Processes 1: Creation & Termination

1. **Creation and Termination**
   - **multi-process programs**
   - **process creation: fork()**
   - **attribute inheritance**
   - **parent vs. child functionality**
   - **fork() vs. threads**
   - **process termination**
   - **parent or child first?**
2. Collecting Children
3. Exec-Family Calls
4. Process Organization
5. Other Process Related Calls
6. Error Handling with Children

Multi-Process Programs

A **multi-process program** is a program that creates **multiple processes** that run **concurrently** to implement the overall functionality of the program.

Example: initial program process loops accepting inputs from the user, but a second process is created that manipulates each input after it is received, allowing the original process to be able to immediately accept further input.

Example: initial server process listens for new client requests, but then creates a new process to handle each individual client.

C supports multi-process programs via OS **system calls**.

Multi-Process Programs (contd.)

A *multi-process program* is one possible approach to **concurrent computing**.

A **thread of execution** (or **control**) refers to one particular sequence of program operations being executed.

In a traditional, *single process, single thread* program, there is a *single thread of execution*.

In a multi-process program, there will be *multiple threads of execution* being concurrently scheduled and run by the kernel, with each thread in a *separate process*.

Multi-Process Programs (contd.)

The alternative approach for concurrent computing is to use a **multithreaded program**: each of the program's threads of execution are **OS threads** within a *single process*.

Multi-process programs are similar to multithreaded programs, except each process has a *separate address space*.

This means that processes will have to explicitly transfer any data needed by other processes using **interprocess communication mechanisms**.

On the other hand, the lack of shared address spaces makes accidental interference (e.g., by changing a variable's value) less likely between processes than between threads.

Process Management System Calls

The following groups of system calls are available to manage the creation and execution of processes:

- process creation
  - `fork()`, `vfork()`, `clone()`
- process termination
  - `exit()`, `_exit()`
- process collection
  - `wait()`, `waitpid()`, etc.
- program execution in a process
  - `execve()` and "exec" library calls

Creating Processes

Every process except the very first one, **init**, must be created by another process.

The creating process is called the **parent process** and the newly created process is called the **child process** or a **subprocess** (of the parent).

Each process has a fairly large set of **attributes**, most of which will be **inherited** by a child process.

However, processes have *separate address spaces*, so the running programs are (mostly) separate.

The only significant exception to this involves *files* that were open prior to the creation of the subprocess.

fork() System Call

`fork()` is the standard system call for creating a subprocess:
`pid_t fork(void)`

- takes no arguments
- return value is critical to test:
  - the PID of the new/child process is the return value in the *parent process*
  - 0 (zero) is the return value in the *child process*
  - -1 is returned (in parent) if `fork()` fails (error)

`fork()` creates a nearly complete *copy* of the parent process, with both parent and child executing the same next instruction following `fork()`, after the `fork()` call has completed.

The resulting processes can determine which one they are (parent or child), by examining the return value they receive from `fork()`.

fork() System Call (contd.)

Since there are *three classes* of outputs from `fork()`, it is common to call it in a `switch` statement:

```
pid_t pid;  //Used to capture child's PID in parent

switch(pid = fork()) {
  case -1:
    //fork error:
    perror("fork failed");
    exit(EXIT_FAILURE);

  case 0:
    //child process:
    printf("Child running!\n");
    ...child code...
    exit(EXIT_SUCCESS);  //exit() to ensure cannot drop into parent code

  default:
    //parent process:
    printf("Parent created child w/PID: %d\n",pid);
    ...parent code...
    wait(...);
  }
```

Attribute Inheritance through fork()

The child duplicates the parent's attributes, except:

- it has its own unqiue PID
- its PPID is the the parent's PID
- it does not inherit pending signals, mutexes, semaphores, various locks, timers/alarms

Attribute Inheritance through fork() (contd.)

One of the more important (and often misunderstood) aspects of attribute inerhitance, concerns *open files*:

- a child inherits copies of the parent's open *file descriptors*
- a file descriptor refers/points to a **file description** object in the kernel's **file table**
- both the parent and child FDs will point to the *same* file description object
- it is the file description that holds file status flags, current file offset, etc., and of course points to the actual file (inode object)
- thus when either the parent or child causes a change in file offset (e.g., via I/O), the other process will automatically see the movement

Different Functionality in Parent/Child

Following a `fork()` call, the parent and child processes will be executing exactly the same program, but it is unlikely we want two process doing exactly the same thing.

One approach is to call different functions after `fork()`:

```
switch(fork()) {
  case -1:
    perror("fork failed");
    exit(EXIT_FAILURE);

  case 0:
    child_func(...);
    exit(EXIT_SUCCESS);

  default:
    parent_func(...);
    wait(...);
  }
```

Different Functionality in Parent/Child (contd.)

The drawback of this approach is that the functionality for both the parent and the child will have to be stored in both the parent's and the child's address spaces.

The use of virtual memory models limits the problems that might arise if both of these functions are extremely large (since the unused functionality will remain swapped out).

However it will often be preferrable to have a completely *separate program* run in the child.

This is the purpose of the **exec-family** of system calls.

Other fork-Related System Calls

Linux also has the following two process creation system calls:

- **vfork**
  - `pid_t vfork(void)`
  - create child process without copying the page tables of the parent
  - slight performance improvement when child will immediately do "exec" call
  - parent is suspended until the child calls `execve()` or `_exit()`
- **clone**
  - `int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...)`
  - main use is to implement **threads**
  - allows child process to share parts of its execution context with parent/caller

fork() vs. Threads

Process creation via `fork()` might sound very expensive since it seems to require copying the entire address space of the parent.

However, Linux/UNIX systems do not actually immediately copy the parent's address space.

Instead they use a scheme called **copy-on-write**:
the child process' **page table** entries initially simply point to the parent process' pages, and are copied only as required by operations in the child that modify memory contents.

