Process Control

Advanced Unix Programming

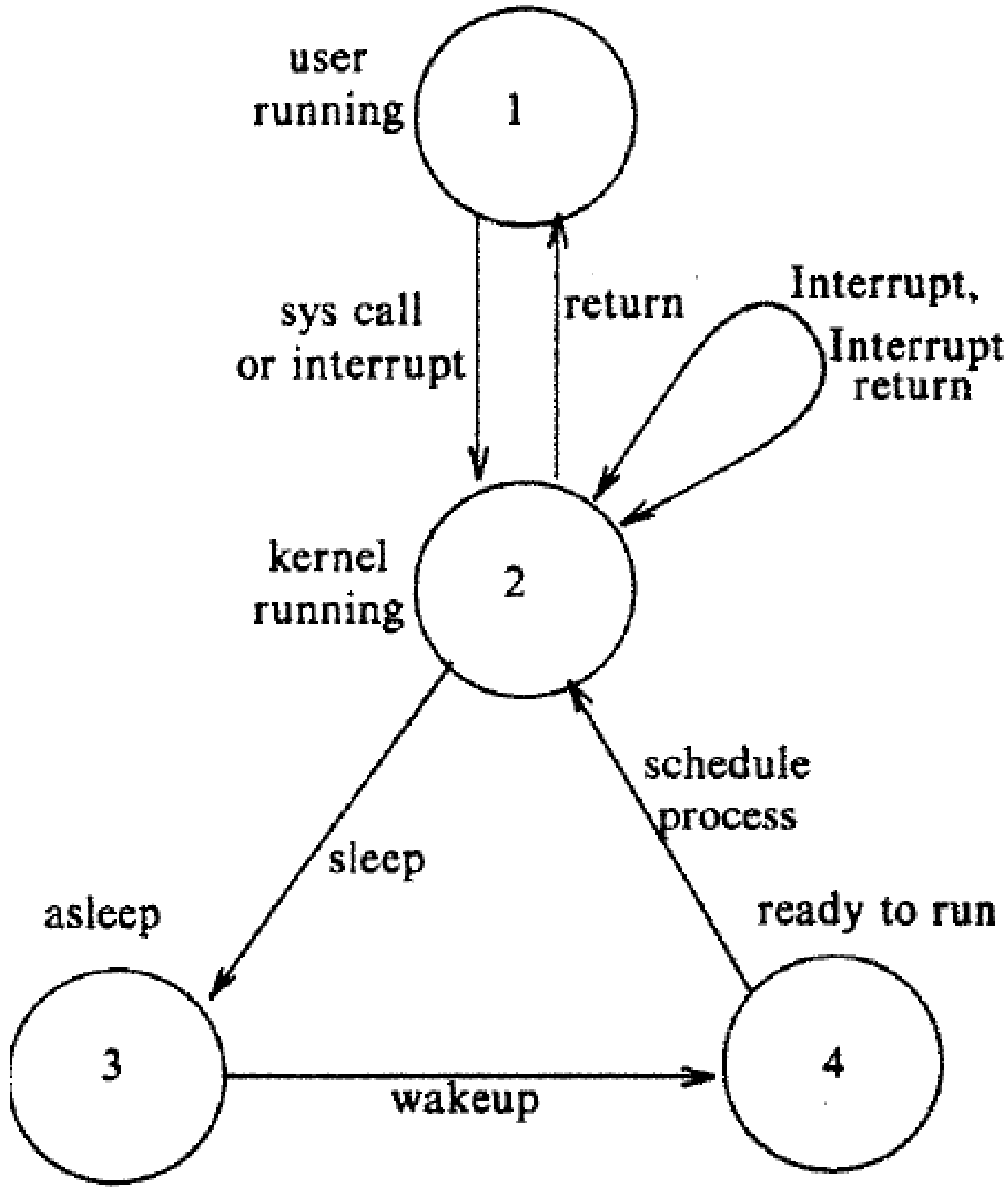
Dr. Jibi Abraham

Books

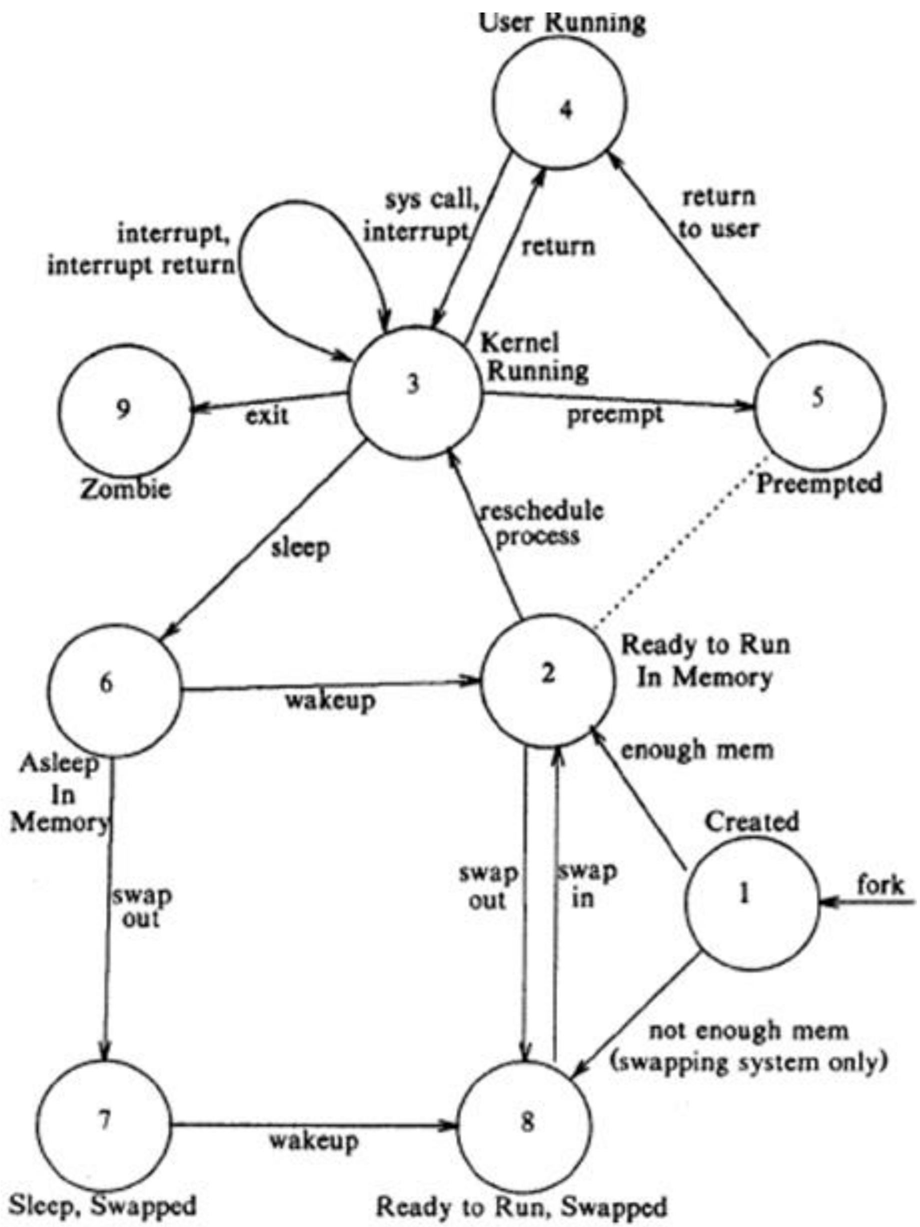
Advanced Programming in the UNIX Environment, Richard Stevens

