Process Programming (Chap7-Chap9)

Hongik University

Eunsung Jung

Disclaimer: The slides are borrowed from many sources!

Overview

1. What is a Process?
2. `fork()`
3. `exec()`
4. `wait()`
5. File Descriptors across Processes
6. Special Exit Cases
7. IO Redirection

What makes up a Process?

- program code
- machine registers
- global data
- stack
- open files (file descriptors)
- an environment (environment variables; credentials for security)

Some of the Context Information

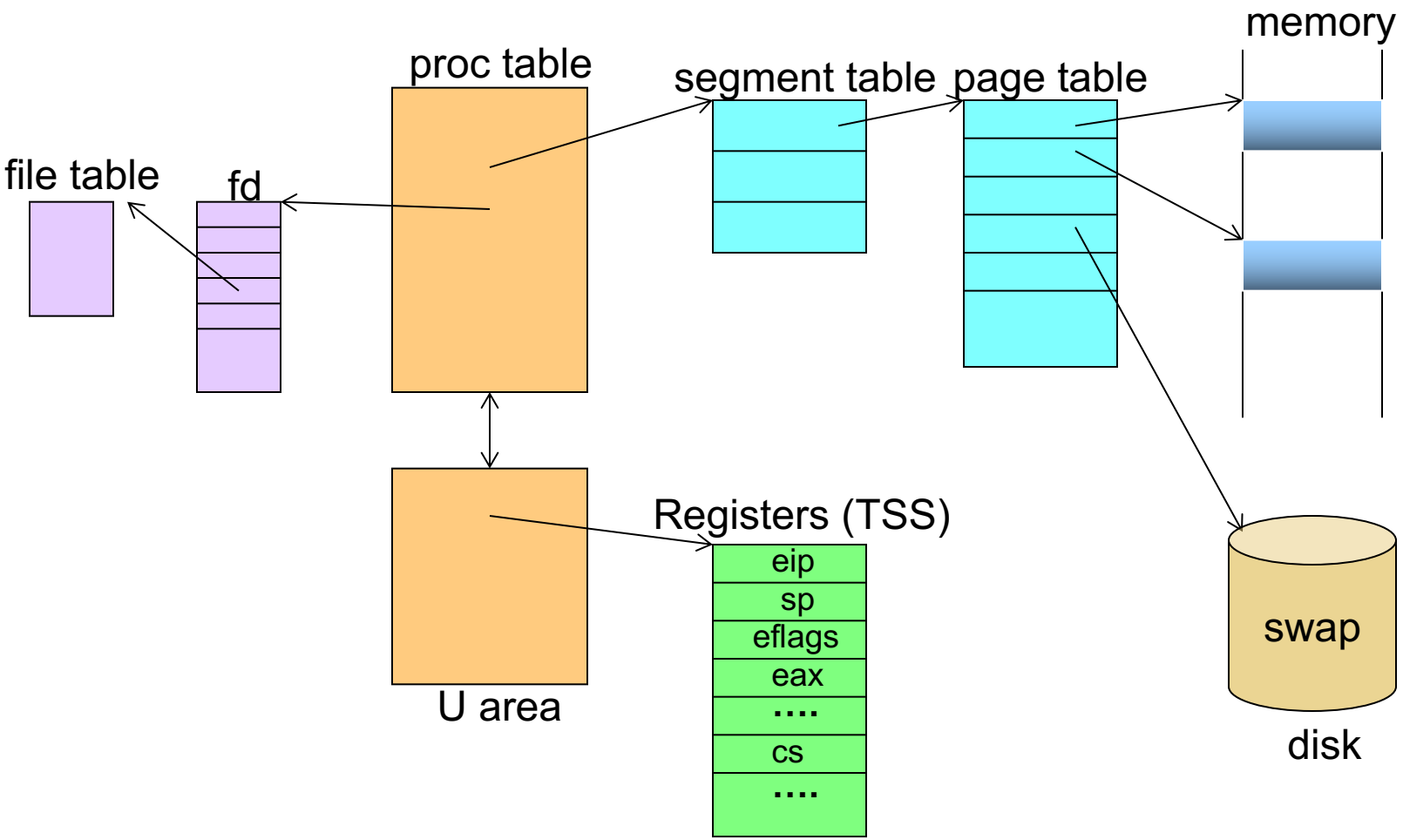
- Process ID (`pid`) — unique integer
- Parent process ID (`ppid`)
- Real User ID — ID of user/process which started this process
- Effective User ID — ID of user who wrote the process' program
- Current directory
- File descriptor table
- Environment — `VAR=VALUE` pairs

- Pointer to program code
- Pointer to data — Memory for global vars
- Pointer to stack — Memory for local vars
- Pointer to heap — Dynamically allocated

- Execution priority
- Signal information

Context

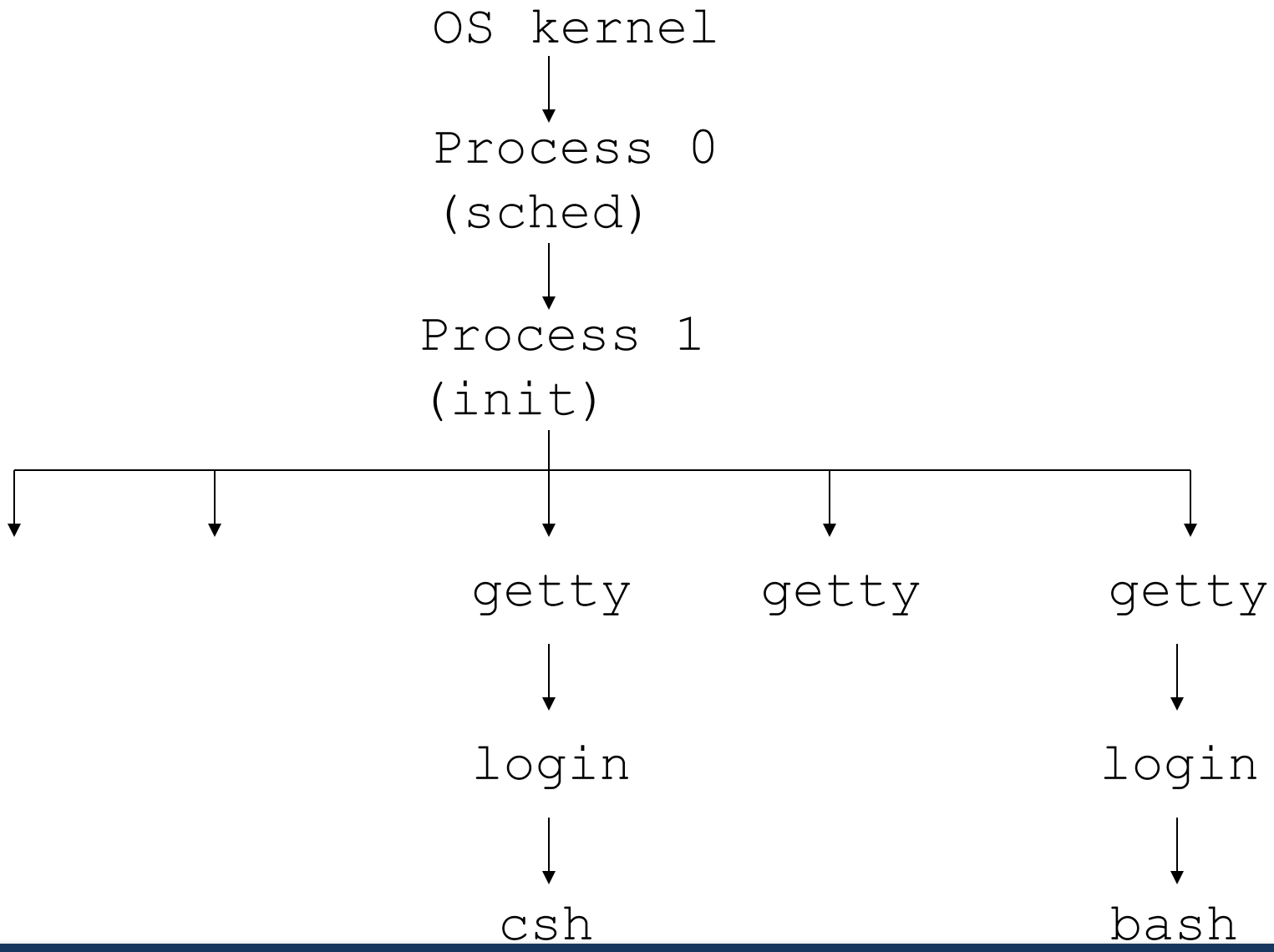
- context : system context, address (memory) context, H/W context