In fact, if a `fork()` is immediately followed by an `exec`-family call, little/no address space copying will occur.

Thus, process creation is not nearly as slow as it might appear (or as it was long ago).

fork() vs. Threads (contd.)

Still, there is overhread in setting up process data structures and creating a process page table pointing to

Recent testing by M. Kerrisk on Linux shows that `fork()` takes on the order of ten times as long as the `clone()` call used to create a new thread, and `fork()` takes increasing time as a process' address space increases in size.

However, the time to `fork()` a fairly large process is less than 10^{-3} sec., so creating a few subprocesses is hardly going have a noticeable impact in most applications.

Thus, one should choose between using multiple processes or multiple threads based on the need for data sharing among the threads and the potential for unwanted interactions (not because using threads might save a few milliseconds of runtime).

Process Termination

Processes can terminate **normally** or **abnormally**.

Normal termination basically means that the program terminated because it was finished doing what it was supposed to do or what it was able to do.

A process that *terminates normally* will have an **exit status**.

The exit status indicates whether it completed "successfully" or not.

E.g., if a program cannot open a file (say, due to insufficient permissions), it will generally terminate itself (after appropriate cleanup) and return failure exit status.

Abnormal termination typically indicates that a process suddenly quit due to severe problems.

Process Termination (contd.)

Processes can **terminate normally** in *five* ways:

- call `exit()` (C library function)
- call `_exit()` (syscall)
- `return` from `main`
- `return` from the start function of the last thread in a process
- call `pthread_exit()`

Processes can **terminate abnormally** in *three* ways:

- **receive a signal** that terminates it
- call `abort()` (sends signal to process itself)
- process **thread cancellation request** in last process thread

Parent or Child First?

A question that can arise when developing multi-process programs is: which runs first after a `fork()`, the parent or the child?

If the order were fixed, one might assume code could be simplified because it would be known which statement after the `fork()` will get run first.

However, this is *not really the case:*

- order is not fixed in Linux/UNIX standards (though may be in practice)
- single code statements may not be **atomic**
- modern computers are all multicore, so processes can get run in **parallel**

Parent or Child First? (contd.)

For code to be portable and reliable, it must be written with *absolutely no assumptions made* about parent-child execution order post-`fork()`!

This means, for example, that if the parent must setup something like a **signal handler** for `SIGCHLD`, the handler must be setup *before* `fork()`.

Luckily, what we need to have ready immediately after a `fork()`, is likely to be *inherited* across the `fork()`.

Attribute inheritance across `fork()` was covered earlier!

Parent or Child First? (contd.)

Setup `SIGCHLD` handler before `fork()` to ensure handler is active before a signal could be generated:

```
// Activate SIGCHLD handler:
struct sigaction act;
act.sa_handler = sigchld_handler;
act.sa_flags = 0;
sigemptyset(&act.sa_mask);
sigaction(SIGCHLD,&act,NULL);
switch(fork()) {
  case -1:
    ...
  case 0:  //Child process:
    ...
    ...make certain child eventually terminates...
  default:  //Parent process:
    ...
}
```

Parent or Child First? (contd.)

Despite what was just said about not making order assumptions, if you run tests, e.g., using `strace`, you might notice that the *parent* seems to always be run first with Linux.

It turns out this is true for current Linux kernels—though it has not always been true.

In fact, due to the way **copy-on-write** works, when a child is going to immediately *exec* another program, it would be faster to run the child first.

This, though, broke old code that assumed the parent would run first, so was reverted.

Despite the parent running first with current Linux kernels, the reasons cited, to avoid writing code that relies on this order, remain valid—so *always write reliable, race condition-free code!*

Parent or Child First? (contd.)

There is a `/proc` setting that can affect paret-child order:
`/proc/sys/kernel/sched_child_runs_first`

Here is what *"man 5 proc"* says:
"If this file contains the value zero, then, after a fork(2), the parent is first scheduled on the CPU. If the file contains a nonzero value, then the child is scheduled first on the CPU. (Of course, on a multiprocessor system, the parent and the child might both immediately be scheduled on a CPU.)"

Note the admonition about dealing with *multiprocess/multicore systems*, since this applies to essentially all modern computers!

Processes 2: Collecting Children

Collecting Child Processes

When a process terminates, the kernel continues to store its "termination status" information in the kernel's **process table**.

In particular, it stores whether termination was normal/abnormal and the process' **exit status** (if normal termination).

This information is stored so that a parent process can determine the outcome of each child process.

A parent can use the **wait**-family of calls to retrieve termination information for its children.

In fact, it is considered the *responsibility* of every process that forks off children to do this for all children before it terminates.

Collecting Child Processes (contd.)

When a parent uses a wait-family call to retrieve a child's termination info, we say:

- the parent has **waited** for the child
- the parent has **collected** the child (or collected its results)
- the child has been **reaped**

Zombies and Orphans

A bit of colorful terminology is commonly used to refer to child processes being in particular states:

zombie – a process that has terminated by not yet been collected (waited for)

orphan – a process whose parent has terminated

Every process must eventually become a zombie at least briefly.

However, because a zombie process continues to *consume system resources* (entry in the **process table**), it is undesirable for processes to remain as zombies indefinitely.

Zombies are also known as **defunct** processes.

Zombies and Orphans (contd.)

Note that zombies and orphans are *independent* concepts:

- a process can be neither (parent and child still running)
- a process can be both (child terminated and parent terminated without collecting child)
- only a zombie (child terminated but running parent has not yet collected it)
- only an orphan (child is still running but parent has terminated)

If you see zombie processes listed when using `ps`, don't bother trying to eliminate them with the `kill` command; zombies are already terminated, so `kill` will do nothing to them!

Zombies and Orphans (contd.)

It is good programming practice for a parent to make sure all its zombie children are collected in a timely manner (and definitely before the parent terminates).