Design of the Unix Operating System, Maurice J Bach

Process State Diagram



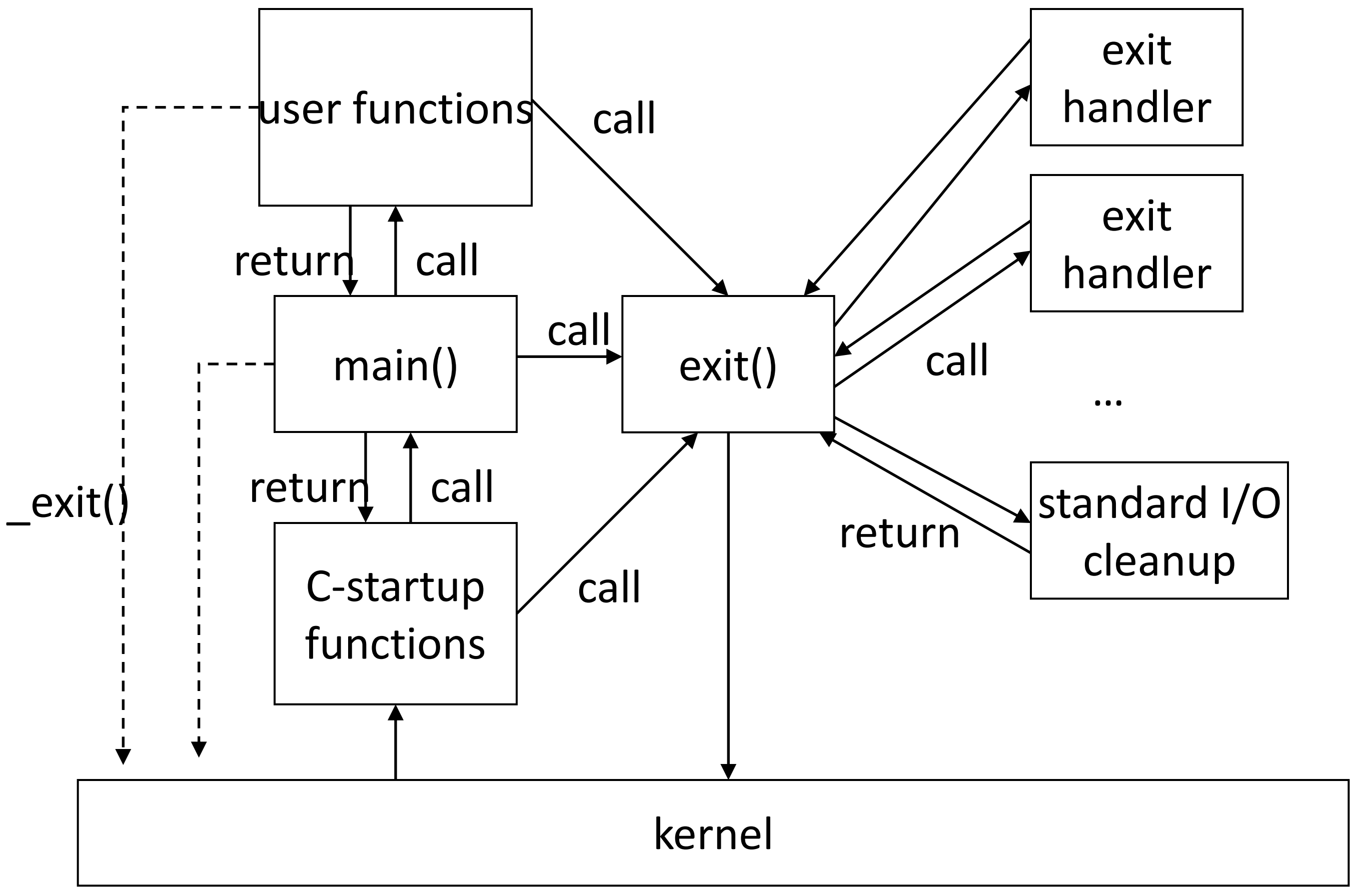
Process State Diagram



Set of process States

1. The process is executing in user mode.
2. The process is executing in kernel mode
3. The process is not executing but is ready to run as soon as the kernel schedules it.
4. The process is sleeping and resides in main memory.
5. The process is ready to run, but the swapper (process 0) must swap the process into main memory before the kernel can schedule it to execute.
6. The process is sleeping, and the swapper has swapped the process to secondary storage to make room for other processes in main memory.
7. The process is returning from the kernel to user mode, but the kernel preempts it and does a context switch to schedule another process.
8. The process is newly created and is in a transition state; the process exists, but it is not ready to run, nor is it sleeping. This state is the start state for all processes except process 0.
9. The process executed the exit system call and is in the zombie state. The process no longer exists, but it leaves a record containing an exit code and some timing statistics for its parent process to collect. The zombie state is the final state of a process.

How a C program Starts and Terminates



C Program Execution

- C program is executed by the kernel by forking and invoking one of the exec functions
- Start-up routine takes values from the kernel—the command-line arguments and the environment—and sets things up so that the main function is called

Ways for a process to Terminate

- Normal termination occurs in five ways:

  1. Return from main

  2. Calling exit

  3. Calling _exit or _Exit

- Abnormal termination occurs in three ways:

  1. Calling abort

  2. Receipt of a signal

- The start-up routine is also written so that if the main function returns, the exit function is called.

```
exit(main(argc, argv));
```

_exit function

- void _exit(int status);
- _exit returns to the kernel immediately without calling exit handlers
- integer argument to the function is the termination status which is available to the parent of this process(to print with echo $?)

```
main()
{
    printf("hello, world\n");
}
```

```
$ cc hello.c
$ ./a.out
hello, world
$ echo $?
13
```

exit function

- void exit(int status);
- exit, which performs certain cleanup processing and then returns to the kernel.
  - clean shutdown of the standard I/O library: the fclose function is called for all open streams

_Exit function

- void _Exit(int status);
- _Exit and _exit are synonymous
- _exit is specified by POSIX.1
- ISO C defines _Exit

Exit handler

- void atexit(void (*func)(void));

  returns: 0 if OK, nonzero on error

- Register exit handler
  - Register a function that is called when a program is terminated
  - Function does not take any arguments and does not return anything
  - A process can register max of 32 functions
  - Called in reverse order of registration

Exit handler

```
static void my_exit1(void),
   my_exit2(void);

int main(void) {
  if (atexit(my_exit2) != 0)
    perror("can't register my_exit2");
  if (atexit(my_exit1) != 0)
    perror("can't register my_exit1");
  if (atexit(my_exit1) != 0)
    perror("can't register my_exit1");
  printf("main is done\n");
  return 0;
}

static void my_exit1(void) {
  printf("first exit handler\n");
}

static void my_exit2(void) {
  printf("second exit handler\n");
}
```

Output :

main is done
first exit handler
first exit handler
second exit handler

Environment Variables

- Like command line arguments to a program execution, environment list is also sent to a program during execution
- Historically, most UNIX systems have provided a third argument to the main function that is the address of the environment list:

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

```
$ env
USER=ysmoon
LOGNAME=ysmoon
HOME=/home/prof/ysmoon
PATH=/bin:/usr/bin:/usr/local/bin:/usr/ccs/bin:/usr/ucb:/usr/ope
   nwin/bin:/etc:.
SHELL=/bin/csh
...
...
```

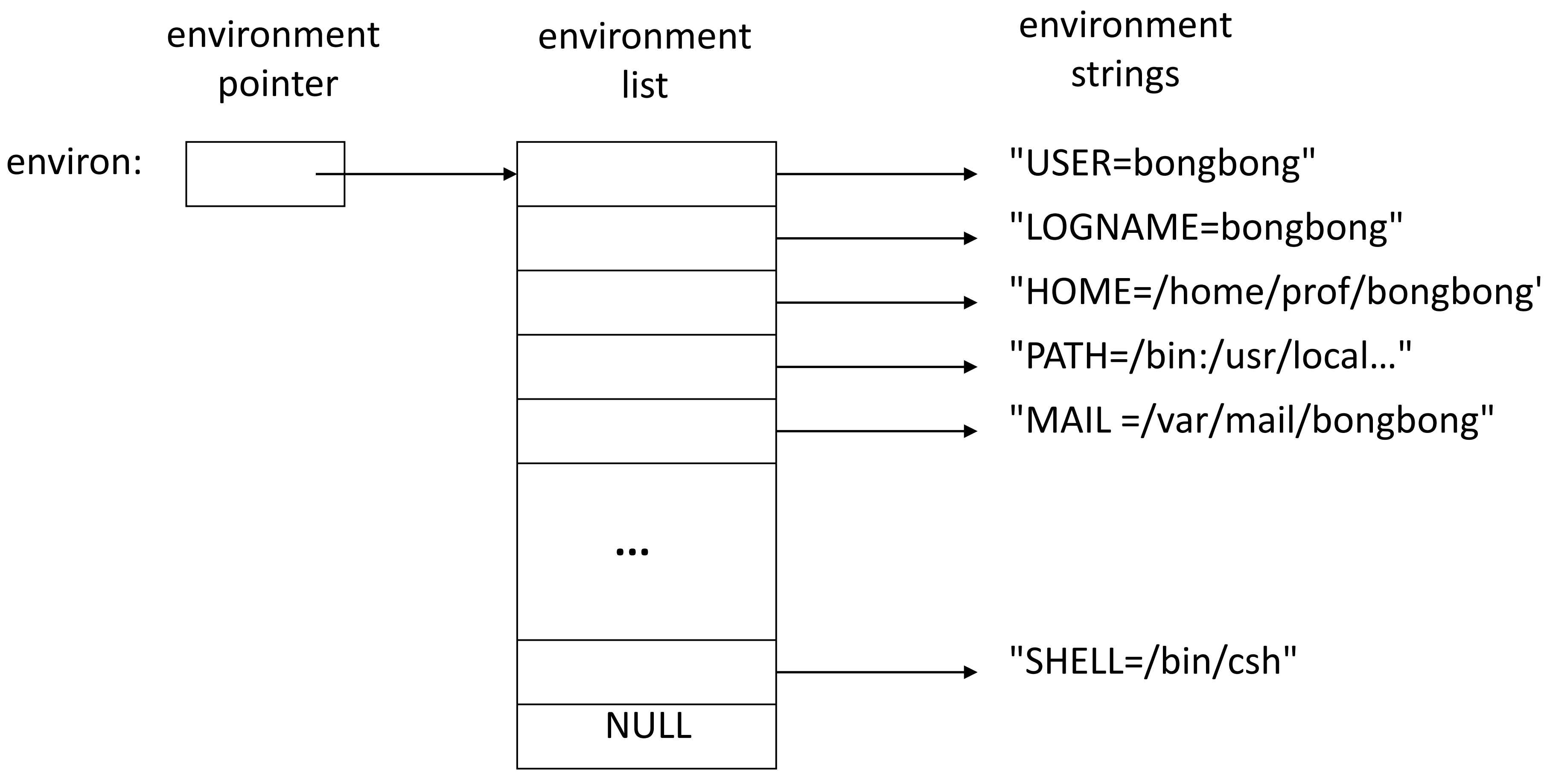
Environment list

- Environment variables are accessed through global variable environ

```
extern char ** environ;
```

- Each element has a form of “Name=Value”
  - Each string ends with ‘\0’
  - Last element of environ is NULL pointer

Environment list



getenv/putenv

- **char *getenv(const char *name);**

  Returns : a pointer to the value of a name=value string associated with name, NULL if not found
- **int putenv(const char *str);**

  Returns: 0 if OK, nonzero on error
  - The putenv function takes a string of the form name=value and places it in the environment list. If name already exists, its old definition is first removed

setenv/unsetenv

- **int setenv(const char *name, const char *value, int rewrite);**

  Returns: 0 if OK, nonzero on error

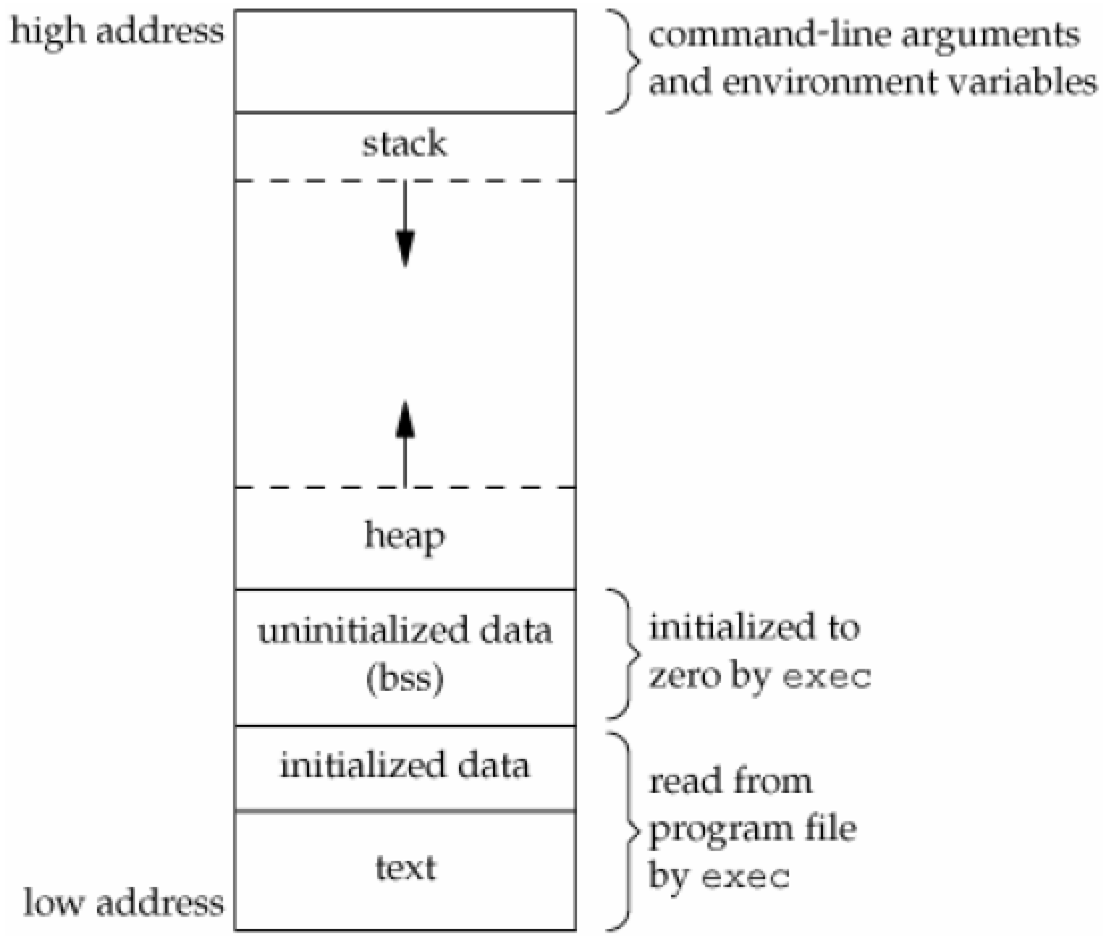
  - The setenv function sets name to value.
  - If name already exists in the environment, then (a) if rewrite is nonzero, the existing definition for name is first removed; (b) if rewrite is 0, an existing definition for name is not removed, name is not set to the new value, and no error occurs.
- **void unsetenv(const char *name);**