Important System Processes

- init – Mother of all processes. init is started at boot time and is responsible for starting other processes.
  - init uses file inittab & directories: /etc/rc?.d
- getty – login process that manages login sessions.

Unix Start Up Processes Diagram



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*.

- You can get the process pid or parent's pid

```
#include <sys/types>
main()
   {
   pid_t pid, ppid;
   printf( "My  PID is:%d\n\n",(pid = getpid()) );
   printf( "Par PID is:%d\n\n",(ppid = getppid()) );
   }
```

2. fork()

- ```
  #include <sys/types.h>
  #include <unistd.h>
  pid_t fork( void );
  ```

- Creates a child process by making a copy of the parent process --- an **exact** duplicate.
  - Implicitly specifies code, registers, stack, data, files

- Both the child *and* the parent continue running.

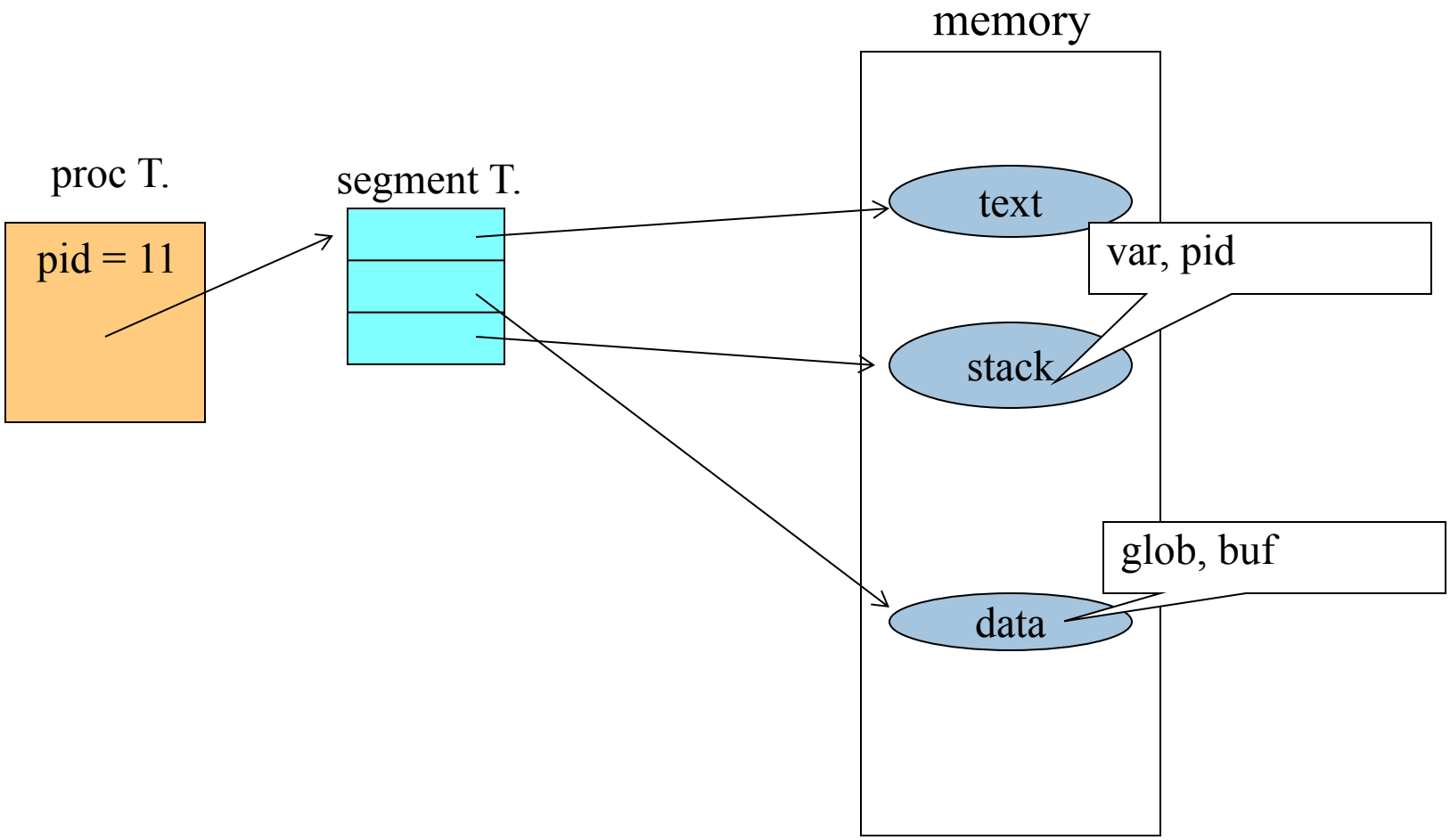
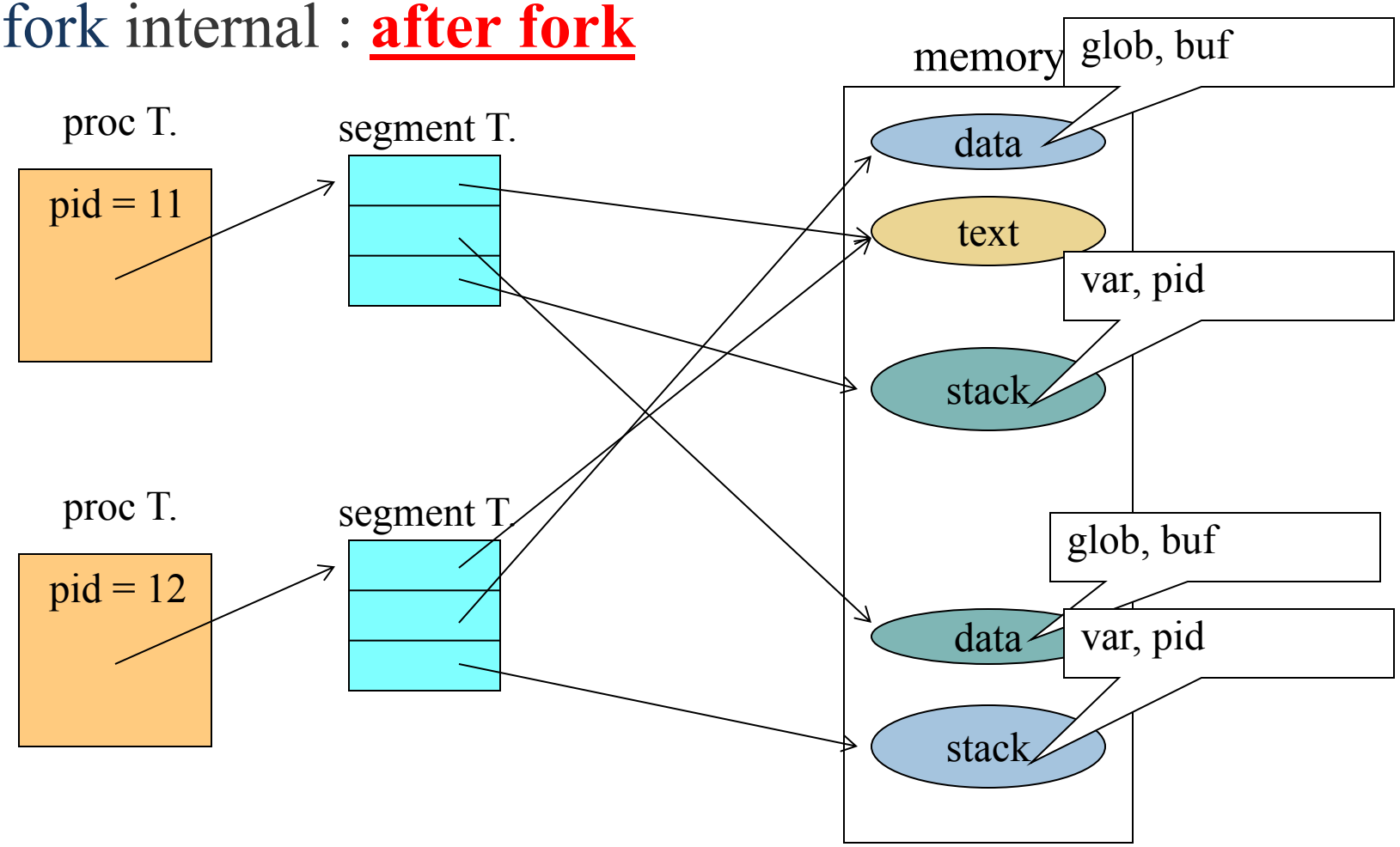

  ```

Process IDs (pids revisited)

- `pid = fork();`
- In the child: `pid == 0;`
  In the parent: `pid` == the process ID of the child.
- A program almost always uses this `pid` difference to do different things in the parent and child.

fork() Example (parchld.c)

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

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