If a process creates a large number of children and never collects them, this can cause resource issues for the program and even the system as a whole (if resource limits have not been set for users/processes).

Note however that if a process terminates without collecting all its children, uncollected children will be **adopted** by the *init process*.

When these children terminate, *init will collect then*.

So zombie problems might arise only if a long running process (e.g., a server) fails to ever collect its terminated children.

wait() System Call

`wait()` is the most basic system call to collect a child:
`pid_t wait(int *status)`

- places the termination status in the `status` argument
- `status` can be `NULL` if parent is not interested in status
- returns PID of collected child (-1 if no children exist)
- returns immediately if there is already a zombie child, else calling process is *suspended* until a child terminates (i.e., parent *waits* for child)
- can be used for **synchronization**, since parent will not continue past `wait()` until a child has terminated

wait() System Call (contd.)

It is important to understand the the `status` parameter to `wait()` is a **termination status**, not a process **exit status**.

`status` can be used with a set of macros to determine how a process terminated and its exit status (if it terminated normally):

WIFEXITED(status) – true if process terminated normally

WEXITSTATUS(status) – returns exit status of process (but only if process terminated normally)

WIFSIGNALED(status) – true if process was terminated by a signal

WTERMSIG(status) – signal that terminated process

wait() System Call (contd.)

Example of parent calling `wait()` and returning final exit status based on child exit status:

```
...
int status;

wait(&status);

//Check if child exited successfully or not:
if (WIFEXITED(status) && WEXITSTATUS(status) == EXIT_SUCCESS)
  exit(EXIT_SUCCESS);
else
  exit(EXIT_FAILURE);
```

waitpid() System Call

The `waitipid()` syscall provides more functionality than `wait()`:
`pid_t waitpid(pid_t pid, int *status, int options)`

- `pid` determines which child process(es) to wait for:
  - < -1: any child whose PGID equals |`pid`|
  - -1: any child
  - 0: any child whose PGID equals that of the caller
  - > 0: child whose PID equals `pid`
- `options` will normally be 0 (zero) or `WNOHANG` (there are additional less used options)
- returns PID of collected child (-1 if no children exist)
- will suspend process if no current child zombies, but if `WNOHANG` was specified, will immediately return 0

waitpid() System Call (contd.)

`wait(&status)` is equivalent to `waitpid(-1,&status,0)`.

A frequent reason for using `waitpid()` is to include the `WNOHANG` flag, which causes `waitpid()` to return 0 (zero) and *not block* if no children have yet terminated.

A parallel server that creates a subprocess to handle each client can use `waitpid()` to periodically collect any zombie children:

```
while (waitpid(-1,NULL,WNOHANG) > 0);
```

[This loop keeps collecting terminated/zombie children until no more remain, at which point it returns 0 and the loop terminates.]

Other wait-family System Calls

Besides the two standard wait-family calls, the following are also available in Linux:

- **waitid()**
  - `int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options)`
  - provides more precise control over which child state changes to wait for
- **wait3()**
  - `pid_t wait3(int *status, int options, struct rusage *rusage)`
  - provides resource usage info
- **wait4()**
  - `pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage)`
  - provides resource usage info

Termination is Asynchronous

Parent are supposed to collect their children promptly, and collection can be done by calling `wait()/waitpid()`.

One problem parents may face, though, is that children's *process termination* is **asynchronous**.

Child process termination is asynchronous because the parent doesn't know at what point in its program any child might terminate.

Thus, it is unclear where in the parent code there should be "wait" calls to collect zombies promptly.

Furthermore, a (concurrent) server can sit waiting indefinitely for the next connection, and while waiting, it cannot collect any children that might terminate.

Children and Signals

Luckily, there approaches for collecting terminating children immediately, involving **signals**.

Signals are a mechanism that allows processes to be informed that some *event* has occurred.

The signal mechanism is inherently *asynchronous*, as signals can be delivered to a process at any time.

In fact, when a child process terminates, the kernel informs the parent of this event.

It does this by sending a **SIGCHLD signal** to the parent process.

Children and Signals (contd.)

The *default action* for the `SIGCHLD` signal is *ignore*, so the signal will have no effect on the parent by default (it is discarded).

The parent can however use signal handling calls to allow children to be "collected" when they terminate.

There are two approaches for doing this via signals:

- set the `SIGCHLD` signal to be ignored
- setup a handler for `SIGCHLD` to call `wait()/waitpid()`

Children and Signals (contd.)

The first approach actually *eliminates the need to collect children that have terminated*.

If the parent *explicitly* sets the `SIGCHLD` signal to be *ignored*, the kernel will remove a child from the process table when it terminates.

Thus, the parent will not have to collect its children because they will not remain as zombies.

`SIGCHLD` can be set to be *ignored* with `signal()`:

```
  signal(SIGCHLD,SIG_IGN);
```

Children and Signals (contd.)

The second signal-based approach is for the parent to set up a **signal handler** for the `SIGCHLD` signal.

A signal handler is a function that will be called whenever the appropriate signal is received.

It is common in servers to register a handler for `SIGCHLD` that executes the `waitpid()` loop shown earlier.

Then, whenever a child terminates, the `SIGCHLD` signal will cause the handler to run, collecting all existing zombie children.

See the **Signals** lectures for information about signal handlers.

Processes 3: Exec-Family Calls

1. Creation and Termination
2. Collecting Children
3. **Exec-Family Calls**
   - **the exec-family syscalls**
   - **attribute inheritance**
   - **system() and popen()**
4. Process Organization
5. Other Process Related Calls
6. Error Handling with Children

The exec-family System Calls

As noted previously, the parent and child processes will be running exactly the same code after a `fork()` call.

By testing the return value from `fork()` it is possible for each process to branch to different code or call different functions.

