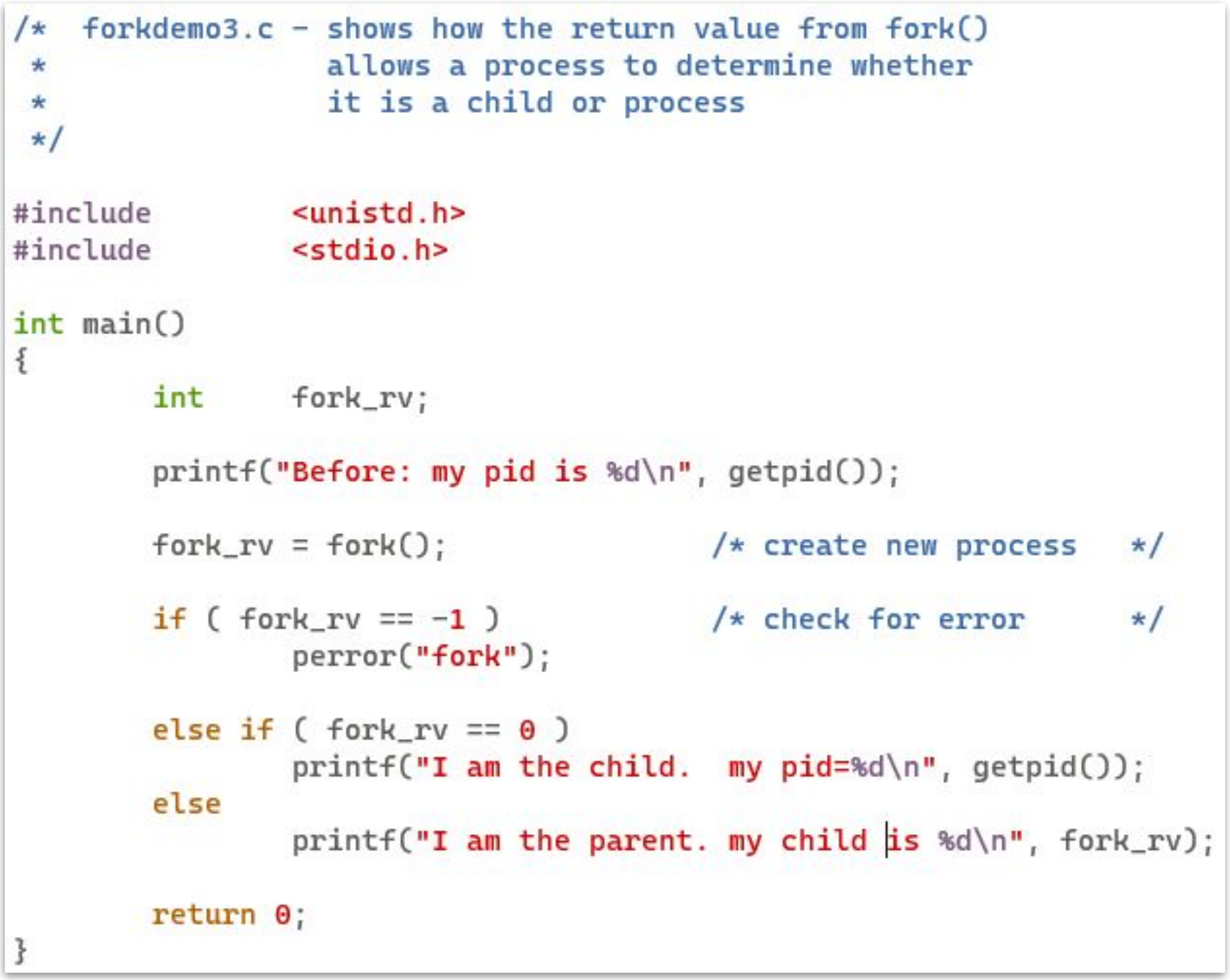
```c
/* forkdemo2.c - shows how child processes pick up at the return
 *               from fork() and can execute any code they like,
 *               even fork().  Predict number of lines of output.
 */

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

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

        return 0;
}
```



```
dynam@DESKTOP-Q4IJBP7:~/lab9$ ./forkdemo2
my pid is 750
my pid is 750
my pid is 751
my pid is 754
my pid is 755
my pid is 753
my pid is 752
my pid is 756
my pid is 757
dynam@DESKTOP-Q4IJBP7:~/lab9$
```



Distinguishing Parent from Child: `forkdemo3.c`



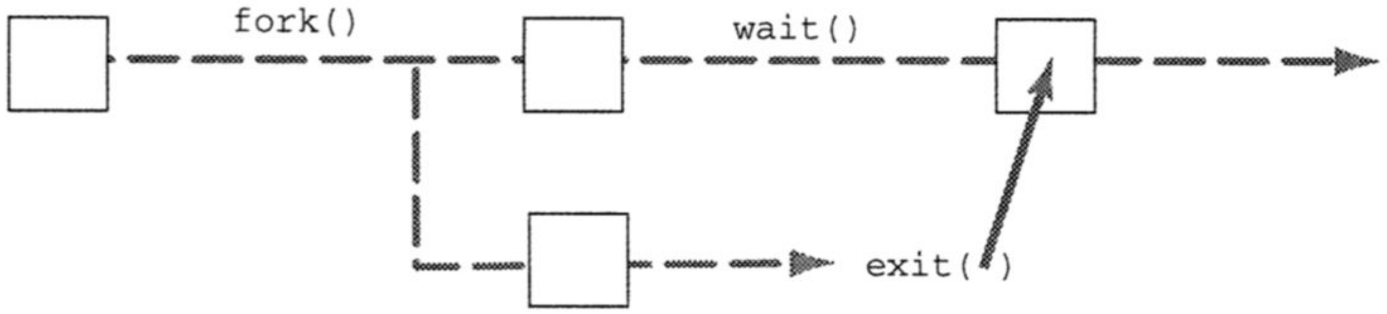
```
/*  forkdemo3.c - shows how the return value from fork()
 *                allows a process to determine whether
 *                it is a child or process
 */

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

int 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);

        return 0;
}
```

Summary: Building a Shell

- How to create a new process
    - `fork()`
        - Creates a new process (child) by duplicating the calling process (parent)
- How to run a program
    - `execvp()`
- How to tell the parent to wait until the child process finishes executing the command
    - `wait()`: to be discussed next
- The fourth system call will be discussed after `wait()`

Question 3: How Does the Parent Wait for the Child to Exit?

