Process management

- Operating-System-Concepts[1]
  - as a process executes, can have one of the following states
    - *new* being created
    - *running*
    - *wating* e.g. for an I/O completion of receiving a signal
    - *ready* waiting to be assigned
    - *terminated*
- **PCB** - **P**rocess **C**ontrol **B**lock
  - the kernel datastructure representing a process in an OS
  - containg information such as process state, list of open files, pointer to process' parent etc
  - in Linux represented by *task_struct* (doubly linked list), exposed via `/proc/<process-id>/...`
- process scheduling queues
  - *ready queue* - ready and waiting to be executed
  - *waits queue* - processes waiting for a certain event such as completion of I/O
- **IPC** interprocess communication (IPC), two fundamental models
  - (1) shared memory e.g. `shm_open` - create/open POSIX shared memory objects
  - (2) message passing
  - *pipes* - allow two processes to communicate in standard producer-consumer fashion
    - ordinary / anonymous pipes
    - names pipes or FIFOs - `mkfifo()` syscall
  - *sockets* - 'socket' = endpoint for communication
  - *RCP* - remote procedure calls
- Linux-Kernel-Development[2]
  - processes provide two virtualizations
    - (1) a virtualized processor
    - (2) virtual memory
  - in current Linux, `fork()` is implemented via the `clone()` system call; clone man page:
    - *By contrast with fork(2), these system calls provide more precise control over*
    - *what pieces of execution context are shared between the calling process and the child process.*
  - process state: `state` field in `task_struct`:
    - `TASK_RUNNING`
    - `TASK_INTERRUPTIBLE` *waiting* - sleeping/blocked, receives and reacts to signals
    - `TASK_UNINTERRUPTIBLE` *waiting* - does not wake and become runnable (even when recv signal)
    - `__TASK_TRACED` - being traced (e.g. debugger, ptrace)
    - `__TASK_STOPPED`

note:
> On Linux, 0 typically represents success; a nonzero value, such as 1 or -1, corresponds to failure.

- in Unix, the creation of a new process and the act of loading a new binary is separated
  - `fork()`
  - `exec()`
- *idle process* (PID=0) process the kernel 'runs' when there's not other runnable process
- *init process* (PID=1) first process kernel executes after booting
  - can be specified with *init* kernel command-line parameter
  - kernel tries four executables:
    - (1) `/sbin/init`
    - (2) `/etc/init`

[1] https://codex.cs.yale.edu/avi/os-book/
[2] https://www.oreilly.com/library/view/linux-kernel-development/9780768696974/

    * (3) `/bin/init`
    * (4) `/bin/sh`

- **process id allocation**
  - kernel: pid is of type `pid_t`, generally `int` e.g. *posix_types.h* `typedef int  __kernel_pid_t;`
  - maximum process id:
    ```
    $ cat /proc/sys/kernel/pid_max
    4194304
    ```
  - allocation is in a strictly linear fashion
  - kernel does not reuse process IDs until it wraps around from top
- **process hierarchy**
  - every process is spawned from another process (exept `init` with pid 1)
  - releationship is recorded in *parent process ID* (**ppid)
  - each process is part of a *process group*
    * e.g. `ls|less` processes belong to same process group

    ```c
    #include <unistd.h>
    #include <stdio.h>
    int main() {
    printf("pid  = %d\n", getpid());
    printf("ppid = %d\n", getppid());
    }
    ```
- **exec()**
  - there's no single exec function, instead a range of exec functions built on singel syscall
  - `execl()` replaces current process image with new one specified with `path`
    ```c
    int main() {
      execl("/bin/ls", "ls", NULL);
    }
    ```
  - Unix convention is to pass the program name as the program's first argument.
  - open files are inherited across an exec
    * often not the desired behavior
    * usual practice is to close files before the exec
    * can be done automatically via kernel with `fcntl()` (fcntl=file control)
- **fork()**
  - fork() creates a new process
  - child and the parent process are (almost) identical except for
    * pid / ppid
    * resource statistics (reset to zero in child)
    * any pending signals are cleared
    * file locks are not inherited by child
      ```c
      int main() {
        pid_t pid = fork ();
        //sleep(1);
        if (pid > 0)
          printf ("Parent of new child; child pid=%d\n", pid);
        else if (!pid)
          printf ("Child with pid=%d and ppid=%d\n", getpid(), getppid() );
      }
      ```
      ```
      # output without sleep -> parent terminates first and kernel reparents child
       Parent of new child; child pid=60525
       Child with pid=60525 and ppid=1
      # output with sleep
       Parent of new child; child pid=60874
       Child with pid=60874 and ppid=60873
      ```
    * `fork()` - returns 0 to the child process, and returns pid of child process to the parent

process
```
pid_t pid = fork();
if (pid==0) printf("I am the child.\n");
if (pid!=0) printf("I am the parent.\n");
printf("..and I will be printed twice. pid=%d\n", getpid());
# output:
 I am the parent.
 ..and I will be printed twice. pid=68782
 I am the child.
 ..and I will be printed twice. pid=68783
```

  - **copy-on-write**
    * modern Unix systems do not copy the parent's address space, but employ copy-on-write (COW) pages
    * COW is a lazy optimization strategy designed to mitigate the overhead of duplicating resources
    * modern machine architectures provide HW support for COW in their memory management units (MMUs)
- **exit() / _ exit()**
  - standard function for terminating the current process
  - `exit()` (Standard C Library function), performs following functions (in order):
    * (1) Call the functions registered with the atexit(3) function, in the reverse order of their registration.
    * (2) Flush all open output streams.
    * (3) Close all open streams.
    * (4) Unlink all files created with the tmpfile(3) function.
  - when done with above steps, `exit()` invokes the syscall `_ exit()`
  - the kernel handles the rest of the termination process

daemons

- process that runs in the background, not connected to a (controlling) terminal
- two requirements:
  - must run as child of *init*
  - must not be connected to terminal
- for a program to become a daemon:
  - (1) `fork()`
  - (2) in the parent call `exit()`; *reparents* process to pid 1 (=init)
  - (3) call `setsid()` - with process itself as leader
  - (4) change the working directory to the root directory "/" via `chdir()`
  - (5) close all file descriptors (`ulimit -a`: open files (-n) 1024)
  - (6) open file descriptors 0, 1, 2 and redirect them to `/dev/null`

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
int main (void) {
        pid_t pid = fork (); // (1)
        if (pid != 0)  // (2)
          exit (EXIT_SUCCESS);
        setsid (); // (3)
        chdir ("/"); // (4)
        for (int i = 0; i < 1024; i++) // (5)
                close (i);
        // (6) note: open() always allocates lowest available fd number
```

```
        int fd = open ("/dev/null", O_RDWR); /* 0 stdin */
        dup (0);                             /* 1 stdout */
        dup (0);                             /* 2 stderror */
        /* daemon code goes here */
        return 0;
}
```