The drawback is that both processes address spaces will contain both the parent and child code.

The alternative is to use one of the **exec-family** system calls immediately after `fork()` *in the child*.

This will cause the child process to start running a different program (executable).

The exec-family System Calls (contd.)

The "exec family" of system calls actually consists of one true kernel system call and five C *library* syscall functions.

The true "exec" *system call* is `execve`:

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

- replaces the process' current address space with the program in `filename` and starts that program running
- `filename` must be either an executable binary file or script
- `argv` will be passed to the new program as its command line arguments (i.e., it will become `argv` for the program)
- `envp` is an array of strings of the form "key=value" that become the **environment** of the new program

The exec-family System Calls (contd.)

The other five "exec" calls provide alternative and often easier to use syntax (i.e., they are **syntactic sugar** for `execve`).

The other five "exec" calls are:

- `int execv(const char *path, char *const argv[])`
- `int execvp(const char *file, char *const argv[])`
- `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[])`

`execve` makes it six "exec" calls altogether:

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

The exec-family System Calls (contd.)

There are different groupings among the six "exec" calls:

- "v" (vector) vs. "l" (list) calls:
  - the "v" calls take an `argv` vector/array as input
  - the "l" calls take an arbitrary size list of arguments as input (which become the elements of `argv`)
- "p" (path) vs. "non-p" calls:
  - the "p" calls look through the search path (`PATH`) for the executable
  - the "non-p" calls require the executable's path
- "e" (environment) vs. "non-e" calls:
  - the "e" calls set up a new environment for the executable
  - the "non-e" calls inherit the current environment

l vs. v exec Calls

The choice between "l" and "v" calls will typically depend upon whether the number of arguments to be passed to the executable is *fixed/known* or is unknown until runtime.

If the number is fixed/known, then an "l" call avoids having to create an array and initialize it.

If the number of arguments must be determined in the program that calls "exec," then you must use a "v" call.

l vs. v exec Calls (contd.)

An example with an "l" call:

```
switch(fork()) {
  case -1:
    perror("fork failed");
    exit(EXIT_FAILURE);

  case 0:
    //Run program/executable mychild in child process:
    execlp("mychild","mychild","norman","110",(char*)NULL);
    //Should not get here, so exec call must have failed:
    perror("execlp failed")
    exit(EXIT_FAILURE);

  default:
    //Do what parent should do:
    ...
    wait(...);  //Must eventually collect child, after mychild terminates
  }
```

l vs. v exec Calls (contd.)

An example with a "v" call:

```
switch(fork()) {
  case -1:
    perror("fork failed");
    exit(EXIT_FAILURE);

  case 0:
    //Setup argument vector:
    char *argv[] = {"mychild","norman","110",(char*)NULL)};
    execvp("mychild",argv);
    perror("execlp failed")
    exit(EXIT_FAILURE);

  default:
    //Do what parent should do:
    ...
    wait(...);
  }
```

Executables Calling Conventions

In the code examples you will see that the program name/path was specified *twice*: as `path/file` and as what will be `argv[0]`.

This is because the standard Linux/UNIX *convention* is that `argv[0]` is to be the program name/path.

Since this is a standard convention, most programs will follow it, so you should generally adhere to it.

However, the examples make it clear that this convention is not imposed on you by "exec."

Executables Calling Conventions (contd.)

Notice in the examples that the final element of the argument list/vector is specified as `NULL`.

Having a `NULL` sentinel for `argv` allows a program to loop through `argv` until it encounters `NULL` (without using `argc`).

This is required for proper operation of some programs (though it is again not forced on you by "exec").

Though the **cast** of `NULL` is not always necessary, it is a good idea to ensure proper semantics—particularly with the "l" calls and their variable argument syntax.

Again: Parent or Child First?

As was discussed earlier with `fork()`, multi-process programs should *not* make assumptions about the parent or child running first after `fork()`.

They must also recognize that the following are **not atomic**:

- `fork()` followed by "`exec()`"
- "`exec()`" and the start of the program it invokes

So, even if we knew e.g. child runs before parent, we have no idea how far the child will get before the parent runs, nor what processes may execute between the above pairs of steps.

Parent or Child First? (contd.)

As noted earlier, the parent-child order uncertainty and lack of atomicity can make it critical to set up files and/or signal handling prior to a `fork()` or "`exec()`" call.

This will prevent possible **race conditions**.

The ability to setup files and signal handling prior to a `fork()` or "`exec()`" call is possible because certain process attributes are *inherited across* `fork()` and/or "`exec()`".

It is critical to understand and make use of this inheritance when working with multi-process programs.

Attribute Inheritance through exec

We saw that most process attributes are inherited by a child process through a `fork()` call.

What happens to process attributes through an "exec" call, where the entire address space is replaced?

Here are the most important attributes that *change or might change* across an "exec" call:

- the effective UID/GID are set according to the program being executed, so may change
- signals with a **registered handler** revert to **default disposition** (since handler code is gone)
- memory mappings, mutexes, locks, etc
- the **environment**, if one of the "e" calls was used

Attribute Inheritance through exec (contd.)

Here are the most important attributes that *do not change*:

- the PID and PPID
- the real UID/GID
- open file descriptors, unless the files are marked *close-on-exec* (false by default in Linux, but settable using `fcntl()`)
- CWD, umask, etc.
- **signal disposition** if set to *default* or *ignore*
- **signal mask** (blocked signals), pending signals, and timers/alarms

Attribute Inheritance through exec (contd.)

The fact that open file descriptors remain unchanged across "exec()" is made use of frequently.

In particular, it is common that the program to exec may do I/O to standard in/out/error, but we need the I/O to go to a pipe setup by the parent.

This is easily handled, by having the file descriptors for standard in/out/error **redirected** to the pipe (using `dup2()`) between the `fork()` and "`exec()`" calls.

