Process-related Linux Functions/System Calls

- **posix_spawn()** ... create a new process, see also
  - **clone()** ... duplicate current process
    address space can be shared to implement threads
    only use clone if posix_spawn can’t do what you want
  - **fork()** ... duplicate current process - don’t use
  - **execve()** ... replace current process - don’t use

- **exit()** ... terminate current process, see also
  - **_exit()** ... terminate current process immediately
    stdio buffers won’t be flushed
    atexit functions won’t be called

- **getpid()** ... get process ID
- **getpgid()** ... get process group ID
- **waitpid()** ... wait for state change in child process
Unix/Linux system calls:

- **kill()** ... send a signal to a process
- **sigaction()** ... specify behaviour on receiving a signal
  - **signal()** simpler version of sigaction, hard to use safely
- **sleep()** ... suspend execution for specified time
posix_spawn()
int main(void) {
    pid_t pid;
    extern char **environ;
    char *spawn_argv[] = {"/bin/date", "--utc", NULL};
    if (posix_spawn(&pid, "/bin/date", NULL, NULL,
                    spawn_argv, environ) != 0) {
        perror("spawn");
        return 1;
    }

    int exit_status;
    if (waitpid(pid, &exit_status, 0) != 0) {
        perror("waitpid");
        return 1;
    }

    printf("date exit status was %d\n", exit_status);
}
fork()

pid_t fork()

• requires `#include <unistd.h>`
• creates new process by duplicating the calling process
• new process is the *child*, calling process is the *parent*
• child has a different process ID (pid) to the parent
• in the child, `fork()` returns 0
• in the parent, `fork()` returns the pid of the child
• if the system call fails, `fork()` returns -1
• child inherits copies of parent’s address space and open fd’s
#include <stdio.h>
#include <unistd.h>

int main(void) {
    pid_t pid = fork();
    if (pid == -1) {
        // the fork failed, perror will print why
        perror("fork");
    } else if (pid == 0) {
        printf("child: fork() returned %d.\n", pid);
    } else {
        printf("parent: fork() returned %d.\n", pid);
    }
}
int execvp(char *Path, char *Argv[])

- transforms current process by executing *Path* object
  - *Path* must be an executable, binary or script (starting with `#!`)
- passes arrays of strings to new process
  - both arrays terminated by a NULL pointer element
- much of the state of the original process is lost, e.g.
  - new virtual address space is created, signal handlers reset, ...
- new process inherits open file descriptors from original process
- on error, returns -1 and sets *errno*
- if successful, does not return
void exit(int status)
  • triggers any functions registered as atexit()
  • flushes stdio buffers; closes open FILE *’s
  • terminates current process
  • a SIGCHLD signal is sent to parent
  • returns status to parent (via wait())
  • any child processes are inherited by init (pid=1)
  • termination may be delayed waiting for i/o to complete

Also void _exit(int status)
  • terminates current process immediately;

Related function: void abort(void)
  • generates SIGABRT signal (normally terminates process)
  • closes and flushes stdio streams
  • used by the assert() macro
Zombie Process

Zombie Process?

Photo credit: Kenny Louie, Flickr.com
Process-related System Calls

When a process finishes, sends SIGCHLD signal to parent

*Zombie process* = a process which has exited but signal not handled
  * all processes become zombie until SIGCHLD handled
  * parent may be delayed e.g. slow i/o, but usually resolves quickly
  * bug in parent that ignores SIGCHLD creates long-term zombies
  * note that zombies occupy a slot in the process table

*Orphan process* = a process whose parent has exited
  * when parent exits, orphan is assigned pid=1 as its parent
  * pid=1 always handles SIGCHLD when process exits
getpid & getppid

Getting information about a process ...

```c
pid_t getpid()

• requires #include <sys/types.h>
• returns the process ID of the current process
```

```c
pid_t getppid()

• requires #include <sys/types.h>
• returns the parent process ID of the current process
```
Processes belong to *process groups*

- a signal can be sent to all processes in a process group

```c
pid_t getpgid(pid_t pid)
```

- returns the process group ID of specified process
- if `pid` is zero, use get PGID of current process

```c
int setpgid(pid_t pid, pid_t pgid)
```

- set the process group ID of specified process

Both return -1 and set `errno` on failure.