  Returns: 0 if OK, nonzero on error

  - The unsetenv function removes any definition of name. It is not an error if such a definition does not exist.

Memory Layout of C Program

- text segment stores program code
  - a program always starts execution from address 0
  - text segment is protected from writing
- Initialized data eg. int i = 0
- Block started by symbol: initialized to zero or null by kernel. Eg. long sum[100]
- Stack holds automatic variables
- Heap dynamic memory allocation
- Unix command to display text, data and bss




```
#size a.out
 text      data      bss      dec            hex        filename
 1663       376        8            2047          7ff     a.out
```

Updation of Memory upon Env update

- The environment list and the environment strings are stored at the top of a process's memory space, above the stack.
- Deleting a string is simple; simply find the pointer in the environment list and move all subsequent pointers down one.
- Adding a string or modifying an existing string is more difficult. The space at the top of the stack cannot be expanded upward and downward

To Modify an Existing Name

a. If the size of the new value is less than or equal to the size of the existing value, copy the new string over the old

b. If the size of the new value is larger than the old one, malloc room for the new string, copy the new string to this area and then replace the old pointer in the environment list for name with the pointer to this allocated area

To add a new name

- call malloc to allocate the name=value and copy the string to this area.

  a. If it's the first time added a new name, call malloc to obtain room for a new list of pointers. Copy the old environment list to this new area and store a pointer to the name=value string at the end of this list of pointers. Also store a null pointer at the end of this list. Set environ to point to this new list of pointers.

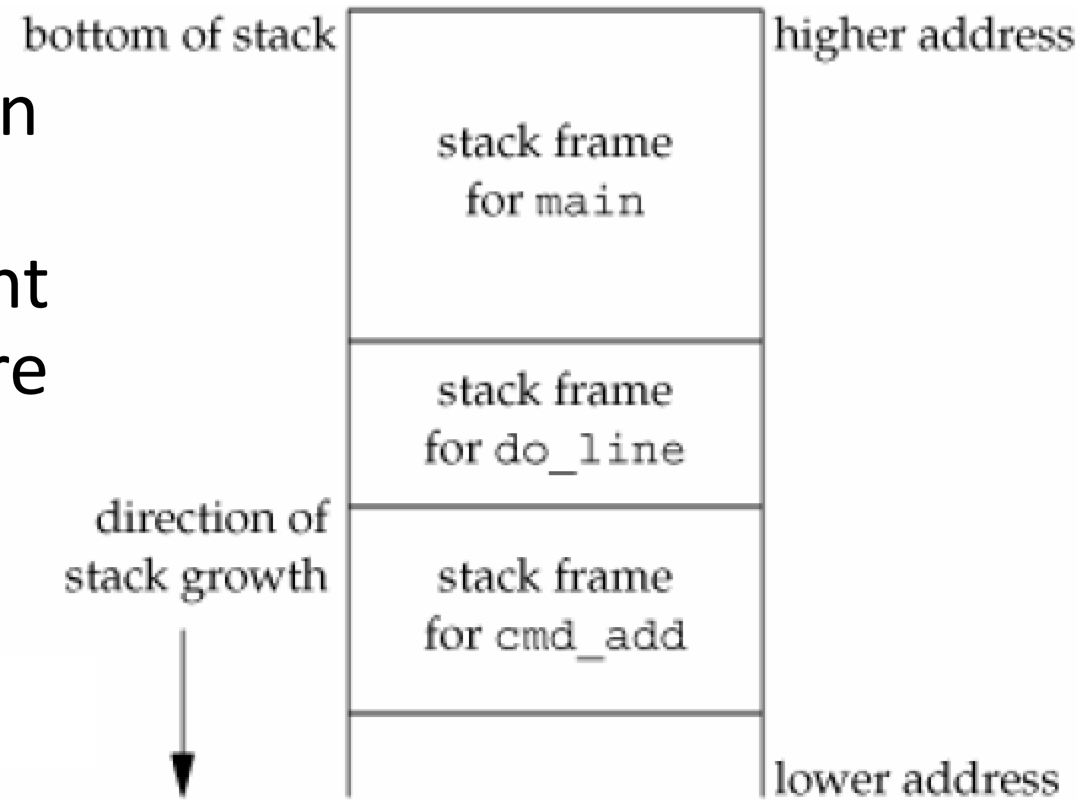
  b. Otherwise, call realloc to allocate room for one more pointer. The pointer to the new name=value string is stored at the end of the list (on top of the previous null pointer), followed by a null pointer.

Deeply Nested Typical Program

```
int main(void){
    char line[MAXLINE];
    while (fgets(line, MAXLINE, stdin) != NULL)
    do_line(line);              exit(0);
}
void do_line(char *ptr) /* process one line of input */{
    int cmd;
    tok_ptr = ptr;
    while ((cmd = get_token()) > 0) {
      switch (cmd) { /* one case for each command */
                case TOK_ADD: cmd_add(); break;}
      }
}
void cmd_add(void) {
    int token;
    token = get_token();
    /* rest of processing for this command */
}
int get_token(void) {
/* fetch next token from line pointed to by tok_ptr */
}
```

Stack frames after *cmd_add* been called

- if the cmd_add function encounters an error—say, an invalid number—it might want to print an error, ignore the rest of the input line and return to the main function to read the next input line
- But when deeply nested numerous levels down from the main function, this is difficult to do in C



setjmp and longjmp Functions

- Solution is to use a nonlocal goto
- int setjmp(jmp_buf env);