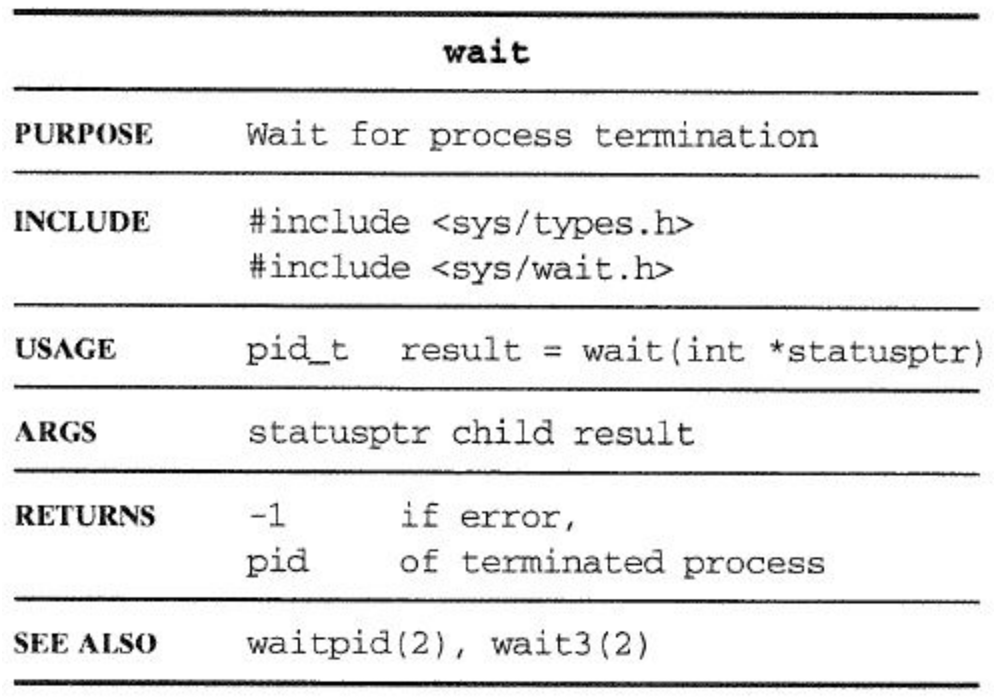
- Answer
    - A process calls `wait()` to wait for a child to finish
    - Usage
        - `pid = wait (&status)`
        - Two things done by `wait()`
            - Pausing the parent process until a child process finishes running
            - Retrieving the value the child process had passed to exit



< `wait` pauses the parent until the child finishes >

Explanation of wait

- A process calls `wait()` to wait for a child to finish



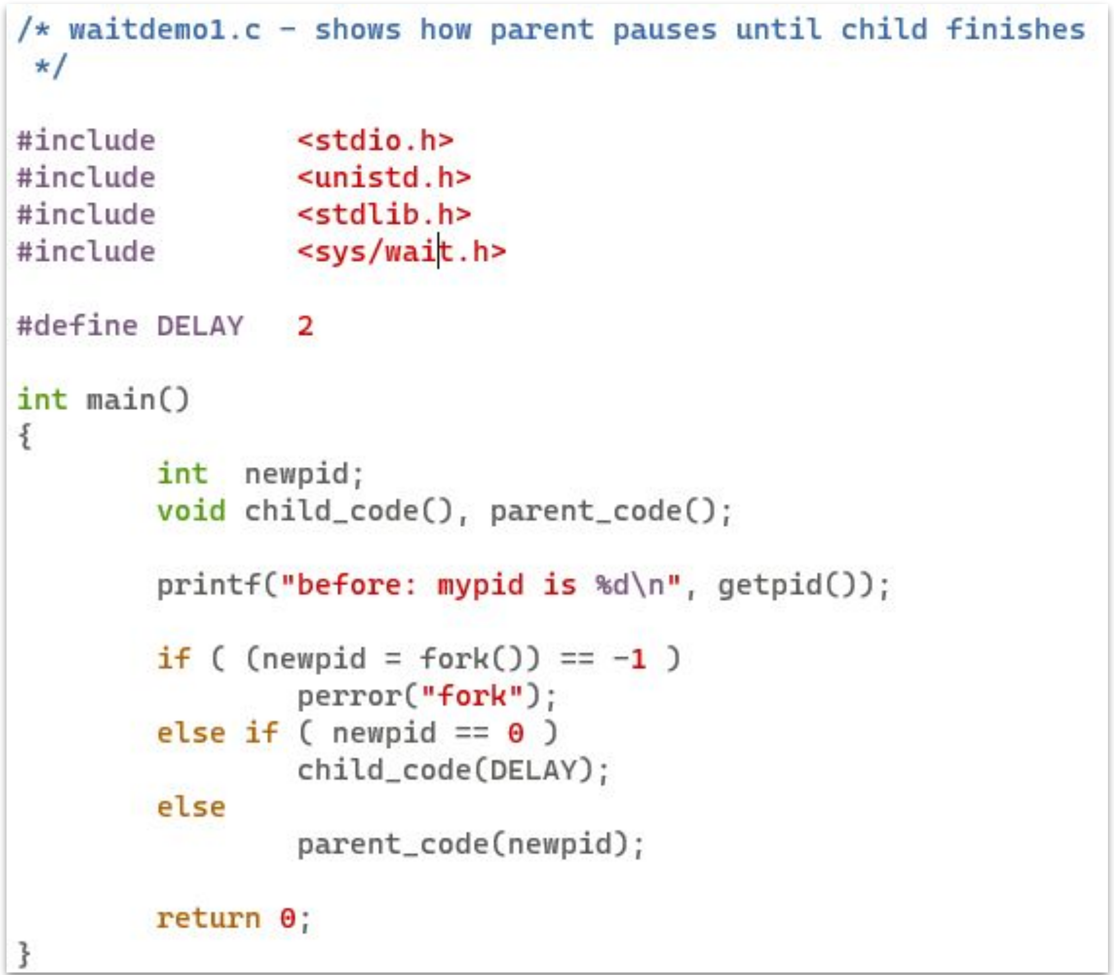
wait	
PURPOSE	Wait for process termination

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

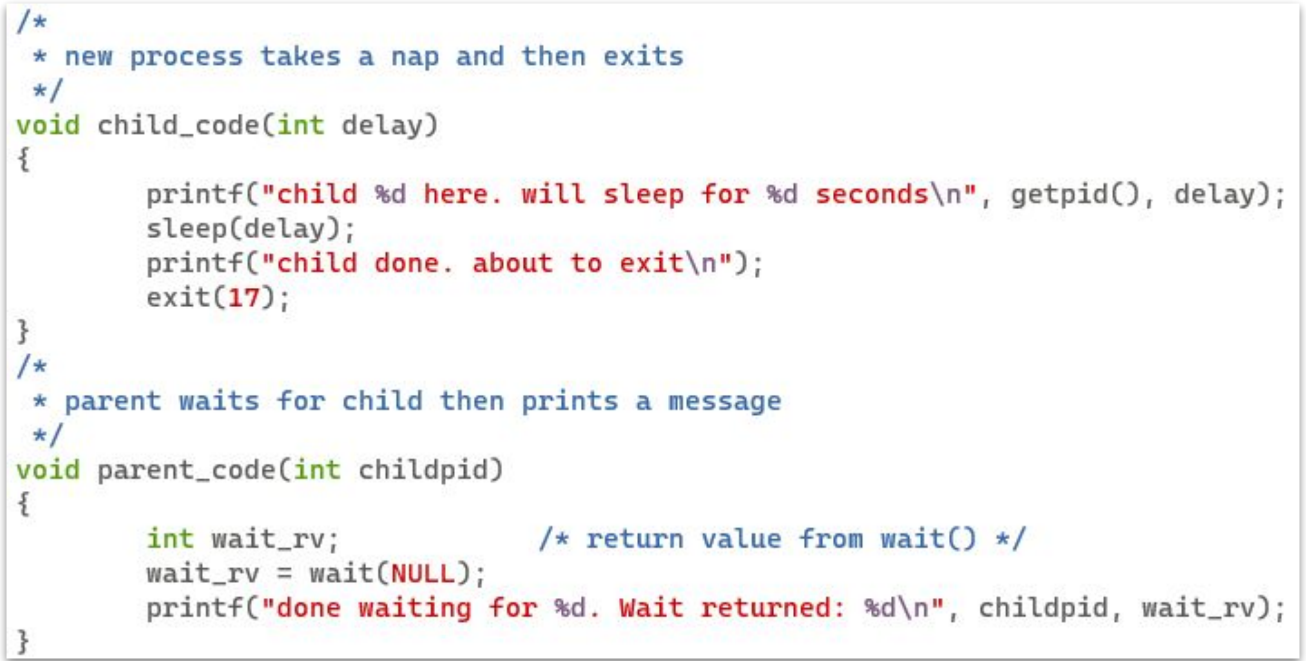
USAGE	pid_t result = wait(int *statusptr)
ARGS	statusptr child result
RETURNS	-1 if error, pid of terminated process
SEE ALSO	waitpid(2), wait3(2)

