Process Control

Unix Systems Programming

CSPP 51081
A process is the unit of work.

Process control involves creating new processes to work concurrently on solving problems. There are four main system calls for process control:

- **fork**: create a new process
- **exit**: normal termination of process (*exit* is not actually a system call, but it calls `_exit` which is.)
- **execve**: load and execute another program (there are five other variations in the *exec* family, but *execve* is the actual system call executed.)
- **wait**: wait for another process to finish
Process control: implementing system

int system(const char *cmd) {
    int pid, status;
    if (cmd == NULL) return 1;
    if ((pid=fork()) == -1) return -1;
    if (pid == 0) {
        execl("/bin/sh", "/bin/sh", ":-c", cmd, NULL);
        perror("'");
        _exit(127); /* get here only if exec error */
    }
    else { /* parent */
        while (waitpid(pid, &status, 0) < 0) {
            if (errno != EINTR) return -1;
        }
        return status;
    }
}
fork: create a child process
**fork: create a child process**

**SYNOPSIS**

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

pid_t fork(void);
```

**Return:**

- `-1` on error
- `0` in child
- `pid` of child in parent

```c
pid_t fork(void);
```
Parent properties and resources inherited by child

- data, stack and heap segments of memory (usually by *copy on write*)
- open file descriptor table
- program execution point
- real and effective user ID and group ID
- process group ID (used in job control)
- session ID and controlling terminal
- current working directory
- file mode creation mask (i.e. umask)
- signal mask and dispositions
- environment
- resource limits
- niceness
**Parent properties not inherited by child**

- return value from `fork`: *child pid* in parent and 0 in child.
- process ID
- parent process ID
- some process accounting such as elapsed time, user time and system time (these are set to 0 in the child).
- pending alarms are cleared for child
- set of pending signals for the child is set to empty set
- file locks set by parent are not inherited by child
Usage of `fork`

- `fork` returns twice: once in parent and once in child.
- Execution begins immediately after `fork()`
- Return value of `fork()` in parent is `child pid`. This is the only way to obtain the process ID of child.
- Return value of `fork()` in child is 0.
- The child receives a `copy` of the parent’s data space, heap and stack (usually `copy-on-write`.)
- You can `never` count on whether the child starts executing before the parent or vice-versa. If the order matters, then parent and child need to synchronize.
Two uses of \textit{fork}

- When a process wants to duplicated itself so that the parent and child can each execute different sections of code concurrently. (Although, when many concurrent threads of execution are needed, as in a network server, it is usually better to use light-weight threads which do not require as much duplication of parent resources.)

- When a process wants to execute a different program. The child has an opportunity to change the per-process attributes, such as I/O redirection, user ID, signal disposition, before executing another program through \texttt{exec}. This is used in shells.
### Possible errors for `fork`

<table>
<thead>
<tr>
<th>errno</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAGAIN</td>
<td>The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user. <strong>CHILD_MAX</strong> would be exceeded. <em>(<strong>CHILD_MAX</strong> is found in <code>limits.h</code> and is set to 999 on admiral.)</em></td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Insufficient storage space is available.</td>
</tr>
</tbody>
</table>
File Sharing
Data Structures for open files

The kernel maintains three distinct structures for open files:

- File Descriptor Table
- System Open File Table
- Kernel I-node Table
- Individual Filesystems Inodes

Each table entry contains a pointer to an entry in the next table.
Each process has its own File Descriptor Table.

- The File Descriptor Table is an array of open file descriptors for a process. A file descriptor is an index to this array.
- Each table entry consists of
  - A file descriptor flag: there is only one FD_CLOEXEC, which closes the file descriptor when an exec function is called. This is set using `fcntl(fd, F_SETFD, FD_CLOEXEC)`.
  - A pointer to an entry in the System Open File Table.
- The File Descriptor Table is copied on `fork` for the new process.
- The maximum number of open file descriptors a process can have is given by `OPEN_MAX` given in `<limits.h>`.
The kernel maintains a system-wide Open File Table.

- Different processes can share the same Open File Table entry, but one process must be a descendant of the other (created by a chain of forks).

- Each table entry consists of
  - The file status flags for the file (read, write, append, nonblocking, etc.) passed by call to `open`.
  - The current file offset
  - A counter of the number of file descriptors which currently point to this entry.
  - A pointer to the inode entry of the file in the Kernel I-node Table.

- A new open file entry is created on each call to `open`, so the same file can have several table entries.
Kernel I-node Table

The kernel maintains a table of recently accessed files.

- There is one entry in the Kernel I-node Table for each file.
- Several Open File Table entries may point to the same file entry in the Kernel I-node Table.
- The Kernel I-node Table is the kernel’s abstraction of a file maintained by the Virtual File System. (Stevens calls these entries v-nodes to contrast them with the i-nodes which are actually implemented in each individual file system.)
- Each entry contains at least the following information:
  - File information as found on struct stat: type of file, size, ownership, etc. But not the name of the file!!
  - Filesystem device number and i-node in this filesystem, which together uniquely identifies the file.
  - Pointers to functions which operate on files in the filesystem (read, write, get i-node, update i-node, etc.)
Individual Filesystems

The Virtual Filesystem is an abstraction by the Kernel to hide many individual filesystems.

- Several operating systems may access the same filesystem. (Think when you log into two or more CS computers.)
- Filesystems have traditionally maintained file information (user, size, etc.) by means of *i-nodes*; but, this is not necessary for the Virtual Filesystem. POSIX now calls i-nodes *file serial numbers* and leaves their meaning and implementation to each filesystem.
- Traditionally, the name of the file is *not* kept with the i-node, but only stored in directories.
- Traditionally, each directory stores the name of the file and the inode of files within the directory.
Putting it together: I/O operations

• When a file is opened the lowest file descriptor available is used on the process File Descriptor Table. An entry in the Open File Table is created, the flag parameter on open is copied to the entry, and the offset is set to 0, unless O_APPEND is specified, in which case the offset is set to the file size. The descriptor count on this table is set to 1. If the file is not located on the Kernel I-node Table, the kernel must create an entry here as well.

• After each lseek the Open File Table offset is adjusted, but no other action is taken.

• After a close the file descriptor entry is closed on the File Descriptor Table and the count for the relevant entry in the Open File Table is decremented. This entry is removed if the count reaches 0.
Putting it together: I/O operations

- After each `write` is complete, the current file offset in the Open File Table entry is incremented by the number of bytes written. If this causes the offset to exceed the current file size, the file size in the i-node table entry of the Kernel I-node Table is adjusted (and eventually, the i-node in the filesystem will be changed as well.)

- If the `O_APPEND` flag is set in the Open File Table, on each `write` the current file offset is set to the file size (the end of the file) obtained from the Kernel I-node Table.

- After a `fork` the File Descriptor Table is copied to the child and all counts for relevant entries in the Open File Table are incremented.
First Lesson in Synchronization: Atomic Operations

An atomic operation is an operation composed of multiple steps, but where all steps are performed together or none are.

- What is wrong with the following code:

```c
lseek(fd, 0L, SEEK_END); /* position to EOF */
write(fd, buff, 100); /* append to EOF */
```

(Multiple processes have access to the same file. It is possible that between the system calls `lseek` and `write` another process writes to the file.)

- Appending to a file is made atomic by using the flag `O_APPEND` when opening.
Second Lesson in Synchronization: File Sharing

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

int main(void) {
    FILE *fs;
    fs = fopen("myfile", "w");
    fork();
    fprintf(fs, "Process %ld writing\n", (long)getpid());
    sleep(1); /* Process doing other things */
    fprintf(fs, "Process %ld writing more\n", (long)getpid());
}

What could happen if parent and child were writing large blocks of data?
dup, dup2: duplicating file descriptors
**SYNOPSIS**

```c
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

**Return:**

- new file descriptor if OK
- -1 on error
Usage of `dup`, `dup2`

- Both `dup` and `dup2` create a copy of the file descriptor `oldfd`:
  - `dup` uses the lowest-number unused descriptor available for the new descriptor
  - `dup2` copies `oldfd` to `newfd` (closing `newfd` if necessary)
- Both `oldfd` and the new descriptor point to the *same* Open File Table entry, so share the same current file offset.
- The file descriptor flag (close-on-exec) is *not* copied on duplication—it is cleared on duplication.
- The most common use of duplicating file descriptors is for redirecting standard input and standard output.
Duplicating standard input

The following steps redirect standard input to \textit{file}:

1. Open: \texttt{fd = open(file,...);} \\
2. Duplicate: \texttt{dup2(fd, 0);} \\
3. Close: \texttt{close(fd);} \\
4. Read, Write: \texttt{read(0,...);, write(0,...);} \\

Since \texttt{fd} was no longer needed it was closed. In general, it is a good practice to close file descriptors that will no longer be used, to conserve a limited system resource.
Filters

A filter is a program which reads from standard input, writes to standard output and reports errors to standard error.

- Examples of filters: head, grep, sort, diff, cat
- The shell handles redirection of standard input and output by duplicating file descriptors. Example:
  
  ```
  cat < my_input > my_output
  ```
  
  - fork for a new process
  - Redirect standard input and output in child
  - exec the cat program.