```
else
{
      /* child */
      for( i=0; i < 1000; i++ )
            printf( "CHILD %d\n", i );
      }
   return 0;
 }
```

Possible Output

```
CHILD 0
CHILD 1
CHILD 2
                    PARENT 0
                    PARENT 1
                    PARENT 2
                    PARENT 3
CHILD 3
CHILD 4
                    PARENT 4
          :
```

Things to Note

- `i` is copied between parent and child.
- The switching between the parent and child depends on many factors:
  - machine load, system process scheduling
- Output interleaving is *nondeterministic*
  - cannot determine output by looking at code

LAB7-1: Address context

- *fork example*

```
int  glob = 6;
char buf[] = "a write to stdout\n";

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

     var = 88;
     write(STDOUT_FILENO, buf, sizeof(buf)-1);
     printf("before fork\n");

     if ((pid = fork()) == 0) {     /* child */
          glob++; var++;
     } else
          sleep(2);          /* parent */

     printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
     exit (0);
}                    (Source : Adv. programming in the UNIX Env., pgm 8.1)
```

☛ guess what can we get from this program?

LAB7-1: Address context

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

LAB7-1: Address context

- fork internal : **before fork** (after run a.out)



We assume that there is no paging mechanism in this figure.

LAB7-1: Address context

- fork internal : **after fork**



LAB7-1: Address context

- *Answer*
  - *Note: we never know which of the parent and the child starts first.*

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89
pid = 429, glob = 6, var = 88
$ ./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
```

Only once. Write() is NOT buffered.

Only once. printf() is buffered, but flushed by newline.

child's variables were changed
parent's copy was not changed

Twice. printf() is buffered and NOT flushed. Copied to the child.

3. exec()

- Family of functions for replacing process's program with the one inside the `exec()` call.