1) Notification: `waitdemo1.c`

- `wait()` notifies the parent that a child finished its running



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

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

#define DELAY   2

int 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);

        return 0;
}
```

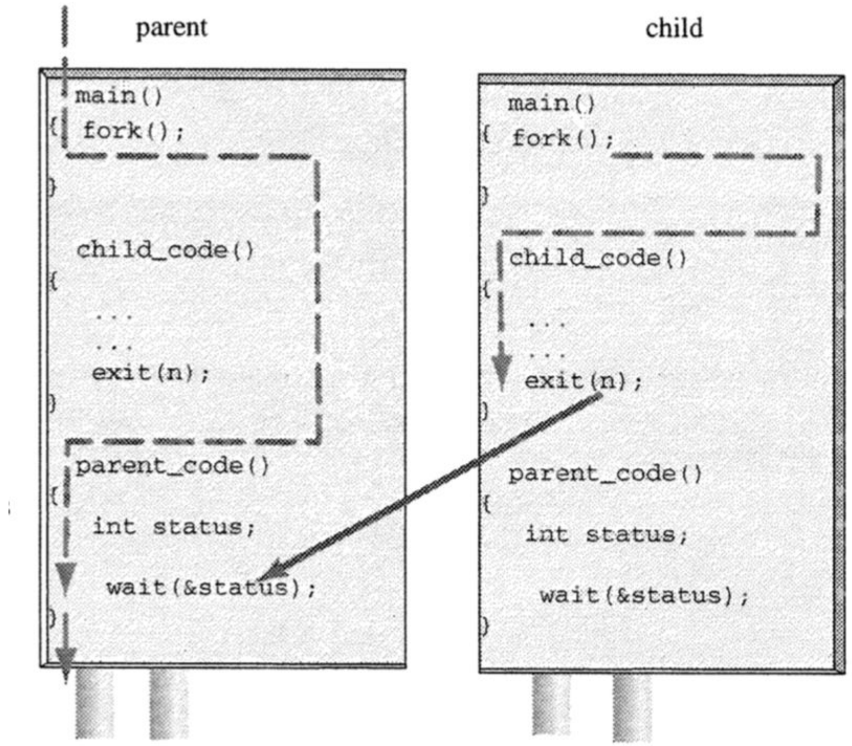


```c
/*
 * new process takes a nap and then exits
 */
void child_code(int delay)
{
        printf("child %d here. will sleep for %d seconds\n", getpid(), delay);
        sleep(delay);
        printf("child done. about to exit\n");
        exit(17);
}
/*
 * parent waits for child then prints a message
 */
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);
}
```

Explanation of wait (cont'd)

- Two important facts from `waitdemo1.c`
  - Fact 1. `wait()` **blocks** the parent process until a child finishes
    - Synchronization
  - Fact 2. `wait()` **returns** the PID of the child process that called exit
    - Effective processing possible:

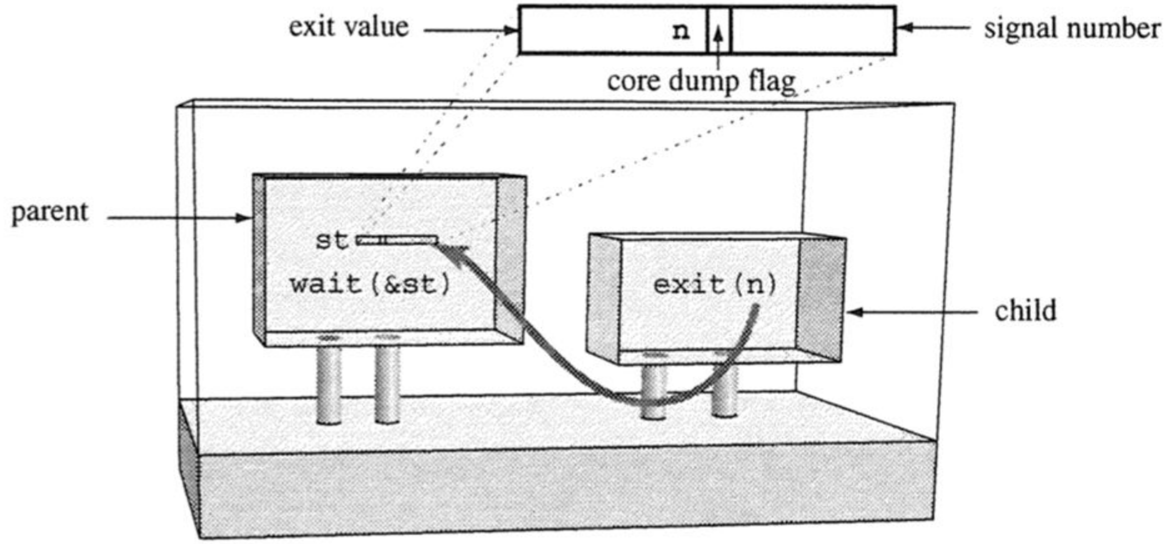
      e.g., A program consolidating data from two different remote DBs



< Control flow and communication with `wait()` >

2) Communication: `waitdemo2.c`

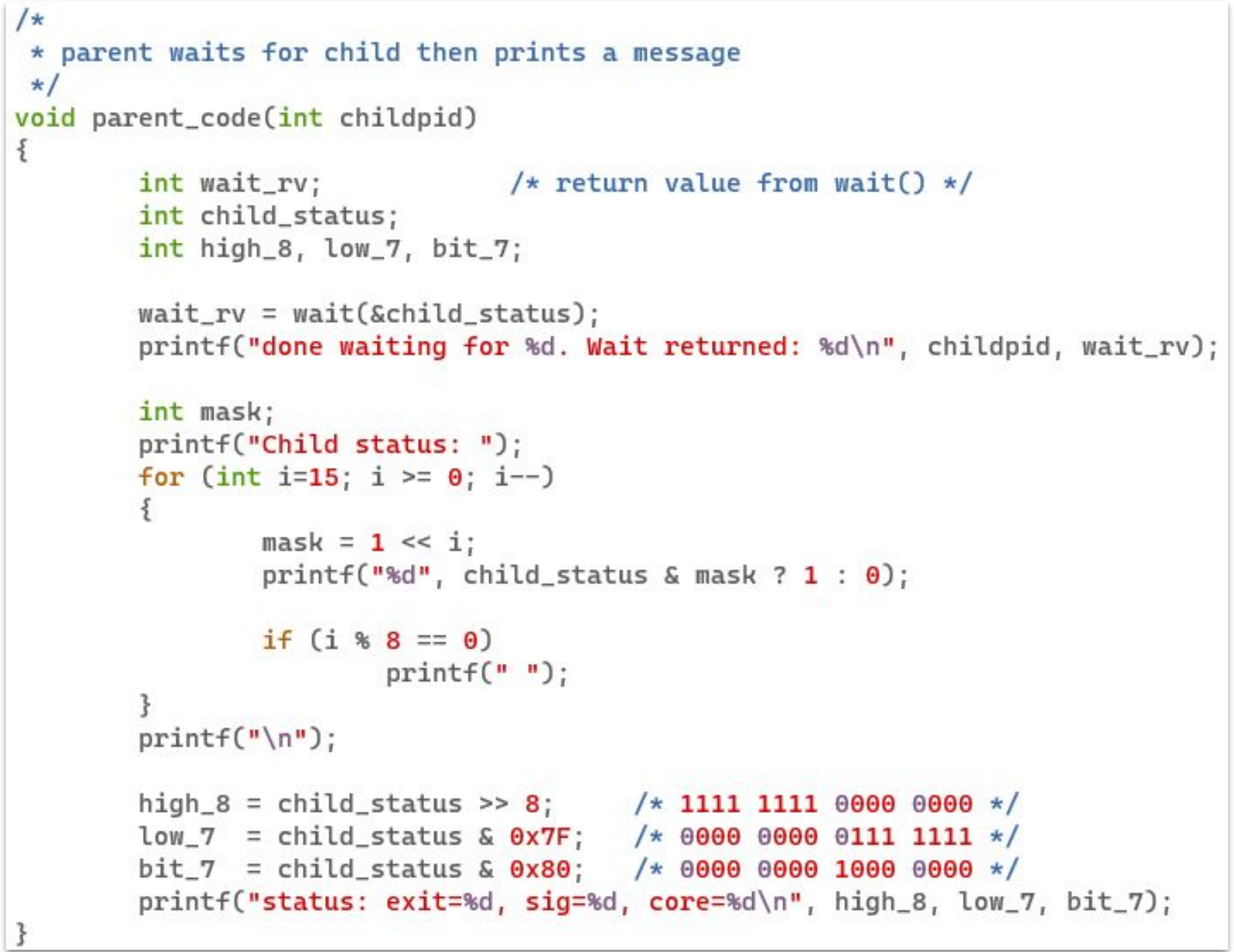