Any redirected meaning of the FDs 0, 1, 2, will be inherited by the program that gets exec'd.

Attribute Inheritance through exec (contd.)

Redirecting standard in/out/error prior to "`exec()`" :

```
int pto[2], pipe(pto);  //pipe to write to command
int pfrom[2]; pipe(pfrom);  //pipe to read from command
...
switch(fork()) {
  case -1:
    ...
  case 0:  //Child process for command to be exec'd in:
    // Dup stdin, stdout, stderr to pipes:
    if (dup2(pto,0) == -1 && dup2(pfrom,0) == -1 && dup2(pfrom,0) == -1) {
      fprintf(stderr,"Error: failed to dup FDs: %s\n",strerror(errno));
      exit(EXIT_FAILURE);
    }
    //Close unwanted pipe ends:
    close(pto[0]); close(pto[1]); close(pfrom[0]); close(pfrom[1]);
    //Exec command:
    execvp(command,cmd_argv);
    exit(EXIT_FAILURE);  //catch exec failure
  default:  //Parent process:
    ...
}
```

Attribute Inheritance through exec (contd.)

Ineritance of signal handling varies significantly between `fork()` and "`exec()`" due to address space changes.

Signal handlers cannot be setup (in a child) prior to an "`exec()`" (since they would be in code wiped out by the "`exec()`").

A common pattern in multi-process programs is to need a signal handler in the *parent* process, to deal with error termination of children (i.e., handler for `SIGCHLD` signal from child termination).

The inability to inherit such a handler across "`exec()`" isn't an issue then, because the handler will need to be in the parent only.

However, such a handler will have to be setup *prior* to the `fork()` call, to ensure the handler is active before a `SIGCHLD` signal can be generated.

Attribute Inheritance through exec (contd.)

Setup `SIGCHLD` handler before `fork()` to ensure handler is active before a signal could be generated:

```
//Activate SIGCHLD handler:
struct sigaction act;
act.sa_handler = sigchld_handler;
act.sa_flags = 0;
sigemptyset(&act.sa_mask);
sigaction(SIGCHLD,&act,NULL);
switch(fork()) {
  case -1:
    ...
  case 0:  //Child process for command to be exec'd in:
    ...
    execvp(command,cmd_argv);  //command could fail and termiate child
    exit(EXIT_FAILURE);  //catch exec failure, so might terminate child
  default:  //Parent process:
    ...
}
```

Attribute Inheritance through exec (contd.)

Sometimes, the parent will need to keep track of information related to a new child, and this will need to happen before a `SIGCHLD` signal could invokes a handler, etc.

How could one ensure that the parent gets to run some code post `fork()`, even if child might run and terminate before the parent gets run?

What must be done is to **block** `SIGCHLD` signals in the parent, until the parent has been able to run the necessary code.

This can be done with `sigprocmask()`.

Attribute Inheritance through exec (contd.)

Block `SIGCHLD` signals in parent before `fork()` to ensure parent can execute some code before handler invoked:

```
//Activate SIGCHLD handler:
...
//Block SIGCHLD:
sigset_t sset;
sigemptyset(&sset);
sigaddset(&sset,SIGCHLD);
sigprocmask(SIG_BLOCK,&sset,NULL);
pid_t cpid;
switch(cpid = fork()) {
  case -1:
    ...
  case 0:  //Child process:
    ...
  default:  //Parent process:
    //Update info about child:
    child_pids[numchildren++] = cpid;
    //Unblock SIGCHILD:
    sigprocmask(SIG_UNBLOCK,&sset,NULL);
}
```

system() and popen()

As we have seen, some *multi-process program* make use of `fork()` followed by "`exec()`" to run a second executable in a subprocess of the main/parent process.

There are two system functions that can make this easier: `system()` and `popen()`.

These functions may be particularly appropriate when the second executable is a system utility like `sort`, and *shell* functionality is required for invocation of the utility.

system()

C provides a function to run a command in a shell subprocess:
`int system(const char *command)`

- creates a subprocess and `exec`'s "`/bin/sh -c command`"
- returns the exit status of `command`, else -1 on error (if `fork()` fails)
- when `command` is `NULL`, returns a nonzero value if shell available or zero if not

popen()

Linux/UNIX defines another function to run a command in a shell subprocess:
`FILE *popen(const char *command, const char *type)`

- creates a pipe, forks subprocess, and exec's shell in subprocess
- as with `system()`, `exec`'s "`/bin/sh -c command`"
- since a pipe is by definition unidirectional, `type` specifies only reading or writing: `"r"` or `"w"`
- return is a standard I/O stream, except it must be closed with `pclose()`
- writing to stream writes to the standard input of the command (standard output is parent's)
- reading from stream reads the command's standard output (standard input is parent's)

Processes 4: Process Organization

Process Organization

A **process** is the OS unit that holds a running program.

Processes are not just individual separate units; there is an organizational structure to the current set of processes.

The key organizational units are:

- **process groups**
- **sessions**

Basically, processes are grouped into *process groups*, which are grouped into *sessions*.

Process Groups

A **process group** is a group/set of processes.

Every process is part of one process group (though it may be the only process in the PG).

Every process has a **process group ID** (**PGID**) attribute, which identifies the process group that the process is a member of.

PGIDs are in the same namespace as PIDs, so a PGID is of type `pid_t`.

The **process group leader** is the process whose PGID is the same as its PID.

Process Groups (contd.)

When a new process is created with `fork()`, the new process automatically *inherits its parent's PGID*.

This means that the new child process is a member of the same process group as its parent.

The process group a process is a member of can be changed with the `setpgid()` syscall.