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

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

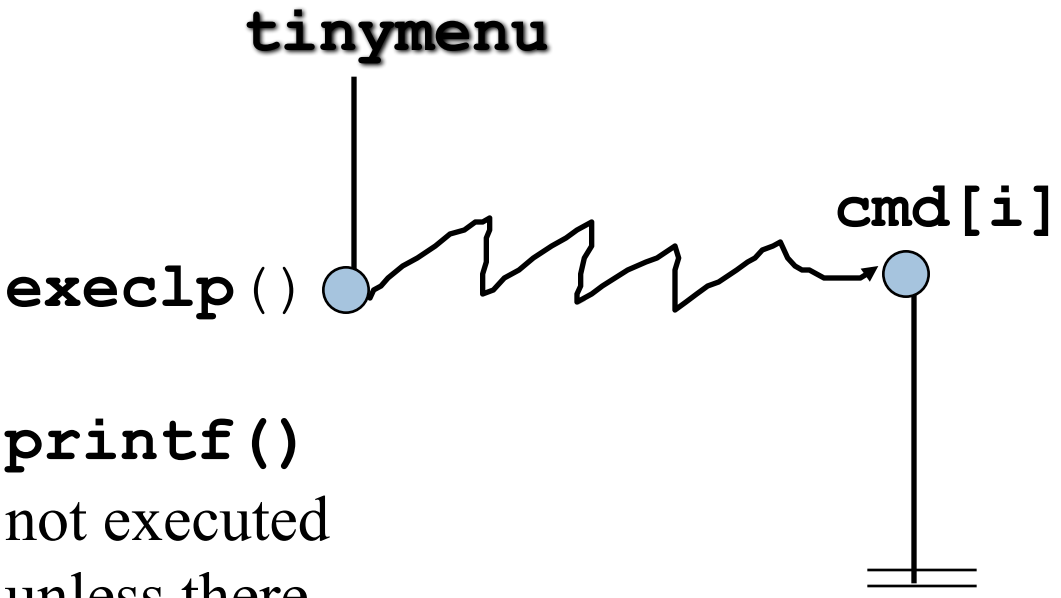
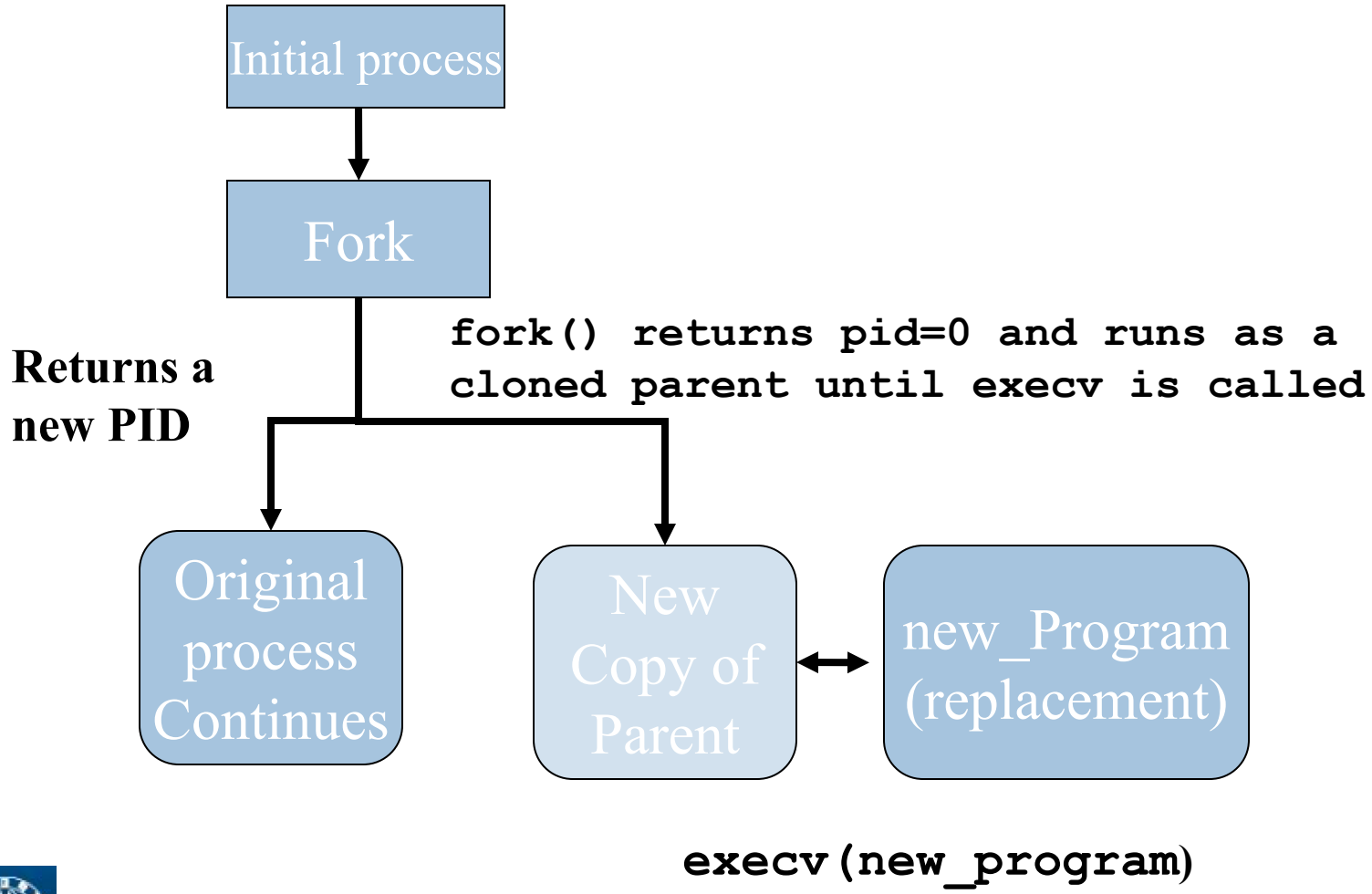
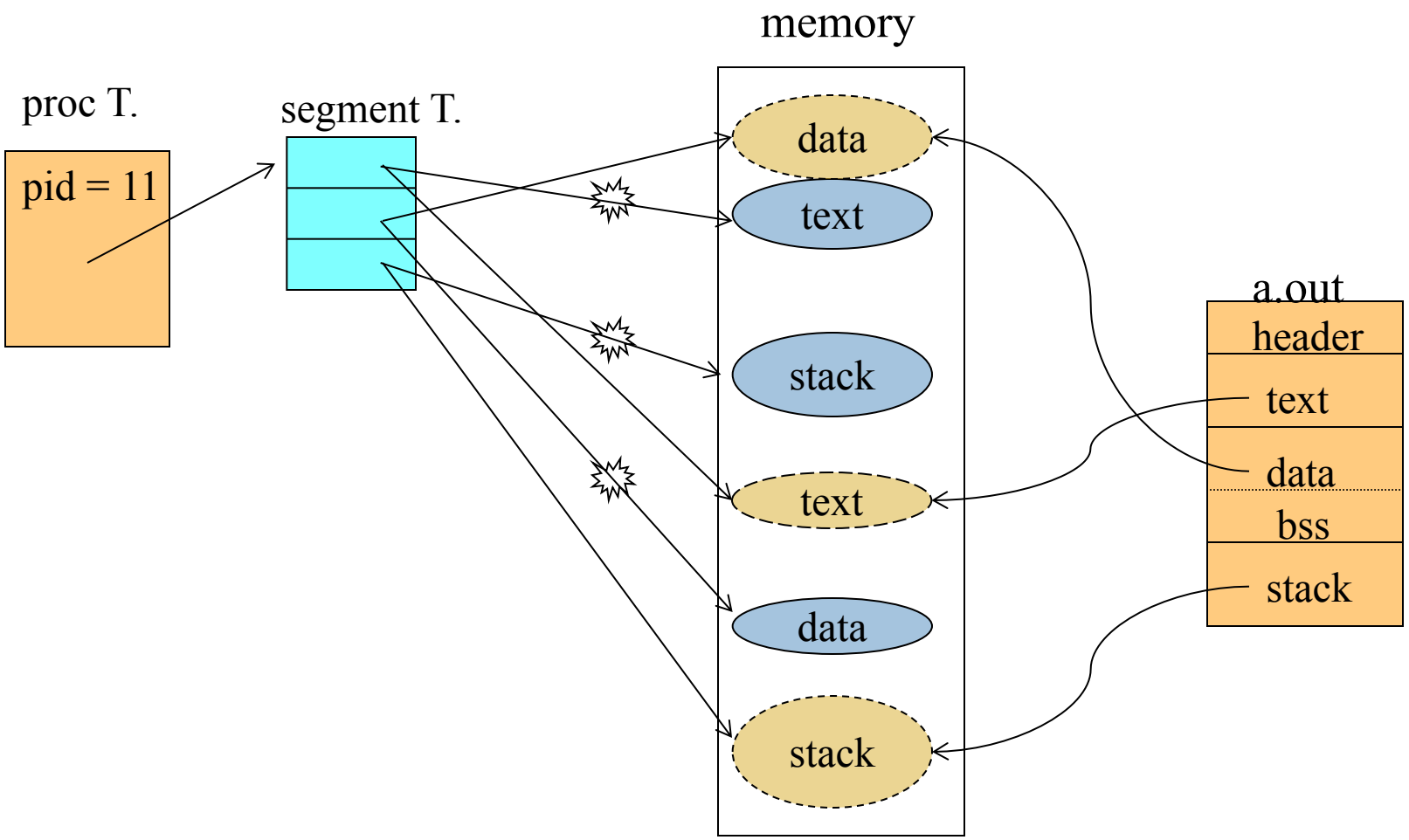
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