- `wait()` tells the parent *how* a child gets terminated.
  - Kernel stores the terminal status in an 2-byte integer value (st shown below)
- Core dump: a memory dump when system crashed or abnormally exited
  - Consists of the recorded state of the working memory of a program at a specific time



< The child status value has three parts >

2) Communication: `waitdemo2.c` (cont.)

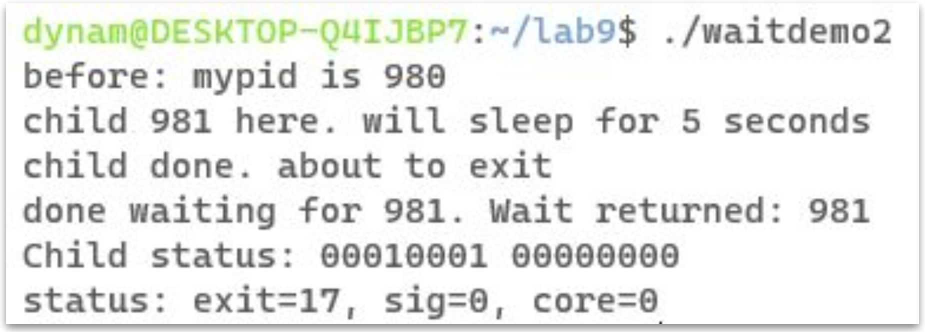
- Revision to `parent_code()` (from `waitdemo1.c`)



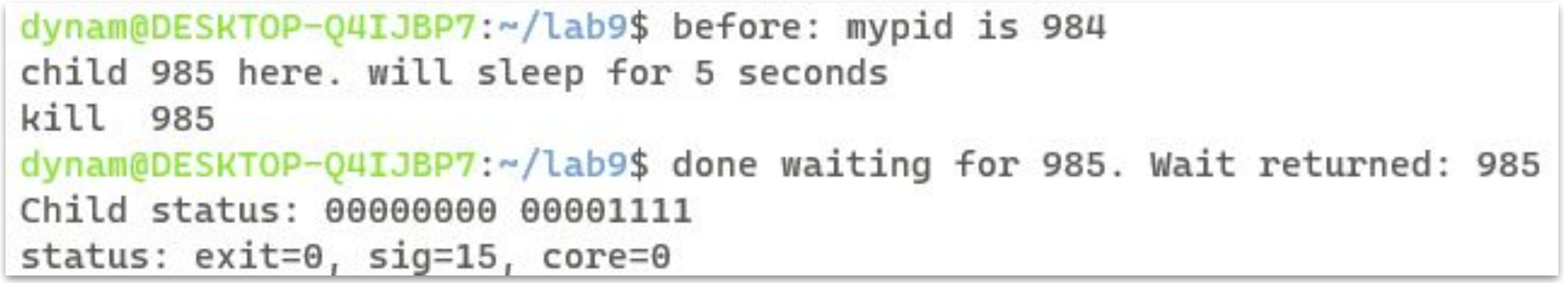
```
/*
 * parent waits for child then prints a message
 */
void parent_code(int childpid)
{
        int wait_rv;            /* return value from wait() */
        int child_status;
        int high_8, low_7, bit_7;

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

        int mask;
        printf("Child status: ");
        for (int i=15; i >= 0; i--)
        {
                mask = 1 << i;
                printf("%d", child_status & mask ? 1 : 0);

                if (i % 8 == 0)
                        printf(" ");
        }
        printf("\n");

        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);
}
```

2) Communication: `waitdemo2.c` (cont.)

- Get child status

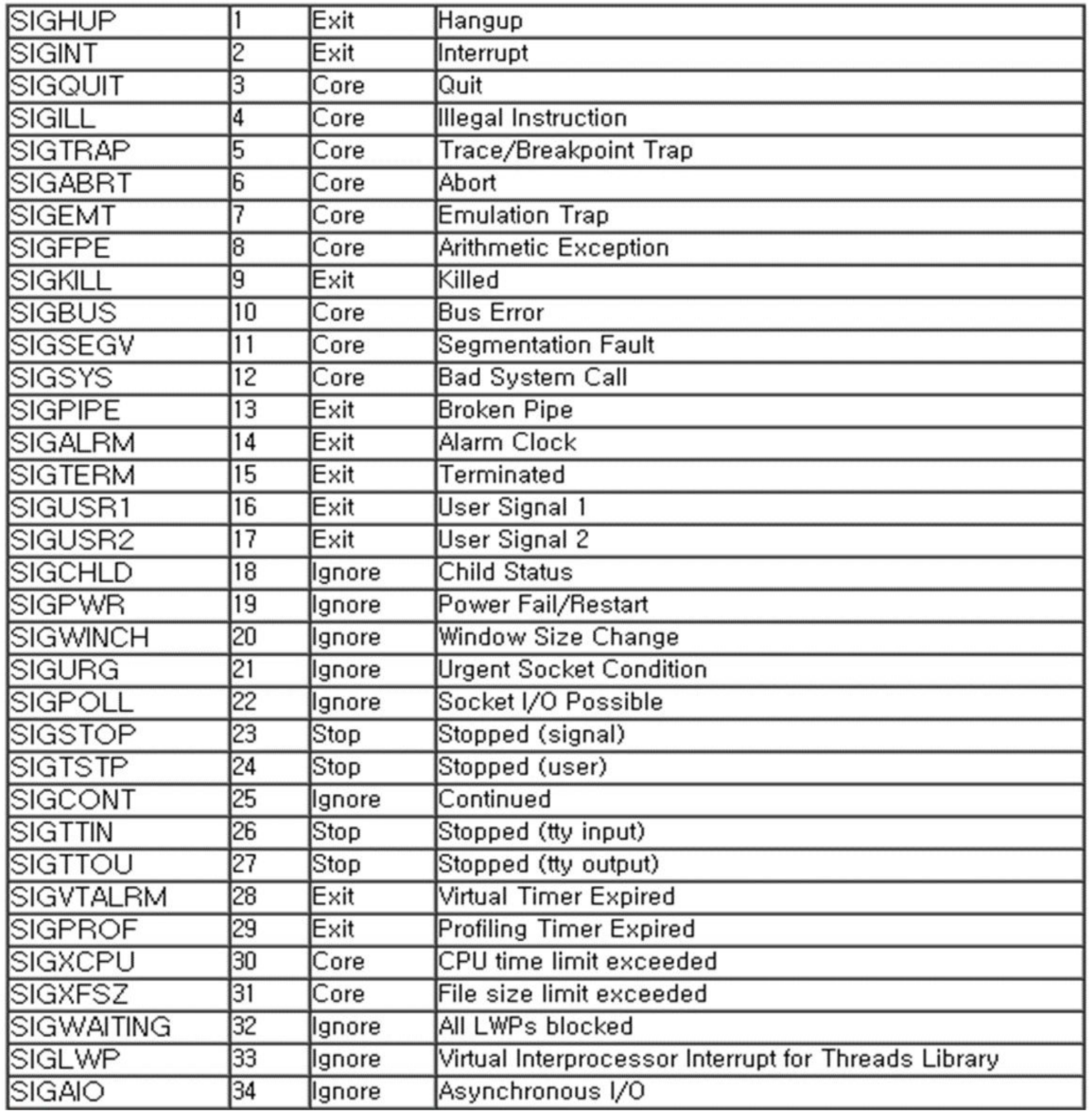