In particular, a process can put itself into its own process group.

setpgid() System Call

`setpgid()` sets the PGID of a process:
`int setpgid(pid_t pid, pid_t pgid)`

- process with PID `pid` has its PGID set to `pgid`
- if `pid` is zero, calling process has its PGID set to `pgid`
- if `pgid` is zero, the PGID of process with PID `pid` is made `pid` (i.e., that process becomes process group leader of a new process group)
- if `pid` and `pgid` are zero, the PGID of the calling process is set to its PID (i.e., the calling process becomes process group leader of a new process group)

If `setpgid()` is used to move a process from one process group to another, both process groups must be part of the *same* **session**.

Sessions

A **session** is a collection of process groups.

Every process has a **session ID** (**SID**) attribute, which identifies the session that the process is a member of.

When a new process is created with `fork()`, the new process automatically inherits its parent's SID (so it is a member of the same session as its parent).

Processes can change sessions with the `setsid()` syscall.

The **session leader** is the process whose SID is its PID (i.e., process that created the session).

Controlling Terminals and I/O

All of the processes in a *session* share a single **controlling terminal**.

The controlling terminal is established when the *session leader* first opens a **terminal device** (**TTY**).

A *terminal device* (TTY) may be the *controlling terminal* of at most one session.

Process groups in a session are classified as **foreground** or **background**.

At any point in time, a *single process group* in a session is the **foreground process group** for the TTY.

All other process groups are **background process groups**.

Controlling Terminals and I/O (contd.)

Only processes in the *foreground process group* can *do I/O* with the controlling terminal.

If a process in a *background process group* attempts to perform I/O with its controlling terminal, the kernel sends *signals* about this non-foreground process I/O.

The relevant signals are:

- SIGTTIN TTY input for background process
- SIGTTOU TTY output for background process

Controlling Terminals and I/O (contd.)

The *default action* for these two signals is to **stop** the process.

Stopping a process is a part of **job control**, where multiple processes can be running in the background.

Being *stopped* is *not* the same as being *terminated;* stopped processes can be **continued** (restarted).

Processes 5: Other Process Related Syscalls

1. Creation and Termination
2. Collecting Children
3. Exec-Family Calls
4. Process Organization
5. **Other Process Related Syscalls**
   - **IPC**
   - **synchronization**
   - **pipes**
6. Error Handling with Children

Other Process-Related System Calls

We have seen the primary syscalls for process management:

- process creation (`fork()`)
- process termination (`exit()`, `_exit()`)
- process collection (`wait()`, `waitpid()`)
- program execution in a process (the "exec" library calls)

Additional process-related syscalls are often required in *multi-process, concurrent* programs, to provide:

- **interprocess communication** (**IPC**)
- process **synchronization**

Interprocess Communication (IPC)

Since processes have separate address spaces, when they need to exchange information, **interprocess communication** (**IPC**) mechanisms must be used.

Linux/UNIX provides a number of IPC mechanisms:

- **pipes**
- **FIFOs** (**named pipes**)
- **sockets** (IP vs. UNIX/local; stream vs. datagram)
- **shared memory** (POSIX vs. System V
- **memory mapping** (mapped file vs. anonymous mapping)
- **message queues** (POSIX vs. System V)
- **signals**
- temporary files

Process Synchronization

Multi-process, concurrent programs will typically require the use of **synchronization** mechanisms.

Synchronization may be required to ensure operations among the processes occur only in desired orders, or to prevent multiple processes from simultaneously accessing **shared resources**.

Linux/UNIX provides several *synchronization mechanisms* that can be used among processes:

- **pipes/FIFOs**
- **signals**
- **semaphores**
- **file locks**
- **wait()** (for child termination)

Pipes

Pipes appear in both lists on the previous slides, and are used in many multi-process programs.

While primarily an *IPC* mechanism, pipes will inherently provide *synchronization* as part of transmitting data between processes.

`pipe()` is the system call to create a pipe:
`int pipe(int pipefd[2])`

`pipefd` is a two-element `int` array into which *file descriptors* for the *read and write ends* of the pipe will be placed.

Logically, using a pipe to transfer data between processes is equivalent to using a file as intermediate storage.

A pipe is a **kernel buffer**, however, so much faster than file I/O.

Pipes (contd.)

Basic, unnamed pipes are represented by file descriptors only, so must be used between "related" processes.

This is appropriate for most multi-process programs, which will typically consist of a parent process and one or more child processes.

Pipes are best used as *unidirectional* IPC mechanisms, to send data from one process to another process.

E.g., a parent can read data produced by a child, or a child can read data produced by a parent, or one children can exchange data.

pipe() Usage Example

Parent sends one message to child via pipe:

```
//Create pipe:  (must do before fork() so FDs inherited by child)
int pipefd[2];  //array to hold pipe FDs
pipe(pipefd);

//Create child and do its reading & printing functionality:
if (fork() == 0) {
  //In child:
  close(pipefd[1]);                //unused in child, close
  char buff[10];                   //storage for message
  read(pipefd[0],buff,10);         //read message from parent
  buff[11] = '\0';                 //turn buff into a string
  printf("pipe read: %s\n",buff);  //print message to stdout
  exit(EXIT_SUCCESS); }            //terminate child

//In parent (after fork()), so do its writing functionality:
close(pipefd[0]);                  //unused in parent, close
write(pipefd[1],"testing #1",10);  //write message to child
wait(NULL);                        //wait for child to terminate, collect
```

Pipes (contd.)

Synchronization comes "automatically" (with **blocking I/O**):

- if a process reads from an *empty* pipe, it will *block* (wait) for data to be written to the pipe by the other process
- if a process writes to a *full* pipe, it will *block* until the other process makes space by reading from the pipe

Using a pipe to transfer data from/to a subprocess is so common, there is a special Linux/UNIX function to do it:
`FILE *popen(const char *command, const char *type)`

`popen()` creates a pipe, forks a new subprocess, and then runs `command` in a *shell* in the subprocess.

`type` specifies whether the calling process can read data from the pipe (produced by `command`), or can write data to `command`.