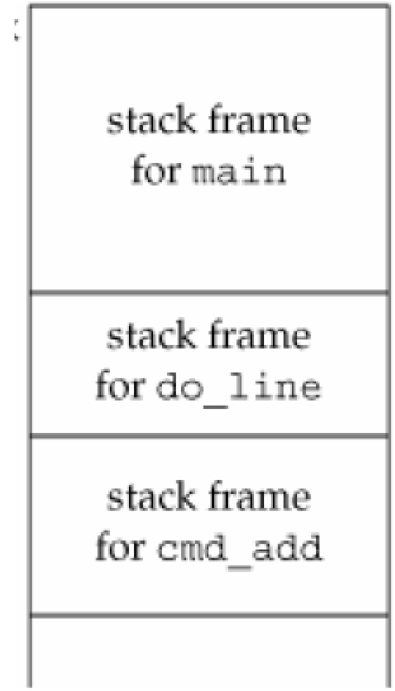
  Returns: 0 if called directly, nonzero if returning from a call to longjmp
- void longjmp(jmp_buf env, int val);
- jmp_buf data type is some form of array that is capable o f holding all the information required to restore the status of the stack to the state when we call longjmp

longjmp

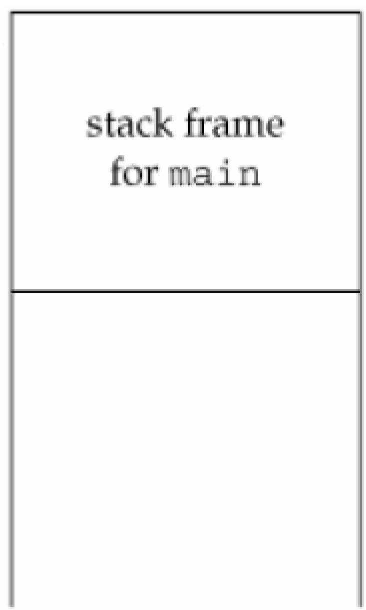
- When an error is encountered—say, in the cmd_add function—call longjmp with two arguments.
  - First is the same env that is used in a call to setjmp and the second, val, is a nonzero value that becomes the return value from setjmp
  - The reason for the second argument is to allow us to have more than one longjmp for each setjmp
  - For example, longjmp can be called from cmd_add with a val of 1 and also from get_token with a val of 2
  - In the main function, the return value from setjmp is either 0 or 1 or 2 and by testing this value, can determine whether the longjmp was from cmd_add or get_token

Example of setjmp and longjmp

```
#define TOK_ADD    5

jmp_buf jmpbuffer;

int
main(void)
{
     char    line[MAXLINE];

     if (setjmp(jmpbuffer) != 0)
         printf("error");
     while (fgets(line, MAXLINE, stdin) != NULL)
        do_line(line);
     exit(0);
}

 ...

void
cmd_add(void)
{
    int     token;

    token = get_token();
    if (token < 0)    /* an error has occurred */
        longjmp(jmpbuffer, 1);
    /* rest of processing for this command */
}
```



Stack frame after `longjmp` has been called



Automatic, Register and Volatile Variables

- "what are the states of the automatic variables and register variables in the main function?“ when main is returned to by the longjmp
- It depends on implementations
- Compile without optimization, store all 3 types of variables into memory
- With optimization, automatic and register variables go to registers and volatile variable stays in memory
- setjmp manual says that **variables stored in the memory will have values as of the time of longjmp, while variables stored in registers are restored to their values** when setjmp was called

Effect of longjmp on variables

```
#include <setjmp.h>
static void f1(int, int, int, int); static jmp_buf  jmpbuffer;
static void f2(void);                static int      globval;
int main(void)
 {
    int autoval; register int    regival;
    volatile int    volaval; static int statval;
    globval = 1; autoval = 2; regival = 3; volaval = 4; statval = 5;
    if (setjmp(jmpbuffer) != 0) {
        printf("after longjmp:\n");
        printf("globval = %d, autoval = %d, regival = %d,"
            " volaval = %d, statval = %d\n",
            globval, autoval, regival, volaval, statval);
        exit(0);
    }
    /*
     * Change variables after setjmp, but before longjmp.
     */
    globval = 95; autoval = 96; regival = 97; volaval = 98;
    statval = 99;
    f1(autoval, regival, volaval, statval); /* never returns */
    exit(0);
 }
```

Contd…

```
static void
f1(int i, int j, int k, int l)
{
     printf("in f1():\n");
     printf("globval = %d, autoval = %d, regival = %d,"
         " volaval = %d, statval = %d\n", globval, i, j, k, l);
     f2();
}
static void
f2(void)
{
     longjmp(jmpbuffer, 1);
}
```

```
$ cc testjmp.c                         compile without any optimisation
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
```

```
$ cc -O testjmp.c                      compile with full optimisation
$ ./a.out
in f1():
globval = 95, autoval = 96, regival = 97, volaval = 98, statval = 99
after longjmp:
globval = 95, autoval = 2, regival = 3, volaval = 98, statval = 99
```

fork() function

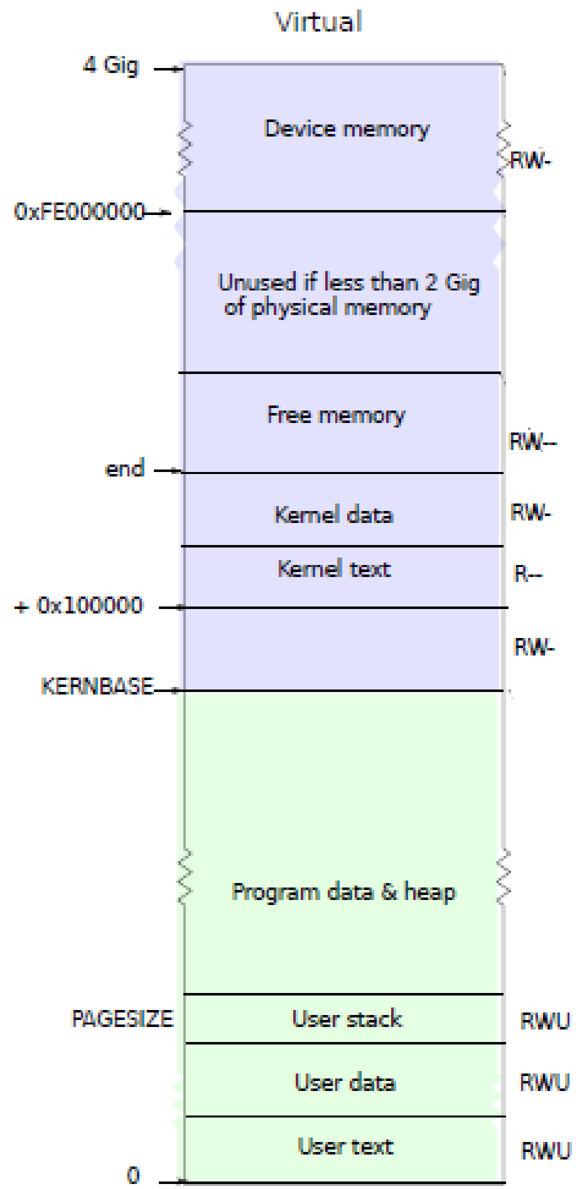
- pid_t fork(void);

  Returns: 0 in child, process ID of child in parent, –1 on error
- An existing process can create a new one by calling the fork function.
- The new process created by fork is called the child process.
- fork is called once but returns twice
- Return value in the child is 0, whereas the return value in the parent is the process ID of the new child
- There is no function at parent that allows to obtain the process IDs of its children
- fork returns 0 to the child because process can have only a single parent, and the child can always call getppid() to obtain the process ID of its parent

Process Address Space

- Both the child and the parent continue executing with the instruction that follows the call to fork.
- Child gets a copy of the parent's data space, heap, and stack.
- Parent and child do not share these portions of memory. The parent and the child share the text segment
- Current implementations use a technique called copy-on-write (COW)
- These regions are shared by the parent and the child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only

Virtual Address Space of a Process



Process Kernel Data Structures

- describe the state of a process
  - process table entry (*process control block (PCB))*: contains fields that must always be accessible to the kernel
    - Permanently resident in memory
  - process user memory (U-*Area*)*:* contains information that the process uses when it is running
    - can be swapped out to disk