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

int fexecve(int fd, char *const argv[], char *const envp[]);
```

All seven return: –1 on error, no return on success

exec(..) Family

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

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

3. exec()

- Example

```
#include <unistd.h>
int execlp(char *file, char *arg0,
           char *arg1, ..., (char *)0);

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

Same as "sort -n foobar"

LAB7-2: tinymenu.c

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

void 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" );
}
```

Execution



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

fork() and execv()

- `execv(new_program, argv[ ])`



Context : address context

- execve internal



4. wait()

- ```
  #include <sys/types.h>
  #include <sys/wait.h>
  pid_t wait(int *statloc);
  ```

- Suspends calling process until child has finished. Returns the process ID of the terminated child if ok, -1 on error.

- **`statloc`** can be `(int *)0` or a variable which will be bound to status info. about the child.


  ```

`wait()` Actions

- A process that calls `wait()` can:
  - *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.

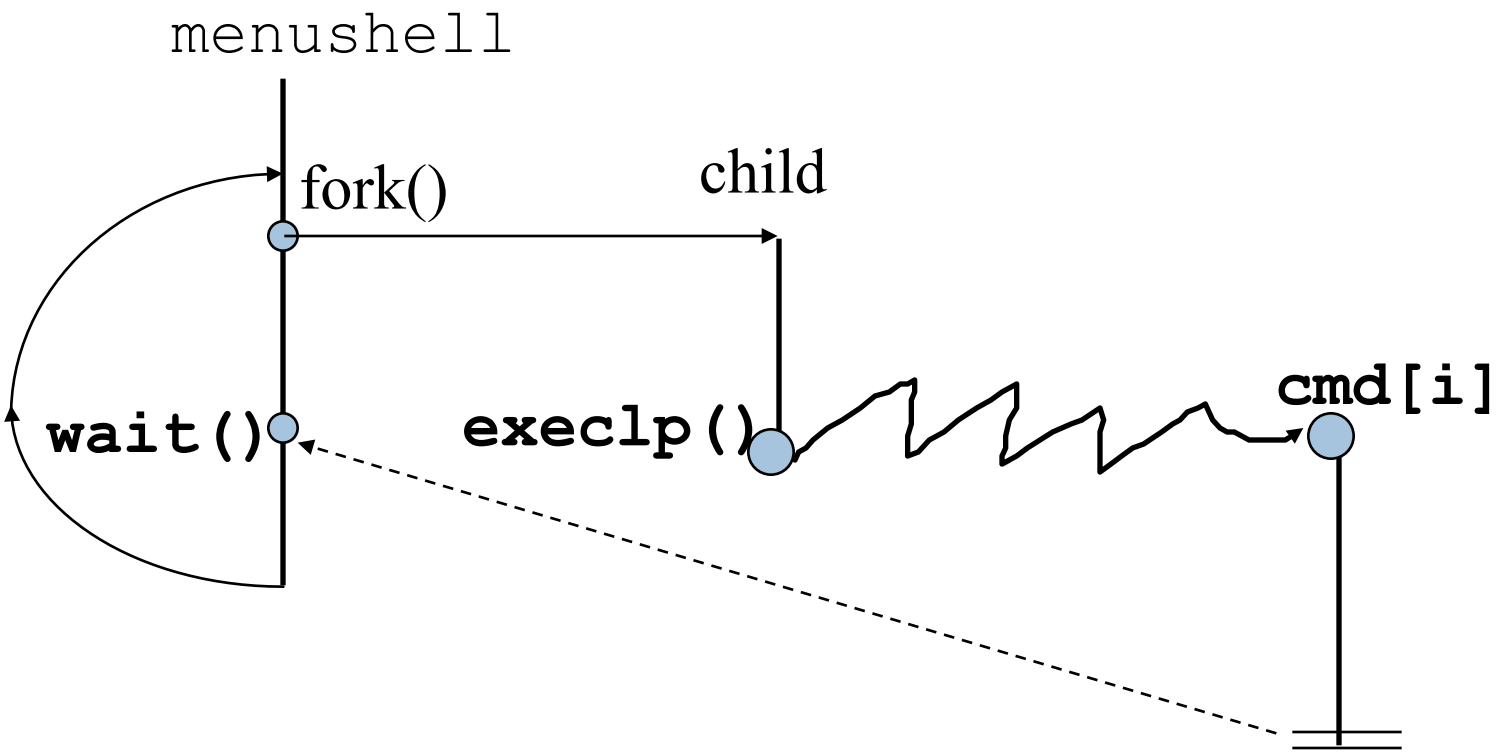
LAB7-3: menushell.c

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

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

continued

```
if(fork() == 0)
    {  /* child */
    execlp( cmd[i], cmd[i], (char *)0 );
    printf( "execlp failed\n" );
    exit(1);
    }
else
    {  /* parent */
    wait( (int *)0 );
    printf( "child finished\n" );
    }
  } /* while */
} /* main */
```

Execution



Macros for wait (1)

- WIFEXITED(*status*)
  - Returns true if the child exited normally.
- WEXITSTATUS(*status*)
  - Evaluates to *the least significant eight 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()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid( pid_t pid, int *status, int opts )
```

- waitpid - can wait for a particular child
- *pid* < -1
  - Wait 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 which wait( ) exhibits.
- *pid* == 0
  - Wait for any child process whose process group ID is equal to that of the calling process.

- *pid* > 0
  - Wait for the child whose process ID is equal to the value of *pid*.
  - *options*
    - Zero or more of the following constants can be ORed.
      - 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 signal).
  - Return value
    - The process ID of the child which exited.
    - -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.

LAB7-4: waitpid

```
#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);
   }
```

```
    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 */
```

5. Process File Descriptors

- A child and parent have copies of the file descriptors, but the R-W pointer is maintained by the system:
  - 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.

LAB7-5: File used across processes (shfile.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
void printpos(char *msg, int fd);
void fatal(char *msg);

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

continued