```
dynam@DESKTOP-Q4IJBP7:~/lab9$ ./waitdemo2
before: mypid is 980
child 981 here. will sleep for 5 seconds
child done. about to exit
done waiting for 981. Wait returned: 981
Child status: 00010001 00000000
status: exit=17, sig=0, core=0
```



```
dynam@DESKTOP-Q4IJBP7:~/lab9$ before: mypid is 984
child 985 here. will sleep for 5 seconds
kill  985
dynam@DESKTOP-Q4IJBP7:~/lab9$ done waiting for 985. Wait returned: 985
Child status: 00000000 00001111
status: exit=0, sig=15, core=0
```

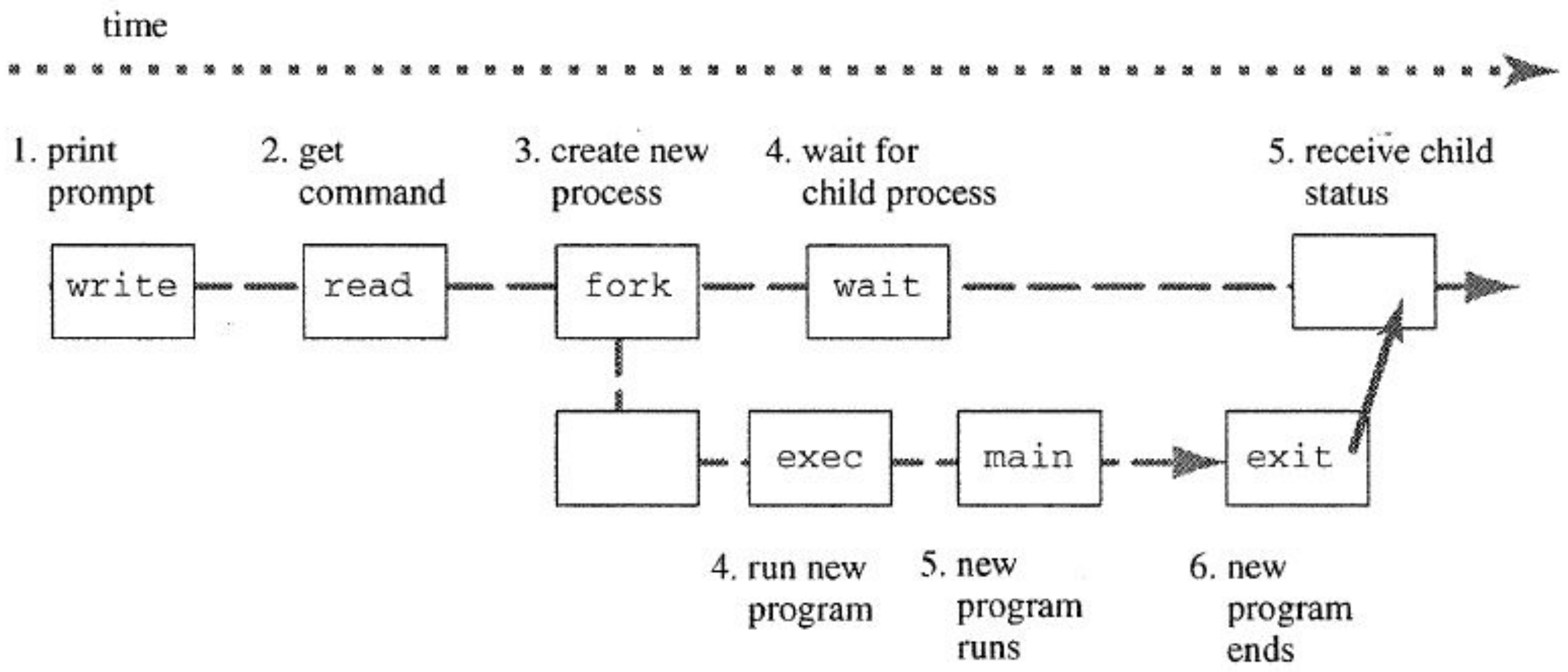
Detailed Signals (from Ch. 7)



SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt
SIGQUIT	3	Core	Quit
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGCHLD	18	Ignore	Child Status
SIGPWR	19	Ignore	Power Fail/Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	21	Ignore	Urgent Socket Condition
SIGPOLL	22	Ignore	Socket I/O Possible
SIGSTOP	23	Stop	Stopped (signal)
SIGTSTP	24	Stop	Stopped (user)
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input)