Processes 6: Error Handling with Children

Error Handling with Children

In a *single process program*, when an error or other unwanted condition occurs so the program should not continue, we typically print a message and call `exit()` to terminate the process.

A *multi-process program* will have a *parent process* and one or more *child processes* that parent creates using `fork()` (there might also be grandchild processes, etc.).

Program termination for multi-process programs is typically more complicated than with single process programs.

For one thing, the error condition could occur in the *parent* process or in a *child* process.

If it occurs in the *parent*, calling `exit()` would terminate the parent, but it will generally *leave at least some children running*.

Error Handling with Children (contd.)

Even if terminating the parent causes some children to also terminate, those children will *not* have been *collected* if the parent terminates first.

How a parent's termination might cause child/ren to terminate:

- one example would be if a child runs, reading from a pipe until it gets an EOF return from reading, and the parent has the only open write end for the pipe
- then, when the parent terminates, the only pipe write end would close, causing the child to get EOF return from `read()`, in turn causing that child to terminate
- not only is it fairly rare for parent termination to cause all child processes to also terminate, the collection issue would remain

Error Handling with Children (contd.)

An error condition that should lead to program termination might also occur in a *child process*.

Calling `exit()` in the child will generally *terminate only that child*, not the parent or any sibling processes.

(As with parent termination, it is possible we could have parent reading from, say, a pipe, and terminating child has the only open write end.)

Plus, there is again the issue of the parent needing to collect all children.

Proper Multi-Process Program Termination

Properly terminating a multi-process program for an error condition requires the following three steps be completed:

1. all child processes must be terminated
2. the parent must collect all terminated children
3. the parent must terminate

These three steps must be completed (in order), whether the error condition occurs in the parent or in a child.

Making certain this always happens no matter where or when the error occurs, can be non-trivial.

Terminating Child Processes

A process can terminate itself by using `exit()`, `abort()`, etc.

How can one process cause another process to terminate?

This is done by the one process sending a **signal** to the other process.

Sending a signal from one process to another is done with `kill()`.

If a signal that causes termination is sent (and not ignored/handled), the receiving process can be terminated.

kill() System Call

Despite its name, the `kill()` system call can send any signal to a *process* or to a *process group:*

```
int kill(pid_t pid, int sig)
```

- `pid` determines which process(es) signal `sig` is sent to:
  - > 0: process with PID of `pid`
  - 0: every process in the *process group* of calling process
  - -1: every process calling process has permission to signal (except PID 1)
  - < -1: every process in the *process group* whose PGID is `-pid`
- returns zero on success, or -1 on error

kill() System Call (contd.)

`sig` should be one of the standard signal symbolic names.

The signals that are generally used to cause termination are:

- **SIGTERM** – preferred because process can **catch** and cleanup appropriately, but can be **blocked** or **ignored**, so may fail
- **SIGKILL** – cannot be **caught**, **blocked**, **ignored**, so will definitely cause termination

If `sig` is 0, then no signal is sent, but existence and permission checks are performed, which can be used to check for the existence of a process or process group.

A non-root user process can send a signal to another process, when the RUID or EUID of the sending process is the same as the RUID of the target process (processes of the same user).

killpg() System Call

A related call, to send a signal to all of the processes in a **process group** is `killpg()`:

`int killpg(int pgrp, int sig)`

- `pgrp` is the *process group ID* (PGID) to send signal `sig` to
- if `pgrp` is 0, `sig` is sent to the calling process's process group

Terminating Child Processes (contd.)

So one process of a multi-process program can terminate another process/processes of the program by using `kill()/killpg()`.

To be able to use these calls, one of three things must be true:

- the target process' PID is known to the sender
- the target process' PGID is known to the sender (and it is OK to terminate all processes in the process group)
- the target process is in the same process group as the sender (and it is OK to terminate all processes in the process group)

There are then two basic approaches for terminating the component processes of a multi-process program:

1. use PIDs with `kill()`
2. setup and use process groups with `killpg()`

Terminating Processes using PIDs

How is it possible for one process in a multi-process program to know the PID of another process in the program?

A process does not have "forward pointers" to its children (child process PIDs are not a process attribute).

However, since `fork()` returns the PID of each newly created child to the parent process, parents can *store* these PIDs for later use.

Child processes *do* have "back pointers" to their parent process, via the *parent process ID* (PPID) attribute of every process.

Children will *not*, though, have easy access to the PIDs of their *siblings* (the other child processes in a multi-process program).

Terminating Processes using PIDs (contd.)

Add in the fact that a *parent* is to collect all its children, and it is clear that having a child terminate all the processes in a multi-process program is *not appropriate*.

So termination via PIDs must be done by the *original/parent process* of the multi-process program.

This is relatively straightfoward when the parent process is the one that encounters the error condition.

An outline of the relevant code is shown next.

Terminating Processes using PIDs (contd.)

Code in parent to make children and store their PIDs:

```
int children_count = 0;
pid_t children_pids[MAX_CHILDREN];
...
while (...make required children...) {
  pid_t cpid;
  switch(cpid = fork()) {
    case -1:
      //fork error:
      ...
    case 0:
      //child process:
      ...
    default:
      //parent process:
      children_pids[children_count++] = cpid;  //store children PIDs
  }
}
```

Terminating Processes using PIDs (contd.)

Code in parent to handle an error condition and terminate children:

```
if (...error condition...) {
  fprintf(stderr,...error message...);
  // Terminate and collect all children:
  for (int i=0; i<children_count; i++) {
    kill(children_pids[i],SIGTERM);
    wait(NULL);
  }
  exit(EXIT_FAILURE);
}
```

We might also terminate and collect as:

```
  // Terminate all children:
  for (int i=0; i<children_count; i++) kill(children_pids[i],SIGTERM);
  // Collect all children:
  while (wait(NULL) != -1);
```

Terminating Processes using PIDs (contd.)