```
if ((fd=open("data-file", 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 );
}
          :
```

continued

```
  else if( pid > 0 )
  {     /* parent */
     wait((int *)0);
     printpos( "Parent after wait", fd );
  }
  else
     perror( "fork" );
}
```

continued

```
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 );
}
```

Output

```
$ shfile

Before fork: 10
Child before read: 10
Child after read: 20
Parent after wait: 20
```

what's happened?

7. Special Exit Cases

Two special cases:

1. 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()`

2. A parent exits when 1 or more children are still running
   - children are adopted by the system's initialization process (`/etc/init`)
     - it can then monitor/kill them

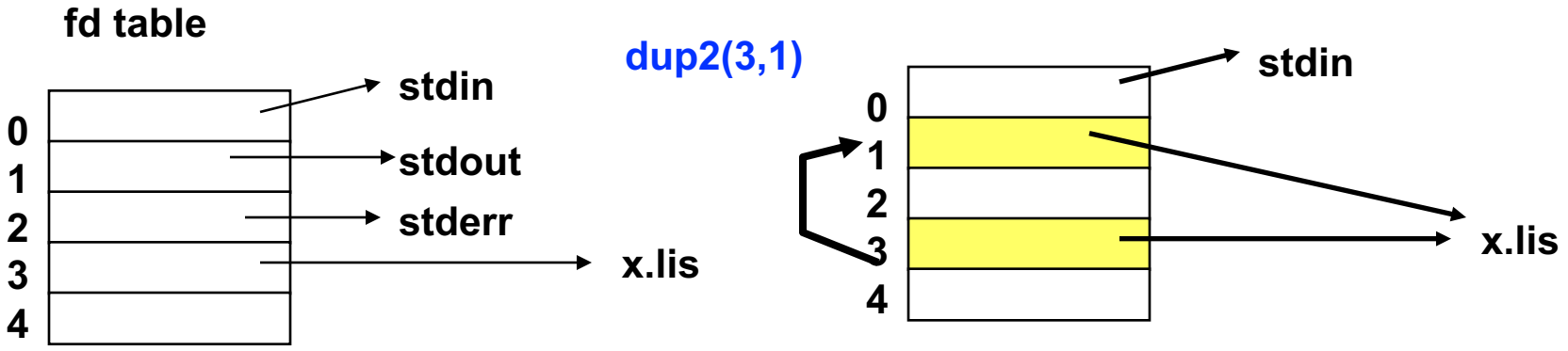
8. I/O redirection

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

Redirection of standard output

- Example implement shell: **ls > x.lis**
- program:
  - Open a new file x.lis
  - Redirect standard output to x.lis using **dup** command
    - everything sent to standard output ends in x.lis
  - execute **ls** in the process

- dup2(int fin, int fout) - copies fin to fout in the fd table



LAB: implement ls > x.lis

```
#include <unistd.h>
int main ()
{
   int fileId;
   fileId = creat( "x.lis",0640 );
   if( fileId < 0 )
   {
       printf( stderr, "error creating x.lis\n" );
       exit (1);
   }
   dup2( fileId, stdout ); /* copy fileID to stdout */
   close( fileId );
   execl( "/bin/ls", "ls", 0 );
}
```

dup, dup2

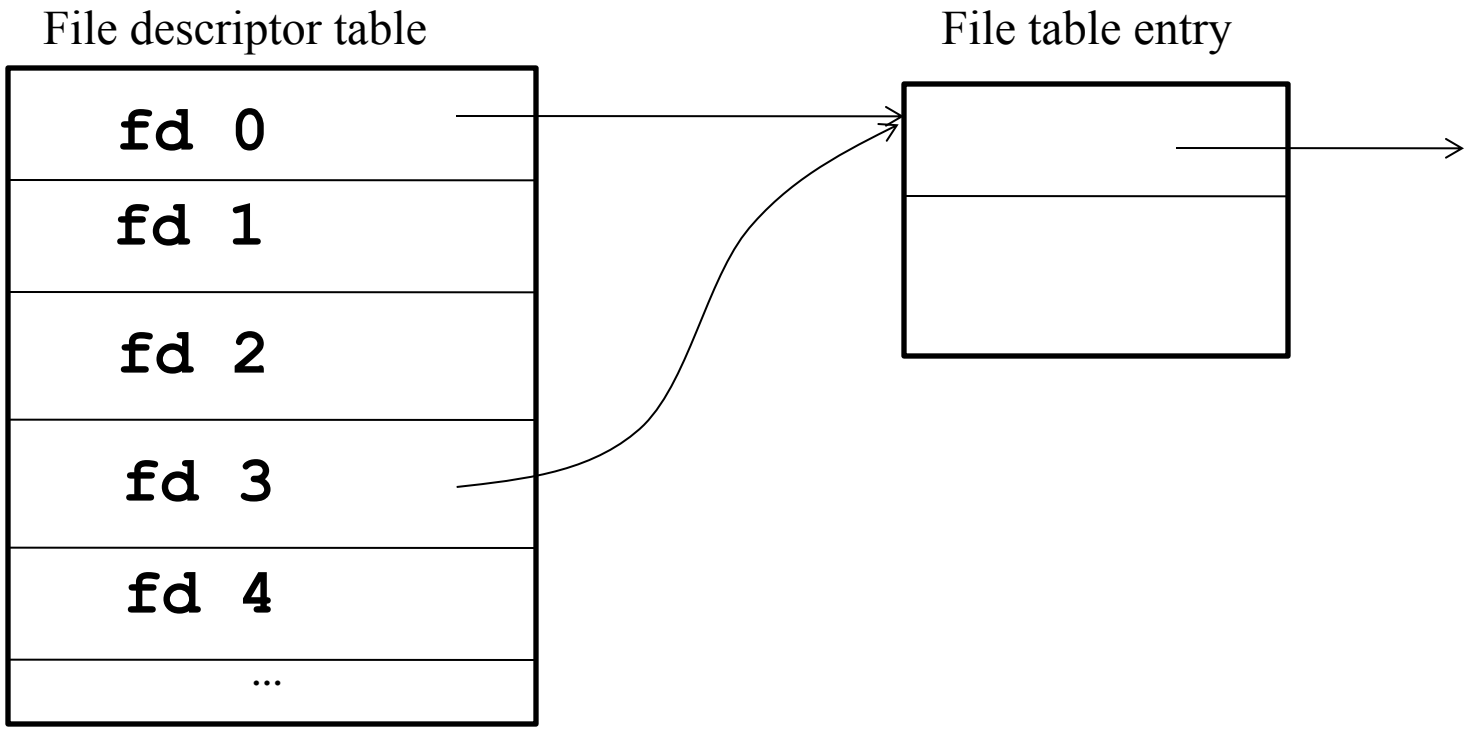
```
#include <unistd.h>

int dup(int filedes);
int dup2(int filedes, int filedes2);