For more details: `man 2 getpgid`
waitpid

pid_t waitpid(pid_t pid, int *status, int options)
  • pause current process until process pid changes state
    • where state changes include finishing, stopping, re-starting, ...
  • ensures that child resources are released on exit
  • special values for pid ...
    • if pid = -1, wait on any child process
    • if pid = 0, wait on any child in process group
    • if pid > 0, wait on the specified process

pid_t wait(int *status)
  • equivalent to waitpid(-1, &status, 0)
  • pauses until one of the child processes terminates
waitpid

More on \texttt{waitpid(pid, \&status, options)}

- **status** is set to hold info about \texttt{pid}
  - e.g. exit status if \texttt{pid} terminated
  - macros allow precise determination of state change
    (e.g. \texttt{WIFEXITED(status)}, \texttt{WCOREDUMP(status)})

- **options** provide variations in \texttt{waitpid()} behaviour
  - default: wait for child process to terminate
  - \texttt{WNOHANG}: return immediately if no child has exited
  - \texttt{WCONTINUED}: return if a stopped child has been restarted

For more information: \texttt{man 2 waitpid}
Processes: review

Process = instance of an executing program

- defined by execution state (incl. registers, address space, ...)

Operating system shares CPU among many active processes

On Unix/Linux:

- each process had a unique process ID (pid)
- `posix_spawn()` creates a copy of current process
- `wait()` parent process waits for child to change state
int kill(pid_t ProcID, int SigID)

• requires #include <signal.h>

• send signal SigID to process ProcID

• various signals (POSIX) e.g.
  • SIGHUP ... hangup detected on controlling terminal/process
  • SIGINT ... interrupt from keyboard (control-C)
  • SIGKILL ... kill signal (e.g. kill -9)
  • SIGILL ... illegal instruction
  • SIGFPE ... floating point exception (e.g. divide by zero)
  • SIGSEGV ... invalid memory reference
  • SIGPIPE ... broken pipe (no processes reading from pipe)

• if successful, return 0; on error, return -1 and set errno
Signals can be generated from a variety of sources

- from another process via `kill()`
- from the operating system (e.g. timer)
- from within the process (e.g. system call)
- from a fault in the process (e.g. div-by-zero)

Processes can define how they want to handle signals

- using the `signal()` library function (simple)
- using the `sigaction()` system call (powerful)
Signals from internal process activity, e.g.

- SIGILL ... illegal instruction  (Term by default)
- SIGABRT ... generated by `abort()`  (Core by default)
- SIGFPE ... floating point exception  (Core by default)
- SIGSEGV ... invalid memory reference  (Core by default)

Signals from external process events, e.g.

- SIGINT ... interrupt from keyboard  (Term by default)
- SIGPIPE ... broken pipe  (Term by default)
- SIGCHLD ... child process stopped or died  (Ignored by default)
- SIGTSTP ... stop typed at tty (control-Z)  (Stop by default)
Processes can choose to ignore most signals. If not ignored, signals can be handled in several default ways:

- Term ... terminate the process
- Core ... terminate the process, dump core
- Stop ... stop the process
- Cont ... continue the process if currently stopped

Or you can write your own *signal handler*

See `man 7 signal` for details of signals and default handling.
Signal Handler = a function invoked in response to a signal
• knows which signal it was invoked by
• needs to ensure that invoking signal (at least) is blocked
• carries out appropriate action; may return
SigHnd signal(int SigID, SigHnd Handler)