SIGTTOU	27	Stop	Stopped (tty output)
SIGVTALRM	28	Exit	Virtual Timer Expired
SIGPROF	29	Exit	Profiling Timer Expired
SIGXCPU	30	Core	CPU time limit exceeded
SIGXFSZ	31	Core	File size limit exceeded
SIGWAITING	32	Ignore	All LWPs blocked
SIGLWP	33	Ignore	Virtual Interprocessor Interrupt for Threads Library
SIGAIO	34	Ignore	Asynchronous I/O

Summary: How the Shell Runs Programs?

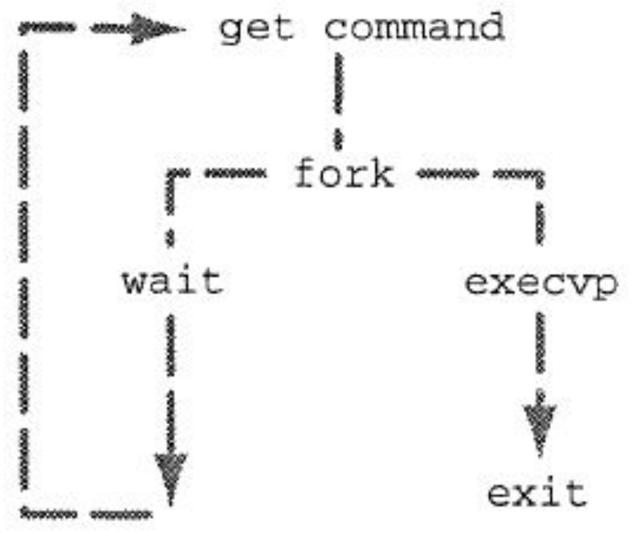
- The shell uses `fork` to create a new process.
  - It then uses `exec` to run the program in the new process.
  - It finally uses `wait` to wait until the new process finishes running the command, getting notified of the new one' status about its termination



< Shell loop with `fork()`, `exec()`, and `wait()` >

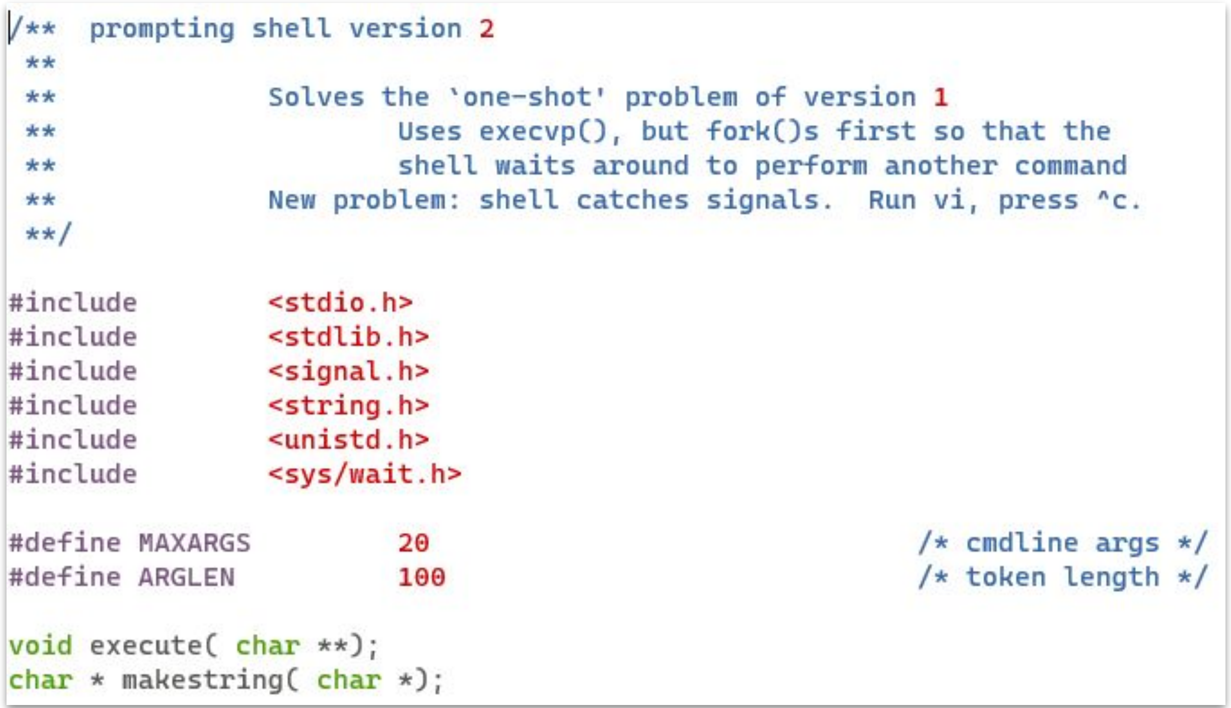
2) Writing the Second Shell: `psh2.c`

- A Shell loop for next command

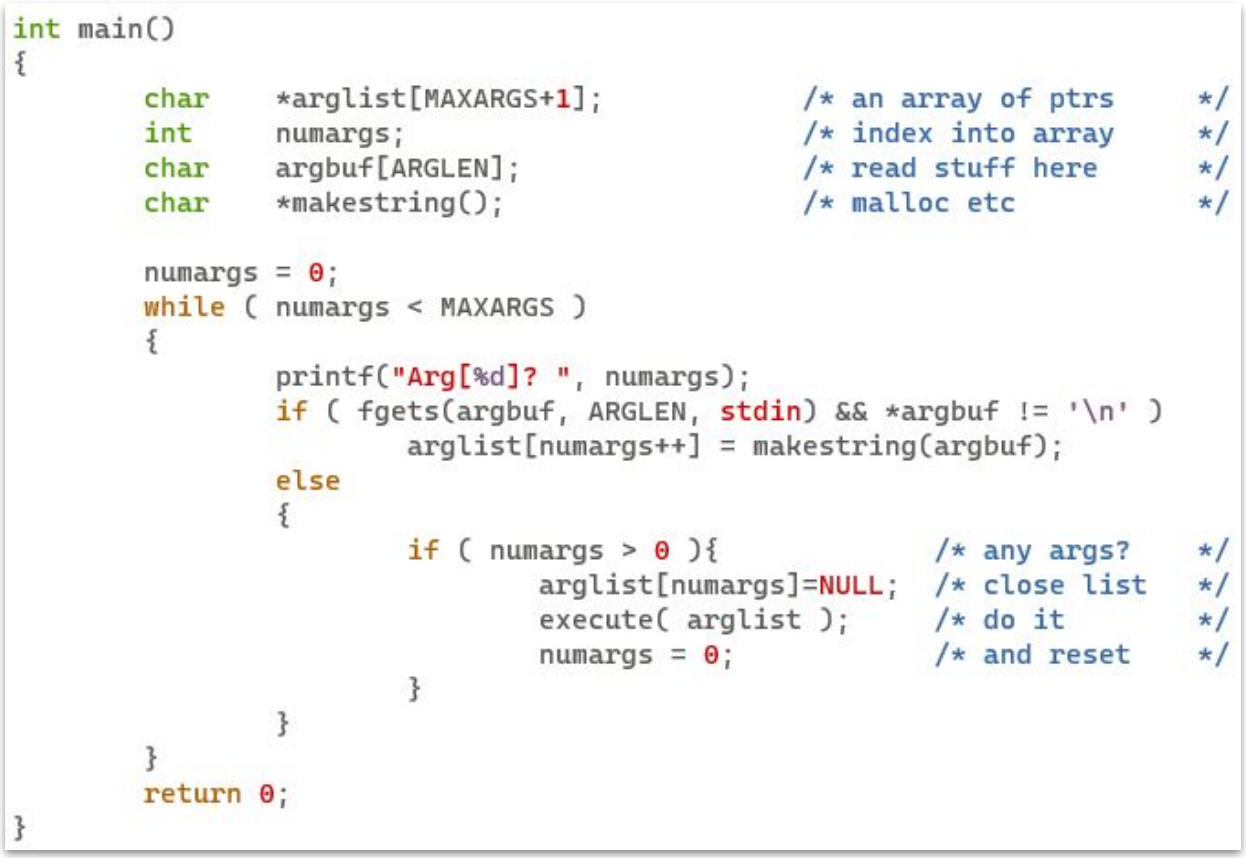


< Logic of a basic Unix shell >

Writing a Shell: `psh2.c`

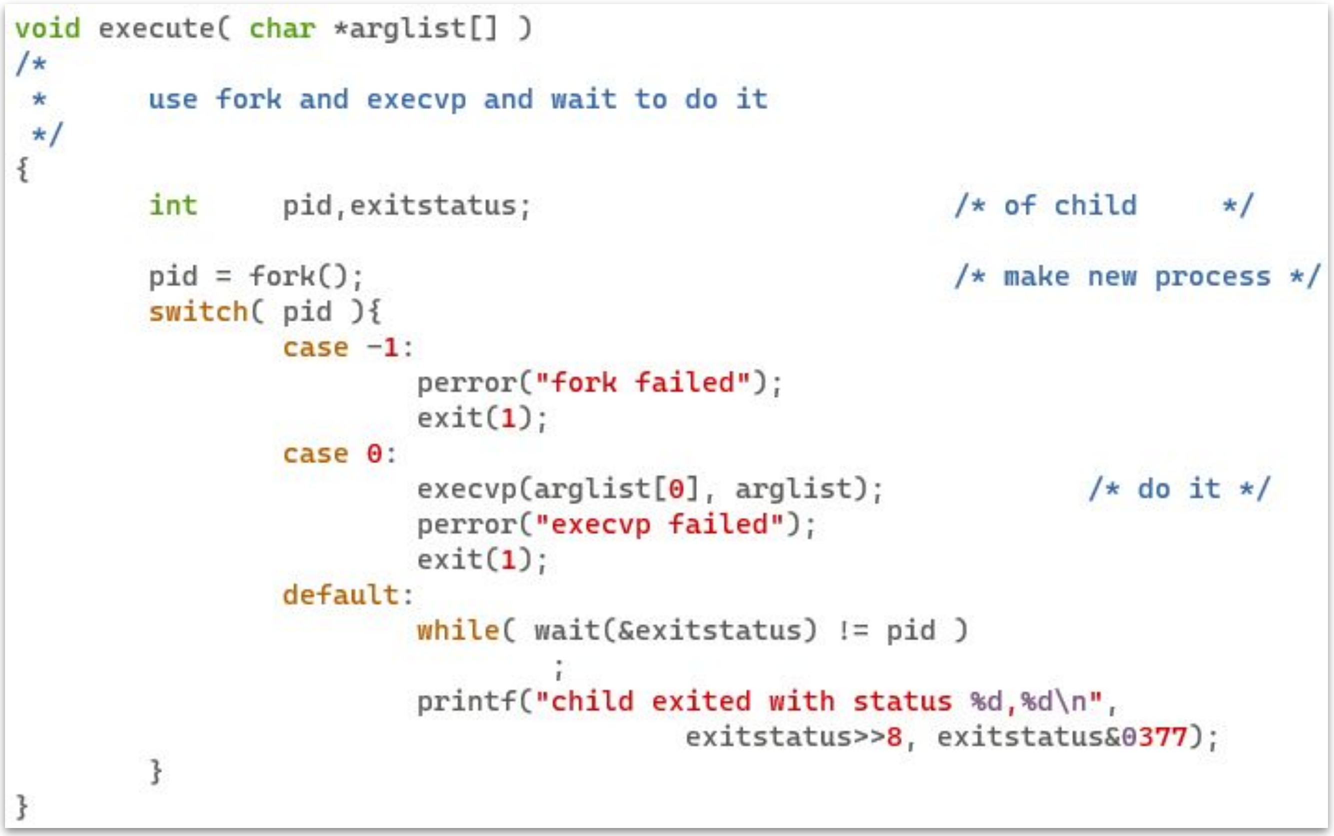