                Both return : new file descriptor if OK, -1 on error
```

- An existing file descriptor (`filedes`) is duplicated
- The new file descriptor returned by `dup` is guaranteed to be the lowest numered available file descriptor
- With `dup2`, we specify the value of the new descriptor with the `filedes2` argument
- If filedes2 is already open, it is first closed.

- Ex) **`dup2(fd,STDOUT_FILENO);`**

example

- dup(1);

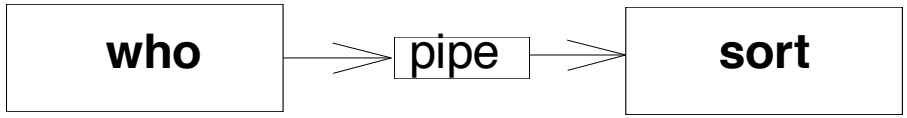


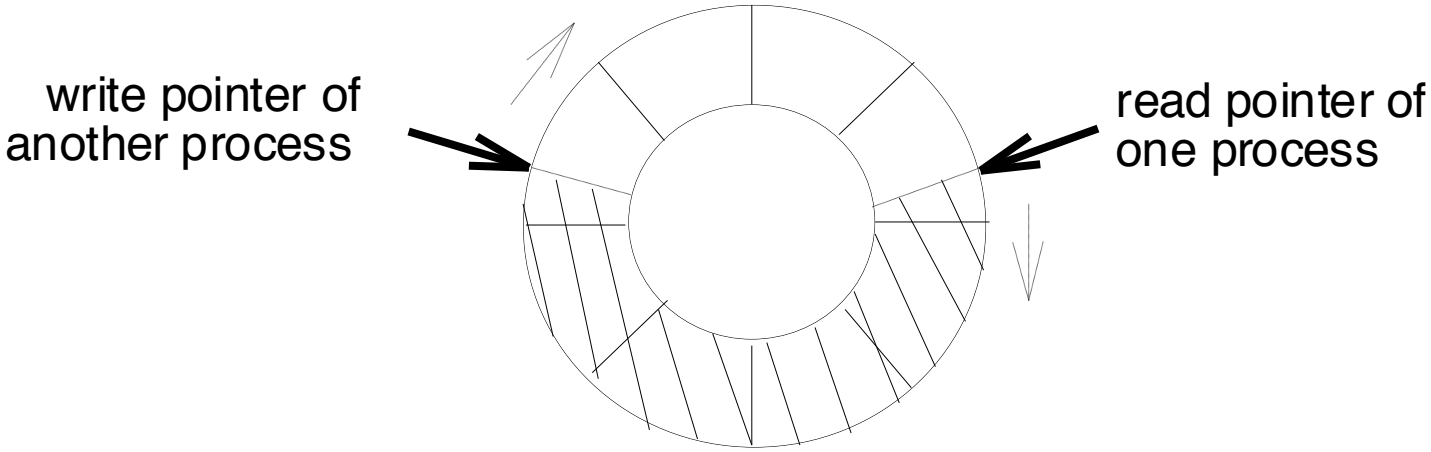
LAB: What does this program do?

```
#include <fcntl.h>          // redirection.c
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <sys/wait.h>

int main(int argc , char* argv[]) {
    char *path = "/bin/ls";
    char *arg0 = "ls";
    pid_t pid;
    int fd;
    int status;

    if (argc!=2) {
        printf("wrong command. ex) a.out filename\n");
        exit(0);
    }
    pid = fork();
    if (pid == 0) {
        fd = open(argv[1], O_WRONLY|O_CREAT|O_TRUNC, S_IRWXU);
        dup2(fd, STDOUT_FILENO);
        close(fd);
        if (execl(path, arg0, NULL) == -1)
            perror("execl");
    } else {
        close(fd);
        wait(&status);
    }
}
```

Pipe

```
$ who | sort
```





pipe

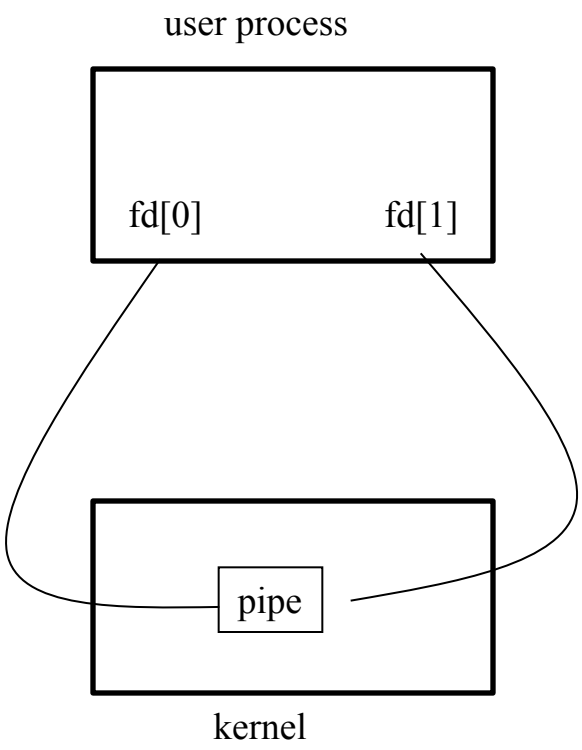
```
#include <unistd.h>

int pipe(int filedes[2]);