Algorithm for fork

input : none

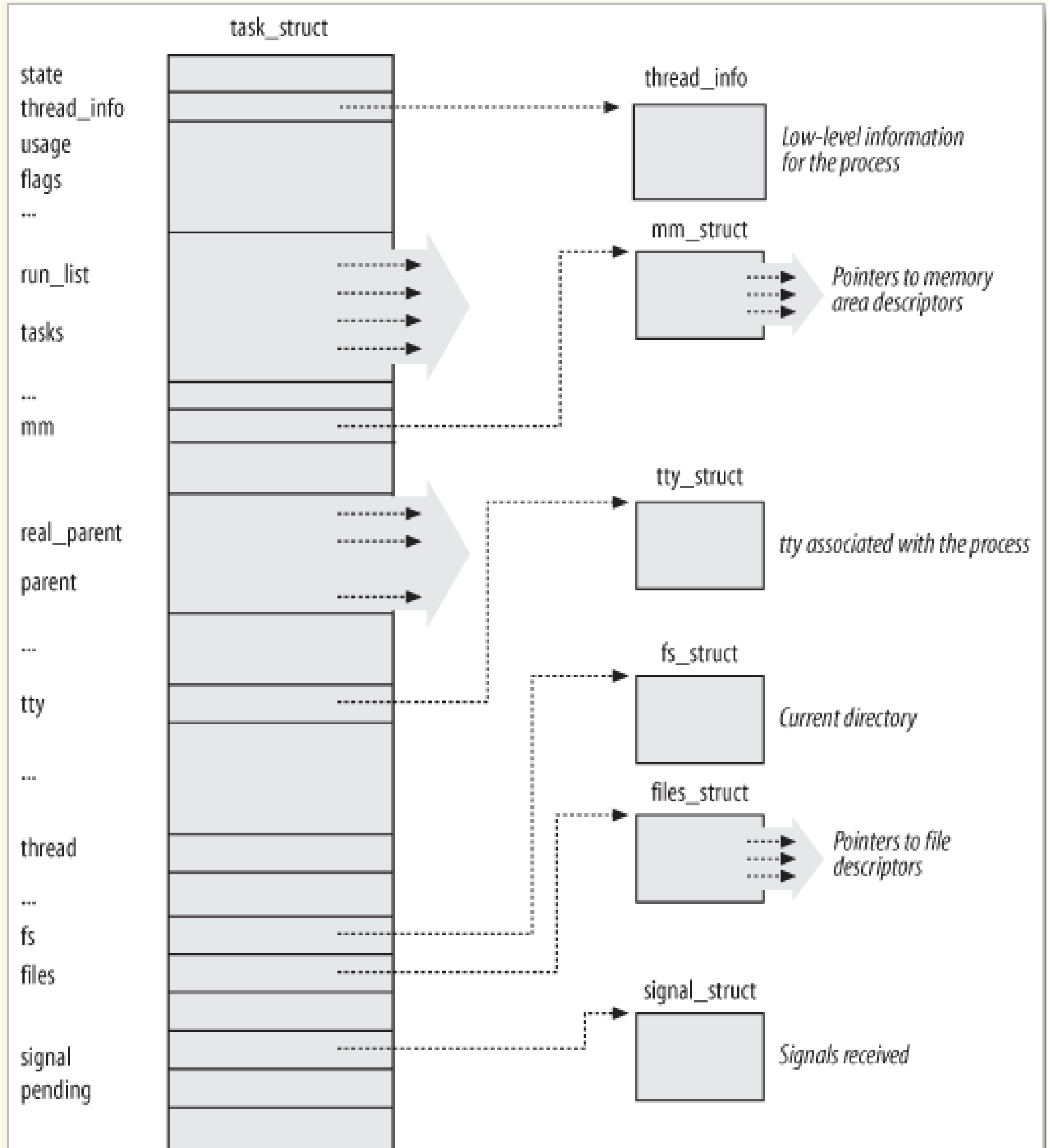
output : to parent process, child PID number; to child process, 0

```
{     check for available kernel resources;
      get free process table slot, unique PID number;
      check that user is not running too many process;
      mark child state "being created";
      copy data from parent process table to new child slot;
      increment open file counts in file table;
      make copy of parent context(U-Area, text, data, stack) in memory;
      push dummy system level context layer onto child system level context; //dummy context
      contains data allowing child process  to recognize itself, and start from here when
      scheduled;
      if ( executing process is parent )
      {       change child state to "ready to run";
              return(child ID);      }
      else    /* executing process is the child process */
      {                    initialize u area timing fields;
              return(0);             }
}
```

Fields in Process Table

- Process identifiers (PID) specify the relationship of processes to each other
- Process State
- process scheduling state: identifies the process state
- Pointers for kernel to locate the process and its u area in main memory or in secondary storage
- process privileges to access to system resources
- Interprocess communication information (flags, signals and messages)
- Timers used for process accounting and for the calculation of process scheduling priority
- Program counter, CPU registers, Memory management information
- IO status information