```c
/**  prompting shell version 2
 **
 **          Solves the ‘one-shot' problem of version 1
 **                  Uses execvp(), but fork()s first so that the
 **                  shell waits around to perform another command
 **          New problem: shell catches signals.  Run vi, press ^c.
 **/

#include        <stdio.h>
#include        <stdlib.h>
#include        <signal.h>
#include        <string.h>
#include        <unistd.h>
#include        <sys/wait.h>

#define MAXARGS         20                              /* cmdline args */
#define ARGLEN          100                             /* token length */

void execute( char **);
char * makestring( char *);
```

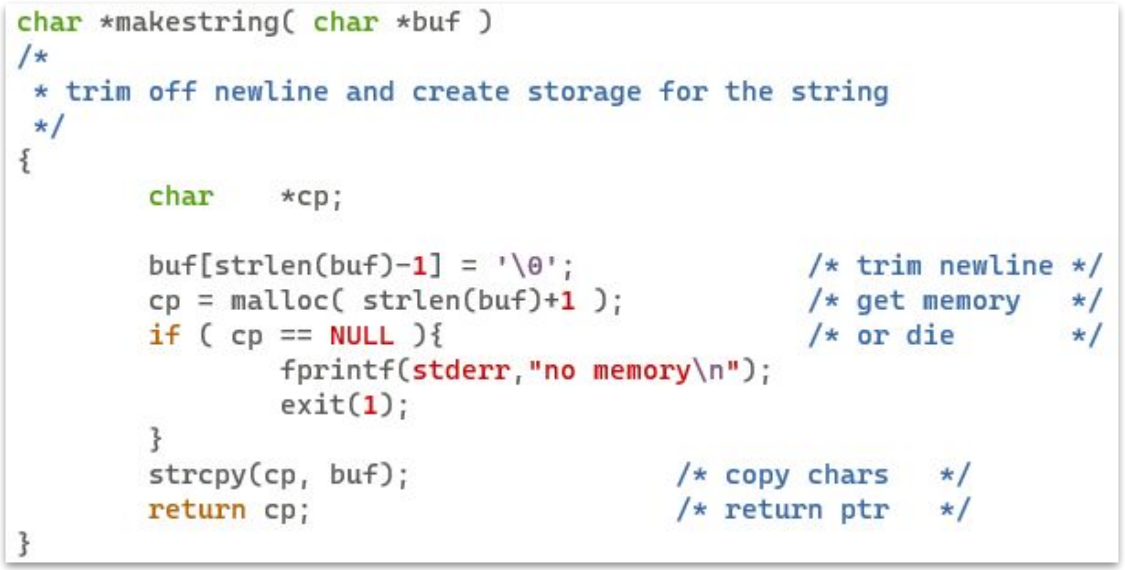


```c
int main()
{
        char    *arglist[MAXARGS+1];            /* an array of ptrs     */
        int     numargs;                        /* index into array     */
        char    argbuf[ARGLEN];                 /* read stuff here      */
        char    *makestring();                  /* malloc 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;
}
```

Writing a Shell: `psh2.c` (cont.)



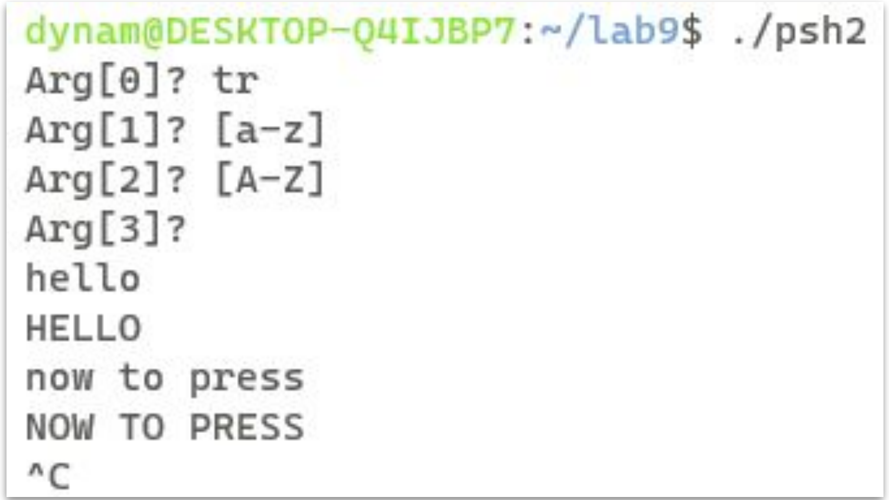
```
void execute( char *arglist[] )
/*
 *      use fork and execvp and wait to do it
 */
{
        int     pid,exitstatus;                         /* of child     */