- The filter reads from standard input, writes to standard output and reports errors to standard error.
Why dup2: Atomic Operations

Consider the following alternative to re-directing standard input

```
close(0);
open("my_input",O_RDONLY);
```

Usually, this is an acceptable alternative to `dup2`, but it duplicates in two operations. `dup2` is atomic and does it in one operation. If you have a signal handler and receive a signal between closing standard input and opening `my_input` then your program will be executing input between these operations which may be a problem. Alternative lines of execution, such as signal handlers called to handle signals, can cause nasty difficulties in ensuring correct execution. Having an atomic duplication operation eliminates this problem.
Possible errors on `dup`, `dup2`:

<table>
<thead>
<tr>
<th>errno</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td><code>oldfd</code> is not an open file descriptor, or <code>newfd</code> is out of the allowed range for file descriptors.</td>
</tr>
<tr>
<td>EMFILE</td>
<td>The process already has the maximum number of open file descriptors.</td>
</tr>
</tbody>
</table>
Starting a program
main function

• The ANSI C prototype for main is

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

where argv is a NULL terminated array of command line words and argc is the length of this array.

• All programs, except init, begin execution by a system call from one of the exec family of functions.

• A start-up routine precedes the beginning of main where the file descriptors for standard input, output and error are opened, the process environment is set, and the arguments for main are passed to main.
Environment List

- On the Unix system, each program is passed an environment list. This list is stored in the global variable `extern char **environ;` from `#include <unistd.h>`.

- An environment list is a NULL terminated array of strings of the form `name=value`. For example,
  
  ```
  HOME=/home/kaharris\0
  SHELL=/usr/local/bin/bash\0
  PATH=/home/kaharris/bin:/usr/local/bin:/usr/bin:/bin:./\0
  NULL
  ```

- With one exception, `PATH`, the environment variables are *never* used by the kernel. They are used by programs.

- You can obtain and set these variables by `getenv(3)` and `setenv(3)`.
exec: execute a file
SYNOPSIS

#include <unistd.h>

extern char **environ;

int execl( const char *path, const char *arg0, .../* NULL */);
int execv( const char *path, char *const argv[]);
int execlp( const char *file, const char *arg, .../* NULL */);
int execvp( const char *file, char *const argv[]);
int execle( const char *path, const char *arg , .../*NULL, char *const envp[]*/);
int execve( const char *path, char *const argv[], char *const envp[]);

**Return:**

no return on success -1 on error
Usage of exec

- When a process calls one of the six exec functions, that process is replaced by a new program which starts execution at its main.

- No exec function ever returns to the calling program, except on error.

- Only execve is an actual system call. The other five functions are wrappers which build the argument list for execve.

- The total size of the argument list to execve is given by ARG_MAX, and is at least 4096 bytes. (On admiral it is 131072 bytes.) This includes both the argument and environment arrays.

- The suffixes on exec family are as follows:
  1: argument list is a NULL terminated parameter list
  v: argument array (NULL terminated) is included
  p: command name may be a partial path to a file, not an absolute path
  e: environment array (NULL terminated) is included
**Changes across exec**

- The process text, data, heap and stack memory segments are completely replaced by the new program.

- If a file descriptor is opened with the FD_CLOEXEC flag set, the descriptor is closed across an `exec`. The default is that this is unset and file descriptors remain open across `exec`

- All open directories are closed across an `exec`.

- The real user ID and real group ID remain the same, but the effective IDs can change, depending on the status of the set-user-ID and set-group-ID bits in the permissions for the program being executed. The effective IDs are used for file access permissions.

- The environment may be replaced if this is specified as an argument to `execve` or `execle`.
Properties inherited on exec

The following properties include the system call (man 2) for accessing value:

- process ID (getpid) and parent process ID (getppid)
- real user ID (getuid) and real group ID (getgid)
- process group ID (getpgid)—used for job control
- session ID (getsid) and controlling terminal (tcgetpgrp)
- current working directory (getcwd)
- file mode creation mask (umask)
- process signal mask (sigprocmask) and pending signals (sigpending)
- time left until alarms (getitimer)
- resource limits (getrlimit)
- Processing times (times)
- file locks (fcntl(fd, FGETLK, FD_CLOEXEC))
- nice value (nice)
### Some possible errors on `exec`