• define how to handle a particular signal
• requires <signal.h> (library function, not syscall)
• SigID is one of the OS-defined signals
  • e.g. SIGHUP, SIGCHLD, SIGSEGV, ... but not SIGKILL, SIGSTOP
• Handler can be one of ...
  • SIG_IGN ... ignore signals of type SigID
  • SIG_DFL ... use default handler for SigID
  • a user-defined function to handle SigID signals

• note: typedef void (*SigHnd)(int);
• returns previous value of signal handler, or SIG_ERR
int sigaction(int sigID,
               struct sigaction *newAct,
               struct sigaction *oldAct)

• `sigID` is one of the OS-defined signals
  - e.g. SIGHUP, SIGCHLD, SIGSEGV, ... but not SIGKILL, SIGSTOP
• `newAct` defines how signal should be handled
• `oldAct` saves a copy of how signal was handled
• if `newAct.sa_handler` == SIG_IGN, signal is ignored
• if `newAct.sa_handler` == SIG_DFL, default handler is used
• on success, returns 0; on error, returns -1 and sets `errno`

For much more information: man 2 sigaction
Signal Handlers

Details on struct sigaction ...

- void (*sa_handler)(int)
  - pointer to a handler function, or SIG_IGN or SIG_DFL
- void (*sa_sigaction)(int, siginfo_t *, void *)
  - pointer to handler function; used if SA_SIGINFO flag is set
  - allows more context info to be passed to handler
- sigset_t sa_mask
  - a mask, where each bit specifies a signal to be blocked
- int sa_flags
  - flags to modify how signal is treated
    (e.g. don’t block signal in its own handler)
Details on \texttt{siginfo\_t} ...

- \texttt{si\_signo} ... signal being handled
- \texttt{si\_errno} ... any \texttt{errno} value associated with signal
- \texttt{si\_pid} ... process ID of sending process
- \texttt{si\_uid} ... user ID of owner of sending process
- \texttt{si\_status} ... exit value for process termination

etc. etc. etc.

For more details: \texttt{bits/types/siginfo\_t.h} (system-dependent)
A *process* is an instance of an executing program. Each process has an *execution state*, defined by:

- current execution point (PC register)
- current values of CPU registers
- current contents of its virtual address space
- information about open files, sockets, etc.

To manage processes, the operating system also maintains:

- process page table (i.e. virtual memory mapping)
- process metadata (e.g. execution time, priority, ...)

**Processes**
Processes

On a typical modern operating system

- multiple processes are active "simultaneously" (*multi-tasking*)

The operating system provides each process with

- control-flow independence
  - each process executes as if the only process running on the machine

- private address space
  - each process has its own address space (N bytes, addressed 0..N-1)

*Process management* is a critical OS functionality
Processes

Control-flow independence ("I am the only process, and I run until I finish")
When there are multiple processes running on the machine
  • each process uses the CPU until pre-empted or exits
  • then another process uses the CPU until it too is pre-empted
  • eventually, the first process will get another run on the CPU

Overall impression: three programs running simultaneously
What can cause a process to be pre-empted?

- it runs "long enough" and the OS replaces it by a waiting process
- it attempts to perform a long-duration task, like input/output

On pre-emption ...

- the process’s entire dynamic state must be saved (incl PC)
- the process is flagged as temporarily suspended
- it is placed on a process (priority) queue for re-start

On resuming, the state is restored and the process starts at saved PC

Overall impression: I ran until I finished all my computation
How does the OS manage multiple simultaneous processes? For each process, maintains context (or state)
- static information: program code and constant data
- dynamic state: heap, stack, registers, program counter
- OS-supplied state: environment variables, stdin, stdout

At pre-emption, performs a context switch
- save context for one process
- restore context for another process

Non-static process context is held in a process control block
Typical contents of *process control block* (PCB)

- **identifier**: unique process ID (*int*)
- **status**: running, ready, suspended, exited
  - if suspended, event being waited for
- **state**: registers (including PC)
- **privileges**: owner, group
- **memory management info**: (reference to) page table
- **accounting**: CPU time used, amount of I/O done
- **I/O**: open file descriptors
Process Management

The operating system maintains a table of PCBs
  • one for each currently active process  (indexed by process ID?)