This is quite straightfoward, but what if the error condition occurs in one of the *child processes* of a multi-process program?

In that case, the child must somehow inform the parent about the error condition, so the parent can terminate and collect children.

The parent must be able to receive the information *asynchronously*, as an error in a child could in general occur at any point in the parent's execution (following the `fork()` that created the child).

Doing this with *signals* was discussed under **Signals and Children** in the earlier **Collecting Children** component of these lectures.

Terminating Processes using PIDs (contd.)

The parent process will need to setup a **signal handler** function for the `SIGCHLD` signal.

The child that encounters the error condition should terminate with failure exit status.

This will cause the signal handler function to be run, and when the handler finds the child failed, it can proceeed to terminate the entire program.

Since we must generally handle error conditions in either the parent or the children of a multi-process program, it is common to define an appropriate "cleanup and exit" function.

This function would be called directly by the parent, or from the `SIGCHLD` handler if a child terminates with failure.

Terminating Processes using PIDs (contd.)

Function in parent to cleanup and terminate if error:

```
void cleanup_and_exit(int exit_status)
{
  // Terminate all children:
  for (int i=0; i<children_count; i++) kill(children_pids[i],SIGTERM);
  // Collect all children:
  while (wait(NULL) != -1);
  //Terminate with appropriate status:
  exit(exit_status);
}
```

`SIGCHLD` handler function in parent to catch child termination:

```
void sigchld_handler(int signal)
{
  // Check on status of child that terminated:
  int status;
  wait(&status);
  If child terminate abnormally or with failure, terminate program:
  if (!WIFEXITED(status) || WEXITSTATUS(status0 != EXIT_SUCCESS))
    cleanup_and_exit(EXIT_FAILURE);
}
```

Terminating Processes using PIDs (contd.)

How parent would handle error:

```
if (...error condition) {
  fprintf(stderr,...error message...);
  cleanup_and_exit(EXIT_FAILURE);
}
```

Child would simply exit with failure:
(parent would catch `SIGCHLD` signal and terminate program)

```
if (...error condition) {
  fprintf(stderr,...error message...);
  exit(EXIT_FAILURE);
}
```

Terminating Processes using PIDs (contd.)

While this is not too complicated, there is one last issue that should be addressed: a **race condition** (since multi-process programs) are **concurrent**).

One cannot rely on whether parent or child runs first after `fork()`.

Consider what would happen if a child ran first, immediately hit an error, and terminated?

The parent might not yet have stored the child's PID when the `SIGCHLD` signal is received (and causes the handler to be run).

The parent would fail to terminate and collect the child, since it effectively doesn't yet know about it.

Terminating Processes using PIDs (contd.)

What is the solution?

Block the `SIGCHLD` signal prior to calling `fork()`, and unblock it only after the child PID has been stored.

Doing this, will prevent a `SIGCHLD` signal being delivered to the parent until after it has updated its list of PIDs.

Race condition eliminated!

Terminating Processes using PIDs (contd.)

Revised code in parent to make children and store their PIDs:

```
int children_count = 0;
pid_t children_pids[MAX_CHILDREN];  //store all children PIDs in parent
sigset_t sset;
sigemptyset(&sset);
sigaddset(&sset,SIGCHLD);
...
sigprocmask(SIG_BLOCK,&sset,NULL);  //block SIGCHLD while making children
while (...make required children...) {
  pid_t cpid;
  switch(cpid = fork()) {
    case -1:
      ...
    case 0:
      ...
    default:
      //parent process:
      children_pids[children_count++] = cpid;
  }
}
sigprocmask(SIG_UNBLOCK,&sset,NULL);  //unblock SIGCHLD since parent updated
```

Terminating Processes using Process Groups

Using *process groups* can slightly simplify terminating children, since a single `killpg()` can terminate all children, instead of having to loop through a list of PIDs.

In addition, the parent must store only the process group PGID for all its children, instead of the separate PIDs for each of its children.

However, putting children all in a single process group—that is separate from the parent—does require a bit of code.

The parent cannot be in the process group with the children, or `killpg()` would terminate the parent as well, meaning it could not collect the children.

By default, children inherit the PGID from their parent—i.e., they are put into the same process group as the parent.

Terminating Processes using PGs (contd.)

So, to be able to use `killpg()` to terminate all children:

1. a new process group must be created
2. each child must be put into that new process group

The `setpgid()` syscall sets the PGID (and so PG) of a process:
`int setpgid(pid_t pid, pid_t pgid)`

A new process group can be created by using a PID that is not a PGID as the `pgid` argument to `setpgid()`.

A process can be put into a process group using `setpgid()` with the new PGID as its `pgid` argument.

The parent can do this as it creates each child.

Terminating Processes using PGs (contd.)

Code in parent to make children with desired PG:

```
int children_count = 0;
pid_t children_pgid;
sigset_t sset;
sigemptyset(&sset);
sigaddset(&sset,SIGCHLD);
...
sigprocmask(SIG_BLOCK,&sset,NULL);  //block SIGCHLD while making children
while (...make required children...) {
  pid_t cpid;
  switch(cpid = fork()) {
    case -1:
      ...
    case 0:
      ...
    default:
      //parent process:
      if (children_count++ == 0) children_pgid = cpid;  //use first child PID as P
  }
}
sigprocmask(SIG_UNBLOCK,&sset,NULL);  //unblock SIGCHLD since parent updated
```

Terminating Processes using PGs (contd.)

Note that we must again deal with a possible *race condition* in this code, by blocking `SIGCHLD` until the parent has completed its post-`fork()` tasks.

A single `killpg()` call can now be used in `cleanup_and_exit()` instead of looping on the stored PIDs with `kill()`.

Not a huge advantage, but makes it easier to handle cases where variable numbers of child processes may need to be created.