<table>
<thead>
<tr>
<th>errno</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EACCES</td>
<td>One of: search permission on directory in path of new process denied, or new process file not executable, or process does not have permission to execute.</td>
</tr>
<tr>
<td>E2BIG</td>
<td>Argument list is bigger than system-imposed <code>ARG_MAX</code> bytes</td>
</tr>
<tr>
<td>ENOENT</td>
<td>Component of <code>path</code> or <code>file</code> does not name an existing file</td>
</tr>
<tr>
<td>ENOEXEC</td>
<td>Image file is in an unrecognized format or for the wrong architecture</td>
</tr>
</tbody>
</table>
Building `execve` from other execs
Building execve

execlp → build argv → execvp

execlp → build argv → execl

execvp → try PATH → execv

excele → build argv → execve

execv → use environ → execve
Creating the argument vector for `execle`, `execlp`, `execl`:

- The argument list is turned into an argument vector for `execve`.
- Each argument in the list but the last, is a null-terminated string.
- The argument list itself must be `NULL` terminated. You may use 0 to terminate, but then you should cast as `(char * 0)`.
- The first argument, `arg0`, point to a filename that is associated with the process being started.
Finding a program with `execvp`, `execlp`

- `execvp` and `execlp` take a `file` as argument, but `execve` requires a `path`. The determination of `path` is as follows:

  1. If `file` begins with a slash, `path` is `file`.

  2. Otherwise, the executable file is searched for in the directories specified by `PATH` environment variable, or if this is unspecified, `/bin:/usr/bin:...` If found, `path` is `file` appended by the path to the directory containing `file`.

  3. Otherwise, kernel assumes `file` is an interpreter (or script) file and `path` is taken to be `/bin/sh` (or whatever the shell path is.) For example,

     ```
     execlp(file, arg0, arg1, NULL)
     ```

     will be executed like

     ```
     execl(/bin/sh, arg0, file, arg1, NULL)
     ```
Passing an environment list

- The functions `execle` and `execve` explicitly pass a NULL terminated environment array. The global variable `environ` is set using this list.

- For the functions `execl`, `execv`, `execlp` and `execvp` the environment list is taken from the value of the global environment variable `environ` currently used by the process.
File Access Test

Effective User and Group ID
Testing File Access

The owner and group IDs are properties of the file. The effective user and group IDs are properties of the process.

File access is determined by comparing the permissions of the file with the effective user and group ID as follows:

1. If the effective user ID of the process is 0 (superuser) access is allowed.
2. If the effective user ID of the process is identical to the owner ID of the file, access is allowed provided the appropriate permissions are granted to the owner of the file. Otherwise, access is denied.
3. If the effective group ID of the process is identical to the group ID of the file, access is allowed provided the appropriate permissions are granted to the group of the owner of the file. Otherwise, access is denied.
4. If the appropriate permissions for others are set, access is allowed. Otherwise, access is denied.
Exiting a Process

Zombies and Orphans
Normal termination

- Calling `exit(3)` function. This function is specified by ANSI C and calls all exit handlers registered by `atexit(3)` and flushes standard I/O buffers and closes standard I/O. In Unix systems it also calls `_exit` (see below) which handles Unix-specific details. This function returns its argument, called the exit status.

- Executing `return` from `main` is the same as calling `exit`. The exit status is the return value or undefined if no value is returned. Falling of the end of `main` is equivalent to `return` without a returning value.

- Calling `_exit(2)`. This function is specified by POSIX and called by `exit` on Unix systems. This is a system call and causes the Kernel to
  - Close all open file descriptors for the process
  - Any children of the process are inherited by `init`, process 1
  - Process’ parent recieves SIGCHLD signal to inform of deceased child

The argument is returned as exit status.
Abnormal termination

- Calling abort(3): sends a SIGABRT to the process which cannot be ignored or blocked, and never returns to the caller. (It is possible for the process to call a signal handler which call exit or _exit, implementing a normal termination procedure.)

- Recieving some signals, generated by the process itself or some other process (using kill(2)) or from the Kernel (such as when making an invalid memory reference or dividing by zero.)
Termination Status

- Upon termination the kernel closes all open descriptors, releases memory, and other duties.

- The kernel prepares an integer value, the *termination status*, indicating the reason for termination. Typically the high bits indicate the signal number for abnormal termination and the low bits indicate the exit status (for normal termination.) There may also be a bit to indicate a core dump or continuation from a job control stop, but these are implemented only be some systems.

- Use macros defined in `#include <wait.h>` to read the termination status (see next section for `wait`.)

- The kernel maintains the termination status, along with the process id and the process CPU time until (and if) the parent fetches the termination value through `wait` or `waitpid`. 
**Zombies and Orphans**

- A child whose parent exits before it, becomes an *orphan* and receives a new parent, *init* with process ID 1. *init* waits on the child when it terminates to collect the termination status.