Example: Process Table



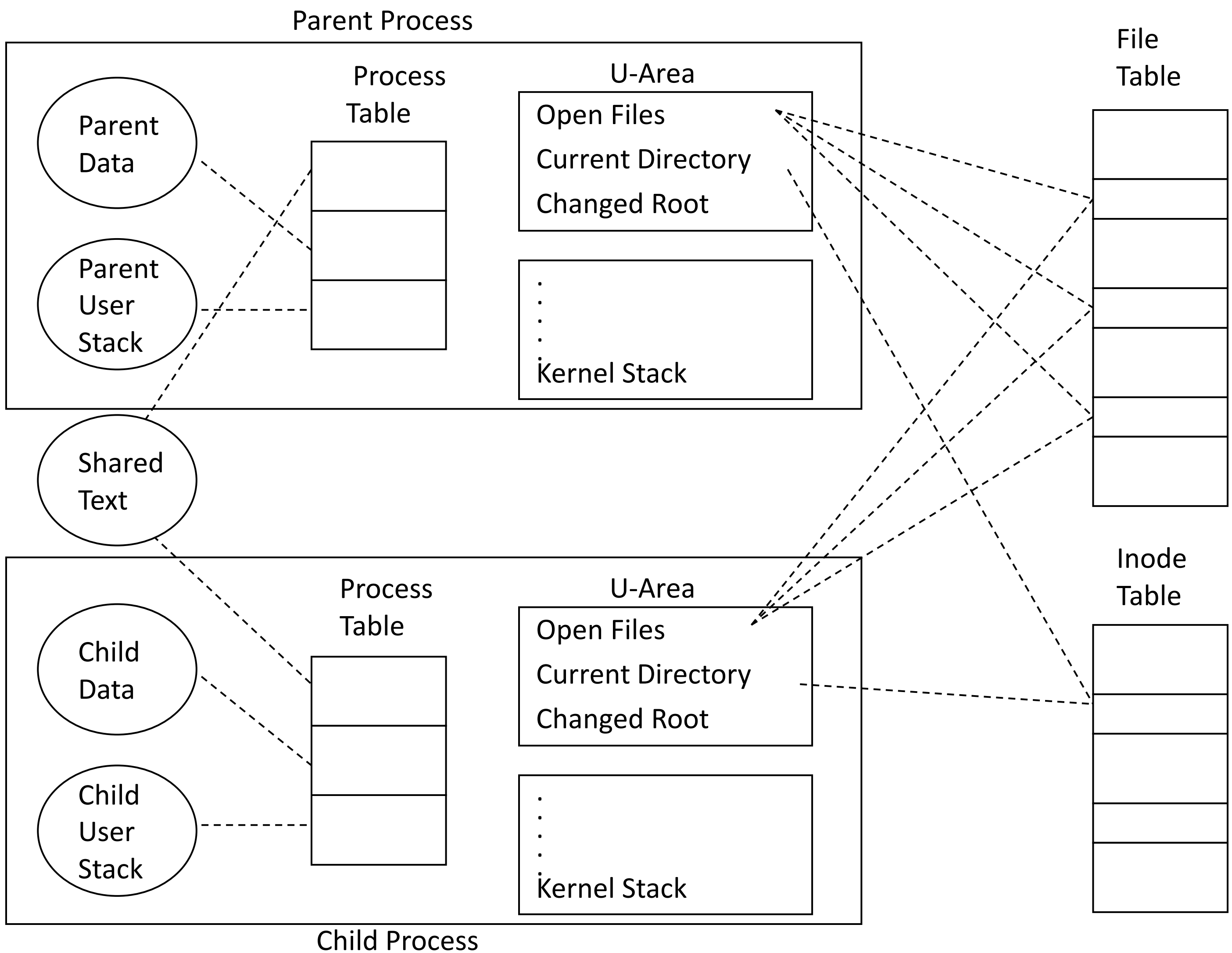
Fields in U-*Area*

- A pointer to the process table identifies the entry
- Real and effective user IDs
- Timer fields record the time the process (and its descendants) spent executing in user mode and in kernel mode
- An array indicates how the process wishes to react to signals
- Control terminal field identifies the "login terminal“
- Error field records errors encountered during a system call
- Return value field contains the result of system calls
- User file descriptor table records the files the process has *open*
- I/O parameters: Amount of data transfer

Example: U-Area

```
> user -f 86
PER PROCESS USER AREA FOR PROCESS 86
USER ID's:        uid: 13297, gid: 1014, real uid: 13297, real gid: 1014
         supplementary gids: 1014 50
PROCESS TIMES:    user: 19, sys: 131, child user: 6085, child sys: 7785
PROCESS MISC:
         command: ksh, psargs: -ksh
         proc: P#86, cntrl tty:  58,6
         start: Fri Jul 15 15:23:21 1994
         mem: 1e5, type: exec su-user
         proc/text lock: none
         current directory: I#360
OPEN FILES AND POFILE FLAGS:
            [ 0]: F#216   r     [ 1]: F#216     w  [ 2]: F#216     w
            [ 3]: F#292   r     [ 4]: F#174        [ 6]: F#156
            [31]: F#246 c r w
FILE I/O:
         u_base: 0x45164c, file offset: 10302696, bytes: 1230
         segment: data, cmask: 0022, ulimit: 2097151
         file mode(s): read
SIGNAL DISPOSITION:
         sig#      signal oldmask sigmask
            1:     0x6ffc    -      1
            2:     0x7718    -      2
...
```

fork Creating a New Process Context



Example of fork file

```
int     glob = 6;       /* external variable in initialized data */
char    buf[] = "a write to stdout\n";

int
main(void)
{
    int         var;    /* automatic variable on the stack */
    pid_t       pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify variables */
        var++;
    } else {
        sleep(2);               /* parent */
    }

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

fork Example (Contd...)

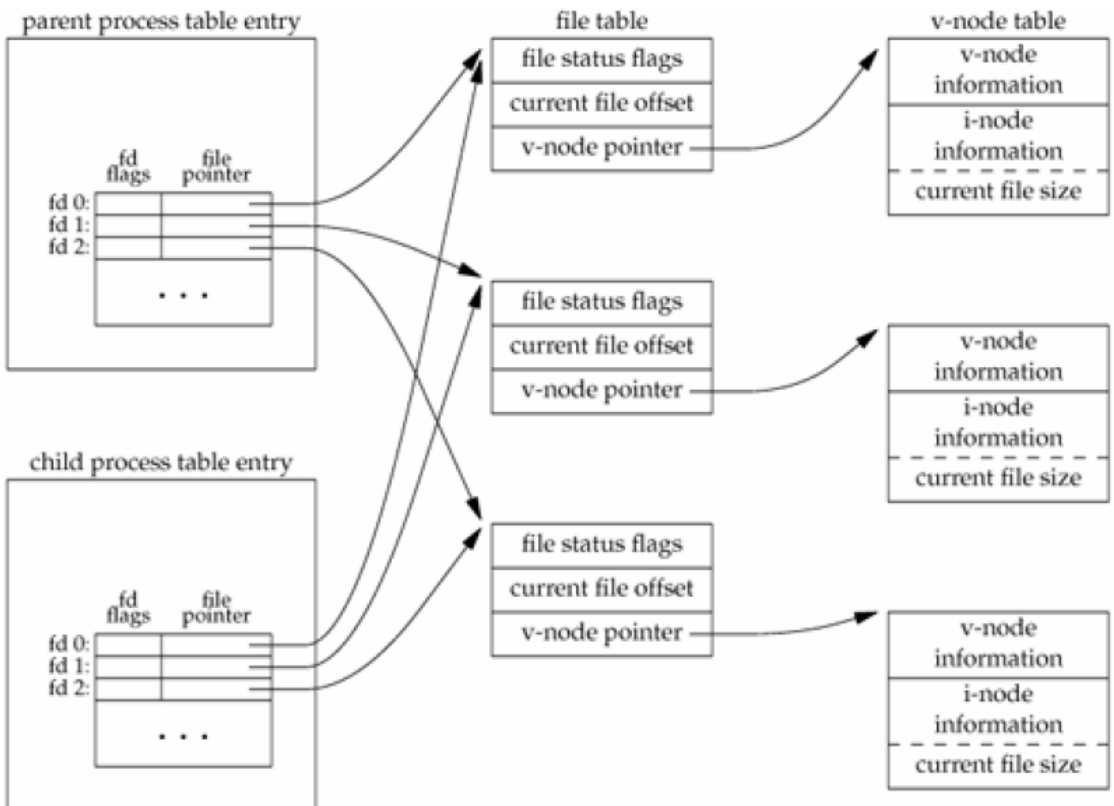
```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89      child's variables were changed
pid = 429, glob = 6, var = 88      parent's copy was not changed
```

```
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

- write function is not buffered
- standard I/O library is buffered
- Standard output is line buffered if it's connected to a terminal device; otherwise, it's fully buffered

File Sharing between Parent and Child

- Parent and the child share the same file offset
- If both parent and child write to the same descriptor, without any form of synchronization, their output will be intermixed