The OS scheduler
  • maintains a queue of runnable processes
  • ordered based on information in the PCBs

When current process is pre-empted or suspends, the scheduler
  • saves state of process, updates PCB entry
  • selects next process to run, and re-starts it
Unix/Linux Processes

Environment for processes running on Unix/Linux systems

argc, argv, envp, uid, gid, ...

Process

stdin (fd:0) → stdout (fd:1) → stderr (fd:2)

return status
(0 = ok, !0 = error)
Unix/Linux Processes

Unix provides a range of tools for manipulating processes

Commands:

• `sh` ... for creating processes via object-file name
• `ps` ... show process information
• `w` ... show per-user process information
• `top` ... show high-cpu-usage process information
• `kill` ... send a signal to a process

System calls:

• `fork()`, `execve()`, `_exit()`, etc.
Exercise: Process Information

How can I find out ...

• what processes I currently have running
• what are all of the processes running on the system
• what are the top CPU-using processes
• who’s logged in and what they’re doing
Unix/Linux Processes

Information associated with processes (PCB):
- **pid** ... process id
- **ruid, euid** ... real and effective user id
- **rgid, egid** ... real and effective group id
- current working directory
- accumulated execution time (user/kernel)
- user file descriptor table
- information on how to react to signals
- pointer to process page table
- process state ... running, suspended, asleep, etc.
Unix/Linux Processes

Process info is split across process table entry and user structure

*Process table* = kernel data structure describing all processes

- memory-resident since very heavily used
- contains PCB info as described above
- content of PCB entries is critical for scheduler

*User structure* = kernel data structure describing run-time state

- holds info not needed when process swapped out
- e.g. execution state (registers, signal handlers, file descriptors, ...)

Unix/Linux Processes

Every process in Unix/Linux is allocated a process ID (PID)

- a +ve integer, unique among currently executing processes
- with type pid_t (defined in `<unistd.h>`)
- process 0 is the idle process (always runnable)
- process 1 is init ("the system")
- low-numbered processes are typically system-related

Process 0 is not a real process (it’s a kernel artefact)

- it exists to ensure that there is always at least one process to run

On older Unix systems, process 0 was called sched
Unix/Linux Processes

Each process has a *parent process*
  - typically, the process that created the current process
A process may have *child processes*
  - any processes that it created
Process 1 is created at system startup
If a process’ parent dies, it is inherited by process 1
Unix/Linux Processes

Processes are collected into *process groups*

- Each group is associated with a unique PGID
- With type `pid_t` (defined in `<unistd.h>`)  
- A child process belongs to the process group of its parent
- A process can create its own process group, or can move into another process group

Process groups allow

- OS to keep track of groups of processes working together
- Distribution of signals to a set of related processes
- Management of processes for job control (control-Z)
- Management of processes within pipelines
System Calls (and Failure)

Reminder ...
System calls are requests for the OS to do something, e.g.
  • create a new process, send a signal, read some data, etc.
Sometimes the request cannot be completed, e.g.
  • invalid PID or file descriptor, resources exhausted, etc.
In such cases
  • the system call returns -1
  • the value of the global variable `errno` is set
In many (most?) cases, a failed system call is a fatal error.
How to deal with failed system calls?
Generally, print an error and terminate the process.
A useful strategy: a wrapper function
• with same arguments/returns as system call
• catches and reports the error
• only ever returns with a valid result
Not always appropriate, e.g.
• failure of open() best handled by caller
Example: a wrapper function for `read()`

```c
size_t read1(int fd, void *buf, size_t nbytes) {
    ssize_t nread = read(fd, buf, nbytes);
    if (nread < 0) {
        perror("read() failed");
        exit(1);
    }
    return nread;
}
```

Use like `read()` but only get non-negative returns.