        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);
        }
}
```



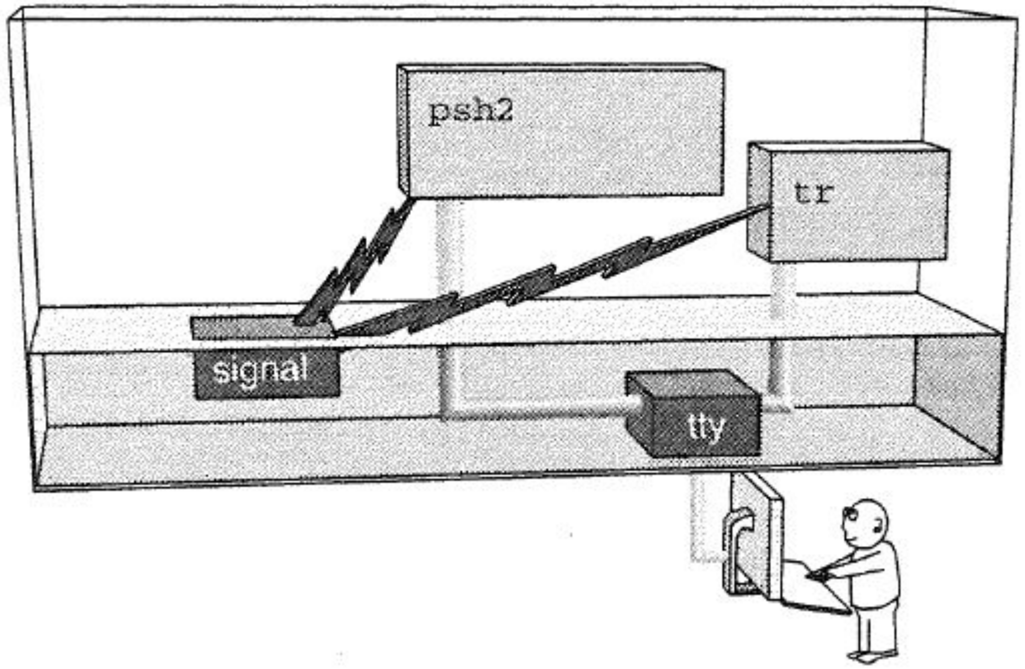
```
char *makestring( char *buf )
/*
 * trim off newline and create storage for the string
 */
{
        char    *cp;

        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   */
}
```

Writing a Shell: `psh2.c` (cont.)

- What happens if CTRL-C arrives when psh2 is waiting?
  - SIGINT is sent to all processes controlled by the terminal running `psh2` and `tr`



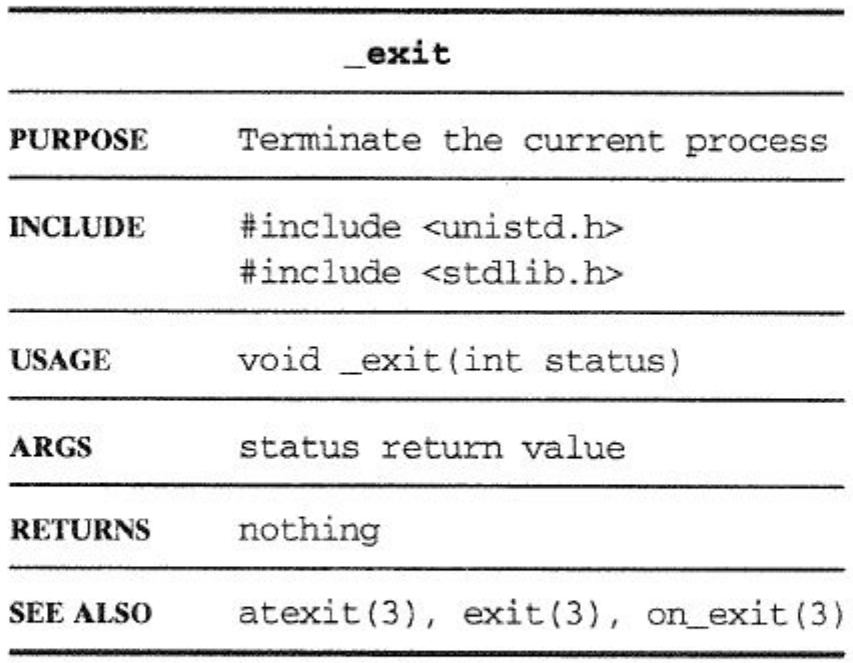
```
dynam@DESKTOP-Q4IJBP7:~/lab9$ ./psh2
Arg[0]? tr
Arg[1]? [a-z]
Arg[2]? [A-Z]
Arg[3]?
hello
HELLO
now to press
NOW TO PRESS
^C
```



< Keyboard signals go to all attached processes >

Death of a Process: `exit` and `_exit`

- The process can stop running by exit.
  - Cf. fork creates a new process.
- How the exit system call works is in the following:
  - (a) Flush all the streams,
  - (b) Call functions that have been registered with `atexit()` and `on_exit()`
  - (c) Perform any other functions associated with exit for the current system,
  - (d) Then call the system call `_exit()`
- `_exit()`: kernel operation (c.f., close(fd))
  - Deallocates all the memory assigned to the finishing process,
  - Closes all files the finishing process has open, and
  - Frees all data structures needed by the kernel for other processes



	_exit
PURPOSE	Terminate the current process

INCLUDE	#include <unistd.h>
	#include <stdlib.h>

USAGE	void _exit(int status)
ARGS	status return value
RETURNS	nothing
SEE ALSO	atexit(3), exit(3), on_exit(3)

Death of a Process: `exit` and `_exit` (cont.)

- What happens to the argument passed to exit by the child process?
  - That value is stored in the kernel until the parent of the process retrieves the value via `wait()`.
  - If the parent is NOT currently waiting, the exit value remains in the kernel until the parent calls wait to be notified of the termination of the child process
    - Then the parent receives the exit value.
- Zombie process: the one that has died with exit but still has an uncollected exit value.
  - Also known as defunct processes
  - Typically occurring for child processes
  - Once the exit value is retrieved via wait, the zombie process is gone
  - Typically caused by an error or a resource leak

The Detailed Procedural Steps of `_exit()`

- 1. Terminates the current process
- 2. Performs all required clean-up operations
- 3. These clean-up operations vary across different versions of Unix/Linux, but all include the following:
    - (i) Close all file descriptors and directory descriptors
    - (ii) If a parent process exits before its children do, children continue to run, not becoming orphans
        - Change the parent PID of all its children to the PID of the init process (the very first process at boot-up) so that the children can continue to run
        - So they become children of the init process
    - (iii) Notify the parent if it is running wait or waitpid
    - (iv) Send SIGCHLD to the parent
        - The default action of SIGCHLD, though, is to be ignored

Summary

- A shell runs programs by calling `fork`, `exec`, and `wait`
- Linux runs a program by loading the executable code into a process and running that code
  - A process is memory space and other system resources required to run a program
- Each running program runs in its own process
  - A process has a unique pid, an owner, a size, and other properties

Summary (cont.)

- The fork system call creates a new (or, child) process by making an almost exact replica of the calling process
- A program loads and executes a new program in the current process by calling the `exec` family
- A program can wait for a child process to terminate by calling `wait`
- A caller can pass a list of strings to main in the new program, which can send back a small char value via `exit`