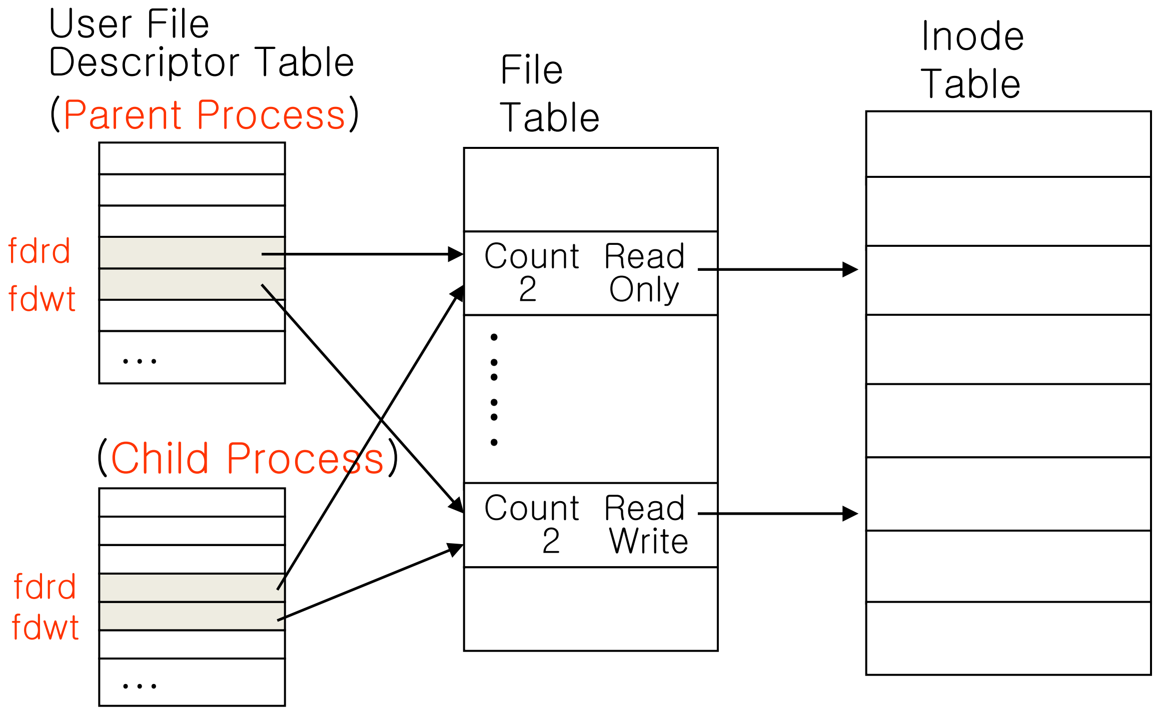
Example of Sharing File Access

```
int fdrd, fdwt;
char c;

main(int argc, char *argv[] )
{
    if ( argc != 3 )    exit(1);
    fdrd=open(argv[1], O_RDONLY));
    fdwt=creat(argv[2], 0666));
    fork();
    rdwt();
    exit(0);
}
```

```
rdwt()
{
    for(;;)
    {
            if (read(fdrd,&c,1)!=-1)
                return;
            write(fdwt,&c,1);
    }
}
```

Example of Sharing File Access(Cont.)

- *fdrd* for both process refer to the file table entry for the source file(argv[1])
- *fdwt* for both process refer to the file table entry for the target file(argv[2])
- Two processes never read or write the same file offset values



Normal File Handling

1. The parent waits for the child to complete.
   - Parent does not need to do anything with its descriptors. When the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly.
2. Both the parent and the child go their own ways.
   - After the fork, the parent closes the descriptors that it doesn't need, and the child does the same thing.
   - Neither interferes with the other's open descriptors.
   - This scenario is often the case with network servers.

Process Identifiers

- Every process has a unique process ID, a nonnegative integer.
- There are some special processes.
  - Process ID 0 is usually the scheduler process and is often known as the *swapper.*
    - It is part of the kernel and is known as a system process.
  - Process ID 1 is usually the *init* process and is invoked by the kernel at the end of the bootstrap procedure.
    - The program file for this process was */etc/init*.
    - This process is responsible for bringing up a Unix system after the kernel has been bootstrapped.
    - *init* usually reads the system-dependent initialization files (the */etc/rc** files) and brings the system to a certain state (such as multi-user).
  - Process ID 2 is the page daemon. This process is responsible for supporting the paging of the virtual memory system.

Process Identifier Related Functions

- pid_t getpid(void);
  Returns: process ID of calling process
- pid_t getppid(void);
  Returns: parent process ID of calling process
- uid_t getuid(void);
  Returns: real user ID of calling process
- uid_t geteuid(void);
  Returns: effective user ID of calling process
- gid_t getgid(void);
  Returns: real group ID of calling process
- gid_t getegid(void);
  Returns: effective group ID of calling process

Properties Inherited by Child

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

Child Properties Different from Parent

- Return value from fork
- Process IDs
- Parent process IDs
- Child's tms_utime, tms_stime, tms_cutime, and tms_cstime values are set to 0
- File locks set by the parent are not inherited by the child
- Pending alarms are cleared for the child
- Pending signals for the child is set to the empty set

Uses for fork

- When a process wants to duplicate itself so that the parent and child can each execute different sections of code at the same time.
  - This is common for network servers, parent waits for a service request from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
- When a process wants to execute a different program.
  - common for shells, the child does an exec right after it returns from the fork.
  - Some OS combine fork followed by an exec into a single operation called a spawn.
  - Separating the two allows the child to change the per-process attributes between the fork and the exec, such as I/O redirection, user ID, signal disposition etc

Main Reasons for *fork* to Fail

a) if there are already too many processes in the system (which usually means something else is wrong), or

b) if the total number of processes for this real user ID exceeds the system’s limit.

Example (1 – with argument 4)

```
int main(int argc, char *argv[])
{  pid_t child_pid=0; int i, n;
   n=atoi(argv[1]);
   for (i=0; i<n; i++)
   if (child_pid=fork())
        break;
   printf("i: %d, PID: %ld,
             PPID:%ld \n",
i, (long)getpid(), (long)getppid());
 }
```

```
Output
i: 0, PID: 100, PPID:shell
i: 1, PID: 101, PPID:1
i: 2, PID: 102, PPID:1
i: 3, PID: 103, PPID:1
```

Example (2)

```
int main(int argc, char *argv[])
{  pid_t child_pid=0; int i, n;
   n=atoi(argv[1]);
   for (i=0; i<n; i++)
   if (child_pid=fork()<=0)
       break;
   printf("i: %d, PID: %ld,
            PPID:%ld \n",
i, (long)getpid(), (long)getppid());
 }
```

```
Output
i: 0, PID: 101, PPID:100
i: 1, PID: 102, PPID:100
i: 2, PID: 103, PPID:100
i: 3, PID: 104, PP ID:100
i: 4, PID: 100, PP ID:shell
```

Example (3)

```
int main(int argc, char *argv[])
{  pid_t child_pid=0; int i, n;
   n=atoi(argv[1]);
   for (i=0; i<n; i++)
   if ((child_pid=fork())==-1)
        break;
   printf("i: %d, PID: %ld,
              PPID:%ld \n",
i, (long)getpid(), (long)getppid());
 }
```

Example (4)

```
int main(void)
{
    pid_t pid = fork();   // fork #1
    pid = fork();         // fork #2
    pid = fork();         // fork #3
    if (pid == 0)
        fork();           // fork #4
    fork();               // fork #5
    printf("pid = ", getpid);
}
```

1. Fork #1 creates an additional processes. You now have two processes.
2. Fork #2 is executed by two processes, creating two processes, for a total of four.
3. Fork #3 is executed by four processes, creating four processes, for a total of eight. Half of those have `pid==0` and half have `pid != 0`
4. Fork #4 is executed by half of the processes created by fork #3 (so, four of them). This creates four additional processes. You now have twelve processes.
5. Fork #5 is executed by all twelve of the remaining processes, creating twelve more processes; you now have twenty-four.