- If the child exits before its parent, and the parent waits for the child (calling *wait* or *waitpid*), the parent receives the termination status, and all remaining resources of the terminated child are released.

- A process that has terminated and whose parent is still around but has not yet waited for it is a *zombie*. The kernel maintains the termination value should the parent eventually wait for it. No child of *init* ever becomes a zombie.

- Zombies do use system resources, so long running programs (such as the shell) need to take steps to prevent zombies: either by waiting for their children or ensuring all descendants are eventually waited upon by *init*. 
wait, waitpid: wait for process termination
wait, waitpid - wait for process termination

SYNOPSIS

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

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

**Return:**

- `pid` of exited child on success
- 0 if `WNOHANG` set
- -1 on error
### Constants for options: `#include <wait.h>`

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WNOHANG</td>
<td><code>waitpid</code> will not block if a child specified by <code>pid</code> is not immediately available. In this case the return value is 0.</td>
</tr>
<tr>
<td>WUNTRACED</td>
<td>return for children which are stopped, and whose status has not been reported (used in job control by shells).</td>
</tr>
</tbody>
</table>
### Possible arguments for *pid* in `waitpid`

<table>
<thead>
<tr>
<th>Value of <em>pid</em></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pid == -1</code></td>
<td>wait for any child process</td>
</tr>
<tr>
<td><code>pid &gt; 0</code></td>
<td>wait for child process whose process ID is <em>pid</em></td>
</tr>
<tr>
<td><code>pid == 0</code></td>
<td>wait for any child whose process group ID equals that of caller</td>
</tr>
<tr>
<td><code>pid &lt; -1</code></td>
<td>wait for any child whose process group ID equals that of <code>-pid</code></td>
</tr>
</tbody>
</table>
### Testing Termination Status: : `#include <wait.h>`

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WIFEXITED(<code>status</code>)</td>
<td>True if child terminated normally. In this case execute WEXITSTATUS(<code>status</code>) to fetch the value child passed to <code>exit()</code> or on <code>return</code>.</td>
</tr>
<tr>
<td>WIFSIGNALED(<code>status</code>)</td>
<td>True if child terminated abnormally. In this case execute WTERMSIG(<code>status</code>) to fetch signal number that caused termination.</td>
</tr>
<tr>
<td>WIFSTOPPED(<code>status</code>)</td>
<td>True if child currently stopped. In this case execute WSTOPSIG(<code>status</code>) to fetch signal number that caused child to stop.</td>
</tr>
</tbody>
</table>
### Possible errors in `wait`, `waitpid`

<table>
<thead>
<tr>
<th>errno</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECHILD</td>
<td>If there are no unwaited for child processes existing. In addition for <code>waitpid</code>, if the process specified in pid does not exist or is not a child of the calling process.</td>
</tr>
<tr>
<td>EINVAL</td>
<td><code>options</code> field of <code>waitpid</code> was invalid</td>
</tr>
<tr>
<td>EINTR</td>
<td>the function interrupted by signal. The value of <code>status</code> is undefined.</td>
</tr>
</tbody>
</table>
Usage of `wait`, `waitpid`:

- When a process terminates (normally or abnormally) the parent is notified by the kernel sending a `SIGCHLD` signal. The default action is to ignore this signal.

- A process that calls `wait` or `waitpid` can
  - block—if all its children are still running—until a child terminates.
  - return immediately with the termination status of a child
  - return immediately without the termination status of a child (only with `waitpid` when `WNOHANG` set)
  - return immediately with error (possibly, an `ECHILD` error if there are no existing unwaited for child processes.) Beware that an error return can be `EINTR`, in which case the wait was interrupted by a signal. The desired action is probably to restart the wait. (see the restart library)
Differences between wait, waitpid

- wait can block the caller until a child process terminates and returns the first child to terminate.
- waitpid has an option to prevent blocking, WNOHANG, and can wait on a particular process or process group by setting pid.
- waitpid supports job control with WUNTRACED option.

```c
pid_t wait(int *status) {
    return waitpid(-1, status, 0);
}
```
How to wait on children

- Call `wait` or `waitpid` and block.
- Poll using `waitpid` with `WNOHANG`.
- Use a signal handler to catch `SIGCHLD` signal and wait in the handler (which will return immediately with the termination status.)
- Fork a grandchild whose parent is `init`:

```c
if ( (pid=fork()) < 0 ) {
    perror("fork error");
} else if (pid == 0) {
    if ( (pid=fork()) < 0 ){
        perror("fork error")
    } else if ( pid > 0){
        exit(0); /* child process of first fork */
    } else /* This is the grandchild, parent is init */
    /* The grandparent needn’t worry about zombies */
```