                return : 0 if OK, -1 on error
```

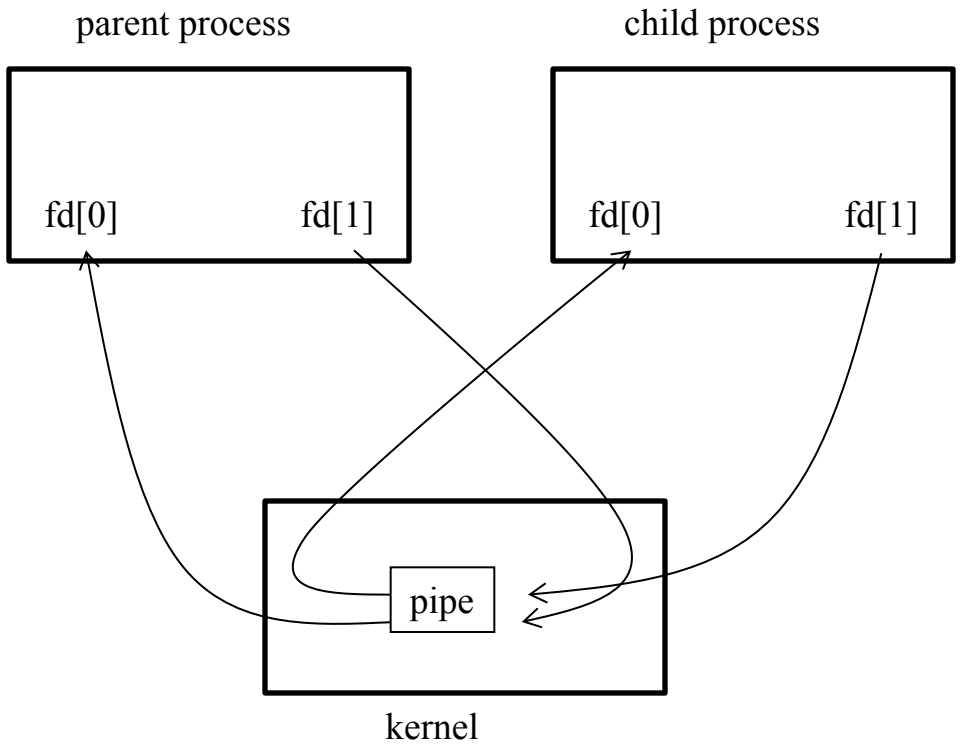
- Two file descriptors are **returned** through the *fd* argument
  - *fd*[0]: can be used to read from the pipe, and
  - *fd*[1]: can be used to write to the pipe
- **Anything that is written on *fd*[1] may be read by *fd*[0].**
  - This is of no use in a single process.
  - However, between processes, it gives a method of communication
- The pipe() system call gives parent-child processes a way to communicate with each other.

pipe

- `filedes[0]` is open for reading and `filedes[1]` is open for writing.
- Output of `filedes[1]` is the input for `filedes[0]`

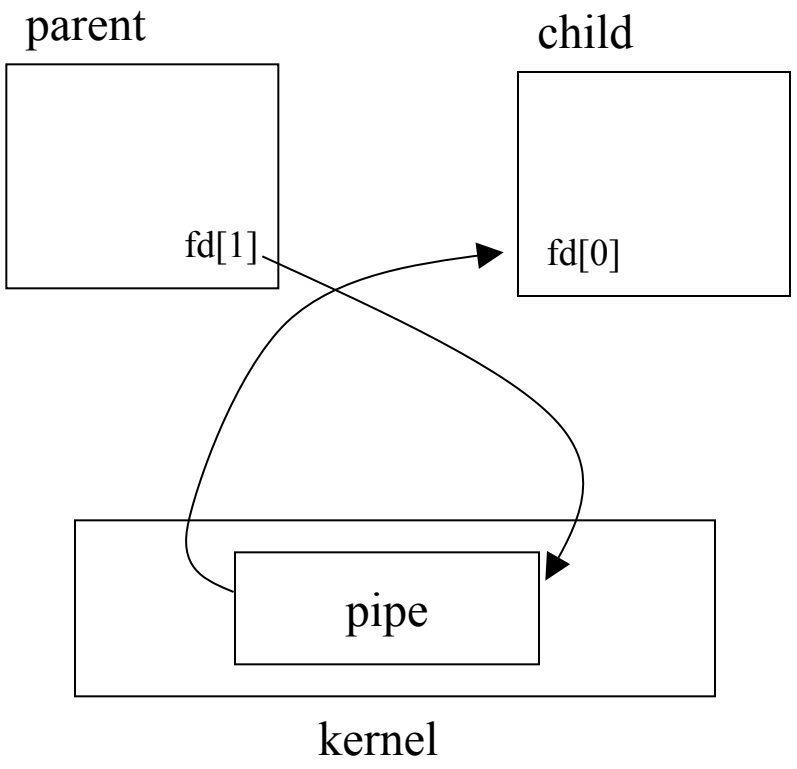
pipe after fork



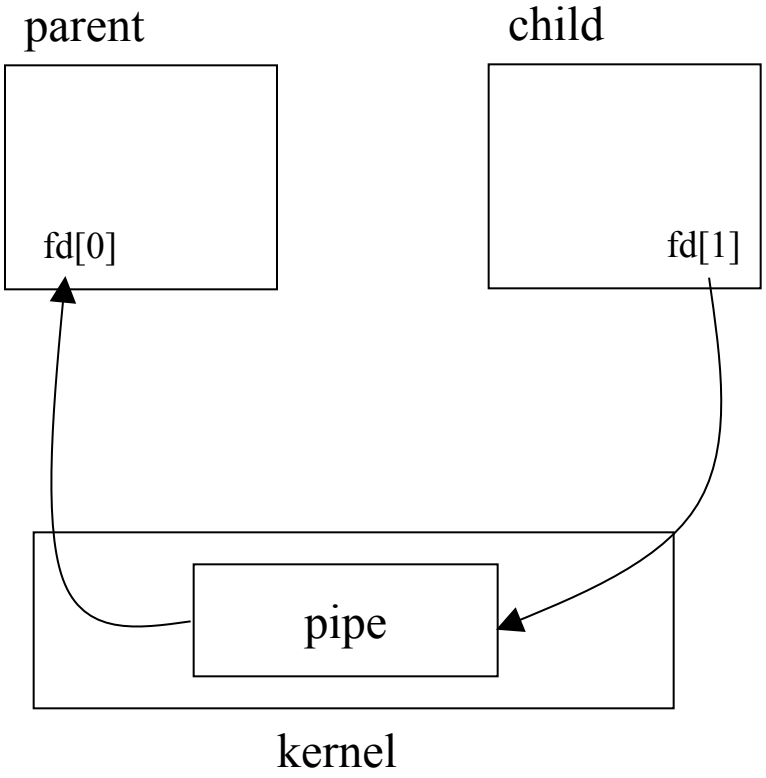


Pipes

parent → child:
parent closes fd[0]
child closes fd[1]



parent ← child:
parent closes fd[1]
child closes fd[0]



LAB: Pipe example results?

```
#include <stdio.h> // pipe2.c

#define READ  0
#define WRITE 1

char* phrase = "Stuff this in your pipe and smoke it";

main( ) {
    int fd[2], bytesRead;
    char message[100];
    pipe(fd);
    if (fork() == 0) {  // child
        close(fd[READ]);
        write(fd[WRITE], phrase, strlen(phrase)+1);
        fprintf(stdout, "[%d, child] write completed.\n", getpid());
        close(fd[WRITE]);
    }
    else {  // parent
        close(fd[WRITE]);
        bytesRead = read(fd[READ],   message, 100);
        fprintf(stdout, "[%d, parent] read completed.\n", getpid());
        printf("Read %d bytes: %s\n", bytesRead,message);
        close(fd[READ]);
    }
}
```

Another Example

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

int main(int argc, char *argv[]) {
    char *path = "/bin/ls";
    char *arg0 = "ls";
    pid_t pid;
    int pipefd[2];
    int status;
    pipe(pipefd);
    pid = fork();
    if (pid == 0) {
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[0]);
        close(pipefd[1]);
        if (execl(path, arg0, NULL) == -1)
            perror("execl");
    } else {
        if (fork() == 0) {
            dup2(pipefd[0], STDIN_FILENO);
            close(pipefd[0]);
            close(pipefd[1]);
            if (execl("/bin/cat", "cat", NULL) == -1)
                perror("execl cat");
        } else {
            close(pipefd[0]);
            close(pipefd[1]);
            wait(&status);
            wait(&status);
        }
    }
}
```